many the cassette interfaces are a high 

low, in the range of a kHz. The 
gain of the recorder is satisfactory in this range 
but not the relative phase shift between the two tones 
used. The read (and write) amplifier 
in the recorder delay the two fundamental tones 
by
This page is not part of the original Wonderbook
The Woz Wonderbook

DOCUMENT

Apple-II

Floating Point Package

This page is not part of the original Wonderbook
FLOATING POINT PACKAGE

The mantissa-exponent, or 'floating point', numerical representation is widely used by computers to express values with a wide dynamic range. With floating point representation, the number $7.5 \times 10^{22}$ requires no more memory to store than the number 75 does. We have allowed for binary floating point arithmetic on the APPLE-II computer by providing a useful subroutine package in ROM, which performs the common arithmetic functions. Maximum precision is retained by these routines and overflow conditions such as 'divide by zero' are trapped for the user. The 4-byte floating point number representation is compatible with future APPLE products such as floating point BASIC.

A small amount of memory in page zero is dedicated to the floating point workspace, including the two floating-point accumulators, FP1 and FP2. After placing operands in these accumulators, the user calls subroutines in the ROM which perform the desired arithmetic operations, leaving results in FP1. Should an overflow condition occur, a jump to location $3F5$ in RAM is executed, allowing a user routine to take appropriate action.

FLOATING POINT REPRESENTATION

![Floating Point Representation Diagram]

Exponent  Signed Mantissa
1. Mantissa

The floating point mantissa is stored in two's complement representation with the sign at the most significant bit (MSB) position of the high-order mantissa byte. The mantissa provides 24 bits of precision, including sign, and can represent 24-bit integers precisely. Extending precision is simply a matter of adding bytes at the low-order end of the mantissa.

Except for magnitudes less than $2^{-128}$ (which lose precision) mantissas are normalized by the floating point routines to retain maximum precision. That is, the numbers are adjusted so that the upper two high-order mantissa bits are unequal.

\[
\text{High-order Mantissa Byte} \\
\begin{array}{c}
01.\text{XXXXX} \quad \text{Positive mantissa.} \\
10.\text{XXXXX} \quad \text{Negative mantissa.} \\
00.\text{XXXXX} \quad \text{Unnormalized mantissa,} \\
11.\text{XXXXX} \quad \text{exponent = -128.}
\end{array}
\]

2. Exponent

The exponent is a binary scaling factor (power of two) which is applied to the mantissa. Ranging from -128 to +127, the exponent is stored in standard two's complement representation except for the sign bit which is complemented. This representation allows direct comparison of exponents since they are stored in increasing numerical sequence. The most negative exponent, corresponding to the smallest magnitude, -128, is stored as $00$ ($\$0$ means hexadecimal) and the most positive, +127, is stored as $\$FF$ (all ones).
<table>
<thead>
<tr>
<th>Exponent</th>
<th>Stored As</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>10000001  (81)</td>
</tr>
<tr>
<td>+2</td>
<td>10000010  (82)</td>
</tr>
<tr>
<td>+3</td>
<td>10000011  (83)</td>
</tr>
<tr>
<td>-1</td>
<td>01111111  (7F)</td>
</tr>
<tr>
<td>-2</td>
<td>01111110  (7E)</td>
</tr>
<tr>
<td>-3</td>
<td>01111101  (7D)</td>
</tr>
</tbody>
</table>

The smallest magnitude which can be represented is $2^{-150}$.

