SYMASS: A Symbolic Assembler For The Commodore 64

Robert Huehn
Neustadt, Ontario

Now Assemble Any Transactor Program, Anytime!

Symbolic assemblers, used to assemble machine language programs, are essential tools for serious programmers. The merits of machine language need not be discussed here. If you haven't broken down and bought one yet, you've probably been using a monitor such as Supermon. Monitors were never meant for program development. After trying to insert a couple of instructions into a long program with a monitor, you must also readjust the rest of your program properly. Then you think very hard about alternatives.

Unfortunately there were very few choices until now. SYMASS was written to fill the gap. It is a very fast, compact, easy to use assembler with enough features for serious programs. Besides, it's in the public domain. After experiencing SYMASS in action, you will gladly demote your monitor to debugger.

You're likely already familiar with SYMASS syntax, since it is totally compatible with PAL. PAL source code is published often in each issue of The Transactor. SYMASS syntax evolved through many changes from its beginning as a BASIC program (which would take over twenty minutes to assemble early versions.) It now includes most of PAL's features, including the ones most often used in Transactor programs. PAL has no problems assembling SYMASS itself, but SYMASS is faster. SYMASS source code is about 18 K bytes long and PAL takes about 17 seconds to assemble it. SYMASS assembles itself in six seconds.

Type in SYMASS 3.0.GEN, then run it. (It's not long, but you might consider getting the Transactor disk for this issue, especially if you want the source code.) It will create the final program, SYMASS 3.0, on disk. (The generator program could also be modified for tape, since SYMASS doesn't use the disk drive.) The SYMASS 3.0 loader will relocate itself at the top of memory when it is run. Source code is entered with the BASIC editor; use 'sys 700' alone on the first line to call SYMASS. Leave out the PAL's .opt xx statement since SYMASS assembles to memory only. Type 'run' and save the object code with a monitor.

Probably the major limitation of SYMASS is both source and the resulting object code must reside in memory along with SYMASS. SYMASS doesn't take up much room, (about 2.6 K) but you will have problems if the source is too long to fit with the object code.

A partial list of SYMASS/PAL compatible features follows:

*=$c000	;define start of program
name=$ff	;assign a value to a symbol
=+n	;skip n bytes for storage
;	;comments follow
$	;hexadecimal value, default is decimal
%	;binary value
'	;ASCII value of character
!	;force absolute addressing
>high, <low	;low or high byte of word
+, -	;add, subtract
.byte $ff	;store bytes
.word $ffff	;store words
.asc "text"	;store string of characters
.end	;end assembly

You can use SYMASS without knowing how it works, but the explanation will help you get the most out of it.

SYMASS itself is composed of small modules, each performing a specific function. In general, each module could be replaced by another section of code, if it performs the function correctly. This makes it easier to modify small sections without any side effects. SYMASS was debugged that way.

SYMASS makes two passes over the source code. During the first pass, SYMASS builds a symbol table of all the symbols which appear in the program. It then stores the object code to memory on the second pass, after all unknown symbol values have already been defined. A variable called FLAG is set to 0 or 1, depending which pass SYMASS is currently on.

The source code has already been tokenized by the BASIC editor, but this causes no problems. It even reduces the amount of memory needed for the source. Opcodes such as 'and' are already stored as tokens internally, as are custom pseudo–ops like '.end'.

WORD is the most basic routine to find the next word. WORD defines a word as a sequence of characters ending with a space, colon, semi–colon, or equal sign. It also ignores leading spaces, and has a quote mode that accepts any character except the end of line.

A pointer, AD, and the .y register is always used to access the source code. When WORD is called, the pointer AD is advanced over leading spaces, then the .y register is advanced to the end of the word and the result is stored in LEN. Therefore, LEN is the length of the word, '(ad),y' gives the stop character when .y equals LEN, and the first character when .y equals zero. Two routines, NEXTWORD and NEWWORD, set up AD and call WORD. NEXTWORD starts at the current stop character, so will only get another word if a space separates them. NEWWORD, on the other hand, skips over the stop, and is used to get the expression after an equals sign.

It's important to understand how those routines work if you wish to use them in your own additions to SYMASS.

SYMASS creates a symbol table which starts at the top of memory and grows downward to the end of the source code. A symbol table overflow results if not enough memory is available. Each entry takes ten bytes; eight to store the name, and another two for the value. If tokens are embedded in the name, its actual length could be longer than eight characters, but it's not a good idea.

CRSYM creates a symbol table entry. It decides if there is enough room, then copies the current word into the table. It is your responsibility to make sure a symbol isn't defined twice. Whenever the value of a symbol is needed, FINDSYM is called. FINDSYM returns with the value in the .a and .x registers, or prints an 'undefined symbol' message.

FINDSYM uses the simplest possible search method, searching from beginning to end. It might be worthwhile to use a different method, such as a hash function, to save time. (Calculate the storage address with a special function, such as the remainder of table size / ASCII sum of name.)

The opcode table makes up 728 bytes of SYMASS. Again, FINDOP does a linear search. The more commonly used opcodes are close to the beginning. You could fine–tune the table to your style by counting the number of times each opcode appears in your programs, then rearranging the table in that order. If you do so, change the brk op# and bit op# in DOOP and PUTOP to their new positions. You could also easily add extra opcodes such as skb (skip byte) to the table, changing NOPS to reflect the change.

Two other major routines are EVAL and PUTOP.

EVAL takes the current word, an expression containing no spaces, evaluates it, and returns the result. It can add or subtract symbols, decimal, and hexadecimal numbers. A character enclosed in single quotes will return its ASCII value. A > or < can be placed at the beginning of the expression to return either the high or low byte of the result. The number conversion routines only convert from BASIC's format as a string of characters to a useful two–byte binary number, not both ways. This is why SYMASS gives the end of assembly as a decimal number instead of hex. The BASIC ROM routine that prints 'in xxxxx' is used.

During the first pass DOOP keep track of the current object address with a pointer called PTR. PUTOP is used on the second pass to store the machine code into memory. It recognizes all addressing modes. Since there is no difference in syntax between zero and absolute modes, the correct mode may sometimes be ambiguous.

Suppose you are storing variables in memory after the end of your program, with a label to identify the location. On the first pass, an instruction such as 'lda variable' would normally cause FINDSYM to give an undefined symbol error. FINDSYM therefore tries to guess your meaning by returning the value of PTR for undefined symbols on the first pass. Other assemblers may use zero, and cause an instruction like 'lda variable+1' to produce a phase error. A phase error results when the assembler makes the wrong guess, and reserves an incorrect number of bytes for an instruction.

SYMASS doesn't have phase errors. You can force SYMASS to use absolute mode with a ! prefix, or to zero page by a <, which works by returning the low byte.

You can add your own specialized commands to SYMASS by adding them to the CUSTOP routine. One such command, '.pad' will add a zero to the object code if the current address is odd. You might use it sometime to make sure a jump table doesn't cross a page boundary.

SYMASS leaves room for optimization; the major goals in its design were simplicity, speed and ease of use. WORD, since it is used so often, is a good candidate. PAL doesn't seem to recognize '=' as the end of a word. If the relevant parts of SYMASS were changed, the check could be taken out of WORD. A useful, but probably more complicated improvement is assembly to disk.

SYMASS's hidden strength is the ease with which it can be modified, compared to commercial programs which do not provide source code. You can also study SYMASS just to learn how to write an assembler. In the end though, SYMASS is a tool which will enable you to write machine language programs as complex as your growing skills allow.