The ACTION65! Cross Compiler for the 6502

REFERENCE MANUAL

Preface:

The original compiler, ACTION!, was released in 1983 for the Atari 8 bit computer line by OSS. The compiler was written by Clinton Parker. ACTION! Was a real game changer. There were several choices for writing programs for the Atari 8 bit computer. There was BASIC, Assembly (of course), FORTH, and pretty much that was it. BASIC was just plain too slow, but it was more or less easy to read, unless the author wrote it as spaghetti code. FORTH was faster, but unless you were a FORTH devotee, it was very difficult to interpret. And of course there was assembly code, which had as it’s limit on speed the capabilities of the core processor. But assembly was labor intensive, difficult to debug, and pretty much unreadable. There were the choices.

But then ACTION! Came along and changed everything. It was not an interpreter, it was a real compiler. It directly generated machine code from the source code, and it did this fairly quickly. Now, if you were to write code that did exactly the same thing in both ACTION! And Assembler, you would find that the assembly code was smaller and faster than the ACTION! Code. But not by much. ACTION! Programs were basically almost as fast, almost as small, and much easier to read than Assembly code.

This Program

Writing a compiler is not easy. We all need to take our hats off to Clinton Parker because he not only wrote a compiler, but he did so in 6502 assembly. An amazing feat.

This program is being written using C++. There are about 160 different classes, occupying 330 files. That is bad enough. But I also chose to use an LL(1) grammar. In the original ACTION! Manual, the grammar that is in one of the Appendixes is an LR(1) grammar. LR grammars are more powerful and much easier to read than an LL(1) grammar. However, writing a parser for an LR(1) grammar is just about, well, impossible. It can be done, but there are many fine tools that will generate the parser for you and makes them practical. An LL(1) parser is pretty easy to implement directly from the grammar by using a technique called Recursive Decent. You can also implement an LL(1) parser using a table. However, the grammar I created would need a 2 dimensional matix with 44,000 entries. Sort of impractical, but again, there are programs that can generate those tables.

I have no idea if the original ACTION! Compiler used an LR model or an LL model. One thing I ran into pretty quickly was an ambiguity I ran into translating the LR grammar in the manual into LL. This is fairly easy to do.

The problem was in the declarations for global variables and functions.

INT a,b,c

INT FUNC afunc()

INT d,e,f

So above there is a declaration for three variables a, b and c. We then follow that by a function and then another declaration for three more variables. Why this is a problem is that the parser can only look ahead by a single token. I did come up with a solution, it was kind of messy, but it worked. In the original ACTION compiler, you could declare global variables until you hit the first function. After that, no more global variables. Clinton seems to have solved the problem using this little trick. I stumbled onto a solution later on that was not only cleaner, but also allowed the mixing of function and variables to your hearts content.

Of course, I basically have no limit on memory resources, where as Clinton was constrained to a 8K block of memory. I am still amazed at the fact that Clinton accomplished this feat.

1. Declarations

Declarations are pretty much the same as they were in the original. The BOOL data type has been added to the set of datatypes.

* 1. Fundamental Data Types
     1. BYTE
     2. CHAR
     3. INT
     4. CARD
     5. BOOL
     6. PROCEDURES
        1. PROC
        2. FUNC
        3. INTERRUPT

This is used to define an Interrupt procedure. It is very similar to a regular procedure except no parameters can be passed to it, it saves the processor context onto the stack, and uses the RTI instruction to return rather than the RTS that is used by PROC and FUNC.