The largest positive magnitude which can be represented is $2^{128} - 1$. 

```plaintext
$7F$ $7F$ $FF$ $FF$
```
### Floating Point Representation Examples

<table>
<thead>
<tr>
<th>Decimal Number</th>
<th>Hex Exponent</th>
<th>Hex Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>+3</td>
<td>81</td>
<td>60 00 00 (1.1₂ x 2¹)</td>
</tr>
<tr>
<td>+4</td>
<td>82</td>
<td>40 00 00 (1.0₂ x 2²)</td>
</tr>
<tr>
<td>+5</td>
<td>82</td>
<td>50 00 00 (1.01₂ x 2²)</td>
</tr>
<tr>
<td>+7</td>
<td>82</td>
<td>70 00 00 (1.1₁₂ x 2²)</td>
</tr>
<tr>
<td>+12</td>
<td>83</td>
<td>60 00 00 (1.10₁₂ x 2³)</td>
</tr>
<tr>
<td>+15</td>
<td>83</td>
<td>78 00 00 (1.11₁₂ x 2³)</td>
</tr>
<tr>
<td>+17</td>
<td>84</td>
<td>44 00 00 (1.0001₂ x 2⁴)</td>
</tr>
<tr>
<td>+20</td>
<td>84</td>
<td>50 00 00 (1.0₁₂ x 2⁴)</td>
</tr>
<tr>
<td>+60</td>
<td>85</td>
<td>78 00 00 (1.11₁₂ x 2⁵)</td>
</tr>
<tr>
<td>-3</td>
<td>81</td>
<td>A0 00 00</td>
</tr>
<tr>
<td>-4</td>
<td>81</td>
<td>80 00 00</td>
</tr>
<tr>
<td>-5</td>
<td>82</td>
<td>B0 00 00</td>
</tr>
<tr>
<td>-7</td>
<td>82</td>
<td>90 00 00</td>
</tr>
<tr>
<td>-12</td>
<td>83</td>
<td>A0 00 00</td>
</tr>
<tr>
<td>-15</td>
<td>83</td>
<td>88 00 00</td>
</tr>
<tr>
<td>-17</td>
<td>84</td>
<td>BC 00 00</td>
</tr>
<tr>
<td>-20</td>
<td>84</td>
<td>B0 00 00</td>
</tr>
<tr>
<td>-60</td>
<td>85</td>
<td>88 00 00</td>
</tr>
</tbody>
</table>
FLOATING POINT SUBROUTINE DESCRIPTIONS

FCOMPL subroutine (address $F4A4)

Purpose: FCOMPL is used to negate floating point numbers.

Entry: A normalized or unnormalized value is in FP1 (floating point accumulator 1).

Uses: NORM, RTLOG.

Exit: The value in FP1 is negated and then normalized to retain precision. The 3-byte FP1 extension, E, may also be altered but FP2 and SIGN are not disturbed. The 6502 A-REG is altered and the X-REG is cleared. The Y-REG is not disturbed.

Caution: Attempting to negate $2^{128}$ will result in an overflow since $2^{128}$ is not representable, and a jump to location $3F5$ will be executed, with the following contents in FP1.

FP1:

$0 \quad \$80 \quad 0 \quad 0$

X1  M1

Example: Prior to calling FCOMPL, FP1 contains +15.

FP1:

$\$83 \quad \$78 \quad 0 \quad 0 \quad (+15)$

X1  M1

After calling FCOMPL as a subroutine, FP1 contains -15.

FP1:

$\$83 \quad \$88 \quad 0 \quad 0 \quad (-15)$

X1  M1
FADD subroutine (address $F46E)
Purpose: To add two numbers in floating point form.
Entry: The two addends are in FP1 and FP2 respectively. For maximum precision, both should be normalized.
Uses: SWPALIGN, ADD, NORM, RTLOG.
Exit: The normalized sum is left in FP1. FP2 contains the addend of greatest magnitude. E is altered but SIGN is not.
The A-REG is altered and the X-REG is cleared. The Y-REG is not disturbed. The sum mantissa is truncated to 24 bits.
Caution: Overflow may result if the sum is less than $-2^{128}$ or greater than $2^{128} - 1$. If so, a jump to location $3F5$ is executed leaving 0 in X1, and twice the proper sum in the mantissa M1. The sign bit is left in the carry, 0 for positive, 1 for negative.

FP1: 

X1 M1

(FOR carry=0, true sum = +X.YYY... x $2^{128}$.)

Example: Prior to calling FADD, FP1 contains +12 and FP2 contains -5.

