The Applesoft Compiler (TASC): We have the source code, in a sense

devblogs.microsoft.com/oldnewthing/20220419-00

April 19, 2022

Raymond Chen

Back in the early 1980's, the Apple][computer was taking the personal computing world by storm, and Microsoft released a compiler for Applesoft BASIC. That compiler went by the name TASC: The Applesoft Compiler.

TASC was written by one person in his dorm room while studying at MIT. Microsoft licensed the product and hired the author, who spent the summer at the Northup building[1] polishing the code.

As noted on page 61 of the manual:

> TASC is a "two-pass" compiler, since it compiles in two major steps. PASS0 simply picks up user inputs and sets up compilation parameters, so it is not really part of the actual compilation process. The Applesoft program TASC runs PASS0.
>
> PASS0 and PASS1 chain to PASS1 and PASS2, respectively. All three passes were written largely in Applesoft, and TASC was used to compile itself.

Chaining refers to a program instructing the system to replace the current program in memory with another program, but preserve the values of some or all variables. Chaining was a common technique when your program got too large to fit into memory all at once, so you broke it into multiple programs that each handed off control to each other.

Even if you hadn't made it that deep into the manual, you could have figured out that TASC was used to compile itself because TASC used its own runtime library.

Chaining was not a feature native to Applesoft BASIC. It was one of a handful of language extension provided by TASC itself, through the use of magic comments that begin with an exclamation point. This meant that once TASC development reached the point that it required chaining, it could be run only in its compiled form. There was no way to bootstrap it from the interpreter.

As the author added features, he kept hitting the Apple][‘s 48KB RAM limit and was forced to delete all the comments from the code, and when that wasn’t enough, he resorted to shortening all the important variable names to one character.

Such is the desperation of developing on a system with very tight memory constraints.

Everything was working smoothly, until the author returned to school for a semester. Upon returning to Microsoft, he found that he no longer understood the code. He had a sprawling compiler, with no comments, and unhelpful variable names.[2]

Yet somehow, he finished TASC, and it shipped.

If you dig through the TASC manual, you can find all sorts of wonderful implementation details.

All of the real work happened in pass 1. This performed code generation and left placeholders for references to other locations like branch targets or variables. Pass 2 consisted of resolving these references and patching up the code. Even though Pass 1 had all the smarts and Pass 2 was just doing clerical work, it was Pass 2 that took the most time because it was I/O-bound, and floppy disks are not speed demons when it comes to random access. Pass 2 was I/O-bound not only because of the need to patch the object code, but also because the table of line numbers was itself written to disk, there not being enough RAM to keep it in memory.

Pass 2 made a pass through the object code once for each variable, since the references to a variable were threaded through the object code as a linked list, similar to how 16-bit Windows threaded external references through the code instead of keeping a separate fixup table. Your program has 100 variables? Then that’s 100 passes through the object code to update references to 100 variables.

When the code generator needed to access a variable, it didn’t do so directly. The 6502 was an 8-bit processor, so none of the variables fit into a register. You needed to call a function to transfer the variable’s value to or from a common staging area.[3] Your traditional code generation went something like this:

```
    ; BASIC code: X = A + B

    ; traditional code generation
    lda #address_of_a_lo
    ldy #address_of_a_hi
    call load_variable_to_accumulator

    lda #address_of_b_lo
    ldy #address_of_b_hi
    call load_variable_to_arg

    call add_arg_to_accumulator

    lda #address_of_x_lo
    ldy #address_of_x_hi
    call store_accumulator_to_variable
```

To save four bytes at each call site, the address-loading is factored out, and each variable gets a dedicated entry point:

```
    ; revised code generation
    call load_variable_a_to_accumulator
    call load_variable_b_to_arg
    call add_arg_to_accumulator
    call store_accumulator_to_variable_x

    ...
    ; block of variable access functions

load_variable_a_to_accumulator:
    lda #address_of_a_lo
    ldy #address_of_a_hi
    jmp load_variable_to_accumulator

load_variable_b_to_arg:
    lda #address_of_b_lo
    ldy #address_of_b_hi
    jmp load_variable_to_arg

store_accumulator_to_variable_x:
    lda #address_of_x_lo
    ldy #address_of_x_hi
    jmp store_accumulator_to_variable
```

This is a net win if each variable is accessed several times, which is a pretty fair assumption.

To save code size further, the access function was itself parameterized on the type of access.

```
store_arg_to_variable_x:
    ldx #4
    .byte 0x2c      ; swallow next two bytes
load_variable_x_to_arg:
    ldx #3
    .byte 0x2c      ; swallow next two bytes
store_accumulator_to_variable_x:
    ldx #2
    .byte 0x2c      ; swallow next two bytes
load_variable_x_to_accumulator:
    ldx #1
    lda #address_of_x_lo
    ldy #address_of_x_hi
    jmp do_something_with_variable ; uses value in X to decide what to do
```

We are using the trick of jumping into the middle of an instruction to provide multiple entry points to a common block of code. The author of TASC was very proud of this optimization.

Related reading: Excuse me, has anybody seen the FOCAL interpreter?

[1] This is the same building that was next door to the restaurant that inspired an important variable name in the 16-bit Windows kernel.

[2] Also, Applesoft BASIC didn't have local variables. All variables were global. That certainly didn't help with understanding the code.

[3] It has been said that when you write code for the 6502, you're not so much writing code as you are writing microcode. The CPU itself has only three 8-bit registers (A, X, and Y), and only A can do arithmetic. Anything of more than ephemeral value must be stored in memory. The real working space was the zero page. For example, you might decide that one region of the zero page was the logical "accumulator", and most of your time was spent transferring values into or out of that accumulator, interspersed with occasionally performing arithmetic on or testing the value in that accumulator.

Perhaps the most famous example of treating the 6502 as microcode is the SWEET16 interpreter, written by Steve Wozniak, which emulated a 16-register 16-bit virtual processor in roughly 300 bytes of memory.

Raymond Chen

Follow