A 6502 macro assembler in FORTH

Motorola's 6502 microprocessor is becoming increasingly popular on small computers. Mike Jarvis has implemented a FORTH macro assembler on the 6502

An account of the writing of a FORTH macroassembler for the 6502 microprocessor is given. The processors mnemonics were divided into groups and taken care of by five routines using two FORTH words. Examples are presented designed to run under the implementation of fig-FORTH for the Microtan microcomputer. Some of the problems encountered in such an implementation are considered.

FORTH 6502 macroassembler

In this article I shall attempt to illustrate the power of the language FORTH by giving a brief account of the writing of a macro assembler. I had already implemented FORTH on a Microtan 6502-based computer, and to make the language as complete as possible an assembler was needed.

The first task in writing the assembler was to divide the acceptable 6502 mnemonics into groups according to the type of code produced. For instance, the SEC (set carry flag) instruction is always 38 hex. A large group of instructions in 6502 code fall into this single byte category. Other instructions require extra bytes of data and change according to the addressing mode chosen. The whole list of 6502 codes can be broken down into five groups.

The process of deciding what to do with a particular mnemonic was taken care of by five small routines using the FORTH words <BUILDS and DOES>. These words effectively divide a definition into two parts one which is performed at compile-time and one executed at run-time. Everything between <BUILDS and DOES> is done at the time of compilation with that following DOES> being executed at run-time.

The bulk of the source code for the assembler is taken up by the definitions of the five routines and the groupings of the mnemonics into their types. The rest is a list of special addresses for subroutine jumps into the FORTH machine code kernel, control structures and macro facilities. The whole program is contained in just four screens on disc (4k of storage) and compiles down to a little over 1k of object code.

```
SCR   1
 ( ASSEMBLER 2.1 )
 FORTH DEFINITIONS VOCABULARY ASSEMBLER IMMEDIATE
 HEX ASSEMBLER DEFINITIONS 0 VARIABLE TWO/THR
 0 VARIABLE FIXIT
 : INIT 0 TWO/THR ! 0 FIXIT ! ;
   SET <BUILDS , DOES> DUP C@ TWO/THR ! 1+ C@ FIXIT !
 FC00 SET X) 0400 SET
 0C00 SET )Y 2001 SET )
   .Y DUP 100 / IF 1 TWO/THR ! 14 FIXIT !
 ELSE 10 FIXIT ! ENDIF ;
   ,X DUP 100 / IF 1 TWO/THR ! 18 FIXIT !
 ELSE 10 FIXIT ! ENDIF ,
   .A 2 TWO/THR ! 4 FIXIT ! ;
   TYPE1 <BUILDS C, DOES> C@ C, ;
   TYPE3 <BUILDS C, DOES> C@ FIXIT @ + C, , INIT
 -->
```

