Understanding flat assembler 1

Contents

This text is of guide for advanced users, that summarizes some of the rules of the flat assembler's language and teaches select advanced techniques of combining its various features. It also has the purpose of explaining some of the behaviors that may be confusing and not conforming with the expectations unless one understands exactly how the things work and cooperate in various layers of the language used by flat assembler, which is in some aspects unique among the assemblers.

This guide cannot, however, replace the manual - it assumes you already have some basic knowledge about the flat assembler's language so now you can go into understanding it deeper.

Assembler as a Compiler and Assembler as an Interpreter

The implementations of programming languages are divided into two main classes: compilers and interpreters. The interpreter is a program which takes a program written in some language and executes it. And the compiler is a program that translates program written in one language into another one - most commonly the machine language, so the result can then be executed by a processor.

From this point of view, the assembler appears to be a kind of compiler, as it takes the program written in the assembly language and translates it into the machine language. However, there are some differences.

When compiler does the translation from one language to another, it is expected to create the equivalent program in the target language, but it has a freedom of choice between many possible language constructs that would do the same. Even though it is usually expected to make possibly the best choice, the various compilers translating between the same two languages may give quite different results.

In case of the assembler, most often there is an exact correspondence between the instructions of assembly language and the instructions of machine language they are translated to. In many cases it is known exactly what sequence of bytes is going to be generated by the construct of the assembly language. This is what makes assembler behave a bit like interpreter. It is the most obvious in the case of directives like:

```
db      90h
```

which tells the assembler to put the one byte with value 90h at the current position in the output. This is more like if the assembler was an interpreter, and the machine language generated by assembler was simply an output of the interpreted program. Even the instructions, which in fact represent the machine language instructions they are translated to, can be considered to be actually the directives that tell assembler to generate the opcode of given instruction and place it at current position in output.

Also, one can put no instruction mnemonics at all into the source code, and use only DB directives to create, for instance, just some text. In such case the output is not a program at all, as it doesn't contain any machine language instructions. This makes assembler appear to be really an interpreter.

But when someone writes the program in assembly language, he usually thinks about the program he writes in machine language - assembler just makes the task of creating program in machine language easier, providing easy to memorize names for instructions (called *mnemonics* for that very reason), allowing to label various places in memory and other values with names and then calculating the appropriate addresses and displacements automatically. When writing some simple sequence of instructions in assembly language:

```
mov ax, 4C00h
int 21h
```

one usually doesn't think about them as interpreted directives that generate the machine language instructions. One does think as if they actually were the instructions which they generate, one thinks in terms of the program he writes *with* the assembly language, not the program he writes *in* assembly language. But there are actually those two programs merged together, two different layers of thinking in one source code. This makes assembler a new quality: a tool that appears to be both compiler and interpreter at the same time.

Run-Time Layer and Interpreted Layer

Let's look at the two simple pieces of assembly language that both do add the EAX to itself, repeating this process five times. The first one uses ECX to count the repetitions:

```
        mov ecx,5
square: add  eax,eax
        loop square
```

This generates the three machine language instructions, and the operation that adds EAX to itself, is performed five times when the processor executes the machine code that is generated from the above source. This is done by decrementing ECX by one and jumping back to that instruction if ECX is still not zero. The second sample looks simpler:

```
repeat 5
   add    eax,eax
end repeat
```

This time the directive of the assembler is used to repeat the instruction five times. But no jumping is done here. What assembler generates when meets the above construct, is actually the same what would be generated when we wrote it this way:

```
add eax,eax
aAdd eax,eax
add eax,eax
add eax,eax
add eax,eax
```

Assembler generates the five copies of the same machine instruction. As LOOP instruction is used to create the run-time loops, the looping that happen when processor executes the machine code, the REPEAT directive makes an assembly-time loop, it repeats interpreting the block of directives that follows it up to the corresponding END directive. What's in this block is in this case is an instruction, but as it was said earlier, the instruction is in fact a directive that causes some machine language instruction be created. Thus, it repeats five times interpreting the ADD directive, which each time generates the code of instruction that adds EAX to itself. This in a perfect example of the interpreted layer of assembly language. Nonetheless there still is also a run-time layer here: what we actually achieve, is getting the five copies of the same machine language instruction, executed one after another. The next sample is more pure interpreted language:

```
A = 1

repeat 5
  A = A + A
end repeat
```

This piece of flat assembler's dialect of assembly language defines the assembly-time variable called A and then five times repeats doubling its value. Everything that happens here is the interpreting, there's no machine code or any other such output generated at all, the only influence it would have on the run-time layer would be if the value of A was later used to affect some instruction.

Using flat assembler as Pure Interpreter

As it was already noted on example of the DB directive, the output of assembler may not be a program at all. In such case the interpretation of assembler as a compiler is lost and the result of assembly is just plainly the output of interpreted language. Copy the below source code to file interp.asm to see it on an example:

```
file 'interp.asm'
repeat $
  load A byte from %-1
  if A>='a' & A<='z'
    A = A-'a'+'A'
  end if
  store byte A at %-1
end repeat
```

This is a program written entirely in flat assembler's interpreted language, using some of its advanced features. It first places the entire contents of interp.asm file (its own source) at current position (it is always 0 when starting assembly) and then for each byte of that file (the $ is the value of current position, so after putting the whole file at position 0, the $ becomes equal to the size of this file) it repeats the process of: taking this byte, converting it to upper-case with help of the simple condition

check and writing the modified byte back. This way it converts the whole contents of the file to upper-case, and this is what finally lands in the output file. So, this is an example of simple text-conversion program written in the interpreted language, and since the output is a text and doesn't contain any machine language, the aspect of assembler being a compiler is completely absent in this case.

