flat assembler Design Principles

Why flat assembler is Different

The purpose of this article is to describe the main ideas that led the flat assembler project during the period of its development. The initial design nominates the direction in which the program can evolve and limits somewhat the possible extensions to its capabilities. I wrote this text in March 2005 to explain how flat assembler has come to the point where it is and what were the reasons for the many design decisions I made.

1. The Roots - Turbo Assembler versus Netwide Assembler

When I began learning assembly language, I was exclusively using the Borland's Turbo Assembler, but it was a commercial product and I didn't own a personal copy. That's why I got interested in developing a new product. Netwide Assembler (NASM), which was free and even open source, contained some good ideas, but I was generally disappointed by the lack of many features I was used to while using TASM. So, I gave it up and never even started using NASM for any of my projects. Instead I tried, with success, writing my own assembler (actually I wrote two, fasm being the latter one, but I will skip over describing the first one, as it had the same syntax and less features), with sufficient capabilities to assemble all my previously written projects with only minor changes in the source code.

It should be obvious that the syntax I've chosen for fasm was primarily imitating the one I was using when programming with TASM. It's important to note that TASM offered two modes, with different syntaxes. The first mode, which was the default MASM-Compatibility Mode. The second one was called Ideal Mode. After learning the basics of assembly language, I quickly switched to the Ideal Mode as I found it easier to use and less confusing.

There are two main characteristics of the Ideal Mode that I followed when designing the syntax for flat assembler. The first one was the syntax for accessing contents of memory. TASM with Ideal Mode selected requires operands to be always surrounded with square brackets to ensure that the given operand will be always interpreted as memory contents - while in MASM mode the square brackets were interpreted differently in various situations, giving me a distressing feeling of chaos. So, the use of square brackets to mark memory operands was something I got used to very quickly. I put the same syntax rule to use in my own assembler when I was designing it. NASM had gone into the same direction and simplified it even further. With NASM, when you define the variable of some name, this name becomes a constant equal to the address of that variable. Therefore, every label is just a constant. Nice and simple, but it was one of those things in NASM that made me dissatisfied. Because I was used to the fact that when I defined some variable as byte:

```
        alpha db 0
```

and then tried to access it like this:

```
        mov [alpha],ax
```

TASM would refuse to accept it because I tried to store some larger data in a smaller variable. This feature was catching many mistakes and I felt I could not waive it. However, I still liked the idea of treating labels just like constants equal to an address. It made such instructions as:

```
        mov ax,alpha
        mov ax,[alpha]
```

a straightforward analogy of:

```
        mov ax,bx
        mov ax,[bx]
```

and with such syntax it's very simple and easy to, for example, adjust some algorithm to use absolute addressing instead of register-based, or vice versa.

The consequence of this approach was the elimination of the OFFSET operator, but it was a change I could accept - it was enough to replace OFFSET word with empty string in all my sources. However, in flat assembler, every label, though being just an address at first sight, still keeps the information about what type of variable is defined behind it - and provides the size checking just like I had with TASM.

Of course, in assembly language programming, there is still a need to force different size operands when you want to use them. With TASM, the size override operator had to be put before the name of the variable, inside a set of square brackets. Since I've followed NASM in interpreting the square brackets as a variable, (with address inside identifying which one is it), it was more logical to require the size operator before the square brackets, and it's also consistent with another feature taken from NASM, which is that any operand can be preceded with a size operator even though it might be redundant. But it's not necessary to use this feature as frequently as with NASM since thanks to keeping the information about variable types fasm is generally able to guess the size - this way I got what I felt was the best of the both worlds and was the first milestone in fasm's syntax design. Fasm only needed small changes in source code to convert the instructions to the new syntax. An example for comparison:

```
        mov [byte cs:0],0 ; TASM Ideal
        mov byte [cs:0],0 ; fasm
```

The second characteristic attribute of syntax which I have taken from TASM's Ideal mode is putting the defining directive before the name of the defined object. This does not apply to data definitions, but directives like LABEL, MACRO or PROC worked this way in Ideal mode, while in MASM mode the name always appeared before the directive. Perhaps because of some previous habits from higher level languages (like Pascal), I also liked the variant of Ideal Mode better.

So, I have copied the syntax of LABEL and MACRO directives from the TASM's Ideal Mode syntax, with only one change, that contents of macroinstructions had to be enclosed with braces instead of ending with the ENDM directive. It was just because I liked the braces. They were simpler to parse too.

I have also implemented the LOCAL directive with the same syntax I had with TASM and in this way I was able to implement all the features I was actually using with TASM. Other, more powerful, macroinstruction features were implemented much later when the influence of TASM waned and other design priorities (which are described next) took precedence.

To the list of things that were taken from TASM I might also add the USE16 and USE32 keywords, though TASM allowed them only in the segment declaration, while fasm allows them to be used to switch the type of generated code anywhere in a program. This is where the second design principle came to life.