M & J Software, 34 Gray's Close, Scholar Green, Stoke-on-Trent, ST7 3LU, UK

```
SCR £ 2
 ( ASSEMBLER 2.2 )
   TYPE2 <BUILDS C, DOES> C@ FIXIT @ IF FIXIT @ + C,
 TWO/THR @ IF , ELSE C, ENDIF ELSE
 >R DUP 100 / IF 8 R> + C, , ELSE R> C, C, ENDIF ENDIF INIT ,
   TYPE4 <BUILDS C, DOES> C@ FIXIT @ -DUP IF + C, TWO/THR @ 1 =
 IF , ELSE TWO/THR @ 0= IF C, ENDIF ENDIF ELSE OVER 100 / IF 8 +
 C, , ELSE C, C, ENDIF ENDIF INIT ,
 : TYPE5 <BUILDS C, DOES> C@ FIXIT @ 4 = IF -4 FIXIT !
 ELSE FIXIT @ 14 = IF 18 FIXIT ! ENDIF ENDIF
 FIXIT @ IF FIXIT @ + C, TWO/THR @ IF , ELSE C, ENDIF ELSE
 >R DUP 100 / IF 8 R> + C, , ELSE R> C, C. ENDIF ENDIF INIT
 00 TYPE1 BRK, 08 TYPE1 PHP, 10 TYPE1 0< 18 TYPE1 CLC,
 38 TYPE1 SEC, 40 TYPE1 RTI, 48 TYPE1 PHA, 50 TYPE1 VS
 60 TYPE1 RTS, 68 TYPE1 PLA, 78 TYPE1 SEI, 88 TYPE1 DEY,
 90 TYPE1 CS 98 TYPE1 TYA. 9A TYPE1 TXS. A8 TYPE1 TAY,
 B8 TYPE1 CLV, BA TYPE1 TSX, C8 TYPE1 INY. CA TYPE1 DEX, -->

SCR £ 3
 ( ASSEMBLER 2.3 )
 D8 TYPE1 CLD. E8 TYPE1 INX. EA TYPE1 NOP. F8 TYPE1 SED,
 28 TYPE1 PLP, 58 TYPE1 CLI. 8A TYPE1 TXA, AA TYPE1 TAX,
 D0 TYPE1 0= 4C TYPE3 JMP, 20 TYPE3 JSR. 05 TYPE2 ORA,
 06 TYPE4 ASL, 25 TYPE2 AND, 26 TYPE4 ROL. 45 TYPE2 EOR,
 46 TYPE4 LSR, 24 TYPE2 BIT, 65 TYPE2 ADC, 66 TYPE4 ROR.
 84 TYPE2 STY, 85 TYPE2 STA, A5 TYPE2 LDA. C5 TYPE2 CMP,
 C6 TYPE2 DEC, E5 TYPE2 SBC, E6 TYPE2 INC, A4 TYPE5 LDY.
 A6 TYPE5 LDX, C4 TYPE5 CPY, E4 TYPE5 CPX. 86 TYPE2 STX,
   BEGIN, HERE 1 ,   IF, BEGIN,  ALLOT 2 ,   UNTIL, 1 ?PAIRS
 HERE 1+ - C, ,   END, UNTIL, ,
   FIXUP DUP >R 1+ >R HERE R> - R> C! .
   AGAIN. 1 ?PAIRS JMP. ,   ELSE. 2 ?PAIRS 0 JMP, FIXUP HERE
 3 .   ENDIF, DUP 3 - IF 2 ?PAIRS FIXUP ELSE - 1+ HERE SWAP
 ! ENDIF .   THEN, ENDIF. , : WHILE, 1 ?PAIRS 1 IF. 1 - .
   REPEAT, 1 ?PAIRS SWAP 1 ?PAIRS SWAP JMP. FIXUP . -->
OK

SCR #4
 ( ASSEMBLER 2.4 )
 FORTH DEFINITIONS  CODE ?EXEC CREATE [COMPILE] ASSEMBLER
 !CSP ASSEMBLER INIT , IMMEDIATE  END-CODE CURRENT @ CONTEXT !
 ?EXEC ?CSP SMUDGE , IMMEDIATE  LABEL ?EXEC Ø VARIABLE SMUDGE
 -2 ALLOT [COMPILE] ASSEMBLER !CSP , IMMEDIATE ' ASSEMBLER CFA
 ' , CODE Ø8 + ! ASSEMBLER DEFINITIONS HEX
  MACRO ' ASSEMBLER 4 + CURRENT @ - IF CR  " ASSEMBLY ERROR "
 CR QUIT ELSE [COMPILE]  ENDIF ,
  BOT Ø ,X ,  SEC BOT 2+ ,
  NOT 2Ø HERE 1 - +! ,
 444 CONSTANT NEXT 43D CONSTANT PUSH 43F CONSTANT PUT
 46C CONSTANT SETUP 554 CONSTANT POPTWO
 556 CONSTANT POP 744 CONSTANT PUSHØA
 A6 CONSTANT N B5 CONSTANT XSAVE AE CONSTANT IP B3 CONSTANT UP
 B1 CONSTANT W
 ,S
OK
```

Note that the examples given in this text are designed to run under the implementation of fig-FORTH for the Microtan. They are however sufficiently general to make conversion easy.

All the special addresses which appear on screen number 4 are installation-dependent and correspond to labels used in the fig-FORTH 65Ø2 assembly source listing. They can all be omitted with the exception of 'NEXT' which is essential to return control to FORTH after a word has been executed.