Code Resolving

The last samples all showed the description of assembler as an interpreter as enough to fully explain what it does. But after we agreed (I hope) that assembler can be read as an interpreter, now it's time to show where this interpretation fails. Because assembler truly is both a compiler and interpreter at the same time, and none of those terms alone is able to explain correctly what the assembler does.

One of the crucial features for the assembler to have is the ability to label different places in machine code or data with freely chosen names, and to use those names instead of raw addresses inside the instructions. This, in particular, applies to jump instructions, so it is possible to write instructions jumping to other places in program using only names labelling those places, and the programmer doesn't need to worry what exact addresses are they. However, when making a jump, a programmer may need to jump forward as well as backwards. When jumping backwards, the assembler has an easy task - it already knows what the address was it when it met that name and interpreted it as label. However, when jumping forward, the assembler that was just interpreter, would fail. There is no way how it could know what is the address of the label it didn't yet encounter.

But assembler still has to do it, and in this aspect, it behaves again like a compiler and not interpreter: given the instructions and labels written in assembly language it needs to find out the equivalent instructions and addresses in machine language. There are few possible methods for achieving this, one of them is to interpret the whole source once, leaving the empty places in created machine code for the values that are not known at the time when they are needed, and then - after this is finished and so all the values of labels are already known - those empty places can be filled with right values. However, there is another problem, when the assembler wants to generate as optimal machine code as possible (with respect to size, what also affects the speed), and the machine language of given processor architecture this assembler works for (like x86 architecture, which is the one for flat assembler) allows different in length forms of instruction codes depending on what range the target address fits in. For example, on x86 architecture there are short forms of jump instructions that can jump only to addresses not further than about 128 bytes forward or backwards, and the longer forms of analogous jumps, that can jump further. In such cases assembler may try to generate the short forms when possible, but if it used the short form in case, when the value of target address is not yet known, it may then realize that this address is too far away, and the longer form would have to be used in this place.

Thus, to make the more effective optimization possible, flat assembler uses the different approach. It first interprets the whole source, and chooses the smallest possible form of instruction every time - when the value of target address is not yet known, it also generates the shortest form, assuming the best possible optimization may happen. But after it finishes, and knows all the values of labels, it doesn't look back at what it generated to check out whether it may be filled with right values. It then interpretes the whole source once again, but this time using the values of labels gathered the previous time to predict the correct target addresses in the places, where they were unknown earlier. If indeed the best possible optimization can be done, everything this time is assembled the same, only now filled with the

right values everywhere. But it also may happen that this time some of the instructions will have the different length than they had previously. In such case all the labels defined after such instruction would get their addresses shifted, and the predictions made with the addresses gathered previously would become not exactly true. But this does not discourage the assembler. When it realizes that some of the addresses have changed during the second interpretation of the source, it decides to interpret it once again, taking more up-to-date values of labels and trying to predict the right values better this time. This process may be repeated many times, but hopefully finally the predicted values match their definitions and only then assembler decides it has finished its job and writes out the output generated during its last pass through the source.

This whole process is called resolving of the code. This approach not only allows the optimal code to be compiled from the given sequence of machine instructions and labels, but also allows combining the interpreted layer of the assembler with a compiled one in an interesting way. Look at this sample:

```
alpha:
    repeat gamma-beta
   inc    eax
    end repeat
beta:
   jmp    alpha
gamma:
```

Assembler needs to resolve the values of beta and gamma labels, since they are used before they are defined (it is usually called that they are forward-referenced). And those values affect how many times the INC instruction gets repeated. The interpreted layer is not really visible here, since everything here needs to be resolved and assembler needs to find the right solution on its own. There even may not exist a solution at all, like if we subtracted alpha instead of beta from the gamma label. In such case the value of this difference would obviously be larger than the number of repeats, so the source would get self-contradictory and assembler would not approach any solution.

In the above sample the interpreted layer became faded out by the resolving. But the next one shows the case when they clearly coexist:

```
        mov     eax,gamma
A = 1
repeat 5
    A = A*%
end repeat
label gamma at $+A-7
```

The A is an assembly-time variable, and its final value is clearly calculated with help of the interpreted language. But then this value is used to define label gamma, which is forward-referenced and needs to be resolved. In fact, we could even use the = here to define the value of gamma, since the assembler would treat such definition (when the definition of gamma name is encountered only once during interpreting the source) as a definition of global constant, not the assembly-time variable, and so it could also be forward-referenced. This leads us to another interesting example of code resolving:

```
        dd      x,y

x = (y-2)*(y+1)/2-2*y
y = x+1
```

Here the = is used to declare numerical constants that are forward-referenced and the assembler needs to resolve their values. Moreover, their values depend on each other and it's even hard to tell immediately whether there exists some solution where all the values would match. However, if we try to assemble it with flat assembler, it manages to find out the solution in a few passes - it is with x=6 and y=7. It's actually a rare case when the code resolving algorithms designed rather for optimization of the machine code happens to be able to solve even such set of equations, but it also gives us a pure sample of what the code resolving has to do.

