Introduction to Macros

by Alan D. Floeter
4333 N. 71 St.
Milwaukee, WI 53216

If someone came up to you and said, "I'll give you a million dollars if you can tell me what macros are and what they're good for", would you be a millionaire? Read on for a lifetime of riches.

Many people are getting interested in assembly language. It provides the speed, freedom and control unmatched by Applesoft or Integer, or any other high level language for that matter. The only problem is that assembly language programming is tedious. It takes many lines of code to do the job of one line of BASIC. This is where Macros can make a difference. Many of the assemblers available for the Apple have macro capabilities. But what is a Macro?

A Macro is a group of assembly language instructions identified by a single Name. If you want to perform a group of instructions many times, you can identify that group as a macro. Whenever that macro name is used, the assembler knows to pull in those instructions that the name represents.

AN EXAMPLE

Let's look at a simple example. Say your program zeroes out a location called **TEMP** several times. You could set up a macro called **ZEROTEMP** that would do a **LDA #0, STA TEMP**. Now whenever you want **TEMP** zeroed, all you need to do is invoke **ZEROTEMP**, and the assembler would automatically supply the instructions to do the job.

In this example, we are only reducing the coding process by one statement. (It took two statements to zero TEMP without a macro, and now it requires only one.) But you can see that macros containing many statements could produce a big savings in source code and writing time. Notice that the final object code turns out to be the same; it's only the source code that gets reduced.

EXTENDING MACROS

But we can take macros another step further by having a **macro accept parameters** being passed to it. This means that when you invoke a macro, you can also send information along with it that can be picked up by the macro definition. The **ZEROTEMP** macro worked fine for zeroing the location **TEMP**, but what if you also wanted to zero a different location called **NUM**? You could write a **ZERONUM** macro, but you should be able to just **pass a location name** to a **ZERO** macro and have the assembler substitute the name you request. So you could say **ZERO TEMP** and it would substitute **LDA #0, STA TEMP**. Or you could say **ZERO NUM** and get **LDA #0, STA NUM**, or any other location you need.

Once you start passing parameters you can really see the convenience of using macros. You could define a macro that sets a location to a certain value, so that by saying **SET NUM,5** the value **5** is stored into **NUM**. Maybe you could use a macro that transfers a value in one location to another. **TRANSFER NUM,TOT** could be set up to transfer the contents of **NUM** to **TOT**. The possibilities are endless.

Besides convenience, another reason to use macros is to make your code more readable. **You can take BASIC commands such as HOME or GR and define macros with those names that call the Monitor routines for those functions.** So instead of seeing a **JSR** to some hex number, you would see the macro name **HOME** which immediately tells a person familiar with BASIC what is going on.

CUSTOM COMMANDS

The third use is to create **customized instructions** and **pseudo-ops**. We have found macros to be very valuable in converting a program developed under another assembler. When the assembler we use is missing a pseudo-op used in another assembler, we can often write a macro to handle the unusual pseudo-op. We have also used macros to handle Sweet-16 code and assemblers for other processors.

If a macro assembler has the ability to examine the parameters character by character, it is even possible to **write macros for high level language instructions**. Look at the statement **LET X=0**. If we develop a macro called **LET**, it could break down the rest of the statement to produce assembly code to store a **0** into the location **X**. However, this is a very advanced topic and unfortunately very few macro assemblers are even capable of handling parameters on a character basis, but you can see the potential for macros.

MACRO LIBRARY

Of course, once you've developed a group of macros, your assembler should be able to store them into some sort of library file that you can call in whenever you want. This way **macros built up previously can be used by all of your files without retyping them in**. As you develop a library of useful macros, you will be able to write source code in less time, and have it be more readable, which should result in less debugging time.

So now that you know what macros are good for, how do you get started? Well, since the assembler does all the work, you must have an assembler that can handle macros. That narrows the choices to a very select few. However, just because an assembler has macro capability doesn't mean it can handle everything you may dream about. Some assemblers don't allow nesting, which means that you can't call a macro inside a macro. This limitation would be unacceptable to us, although it may not be to someone who is just starting out. Likewise we think recursion (macros that can call themselves) is extremely important. But when it comes down to it, parameter handling is the key. You need to really examine how well you can get to the information passed to a macro. If you are always going to pass parameters with commas, then you don't need string capability, but if you want to search a parameter, make sure that you can do so.

Well, do you know enough to be a millionaire? Purchasing a powerful macro assembler and using the macro capability multiplies the value of assembly language programming many times over. If you are still writing assembly language programs without macros, you may be unnecessarily slowing your software development. It may be time to take a macro step forward.