2. Flexibility - OS Development Appliances

To understand the origins of flat assembler it's also important to notice that I've been trying some OS development at the same time. I was designing fasm as a tool aimed mainly for this purpose. That's why it was important to make it easily portable and as I soon as I finished it, I ported it into the OS I was developing. This allowed me to write programs for it in their native environment. That also might be considered the reason why I wrote fasm as assembly language. However, it's more likely because I was doing all my programming in assembly language at the time - if I really preferred some high-level language I would make some self-compiling high-level compiler instead.

Still, for the OS development, it was necessary to assemble some sophisticated pieces of assembly code, with switching of code type and addressing modes, which was actually quite complicated when done with TASM. I especially hated the necessity of manually building some instruction opcodes with DB directives. So, I built into my assembler all the instruction variants and size settings that were needed to declare any instruction without doubting what operation would be perform - like 32-bit far jumps in 16-bit mode and other similarly rare, but required in OS startup code instructions. Also, the decision to require the size operator before the whole memory operand, that is outside the square brackets, became profitable at this time as it allowed me to insert the size operator inside the square brackets when applying the size of address value.

Also, for the purposes of OS development I have implemented the ORG directive, which behaves a bit differently than the original one found in TASM. What I needed was the ability to set the origin address of given code, but without actually moving the output point in the file. I thought it should be the programmer's responsibility to load the code at the desired origin like DOS does with .COM programs - this is again important in OS kernel development, where you may have many different pieces of code that will be put in many different places and can be addressed in many different ways. The ORG directive in my version allows the design of code to work correctly when loaded at specified origins while its placement in a file is just determined by the order of source code. My assembler, generating the code in flat addressing space, was always outputting the code exactly in the same order as it was defined in source. Thus, came the name for it - flat assembler.

And for the same reason I invented a completely new feature - the VIRTUAL directive. With TASM, when I wanted to access some OS kernel structures I placed at addresses different than the ones in the kernel code space, I had to calculate addresses manually (usually defining the chains of constants, where each one was equal to the previous one incremented by the size of data it addressed). My new directive allowed declared structures to be at the given address without putting any actual data into the output. Some other applications of the VIRTUAL directive were invented much later, but initially, it was only this one.

The output of flat assembler was by default just the plain binary, as it was the most convenient for OS programming and allowed me to create .COM programs as well. But I soon added the option to output relocatable formats, which I designed for my operating system (I have removed this feature in official releases). However, the output format was not selected using a command line switch, but with a directive - this was an idea completely different from what other assemblers offered, the direct consequence of the new principle I came up with in fasm.

3. Same Source, Same Output

The problem with command line switches selecting output options in low level assembly programming is that the given code will most probably assemble and execute correctly only when the same output is selected that the programmer had in mind when writing the source code. Also, I remembered many cases when I had a source for TASM written by someone else, and to compile it correctly, I had to follow the directions given in the comments at the beginning of the source code and just rewrite all the command line switches as described there. My though was: then why don't I just make assembler look for such options in the source instead so nobody will experience problems when recompiling. Thus, came the SSSO principle - all the settings that can affect the output of assembler are selected only from source and source is always enough to generate exactly the file which was intended by programmer. The consequence of the SSSO idea was that no matter what version of fasm (considering ports to different operating systems), it always generated the same output file. So, when you have written a DOS program, the Linux version of fasm would still make the same DOS executable from such source.

Some people seem to dislike the implications of this principle because all other assemblers and compilers have the command line settings that affect the output (or even the source constants) and this different approach needs to change the way of thinking in some cases (this actually happens in even more areas when programming with flat assembler, and it's the purpose of this text to show the origin and reason for those differences). The SSSO rule became one of the guidelines for the design of flat assembler and I don't plan to put it away.

Nonetheless, it is still possible that the same source file will be assembled differently in other environments because it may include some other files. Their contents and availability may vary from computer to computer and from system to system. The paths to files need to follow the rules for a given operating system, and what file gets loaded also may depend on environment variables as flat assembler allows them to be expanded inside the path values. To eliminate this dependency on environment the assembler would have to abandon the file inclusion features. Well, I know of one assembler that did choose this way, but it is not something I could even consider for my assembler. So, my rationale is that the system-dependent file paths define what source (composed from perhaps many different files) the assembler will get to finally process, but after that the SSSO principle kicks in.