By moving the DD directive to the last line in above sample, we would leave only y as a forward-referenced symbol, while x would appear to be just auxiliary variable to help calculate the self-dependent value of y. We could even split those calculations into many separate operations on x:

```
x = y-2
x = x*(y+1)
x = x/2-2*y
y = x+1

    dd     x,y
```

to again emphasize a bit more the interpreted layer. But note that we could not define the self-dependence of y without such intermediate variable, as flat assembler doesn't allow forward-referencing a symbol inside its own definition, so that definitions involving own previous value are reserved for the assembly-time variables (like the x in this sample).

The resolving of code becomes even more complex issue when we consider the IF directive with all the possible complex dependencies it can create. You can find some examples of such problems in the section about multiple passes in the flat assembler's manual.

Of course, for the given set of dependencies more than one correct solution may exist. In case of simple dependencies between instruction forms and label addresses flat assembler tries to find the one with possibly smallest instruction encodings, but this doesn't mean it always find the smallest existing solution in general, and in some cases, it may not be able to find the solution even though it exists. This is because the prediction algorithms it uses were designed with focus on generating the quality machine code, and not solving any complex arithmetic problems that can be encoded with its language. But if the assembler finishes its job without signaling any error, you can at least be sure that all the dependencies are fulfilled and the output is correct with respect to them.

Preprocessor

Still, flat assembler has one more layer, which makes the whole thing even more complex. It's the preprocessor.

The main point of preprocessor is that it operates on the whole source code before it goes into the assembler, and what it does is mainly the text processing, that allows to create with simple statements the much more complex sets of assembly instructions. There is a set of special directives, called preprocessor directives, which are interpreted only by preprocessor and removed from the source before passing it to the assembler. Everything else that preprocessor finds in source, it passes for the assembler to process.

For example, let's consider we've got this source:

```
mov ax,bx
include 'nop.inc'
mov cx,dx
```

and the contents of the NOP.INC file is just:

```
nop
```

What preprocessor does with it? It interprets the things it recognizes, like INCLUDE directive, which tells it to put the whole contents of NOP.INC file in place of it. The things it doesn't recognize it leaves intact. So, what is finally passed to the assembler is:

```
mov ax,bx
nop
mov cx,dx
```

Note that for the assembler itself there is no such thing as INCLUDE directive here. The preprocessor already prepared the simple sequence of instructions for him. There could be an IF directive put just before the INCLUDE and END IF inside the included file, and none of them would complain, as for preprocessor the IF and END IF anyway have no meaning, it just leaves them for the assembler; and the assembler wouldn't see any discontinuity, since after the preprocessing it all would become a continuous and correct sequence of the assembly directives.

When preprocessor puts some new lines into the source, like when it replaces the INCLUDE directive with all the lines from that given source file, it also preprocesses all those new lines before it goes further. Thus, the included file can also contain directives for preprocessor that get recognized and appropriately processed.

Also, the lines that do not contain preprocessor directives may still get altered during preprocessing. This is because of the text replacement features processor provides. Such replacements are done with the so-called symbolic constants. You define a symbolic constant with an EQU directive, like:

```
A equ +
```

With such definition, the name A is replaced with the + symbol everywhere where it is found by preprocessor after that definition. Note that because preprocessor just goes once through all the source (so acts much like just a pure interpreter), the replacement is applied only after the A gets defined. For example:

```
mov eax,A
A equ ebx
mov eax,A
```

will after preprocessing become:

```
mov eax,A
mov eax,ebx
```

Another important thing about symbolic constants is that they may be in fact not constant - they can act as a preprocessing-time variable, analogously to the assembly-time variables defined with = directive. So, they can be re-defined, and perhaps we should call them symbolic variables instead, even though in manual they are called constants (for historical reasons). Just like with the assembly-time variables, you can re-define preprocessing-time variables using the previous value to make the new one. For example:

```
A equ 2
```

```
A equ A+A
```

defines symbolic variable with value 2+2. This works because the replacements of symbolic variables with their assigned values is done also in the line containing the EQU directive, though only after the EQU. Thus, also this:

```
A equ 2
B equ +
A equ A B A
```

defines symbolic variable with value 2+2. From preprocessor's point of view the whitespace is important only where it is needed to separate names that would become one longer name if they were not separated this way. Any other whitespace is ignored and stripped out - for preprocessor it's only the sequence of symbols that counts.

