AM 100 Notes
3/19/73

Macros were the subject of the day. A macro "... provides a convenient way to generate a desired sequence of assembler statements many times in a program. The macro definition is written only once, and a single statement, a macro instruction statement, is written each time the programmer wants to generate the desired sequence of statements" (from OS/360 assembler manual). In other words, they can save you a lot of time since code that is similar in different parts of your program need only be written once, as a macro.

Macros are placed at the front of your OS program, and consist of a header statement, a prototype statement, assembly language statements, and a trailer statement. For example,

```
    macro
div $a,$b,$c
l $r0,$a
srda $r0,32
d $r0,$b
st $r1,$c
end
```

is a macro definition.

Note that the prototype statement (the second one) includes a macro name and contains symbolic parameters (variables preceded by an ampersand) -- it is these symbolic parameters which act like parameters of a function call of sorts, with substitution taking place for all occurrences within the macro. Thus a macro call of DIV as such

```
div x,y,z
```

will result in the following code being generated by the macro processor.

```
l $r0,x
srda $r0,32
d $r0,y
st $r1,z
```

Notice that since the macro preprocessor performs substitutions before the assembler gets to look at the code that is generated (it's just a text processor), a macro instruction may indeed form code which is incorrect assembler language instructions. All the macro processor does is perform whatever substitutions you ask of it, and check for invalid macro statement syntax.

The macro language also allows you to have assumed values to be specified within the macro definition. For example,

```
div $a,$b,$c,$r=0
```

performs a substitution for the first three positional parameters as before, but if no parameter is specified in the macro call, $r=0 (a keyword parameter) will be assumed to be $r0. If you give a value for $r=0 in the macro call, that value will override the $r0.

```
div x,y,z,$reg=14
```

is an example of this, and a substitution of $r14 will occur
wherever &reg appears in the macro definition statement. Note that positional parameters must precede keyword parameters, not only in the macro definition, but also the macro call. A series of keyword parameters, however, may appear in any order, assuming that they all follow any positional parameters.

As the macro processor manipulates text, you might expect it to be able to do concatenation and all sorts of glitzy things that would make your mouth water. Well, we hate to disappoint you, but the processor can't do everything (at least not yet -- watch this space for further developments). It can do concatenation, though. If &a has a value of 'abc' and &b 'cde', then &a&b will be the string 'abcde'. To concatenate &a and 'd' together, &a'd would be incorrect, as this specifies a parameter ad of length two rather than concatenation. Use the delimiter '.' to solve the problem. Thus &a.d would do the trick for the previous hypothetical problem.

There exists an interesting little system parameter that can be used within macro definitions, &sysndx. This can be used to generate a unique label each time the macro definition is used in your program (remember that duplicate labels are a no-no in SOS). It's initial value is '002', and this is increased by 1 each time &sysndx is encountered by the macro generator. You can even nest macros within other macros and still use &sysndx, though you're playing with real hot stuff in this case, friends.

As usual, our friendly SOS system provides some macros for us, and you've probably used them without knowing it. Yes, getx, putx, putc, and the rest are macros, and expand to real live SOS code.

Comments within macros are just the same as regular SOS, and will be printed on your listing when the macro is expanded. If you do not want the comment to appear in the expansion, but just in the macro definition, preceed the line by '.*' instead of just '.*'.

As usual, refer to the SOS manual for a better explanation of all this.

yours truly,

Neal Pierman