ASSEMBLER

This is a structured, one-Pass assembler which supports labels and macros. All the usual 65Ø2 mnemonics are used (with the exception of the branch instructions which are treated slightly differently to fit in with the assembler's structuring). A fig-FORTH convention is that the

 0141–9331/83/040213–04 $03.00 © Butterworth & Co (Publishers) Ltd

mnemonics should be the usual three letters but followed by a comma (eg LDA,). Words created by the assembler can be mixed with other words in program development provided each word is complete in its own right (ie FORTH words and assembly language mnemonics cannot be used together in the same word). A typical assembly language word has the form·

CODE <name> <assembly code> END-CODE

Here, the word CODE selects the assembler vocabulary and saves the FORTH stack pointer for error checking later, and the name is any name of up to 31 characters as is usual for fig-FORTH.

The assembly code is the tricky bit. The normal FORTH conventions are followed – ie reverse Polish notation with all numbers assumed to be addresses unless followed by #. (A complete program is given later as an example). This section should normally finish with NEXT JMP, to return control to FORTH after the word is executed.

END-CODE performs similar functions to the semicolon in normal FORTH definitions such as checking control structures for completeness and making sure that the stack is at its original level.

Subroutines are catered for by using LABEL. This ensures that the word can be called from another assembly code definition by using the JSR, instruction. For this reason NEXT JMP, is replaced by RTS,. For example·

LABEL GETCH FDFA JSR, 1 LDA, RTS, END-CODE

Note that the assembler works in the current base   hex has been assumed here. The word GETCH, which polls the keyboard and returns a keypress in the accumulator, can now be called as follows:

CODE <nameXassembly code> GETCH JSR, <assembly code> END-CODE

The macro facility is used to define common segments of code to be inserted repeatedly in future definitions. A macro is *not* a subroutine - it produces in-line code whenever called, and for this reason can only be used when defining words in the assembler. A macro has the form:

MACRO GETCH FDFA JSR, 1 LDA, ;

and is used thus:

<assembly code> GETCH <assembly code>

This has the effect of actually creating FDFA JSR, 1 LDA, at the point of call.

Control structures are the normal FORTH ones but again followed by a comma. They are:

AGAIN, BEGIN, ELSE, END, ENDIF, IF, REPEAT, THEN, UNTIL, WHILE,

They are used in similar ways to normal control words, but combined with the following tests:

Ø<, Ø= , VS (overflow set), CS (carry set)

all of which can be followed by NOT if necessary. In this way all eight of the 65Ø2 branching conditions can be produced.

The various 65Ø2 addressing modes are catered for by using:

,Y ,)Y , X) ,) , ,X ,.A

Some examples:

FDFA JSR, is equivalent to JSR $FDFA

1 LDA, -------------------------------- LDA $ØØØ1

69) JMP, ------------------------------ JMP ($ØØ69)

70)Y STA, ------------------------------ STA ($ØØ7Ø),Y

71 #LDA, ------------------------------ LDA #ØØ71

72 ,X STA, ------------------------------ STA $ØØ72,X

.A ROL, ------------------------------ ROL A (accumulator)

Special addresses

FORTH has a number of subroutines which can be usefully incorporated into assembly definitions. Some of the more useful have been given names and included in the assembler'

BOT . . . . . address of low byte of a 16-bit stack item with ,X address mode. X register locates computation stack in zero-page, relative to address $ØØØØ;

SEC . . . . address of the second stack item as for BOT;

N . . . . . address of an eight-byte scratch area in zero-page;

UP . . . . . address of the User Pointer in zero-page.

IP . . . . . address of the Interpretive Pointer in zero-page;

W . . . . . address of the code field pointer in zero-page,

XSAVE . . . address of a temporary store for the X-register in zero-page. This register is used extensively by FORTH as a stack pointer so it is important that its contents should be restored after use;

PUT . . . . . address of routine to replace the present computation stack high byte from the accumulator; and put from the machine stack one byte which replaces the present low stack byte; continue on to NEXT,

PUSH . . . . . address of routine to repeat PUT, but creating a new bottom item on the computation stack,

PUSHØA, PUTØA . . . . . address of routine to place the accumulator at the low stack byte, with the high byte zero. PUTØA overwrites, whilst PUSHØA creates a new item;

POP, POPTWO . . . . . address of routine to remove one or two 16-bit items from the computation stack;

SETUP . . . . address of routine to move 16-bit items to zero-page. Item quantity is in accumulator;

NEXT . . . . address of the inner-interpreter, to which all code routines must return. NEXT fetches indirectly referred to IP the next compiled FORTH word address. It then jumps indirectly to the pointed machine code.