Let's now summarize the differences between the preprocessing-time variables and assembly-time variables. The one that is obvious is that the assembly-time variables are purely numerical and always hold just some number or address values, while symbolic variables can have just about anything as a value (they can even have an empty value, when there is nothing else that whitespace and comments after the EQU directive). The fact that symbolic variables do just a kind of text substitution can be demonstrated on the following example:

```
nA = 2+2
   mov    eax,nA*2
sA equ 2+2
   mov    eax,sA*2
```

The first line defines nA to have numerical value of 4, and the next one calculates the value to put into EAX as nA multiplied by 2, so the instruction that is assembled is MOV EAX,8. To the contrary, the third line defined sA to be equivalent to 2+2 text, and thus the instruction in last line is changed by preprocessor into MOV EAX,2+2*2, what is later assembled into MOV EAX,6. This example demonstrates that you should be careful with symbolic variables and always think what effect they may cause on the source when they are replaced with the text you assigned to it.

Another subtle thing here is that EQU and = directives are processed by different layers. All the replacements that EQU causes are done before the whole source is passed to the assembler. Let's look at the effects of this in this sample:

```
A = 0
X equ A = A+
X 4
```

dd A

After this source being chewed by the preprocessor, this is the result that is fed into the assembler:

```
A = 0
A = A+4
dd A
```

Thus, what you finally get is the 32-bit data field filled with the value of 4. This example shows how you can get the different layers to cooperate - however such task requires that you realize exactly what belongs to which layer - we will talk more about mixing of layers later.

Macroinstructions

The macroinstructions (often called in short *macros*) are another feature of preprocessor. The macro is a recipe for the preprocessor, and when you use macro with some set of parameters, preprocessor applies this recipe to create some new source lines and it puts them in place of the line that invoked the macro.

The definition of macro is treated by the preprocessor as a one large directive (as it may span multiple lines), so the whole recipe is itself unaffected by any preprocessing, and is not passed to the assembler. When you invoke macro, however, and preprocessor uses the recipe to produce new lines, it also preprocesses all those new lines before it goes further - just like it is with INCLUDE directive. Let's consider this simple macro:

```
macro Def name
{
    name equ 1
}
```

As everything between the braces is the contents of the macro, the preprocessor doesn't notice the EQU directive here - all the lines here are parts of the recipe, and all preprocessor does with it is that it notes to itself what the recipe for the Def macro is and goes further, without leaving anything for the assembler out of those lines. Now what happens when we use that macro? Let's say we do it this way:

```
Def A
```

Preprocessor will replace this line with the lines generated out of recipe for the Def macro, in this case it will be just one line:

```
A equ 1
```

This new line is then interpreted in a standard way - thus preprocessor recognizes the EQU directive here and applies it to define A constant.

This also means that a line generated by macro can contain an invocation of another macro. Note however that is not possible for a macro to generate invocation of itself, as while processing the lines the macro that generated them is disabled and the previous meaning of that macro is applied (in a similar way like if you used "purge" directive, the only difference is that this macro is enabled again when processing its invocation is finished). This makes it harder to make recurrent macros (but it's still possible, we will discuss it later), however the quality of such behavior is that you can stack the definitions of macros, like it is shown in the manual.

There are some special operators and directives that can be used inside macro definitions - they rule the way in which preprocessor applies the recipe to generate the new lines. So, it is obvious that as those operations are performed in order to generate the new lines, they are done before the preprocessing that happens on those lines when they are finished. For instance:

```
macro Inc f
{
    include `f#'.inc'
}

Inc win32a
```

Here the recipe tells preprocessor to convert the first symbol of f parameter into quoted string and then attach the '.inc' string to the result. If the parameter consists of exactly one symbol, the result of those operations will be just one quoted string, and the invocation of Inc macro in above sample should generate such line:

```
    include 'win32a.inc'
```

Which is then preprocessed as usual, so it includes the win32a.inc file.

The backslash-escaping is useful when you need to put into the lines generated by macro some symbols that would get interpreted as a macro recipe operation otherwise (if they were not escaped). Like:

```
macro Defx x
{
    \x db x
}
```

Here x is the parameter of macro, so when applying the recipe preprocessor puts the value of that parameter everywhere in place of it. However, this macro is intended to define the byte variable labelled x, with the value given by the parameter to macro. The conflict of names could be resolved by giving some distinct name to the parameter, but here you can see how the escaping can also be used to give the desired result. When generating a line from a macro, preprocessor cuts out the first backslash of any escaped symbols and this is all what it does with them.

The most frequent use of escaping is however related to defining macros by macros. Since the lines generated by macro are preprocessed just like any others, they may themselves contain definitions of macros. But you need to escape any operators for such child macro to prevent the parent macro from interpreting them while unrolling itself. Let's go for a bit more complicated example this time:

```
macro Parent [name]
{
  common
    macro Child [\name]
    \{
    \common
  forward
      name dd ?
  common
    \forward
      \name dd ?
    \}
}
```

and see what happens when we invoke such macro this way:

```
Parent x,y
```

The macro directives and parameter names that are not escaped are interpreted by the Parent macro, while the escaped ones will go into the definition of Child. The result is:

```
macro Child [name]
{
  common
    x dd ?
    y dd ?
  forward
    name dd ?
```

```
}
```

It is recommended to stop over this example until you understand exactly what is happening here.