FP1: 

$S3$ $S60$ 0 0 (+12)

X1 M1

FP2: 

$S82$ $SB0$ 0 0 (-5)

X2 M2

After calling FADD, FP1 contains +7 (FP2 contains +12).

FP1: 

$S82$ $S70$ 0 0 (+7)

X1 M1
FSUB subroutine (address $F168)

Purpose: To subtract two floating point numbers.

Entry: The minuend is in FP1 and the subtrahend is in FP2. Both should be normalized to retain maximum precision prior to calling FSUB.

Uses: FCOMPL, ALGNSWP, FADD, ADD, NORM, RTLOG.

Exit: The normalized difference is in FP1 with the mantissa truncated to 24 bits. FP2 holds either the minuend or the negative subtrahend, whichever is of greater magnitude. E is altered but SIGN and SCR are not. The A-REG is altered and the X-REG is cleared. The Y-REG is not disturbed.

Cautions: An exit to location $3F5 is taken if the result is less than $-2^{128}$ or greater than $+2^{128}-1$, or if the subtrahend is $-2^{128}$.

Example: Prior to calling FSUB, FP1 contains +7 (minuend) and FP2 contains -5 (subtrahend).

\[
\begin{array}{c}
\text{FP1:} \\
\text{FP2:} \\
\end{array}
\begin{array}{cccc}
\$82 & \$70 & 0 & 0 \\
X1 & M1 \\
\$82 & \$80 & 0 & 0 \\
X2 & M2 \\
\end{array}
\]

After calling FSUB, FP1 contains +12 and FP2 contains +7.

\[
\begin{array}{c}
\text{FP1:} \\
\text{FP2:} \\
\end{array}
\begin{array}{cccc}
\$83 & \$60 & 0 & 0 \\
X1 & M1 \\
\end{array}
\]
FMUL subroutine (address $F48C)

Purpose: To multiply floating point numbers.

Entry: The multiplicand and multiplier must reside in FP1 and FP2 respectively. Both should be normalized prior to calling FMUL to retain maximum precision.

Uses: MD1, MD2, RTLOG1, ADD, MDEND.

Exit: The signed normalized floating point product is left in FP1. M1 is truncated to contain the 24 most significant mantissa bits (including sign). The absolute value of the multiplier mantissa (M2) is left in FP2. E, SIGN and SCR are altered. The A- and X-REGs are altered and the Y-REG contains $FF upon exit.

Cautions: An exit to location $3F5 is taken if the product is less than $-2^{128}$ or greater than $+2^{128}-1$.

Notes: FMUL will run faster if the absolute value of the multiplier mantissa contains fewer '1's than the absolute value of the multiplicand mantissa.

Example: Prior to calling FMUL, FP1 contains +12 and FP2 contains -5.

\[
\begin{align*}
\text{FP1:} & \quad \begin{array}{c|c|c}
\$83 & \$60 & 0 \\
X1 & M1 & 0 \\
\end{array} \quad (+12) \\
\text{FP2:} & \quad \begin{array}{c|c|c}
\$82 & \$80 & 0 \\
X2 & M2 & 0 \\
\end{array} \quad (-5)
\end{align*}
\]

After calling FMUL, FP1 contains -60 and FP2 contains +5.

\[
\begin{align*}
\text{FP1:} & \quad \begin{array}{c|c|c}
\$85 & \$88 & 0 \\
X1 & M1 & 0 \\
\end{array} \quad (-60) \\
\text{FP2:} & \quad \begin{array}{c|c|c}
\$82 & \$50 & 0 \\
X2 & M2 & 0 \\
\end{array} \quad (+5)
\end{align*}
\]
FDIV subroutine (address SP4B2)

Purpose: To perform division of floating point numbers.

Entry: The normalized dividend is in FP2 and the normalized divisor is in FP1.

Exit: The signed normalized floating point quotient is left in FP1. The mantissa (M1) is truncated to 24 bits. The 3-bit M1 extension (E) contains the absolute value of the divisor mantissa. MD2, SIGN, and SCR are altered. The A- and X-REGS are altered and the Y-REG is cleared.

