flat assembler

Documentation and tutorials.

Design Principles

or why flat assembler is different?

The purpose of this article is to describe the main ideas that led the flat assembler project during the all time 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 to explain how flat assembler has come to the point where it is and what were the reasons for the many desing decision I made.

1. The Roots - Turbo Assembler versus Netwide Assembler

When I have learned the 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 the new these days product, Netwide Assembler, which was free and even open source. But, though I did like some of its ideas, 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 capabilities enough to assemble all my previosly written projects with only small source changes.

It should be then obvious, that the syntax I've chosen for fasm, was primarily imitating the one I was using when programming with TASM, and it's important to note that TASM offered two modes, with different syntaxes, first and default being the MASM-compatiblity mode, and the second one called Ideal mode. After learning the basics of assembly language I have quickly switched to the Ideal mode, as I found it easier 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 is the syntax for accessing contents of memory. TASM with Ideal mode selected requires such operand to be always surrounded with square brackets, and they also ensured 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 fast and I had put the same syntax rule to 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. But I still liked the idea of label to be treated just like a constant equal to address, as it made such instructions:

```
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 it was also getting rid 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 address at the first sight, still keeps the information about what type of variable is defined behind it - and provides the size checking just like I had it with TASM. Of course in assembly programming there is still needed some way to force the different size when you want to. With TASM the size override operator had to be put before the name of variable, inside the square brackets. But since I've followed the 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 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 two worlds, and was the first milestone in fasm's syntax design. And it still needed only small changes in my sources to convert them to the new syntax, a small example for comparision:

```
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 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 name was always before the directive. Perhaps because of some previous habits from higher level languages (like Pascal) I also liked the variant of Ideal mode more.

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 macroinstruction had to be enclosed with braces instead of ending with ENDM directive. It was just because I liked the braces and they were simpler to parse, too. I have also implemented the LOCAL directive with the same syntax I had with TASM and this way I got implemented all the features I was actually using with TASM. Other, more powerful macroinstruction features were implemented much later, when the influence of TASM was already lost and other design priorities (which will be described next) have taken its place.

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 using them to switch the type of generated code just anywhere. This is where the second design principle came on.

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 developing at the same time, and I was designing fasm as a tool aimed mainly at this purpose. That's why it was important to make it easily portable and as I soon as I finished it, I have ported it into OS I was developing, to become able to write programs for it in their native environment. That might be also considered the reason why I have written fasm in the assembly language, however it's more likely because I was doing all my programming in assembly language these days - if I really preferred some high level language I would make some self-compiling high level compiler instead.

However for the OS development it is necessary to assemble some sophisticated pieces of assembly code, with switching of code type and addressing modes, and this was actually quite complicated, when you wanted to do it with TASM. I especially hated the necessity of manually building some instruction opcodes with DB directive. So I have put into my assembler all the instruction variants and size settings that are needed to declare any instruction without any doubt, what operation will it perform - like 32-bit far jumps in 16-bit

mode and other similar, rare but needed 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 to interprete the size operator inside the square brackets as applying to the size of address value.

Also for the purposes of OS development I have implemented ORG directive behaving a bit differently than the original one in TASM. What I needed was setting the origin address of given code, but without actually moving the output point in file. I though it should be programmer's responsibility to load the code at the origin he specified, like DOS does it with the .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 to design code to work correctly when loaded at specified origin, while its placement in file is just determined by the order of source. 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 similar reason I have invented the 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 to declare structures at the given address without putting any actual data into the output. Some other applications of VIRTUAL directive were invented much later, 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 to create .COM programs aswell. But I have soon added also the option to output relocatable format, which I have designed for my operating system (I have removed this feature before the official releases). However the output format was not selected with 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 with.

3. Same Source, Same Output

The problem with command line switches selecting output option in case of low level assembly is that the given code will anyway most probably assemble and execute correctly only when the same output is selected that programmer had in mind when writing this code. Also I remembered many cases when I had a source for TASM written by someone other, and to compile it correctly I had to follow the directions given in the comment at the beginning of source and just rewrite all the command line switches as described there. And my though was: then why don't just make assembler look for such options in the source instead, so nobody will have any problem with recompiling. Thus came the SSSO principle - all the settings that can affect the output of assembler are be selected only from source and source is always enough to generate exactly the file which was intented by programmer. The consequence of SSSO idea was also 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 a bit 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.

4. Resolving The Code

There was one more feature of Turbo Assembler I wanted to have in my assembler aswell: optimizing the size of displacements with doing multiple passes to resolve which displacements can fit in shorter range and which not. To make this feature possible I had to make labels - which from the programmer's point of view are actually constants - an 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 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 the passes, the truthfulness of the condition may also vary between the passes, and this can lead to 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 matches. 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 case assembler will do more and more passes, never approaching any solution. But since there is a limit of possible number of passes built into assembler, it will finally 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 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 understand the specific self-dependent sources. Let's consider such sample:

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

Assuming that this label is not defined anywhere else in the source, in first pass assembler will of course execute the block and define the label as one would expect. But in second pass it predicts that the label will be defined (since it was defined in 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 get 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 value of label assembler of course cannot allow to define label in more than one place. This however does not apply to the constants, defined with = operator, which - contrary to their name - can be redefined, but in such case assembler simply forbids forward-referencing them (what means using the value of symbol earlier in source than it gets actually defined) and no predicting is needed then. But if the constant is defined only in one place, the 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 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 allows to use it - this is similar to the behavior of TASM, but in case of absolute addresses and other such values fasm gives the very high level of freedom 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 Windows version of flat assembler - there was need to allow some more high level syntaxes. I was afraid that adding a lots of new features that wasn't initially planned can 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 logical, unequivocal way. Therefore instead of writing a whole bunch of new features for this purpose, I was trying to implement them as a macroinstructions, only extending the capabilities of preprocessor when the good macro solution for given problem could not be achieved without such extensions. But even when adding some new feature, I was always doing it restively, first wanting to make sure it won't interact with the existing ones in any unwanted way. And always tried to find the most simple extension, some really low level feature, which would be applied to solve many different problems.

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 doing. Anyway the reason behind this all is also that I'm keeping the flat assembler project as "one man's vision", stressing the efforts to keep the overall logic and consistency. I hope this text will help others to understand my motives and vision itself.