It is clear why we have to distinguish various levels with the escaping like in the above sample. The questions that is sometimes asked, is why do we need to escape the enclosing braces, too. Such escaping obviously helps keeping track of what escaping should be used where, especially when there may be many levels of macro definitions - the symbols that we want to be recognized by some appointed macro has to be escaped with as many backslashes as its braces. And the manual explains that escaping of the closing brace is needed, because the first closing brace preprocessor meets is always interpreted as the end of macro definition, so if we did not do backslash-escaping on closing brace of Child macro, it would be interpreted as the end of recipe for Parent macro. But why preprocessor cannot just count all the braces to determine which one is closing which block?

The answer is already hidden it what it was said here. It is possible for macro to only begin the definition of another macro and not end it, like it is shown in the example of creating alternative definition macro syntax in the manual. After such macro is processed, the preprocessor gets the started definition of another macro, and then gathers the following lines into this definition until it finds the closing brace (note that this means that no preprocessing is done on the lines following that macro, and thus the only way to close such definition then is to provide the closing brace directly - but the FIX directive may also provide such closing brace by producing it from other symbol, as it is shown in the manual).

Also, there might be some other out-of-order effects on the blocks generated by macro, like when there are some opening braces that do not open any block at all, or when they are affected by the repeating blocks. For all those reasons combined, the preprocessor does always treat first closing brace that it meets as the end of the macro definition, and thus you have to backslash-escape all the braces that you want to be generated by this macro. Well, you can omit backslash-escaping the opening braces, but it is anyway recommended to do it, to help keep track of the things.

Instantaneous Macroinstructions

There are some directives that define the macros that are not given any names, but instead are invoked just when they got defined. Those directives are REPT, IRP, IRPS and also MATCH (which deserves a separate section), and we may call them *instantaneous macroinstructions* to emphasize the fact that they are applied just once, immediately after being defined.

They may also differ from the standard macros in a way in which the parameters are provided, but their definition blocks are just the same kind of recipes, and generate the new lines in the same way as regular macros. This also means that when you put instantaneous macro inside some other macro, you must do the appropriate backslash-escaping for it to work properly.

To demonstrate how the instantaneous macros are related to regular macros, here are the four equivalent constructions:

```
rept 4 i
{
  ; ...
}

irp i, 1,2,3,4
```

```
{
  ; ...
}

irps i, 1 2 3 4
{
  ; ...
}

macro Inst [i]
{
  ; ...
}
Inst 1,2,3,4
```

If you put the identical recipes into each of those four blocks, each of them will do exactly the same. In particular, the FORWARD, REVERSE and COMMON directives will have indistinguishable effects in all the cases.

From the three types of instantaneous macroinstructions demonstrated above, the IRP is the only one where the values of parameters are given in basically the same way as for the named macroinstructions - they are separated with commas, and thus may happen to be empty (unless you tell to preprocessor that the value of parameter cannot be accepted as empty, by putting * character after the name of parameter), and you can enclose the value of parameter with < and > characters if you need to provide value that contains the comma itself.

The REPT directive on its own generates all the possible values for the counter parameters, you can only adjust the base value for each counter - or not use counters at all. Still, those counters behave in the same way as macro parameters given the lists of possible values.

As for the exact explanation of IRPS directive, we first need to know a few more details about how preprocessor perceives the source text.

Tokenization of Source and Line Makers

Preprocessor is not in fact working on the source text in the exact form how is it stored in file. It extracts from each line its meaningful contents, ignoring the redundant whitespace and comments. What it does, is actually splitting each line of source into the sequence of simple tokens, and since then all the processing is performed on those tokens (in manual they are called symbols, for the historical reasons, here we will use both terms interchangeably).

The first class of tokens are the symbol characters. The manual states that all of them are:

```
+-/*=<>()[]{}:,|&~#`
```

Each of those characters, when it occurs somewhere in the source text, is an independent entity and becomes one separate token. The other special characters are also the whitespace ones - space and the tab, which don't form any token themselves (though may separate some entities that become individual tokens), the line breaking characters (with obvious role), the semicolon (that marks the beginning of a comment and thus all the rest of line is ignored entirely) and also the quotes and backslash character, which will get covered later.

Any sequence of the characters that are not special ones, like the continuous sequence of letters and digits, becomes a name token. Such sequence can be split into separate token by either whitespace or some symbol character. For example, this line:

```
mov ax,2+1
```

contains six tokens: first the MOV name token, then the AX name token (they are separated by whitespace), the comma symbol character, the name token 2, the plus symbol character and the name token 1. Putting any additional whitespace into this line wouldn't change in any way how it is seen by preprocessor. However, removing all the whitespace between MOV and AX symbols would make them become a single name token.

There is still one more type of tokens, the quoted strings. When the first character of token is either a single or double quote, it is interpreted as a quoted string, and all the following characters other than line break are fed into this single token until the closing quote is met (but, following the standard of many assemblers, the two quotes in a row do not close the string and are included into it as a single character). So, this line:

```
db '2+2'
```

contains two tokens, a name token followed by a quoted string. Nevertheless, if the quote occurs not as a first character of token, but is placed somewhere inside, it doesn't have any special meaning (quotes are not special characters by themselves). Thus:

```
Jule's:
```

is just a regular name token followed by the colon symbol character.