The assembler is strange to say the least, but if you follow a few simple rules you should be able to produce results fairly easily

1 Macros can only be declared if the *current* vocabulary is the *assembler*   ie prefix MACRO by ASSEMBLER DEFINITIONS. The reason for this apparently needless restriction is that subsequent calling of the word defined as a macro will insert in-line machine code at the point of call. This is clearly only acceptable in a machine code word.
2 End MACROs with a semi-colon ;.
3 Exit from LABEL definitions with RTS, .
4 Exit from CODE definitions with NEXT JMP, .

Having explained the necessary 'do's and 'don't's, I shall now take you through the development of a morse code generating program. Unlike BASIC, FORTH demands a degree of planning before any sizeable programs can be written. The technique I find most productive is top-down or stepwise refinement. The idea is to define the problem and then refine it bit by bit until the complete program emerges. The steps are as follows:

1 Get the computer to speak in morse code.
2 Write out an algorithm in as general a way as possible:

Read a letter.

Output it as morse.

Do it again until finished.

3 At this point I have decided on three main functions which we can go on to refine further. It is probably a good idea to decide on the most appropriate data structures now. I decided that two arrays were needed – one to hold the morse representations of the alphabet and the other for the text to be converted. The input text array is easy enough to code at this point – see lines 13 and 14 of screen number 1. The conversion table, however, will have to wait until I decide how to get the computer to make noises.

4 How are we going to stop reading characters for output? The word EXPECT in INDATA finishes putting characters in our DATA array when either 1ØØ characters have been typed in or a carriage return received. It then puts a couple of nulls at the end. We stop reading from DATA when we get a null. ?QUIT does this – line 2 screen number 4.

5 We can write a subroutine now which will read the next character – lines 11 and 12 screen number 4. I decided to use the Y register as an index into DATA, but as this register will probably be used elsewhere before we have finished, I have designated a byte in the zero-page scratch area as a temporary store.

6 We are now ready to work on making a noise. The simplest way of getting a sound from the computer is via the tape interface. We can write primitives now to produce dots and dashes. A dash is three times the length of a dot so we need some sort of delay routine. DAH, DIT, BEEP and DELAY are on screen number 2. DELAY is the usual nested decrement but its length can be varied by passing parameters in N. In a normal assembler the code would be:

```
LOOP:  DEC N + 1
       BNE LOOP
       DEC N
       BNE LOOP
```

BEEP produces a continuous tone through the tape interface. DIT and DAH call BEEP and they DELAY for the appropriate length of time before switching the tone off.

7 Now we come to the crunch. How do we convert an ASCII to a morse character? Morse characters vary in length from one to four dots and dashes so we would need at least four bits of a byte to represent any letter. This leaves four bits which could be used to hold a count of the number of elements in the character.

The algorithm involves using the three least significant bits of a byte to hold the element count and then using the most significant four bits to hold the elements. A 1 represents a dash and a Ø represents a dot. We access the various parts by ANDing with a suitable mask – line 2 on screen number 3. We then ASL the byte to get the next bit into the carry flag and branch to either DIT or DAH. This is repeated until the element count reaches zero. As an example consider 'A'. This letter is represented in morse by 'dot dash'. There are two elements, so the lower half of the byte representing it will contain the number 2. The upper half of the byte will contain Ø1ØØ in binary or 4 in hex. Our full byte for 'A' then is Ø1ØØØØ1Ø or 42 hex.

We have now defined all the major parts of the program in considerable detail. It only remains to iron out the bumps. I should like to point out the method I have used to fill the conversion table with constants – lines 8 to 12 of screen number 1. We need a temporary word to poke the values in but we do not need it again, so we declare the word, call it and immediately forget it.