* + 1. RECORD TYPE
    2. TYPE MODIFIERS
       1. POINTER
       2. ARRAY

1. Statements
   1. Assignment
      1. ‘=’
      2. '==+'
      3. '==-'
      4. '==\*'
      5. '==/'
      6. '==MOD'
      7. '==&'
      8. '==!'
      9. '==%'
      10. '==LSH'
      11. '==RSH'
   2. FOR
      1. FOR
      2. TO
      3. STEP
      4. DO … OD
      5. Statements
   3. WHILE
      1. WHILE
      2. Loop Break Condition
      3. DO … OD
      4. Statements
   4. DO … OD
      1. Infinite Loop.
      2. Breaking from the loop
         1. UNTIL
         2. EXIT
   5. IF
      1. IF
         1. Conditional
      2. THEN
      3. ELSEIF
      4. ELSE
      5. FI
      6. Statements
   6. IFF
      1. Differences between IFF and IF
      2. Conditionals
         1. BIT
      3. THEN
      4. ELSE
      5. FFI
      6. Statements
   7. UNTIL
      1. UNTIL
      2. Conditional
   8. EXIT
   9. RETURN
      1. RETURN
      2. Return value
         1. PROC
         2. FUNC
   10. Code Blocks [ …. ]
       1. Boundaries
       2. Values
          1. Numbers
          2. Labels
   11. Inline Assembler ASM
       1. ASM
   12. PUSH
       1. PUSH
       2. PARAMETERS
          1. REGOSTERS
          2. MEMORY CONTENTS
          3. CPMSTAMTS
   13. POP
       1. POP
       2. PARAMETERS
          1. REGISTERS
          2. MEMORY LOCATIONS
   14. BREAK
   15. RTI
2. Program Structure
   1. Parameters
   2. Local Variables
   3. Procedure/Function/Interrupt Body
   4. Operators
   5. Constants
   6. MODULE
3. Inline Assembler
   1. PROCESSOR
      1. 6502
      2. W65C02
      3. W65C816
   2. SECTION
      1. Start Address
      2. Size
      3. Read Only/Read Write
      4. Section Name
      5. Default Section Names
   3. ORG
   4. Define Memory
      1. .DB
      2. .DW
      3. .DL
      4. .DAS
      5. .DCS
   5. Define Storage
      1. .DS
   6. Assembly Procedure
      1. PROC
      2. EPROC
   7. Labels
      1. Global Labels
      2. Private Labels
   8. Instructions
      * 1. ADC
        2. AND
        3. ASL
        4. BIT
        5. BCC
        6. BCS
        7. BVC
        8. BVS
        9. BMI
        10. BPL
        11. BNE
        12. BEQ
        13. BRK
        14. CLC
        15. CLD
        16. CLI
        17. CLV
        18. CMP
        19. CPX
        20. CPY
        21. DEC
        22. DEX
        23. DEY
        24. EOR
        25. INC
        26. INX
        27. INY
        28. JMP
        29. JSR
        30. LDA
        31. LDX
        32. LDY
        33. LSR
        34. NOP
        35. ORA
        36. PHA
        37. PHP
        38. PLA
        39. PLP
        40. ROL
        41. ROR
        42. RTI
        43. RTS
        44. SBC
        45. SEC
        46. SED
        47. SEI
        48. STA
        49. STX
        50. STY
        51. TAX
        52. TAY
        53. TSX
        54. TXA
        55. TXS
        56. TYA
   9. Addressing Modes
      1. Accumulator
      2. Implied
      3. Immediate
      4. Absolute
      5. Absolute Index X
      6. Absolute Index Y
      7. Zero Page
      8. Zero Page Index X
      9. Zero Page Indexed Y
      10. Zero Page Indexed Indirect (X)
      11. Zero Page Indirect Indexed (Y)
4. The Little Details
   1. Memory Layout
   2. ROM programs
   3. Parameter Passing
      1. Through Page Zero
      2. Registers
      3. On the Stack
   4. Local Variables
      1. Static Allocation
      2. Dynamic Allocation
      3. Recursion

You can think about it, but don’t do it.

Seriously, given the very limited nature of the stack resources on the 6502, this is just probably not a good idea. Using the 65C816 will make this less of a problem.

1. Run Time Library
2. Grammar
   1. Declarations
   2. Statements
   3. Inline Assembly Code
   4. Memory Content
   5. Constants
3. Compatibility with the Original ACTION!