The backslash is a special character that has two different meanings depending on its position. If backslash is followed by some token, it is integrated with that token into single one. This may happen recursively, so if the backslash is followed by the backslash followed by some other token, they will all finally become one token. This feature exists solely for the purpose of escaping the symbols in the macroinstructions.

If backslash is not followed by any token, it is interpreted as a line continuation character, and the tokens from the next line from source are attached to the tokens of current line. This way from many lines of source text a single tokenized line in the preprocessor's sense may be formed.

Thus, now we understand that what is here called a line that are preprocessed, are actually the sequences of tokens, not the raw text of the source. And such sequences can be born in two different ways - either created directly from the source text, or generated by macroinstruction. Thus, we've got a two different "line makers" - one being "source reader" and one "macroinstruction processor".

As we have seen, there is a bunch of special commands, like concatenation operator or COMMON directive, that may be issued while macroinstruction generates a new line. In a similar way, there are special commands that are understood and obeyed only by the source reader. The backslash character mentioned above is an example, and the FIX directive is another.

The FIX directive provides a kind of textual replacements (actually the token replacements, as it defines replacement of single name token with some specified sequence of tokens) like EQU or DEFINE, however those definitions and replacements are done by the source reader, while preparing a line to be

then preprocessed. This way you can have some replacements done before the other preprocessing happens.

Because the replacement defined by FIX is done by the source reader, it is only really useful for some syntax adjustments, when it is required that some textual construction used by the programmer in source file is perceived by preprocessor as something different. The official manual has an example in it that uses FIX to define a new way of closing the definition of a macro - it replaces the ENDM token with a closing brace. It is one of the very few cases when this feature might be really needed.

The macroinstruction processor has much more special commands and operators than the source reader. It not only replaces the parameters of macro with their values and processes some special commands like COMMON or FORWARD, but it also allows to concatenate tokens with # operator, or convert a token into quoted string with ` operator. These operators are not recognized by source reader, and thus they only work inside the macros. As for the backslash, it is stripped from the beginning of any token that macroinstruction processor puts into a newly generated line. If a token starts with more than one backslash, only the first one is removed, so this way a token can be passed many times through macroinstruction processing until it finally is stripped of all the backslashes and is recognized as something different.

Another of the commands specific to macroinstruction processor it the LOCAL directive. It tells it to define some new macro parameters in addition to the regulars one, and the value it assigns to each one of them should be a unique name token, generated differently each time the macro is processed. They are then replaced with these values in every place they are encountered in the macro text, just like the regular parameters. This way macro can generate some unique label names each time it is used. If you define LOCAL parameter with the same name as one of the regular parameters, it just gets redefined.

```
macro Testing a
{
  db `a,13,10
  local a
  db `a,13,10
}

Testing one
```

By assembling the above example and looking at the text generated as output you can see with what kind of value the LOCAL directive gives to a parameter.

There is one more important detail demonstrated by this sample: that the macroinstruction processor performs the token conversion after it has already replaced the parameters with their values. The same rule applies to the token concatenation operator.

On the other hand, as it was already mentioned, the conversion operator works only on a single token. If the parameter is replaced by a sequence of more than one symbol, the ` operator applies just to the first one of them. This may be a bit counter-intuitive, for example the above macro will not work properly when the parameter is a sequence of a few separate words, because only the first one of them will get converted into quoted string, and thus the input to DB directive will be incorrect. But this is where the IRPS directive comes in handy.

Processing Individual Tokens

The IRPS is a kind of instantaneous macroinstruction similar to IRP, as it has one parameter that iterates through a list of values. However, in this case it is not the list of entries separated with commas, but it simply a sequence of tokens (and every kind of token is allowed there), and for the value of parameter in each iteration it takes a single token, one after another.

This way any sequence of symbols that gets passed to a macro, can be taken apart and processed in detail, one token at a time. For instance, it allows to get around the problem that token conversion operator is able to convert only a single symbol at once. This macro iterates through all the tokens given in its parameter and displays them (spaced out) during the assembly time:

```
macro Tokens sequence
{
  irps token, sequence
  \{
    display \`token,32
  \}
}

Tokens eax+ebx*2+1
```

Note that token conversion operator had to be escaped here, otherwise it would get processed by the outer macro and the IRPS macro would not see it (instead, it would have the quoted string containing the "token" text in there).

The above example is not going to work if the sequence in the last line is modified to contain the comma character - simply because that comma would be regarded as a delimiter separating the different values for parameters. Of course, it is possible to specify a parameter that contains a comma by enclosing the complete value with < and > characters, but it is also possible to modify the above macro so that it works even when there are commas treated as delimiters:

```
macro Tokens [sequence]
{
  common irps token, sequence
  \{
    display \`token,32
  \}
}

Tokens mov ax,2+1
```

The COMMON directive takes all the separate values of the "sequence" parameter and joins them back with all the delimiters into a single chain, which in this case reflects exactly what was in the original line.