There is plenty of room for further refinements of this program. For instance, I have written a small routine which will alter the speed of the morse from the default values – eg 2Ø WPM will produce morse at approximately 2Ø words per minute. You could write another routine which would vary the frequency of the tone produced by altering FREQ in screen number 1.

An obvious area for improvement is to refine the character recognition word to take in numbers as well as letters and to discard any not within limits. Numbers in morse are made up of five elements, so there is still room in one byte. At the moment, if the program comes across anything which is not a space or a letter, then unpredictable noises are produced.

Use the program by typing:

INDATA ⟨CR⟩ ⟨text, up to 1ØØ chars⟩ ⟨CR⟩
MORSE ⟨CR⟩

```
SCR #1
 Ø  ( MORSE 1 )
 1  FORTH DEFINITIONS DECIMAL
 2  2ØØ VARIABLE FREQ
 3
 4  Ø VARIABLE ONE 5Ø VARIABLE TWO
 5  15Ø VARIABLE THREE
 6  5Ø VARIABLE FOUR
 7
 8  Ø VARIABLE ALPHA 26 ALLOT
 9  HEX C4 B4 94 63 14 23 81 3 43 D4 64 E3 82 C2 44 A3 74
1Ø  2 4 C3 24 1 83 A4 84 42
11   FILL-ALPHA 1A Ø DO ALPHA I + C! LOOP ,
12  FILL-ALPHA FORGET FILL-ALPHA
13  Ø VARIABLE DATA 1Ø3 ALLOT
14   INDATA DATA 1ØØ EXPECT ,
15  - ->

SCR #2
 Ø  ( MORSE 2 )
 1  FORTH DEFINITIONS DECIMAL
 2  LABEL DELAY BEGIN, BEGIN, N 1+ DEC, Ø = UNTIL,
 3  N DEC, Ø = UNTIL, RTS, END-CODE
 4
 5  HEX
 6  LABEL BEEP Ø # LDA, BFCE STA, BØ #LDA, BFC2 STA,
 7  CØ # LDA, BFCB STA, FREQ LDA, BFC6 STA, FREQ 1+ LDA,
 8  BFC7 STA, BFC5 STA, RTS, END-CODE
 9
1Ø  LABEL DIT BEEP JSR, TWO LDA, N STA, DELAY JSR,
11  Ø # LDA, BFCB STA, BFC2 STA, RTS, END-CODE
12
13  LABEL DAH BEEP JSR,THREE LDA, N STA, DELAY JSR,
14  Ø # LDA, BFCB STA, BFC2 STA, RTS, END-CODE
15  - ->

SCR #3
 Ø  ( MORSE 3)
 1  LABEL    OUTMOR
 2           TAY, 7 #LDA, ALPHA ,Y AND,
 3           N 3 + STA, ALPHA, Y LDA, N 4 + STA,
 4           BEGIN,
 5              N 4 + ASL,
 6              CS IF, DAH JSR,
 7              ELSE, DIT JSR,
 8              ENDIF,
 9              TWO LDA, N STA,
1Ø              DELAY JSR,
11              N 3 + DEC,
12           Ø= UNTIL,
13           RTS,
14           END-CODE
15  -  ->
OK
```

```
SCR # 4
( MORSE 4 )
ASSEMBLER DEFINITIONS HEX
MACRO ?QUIT Ø= IF, NEXT JMP, ENDIF, ;

MACRO    ?SOUND
         2Ø # CMP,
         Ø= IF, FOUR LDA, N STA, DELAY JSR,
         ELSE, SEC, 41 #SBC, OUTMOR JSR,
         ENDIF,


LABEL    LETTER N 2+ LDY, N 2+ INC,
         DATA ,Y LDA, RTS, END-CODE



   >
```

```
SCR # 5
( MORSE 5 )
FORTH DEFINITIONS

CODE MORSE Ø # LDY, N STY, N 1+ STY, N 2+ STY,
       BEGIN,
         LETTER JSR,
         ?QUIT
         ?SOUND
         THREE LDA, N STA, DELAY JSR,
       AGAIN,
       END-CODE

 WPM A * 1ØØØ SWAP / DUP 3 * THREE !
       DUP 6 * FOUR ! TWO ! ,
 S
```