Uses: MD1, MD2, MDEND.

Cautions: An exit to location $3F5 is taken if the quotient is less than $2^{128}$ or greater than $+2^{128}-1$.

Notes: MD2 contains the remainder mantissa (equivalent to the MOD function). The remainder exponent is the same as the quotient exponent, or 1 less if the dividend mantissa magnitude is less than the divisor mantissa magnitude.

Example: Prior to calling FDIV, FP1 contains -60 (dividend)
and FP2 contains +12 (divisor).

\[
\begin{align*}
FP1: & \quad [85] \quad [88] \quad 0 \quad 0 \quad (-60) \\
& \quad X1 \quad M1 \\
FP2: & \quad [83] \quad [60] \quad 0 \quad 0 \quad (+12) \\
& \quad X1 \quad M1
\end{align*}
\]

After calling FMUL, FP1 contains -5 and M2 contains 0.

\[
\begin{align*}
FP1: & \quad [82] \quad [80] \quad 0 \quad 0 \quad (-5) \\
& \quad X1 \quad M1
\end{align*}
\]
FLOAT subroutine (address $F451)

Purpose: To convert integers to floating point representation.

Entry: A signed (two's complement) 2-byte integer is stored in M1 (high-order byte) and M1+1 (low-order byte). M1+2 must be cleared by the user prior to entry.

Uses: NORM1.

Exit: The normalized floating point equivalent is left in FP1. E, FP2, SIGN, and SCR are not disturbed. The A-REG contains a copy of the high-order mantissa byte upon exit but the X- and Y-REGs are not disturbed. The carry is cleared.

Notes: To float a 1-byte integer, place it in M1+1 and clear M1 as well as M1+2 prior to calling FLOAT.

FLOAT takes approximately 3 msec. longer to convert zero to floating point form than other arguments. The user may check for zero prior to calling FLOAT and increase throughput.

* * LOW-ORDER INTEGER BYTE IN A-REG
* * HIGH-ORDER BYTE IN Y-REG
* *

85 FAXFLOAT STA M1+1
84 F9STY M1 INIT MANT1.
A0 00LDY #$0
84 F8STY M1+2
05 D9ORA M1 CHK BOTH BYTES
D0 03BNE TOFLOAT FOR ZERO.
85 F8STA X1 IF SO, CLR X1
60RTS AND RETURN.
4C 51 F4TOFLOAT JMP FLOAT ELSE FLOAT INTEGER.
(FLOAT continued)

Example: Float +274 ($0112 hex)

Calling sequence

A0 01 LDY #$01 HIGH-ORDER INTEGER BYTE
A9 12 LDA #$12 LOW-ORDER INTEGER BYTE
84 F9 STY M1
85 FA STA M1+1
A9 00 LDA #$00
85 F8 STA M1+2
20 51 F4 JSR FLOAT

Upon returning from FLOAT, FP1 contains the floating point representation of +274.

FP1: $88 $44 $80 0 (+274)
**FIX subroutine (address $F640)**

**Purpose:** To extract the integer portion of a floating point number with truncation (ENTER function).

**Entry:** A floating point value is in FP1. It need not be normalized.

**Uses:** RTAR.

**Exit:** The two-byte signed two's complement representation of the integer portion is left in M1 (high-order byte) and M1+1 (low-order byte). The floating point values +24.63 and -61.2 are converted to the integers +24 and -61 respectively. FP1 and E are altered but FP2, E, SIGN and SCR are not. The A- and X-REGs are altered but the Y-REG is not.

**Example:** The floating point value +274 is in FP1 prior to calling FIX.

```
FP1:  $88 | $44 | $80 | 0   (+274)
    X1  M1
```

After calling FIX, M1 (high-order byte) and M1+1 (low-order byte) contain the integer representation of +274 ($0112).

```
FP1:  $8E | $01 | $12 | 0
    X1  M1
```

**Note:** FP1 contains an unnormalized representation of +274 upon exit.
AUXILIARY SUBROUTINES.

NORM subroutine (address $F463)

Purpose: To normalize the value in FP1, thus insuring maximum precision.