But there is another weak spot of this macro. The token conversion operator does nothing when the token already is a quoted string. So, in the output displayed by this macro there is no way to tell whether a given token was in quotes or not. This can be solved with some conditional assembly:

```
macro Tokens [sequence]
{
  common irps token, sequence
  \{
    if '' eqtype token
      display "'",token,"'",32
```

```
    else
      display \`token,32
    end if
  \}
}
```

But this solution is a little bit like mixing apples with oranges. The EQTYPE operator is processed at the assembly stage, and it allows to distinguish a few classes of syntactical structures that are recognized by assembler, for example it can tell apart the floating-point number from the register name. In particular it can recognize the quoted string, because this is a separate class of syntactical elements in flat assembler - and for this reason EQTYPE can be used here to react when the token was a quoted one.

However, it is possible to make this distinction already during the preprocessing stage. What is needed to accomplish this goal is the ability to execute some preprocessor's commands conditionally, depending on the kind of value held in a parameter - and it is the MATCH directive that provides such functionality.

Conditional Preprocessing

The simplest description of MATCH is that it is a kind of instantaneous macroinstruction that only gets processed when specified value matches the provided pattern. But there are many specific details that need to be known in order to use this directive correctly.

The syntax for MATCH is such that you first specify a pattern, and then after a comma the value that needs to match this pattern. The value has to come last because it may contain any sequence of tokens, including comma and other special symbols (except for the opening brace, which - as in case of all the macroinstruction - marks the beginning of macro body and thus forcibly ends the matched value). So, there is one syntactical comma in the MATCH definition, the one that marks the end of pattern and the beginning of matched value, and anything that follows it up to the end of line of opening brace is the value, which can be just about anything.

The pattern, on the other hand, needs to follow some strict rules. It is a sequence of elements which define what tokens or blocks of tokens are expected in the value. These elements come in two kinds - the ones that need to be matched literally, and the "wild card" ones.

Every symbol character (except =) and every quoted string is by itself a literally matched element. Also, any token (even the one that would be matched as-is anyway) can be preceded with = character to define the literally matched element. In particular the == and =, constructions can be used to match the equality sign and comma respectively, since they cannot be specified directly as they both have a special meaning in the context of the pattern.

All the examples below use the patterns composed of literally matched elements to get the positive match with the value that follows the comma:

```
match +,+
{ display "special character is matched as-is",13,10 }

match 'a','a'
{ display "the same with quoted string",13,10 }

match =a,a
{ display "the name token must be preceded with =",13,10 }
```

```
match =a=+='a' , a+'a'
{ display "and = may be actually used with any token",13,10 }
```

As for the name tokens, the reason why they need to be prefixed with = to make literal matches is that otherwise they define the wild card elements, which can match just any sequence of tokens, as long as there is at least one token to be matched with. Moreover, the macro parameter of such name gets defined with the value being exactly the sequence of tokens that it was matched with.

In the simplest case, if the pattern consists of nothing more than one name symbol, it matches the entire value that follows the comma, as long as there is at least one token to be matched with. Therefore, it can be used to execute the contents of MATCH block only when given value is not empty:

```
macro check value
{
  match any,value
  \{ display "the value is not empty" \}
}
```

In this example the MATCH is embedded inside another macro (so its enclosing braces had to be escaped) and therefore it is able to check what is passed in parameter, because parameters are replaced with their values during the preparation of line by the macro processor. So when the line produced by the outer macro gets preprocessed and the MATCH directive is interpreted, it already has the value that was passed to macro in place after the comma.

It does not mean, however, that the MATCH directive is only useful when put inside another macro. The standalone MATCH may too become handy, thanks to the fact that symbolic constants in the matched sequence are replaced with their values before performing the match.

```
define X

match any,X
{ not preprocessed }

define X +

match any,X
{ display `any }
```

In the above sample the first MATCH tests the value that is actually empty, because the symbolic contents get replaced with its empty value before the match is performed. Since the wild card must be matched with something, the condition is not fulfilled and the block is not preprocessed. Before the second MATCH the same symbolic constant is defined with a new value, and this time the analogously defined value is not empty - the wild card element is matched with the contents of X and becomes a parameter with the same value.

It is only possible to have a positive match with an empty value if the pattern itself is empty, since the wild card element needs to be matched with some non-empty sequence in the value. On the other hand, every consecutive wild card will try to consume as few tokens as possible. If the pattern consists of two or more wild card elements in a row, each one but the last will be matched with exactly one token (and the successful match is only possible when the value contains at least as many tokens as there are wild cards). Consider this example:

```
match car cdr, 1+2+3
{
```

```
  db car
  db cdr
}
```

We have two wild cards here, and they both could theoretically be matched with any non-empty sequence. However, the way fasm implements it, the first parameter takes as few tokens as possible, leaving no other choice than to throw all the following tokens into a second one. Thus, the first DB gets the value "1", while the second DB gets "+2+3".

Adding some literally matched elements between wild card put additional conditions on the match process, but the same general rules are enough to determine the result, as in this example:

```
match first:rest, 1+2:3+4:5+6
{
  db first
  dd rest
}
```

Here the first wild card matches everything up to the first colon (which is matched literally), because it tries to consume as few tokens as possible. Therefore, input for DB is "1+2", while DD receives the rest of the value after the matched colon, that is "3+4:5+6" (note that DD allows this kind of syntax, as it defines the segmented address combined of two 16-bit values - that's why this assembles correctly).