4. Resolving the Code

There was one more feature of Turbo Assembler I wanted to have in my assembler as well: optimizing the size of displacements during multiple passes to resolve which displacements can fit in shorter ranges and which cannot. To make this feature possible I had to make labels - which from the programmer's point of view are actually constants - assembly-time variables, which are constantly updated on each pass to reflect the changes of code due to optimization. And for this reason, I had to do processing of structures like IF or REPEAT - which use the expressions that may be dependent on the value of such labels - during those passes, not earlier. Therefore, in fasm the IF directive does not affect processing of macroinstructions or other directives interpreted by the preprocessor - this may be confusing for people starting to learn the syntax of fasm, but was really necessary to resolve correctly the sources like:

```
        if alpha > 100
```

```
            ; some code here
        end if
```

Since this checks the value of some label, which may vary between passes, the truthfulness of the condition may also vary between passes, and this can lead to a chain of even more complicated changes. The fundamental rule for flat assembler always was, that it cannot output code that is not resolved completely and trustworthy. So, if there's even a slightest suspicion that some value might have been used during code generation with other value than it got finally, fasm does more passes, until everything is resolved. This process can be described like trying to solve a complex and sophisticated equation by doing iterated approximations. Of course, sometimes the solution does not exist, like in this case:

```
        alpha:

        if alpha = beta
           db 0
        end if

        beta:
```

In such a situation the assembler will do more and more passes, never approaching any solution. But since there is a limit of possible number of passes built into the assembler, it will eventually exit with the error message stating that "code cannot be generated".

The resolving process has been improved many times since the first versions of flat assembler, also because many new features were added that made more complex self-dependencies of source possible. During each pass, the assembler does the predictions of values it doesn't know the final values yet (and these predictions are based on the results of previous passes) and finishes the process only when all the predictions match the final values.

Knowing how flat assembler resolves the code is important to understanding the specific self-dependent sources. Let's consider one such example:

```
        if ~ defined alpha
           alpha:
        end if
```

Assuming that this label is not defined anywhere else in the source, during the first pass the assembler will of course execute the block and define the label as one would expect. But during the second pass it predicts that the label will be defined (since it was defined in the previous pass) and will skip over this block. This will lead to the dead loop and stop on the limit of passes with error. To make fasm correctly resolve the source one should do it like:

```
        if ~ defined alpha | defined got_alpha_here
           alpha:
           got_alpha_here = 1
        end if
```

This way, in the first pass the block gets assembled because the label is not yet defined and in the later passes the block gets assembled because of the constant which marks that this block was assembled in the previous pass and therefore should be assembled again.

To match the values of predicted and actual values of labels the assembler of course cannot allow to a label to be defined in more than one place. This; however, does not apply to constants defined with the = operator, which - contrary to their name - can be redefined, but in such case, assembler simply forbids forward-referencing them (which means using the value of a symbol earlier in the source than it gets actually defined) and no predicting is needed then. But if the constant is defined only in one place, then forward references are allowed just as with any other type of label.

The rule that flat assembler always tries to ensure that the values used by instructions are exactly what they should be at run time implies also, that the assembler is very strict with the usage of relocatable symbols - only in cases, when it's sure that even after relocating the value will still be correct, it is allowed to be used - this is similar to the behavior of TASM, but in case of absolute addresses and other such values, fasm great discretion in using them in any kinds of expressions, thanks to its resolving techniques.

5. Complex Solutions with Simple Features

This last principle evolved later, when - after the release of the Windows version of flat assembler - there was a need to add additional high-level syntaxes. I was afraid that adding a lot of new features that weren't initially planned for could lead to unpredictable interactions between the existing and the new and that was the last thing I wanted in my assembler, when one of my main rules was to make it always resolve the code in a logical, unequivocal way. Therefore, instead of writing a whole bunch of new features for this purpose, I attempted to implement them as macroinstructions, only extending the capabilities of the preprocessor when a good macro solution for a given problem could not be achieved without such extensions. But even when adding some new feature, I was always doing it resistively, first wanting to make sure it wouldn't interact with the existing ones in any unwanted way. And I always tried to find the simplest extension possible for some really low-level feature, which would be applied to solve many different problems.

This way a kind of emergent system was created. It may bear a similarity to some esoteric language that only has a few basic instructions, but still allows the implementation of any algorithm, though sometimes in quite a complex way. Some people may smile hearing this comparison, as I know few of them perceive the macroinstruction language of flat assembler exactly this way - as an unnecessarily obscure and hard to master dialect similar to the esoteric languages created to play and exercise with and not for any serious application. In fact, this may not be far from the truth, since I personally have a weakness for some kinds of emergent esoteric languages and it might have affected my choices in the design of many macroinstruction features in flat assembler.

Still, this approach has some strong advantages. The building blocks are defined in a simple way and it is easy to maintain these features - while at the same time they can be used to build very complex constructs that may even define new syntaxes or target different machines for which flat assembler was written. I am hoping that I can keep maintaining this simple and lightweight core of flat assembler, while other people may create packages of advanced macroinstructions for various specific purposes. And over the years this vision was at least partially fulfilled - some of the contributors on the message board are already creating packages way more complex than any macroinstructions I wrote myself, and are doing some amazing things with them. It gives me a sense of accomplishment when I see others using my assembler to create things that are more impressive than anything I undertook so far.

Conclusion

Of course, this text is far from being complete in terms of describing the design of flat assembler. But it shows the main directions and should be enough to explain most of the choices I've been applying. Anyway, the reason behind all of this is that I'm keeping the flat assembler project as "one man's vision," focusing on the efforts to keep the overall logic and consistency. I hope this text will help others to understand my motives and vision itself.

Top of Document