Entry: A normalized or unnormalized value is in FP1.

Exit: The value in FP1 is normalized. A zero mantissa will exit with X1=0 ($2^{-128}$ exponent). If the exponent on exit is -128 (X1=0) then the mantissa (M1) is not necessarily normalized (with the two high-order mantissa bits unequal). E, FP2, SIGN, and SCR are not disturbed. The A-REG is disturbed but the X- and Y-REGs are not. The carry is set.

Example: FP1 contains +12 in unnormalized form (as $0.0111_2 \times 2^6$).

\[
\begin{array}{c}
\text{FP1} \\
\begin{array}{c}
\$86 \\
\$0C \\
\end{array} \\
\begin{array}{c}
\text{X1} \\
\text{M1} \\
\end{array} \\
\end{array} \\
(+12)
\]

Upon exit from NORM, FP1 contains +12 in normalized form (as $1.1_2 \times 2^3$).

\[
\begin{array}{c}
\text{FP1} \\
\begin{array}{c}
\$83 \\
\$60 \\
\end{array} \\
\begin{array}{c}
\text{X1} \\
\text{M1} \\
\end{array} \\
\end{array} \\
(+12)
\]

NORM1 subroutine (address $F455$)

Purpose: To normalize a floating point value in FP1 when it is known the exponent is not -128 (X1=0) upon entry.

Entry: An unnormalized number is in FP1. The exponent byte should not be 0 for normal use.

Exit: The normalized value is in FP1. E, FP2, SIGN, and SCR are not disturbed. The A-REG is altered but the X- and Y-REGs are not.
ADD subroutine (address $F425)

Purpose: To add the two mantissas (M1 and M2) as 3-byte integers.

Entry: Two mantissas are in M1 (through M1+2) and M2 (through M2+2). They should be aligned, that is with identical exponents, for use in the FADD and FSUB subroutines.

Exit: The 24-bit integer sum is in M1 (high-order byte in M1, low-order byte in M1+2). FP2, X1, E, SIGN, and SCR are not disturbed. The A-REG contains the high-order byte of the sum, the X-REG contains $FF, and the Y-REG is not altered. The carry is the '25th' sum bit.

Example: FP1 contains +5 and FP2 contains +7 prior to calling ADD.

\[
\begin{array}{c|c|c|c|c|c}
FP1 & $82$ & $50$ & 0 & 0 & (+ 5) \\
X1 & & & & & \\
M1 & & & & & \\
FP2 & $82$ & $70$ & 0 & 0 & (+ 7) \\
\end{array}
\]

Upon exit, M1 contains the overflow value for +12. Note that the sign bit is incorrect. This is taken care of with a call to the right shift routine.

\[
\begin{array}{c|c|c|c|c|c}
FP1 & $82$ & $C0$ & 0 & 0 & (+12) \\
\end{array}
\]
ABS\text{WAP} subroutine (address $F437$)

Purpose: To take the absolute value of FP1 and then swap FP1 with FP2. Note that two sequential calls to ABS\text{WAP} will take the absolute values of both FP1 and FP2 in preparation for a multiply or divide.

Entry: FP1 and FP2 contain floating point values.

Exit: The absolute value of the original FP1 contents are in FP2 and the original FP2 contents are in FP1. The least significant bit of SIGN is complemented if a negation takes place (if the original FP1 contents are negative), by means of an increment. SCR and E are used. The A-REG contains a copy of X2, the X-REG is cleared, and the Y-REG is not altered.

RTAR subroutine (address $F47D$)

Purpose: To shift M1 right one bit position while incrementing X1 to compensate for scale. This is roughly the opposite of the NORM subroutine.

Entry: A normalized or unnormalized floating point value is in FP1.