The literal matching of quoted strings, although generally less helpful than the other ones, can be used for a specific purpose of recognizing whether the value of macro parameter is a quoted string. The macro from previous chapter can be rewritten to use the conditional preprocessing, as demonstrated by the sample below.

Note the unusual placement of LOCAL command - because it is not escaped (as it does not matter in this case whether we localize with respect to the outer macro or the inner IRPS) it gets processed by the outer macro and does not produce any output into the generated text of the inner one, and thus it does not disrupt the syntax:

```
macro Tokens [sequence]
{
  common irps token, sequence
  local output
  \{
    output equ \`token,32
    match \`token,token
    \\{ output equ "'",token,"'",32 \\}
    display output
  \}
}
```

Since the conversion operator does nothing when it is followed by a quoted string, the converted token is equal to the original if and only if the original was already a quoted string. To check for this condition, the MATCH directive receives the converted token as a pattern, and this ensures that the literal matching is performed each time (because the pattern is always a quoted string). Therefore, the content of the MATCH block (which needs to be doubly escaped because of the nesting) is processed only for the tokens that are quoted strings.

Handling the Custom Syntax

Thanks to the wild card matching, the MATCH directive can do much more than check for simple conditions. It is possible to use it to parse some custom syntax constructions and therefore define macroinstructions with a less orthodox structure of the arguments.

Let's say that we wish to define a macro that would allow us to write "let al=4" instead of "mov al,4".

Normally the macroinstructions accept parameters separated with commas, so "let"defined as a macro would see "al=4" as one parameter. Therefore, we need to embed a MATCH directive in that macro - to recognize the structure of such parameters and split it into the parts we need:

```
macro let param
{
  match dest==src,param
  \{
     mov dest,src
  \}
}
```

Since = has a special meaning in the MATCH pattern, the == construction has to be used to match the = character literally. Now the match is going to be positive only when the parameter to the macro contains something followed by the = character and again by something. Otherwise the macro will do exactly nothing. But we can add more possible syntaxes by putting more of the MATCH directives in a row, like:

```
macro let param
{
  match dest==src,param \{ mov dest,src \}
  match dest++,param \{ inc dest \}
}
```

This version not only allows to generate "mov al,4" by writing "let al=4", but also allows to write "let al++" and generate the "inc al" instruction.

But now consider we wanted to additionally allow the "let al+=4" to generate the "add al,4". We could simply add this line to the above macro:

```
match dest+==src,param \{ add dest,src \}
```

But even though it would correctly generate the ADD instruction if we wrote "let al+=4", also the "dest==src" would be matched in this case, with "al+" being assigned to "dest", and we would get the erroneous instruction "mov al+,4" generated at the same time.

To prevent such problem, we need to somehow handle the information that the syntax has been already recognized and prevent any more matches from happening in such case. Every MATCH directive is independent, so to pass such information we need to employ some symbolic variable. The first solution coming to mind may be like the one below. Note that we need to match the "dest+==src" pattern first, to make "al+=4" be recognized as += syntax, not the = one.

```
macro let param
{
  local status
  define status 0
  match dest+==src,param
```

```
  \{
     add dest,src
     define status 1
  \}
  match =0,status
  \{
     match dest==src,param
     \\{
    mov dest,src
    define status 1
     \\}
  \}
}
```

But it can be done simpler. These nested matches can be actually composed into a single one:

```
  match =0 dest==src  ,        status param
  \{
     mov dest,src
     define status 1
  \}
```

The comma that separates the matching expression from matched text has been spaced out, so you can see what is actually matched. Since the first token that has to be matched is the literal 0, and the first token of text is either 1 or 0, as those are only possible values of symbolic variable, the match can happen only when it is defined to be 0 and, in such case, the remaining part of text (which happens to be exactly the value of parameter) is matched to "dest==src".
Thus, the complete macro may look like this:

```
macro let param*
{
  local status
  define status 0
  match =0 dest+==src , status param
  \{
     add dest,src
     define status 1
  \}
  match =0 dest==src , status param
  \{
     mov dest,src
     define status 1
  \}
  match =0 dest++ , status param
  \{
     inc dest
     define status 1
  \}
  match =0 any , status param
  \{
     err "SYNTAX ERROR"
  \}
}
```

Each one of the patterns contains the =0 matched to the value of symbolic variable, even though in the first one it is completely redundant - but it makes the macro uniform. The last MATCH checks for the case when the syntax was not recognized by any of given rules, and it signals an error then. Checking for the case of empty parameter is not needed, since it is ensured by the * following the name of parameter

in the definition of macro. Otherwise we would need to add another (and the last one) MATCH to check for the empty value.

The use of the ERR directive requires some explanation. This command is processed during the assembly stage (so it still may be suppressed when put inside an IF block), but it causes an assembler to abort immediately as soon as it is encountered, even if it happens only in some intermediate stage of code resolving process. Therefore, the purpose of this directive is to signal an unrecoverable error; in this case - a syntactical one. The ERR directive also does not have any rules concerning its arguments - it breaks the assembly with an error message without even looking at the remaining text of the line. And the above sample simply has an additional message put there, to clarify what kind of problem occurred - the flat assembler always presents the line that caused an error, so this message is going to be in sight.

Table of Contents