Exit: The 6-byte field MANT1 and E is shifted right one bit arithmetically and X1 is incremented by 1 to retain proper scale. The sign bit of MANT1 (MSB of M1) is unchanged. FP2, SIGN, and SCR are not disturbed. The A-REG contains the least significant byte of E (E+2), the X-REG is cleared, and the Y-REG is not disturbed.
RTAR subroutine (continued)

Caution: If X1 increments to 0 (overflows) then an exit to location $3F5 is taken, the 'A-REG contains the high-order MANT1 byte, M1, and X1 is cleared. FP2, SIGN, SCR, and the X- and Y-REG's are not disturbed.

Uses: RTLOG

Example: Prior to calling RTAR, FP1 contains the normalized value -7.

\[
\begin{array}{cc}
\text{FP1} & \text{Value} \\
\hline
\text{X1} & \text{M1} \\
\$83 & \$A0 \\
0 & 0 \\
\end{array}
\]

(-7)

After calling RTAR, FP1 contains the unnormalized value -7 (note that precision is lost off the low-order end of M1).

\[
\begin{array}{cc}
\text{FP1} & \text{Value} \\
\hline
\text{X1} & \text{M1} \\
\$84 & \$D0 \\
0 & 0 \\
\end{array}
\]

(-7)

Note: M1 sign bit is unchanged.
RTLOG subroutine (address SF480)

Purpose: To shift the 6-byte field MANT1 and E one bit to the right (toward the least significant bit). The 6502 carry bit is shifted into the high-order M1 bit. This is useful in correcting binary sum overflows.

Entry: A normalized or unnormalized floating point value is in FP1. The carry must be cleared or set by the user since it is shifted into the sign bit of M1.

Exit: Same as RTAR except that the sign bit of M1 is not preserved (it is set to the value of the carry bit on entry).

Caution: Same as RTAR.

Example: Prior to calling RTLOG, FP1 contains the normalized value -12 and the carry is clear.

\[
\begin{array}{cccc}
\text{FP1:} & $83 & $A0 & 0 & 0 \\
& X1 & M1 & & \\
\end{array}
\]

(-12)

After calling RTLOG, M1 is shifted one bit to the right and the sign bit is clear. X1 is incremented by 1.

\[
\begin{array}{cccc}
\text{FP1:} & $84 & $50 & 0 & 0 \\
& X1 & M1 & & \\
\end{array}
\]

(+20)

Note: The bit shifted off the end of MANT1 is rotated into the high order bit of the 3-byte extension E. The 3-byte E field is also shifted one bit to the right.
RTLOG1 subroutine (address $F484)

Purpose: To shift MANT1 and E right one bit without adjusting X1. This is used by the multiply loop. The carry is shifted into the sign bit of MANT1.

Entry: M1 and E contain a 6-byte unsigned field. E is the 3-byte low-order extension of MANT1.

Exit: Same as RTLOG except that X1 is not altered and an overflow exit cannot occur.

MD2 subroutine (address $F4E2)

Purpose: To clear the 3-byte MANT1 field for FMUL and FDIV, check for initial result exponent overflow (and underflow), and initialize the X-REG to $17 for loop counting.

Entry: The X-REG is cleared by the user since it is placed in the 3 bytes of MANT1. The A-REG contains the result of an exponent addition (FMUL) or subtraction (FDIV). The carry and sign status bits should be set according to this addition or subtraction for overflow and underflow determination.

Exit: The 3 bytes of M1 are cleared (or all set to the contents of the X-REG on entry) and the Y-REG is loaded with $17. The sign bit of the A-REG is complemented and a copy of the A-REG is stored in X1. FP2, SIGN, SCR, and the X-REG are not disturbed.

Uses: NORM.

Caution: Exponent overflow results in an exit to location $3F5. Exponent underflow results in an early return from the
MD2 subroutine (continued)

calling subroutine (FDIV or FMUL) with a floating point zero in FP1. Because MD2 pops a return address off the stack, it may only be called by another subroutine.
FLOATING POINT ROUTINES

1:49 P.M., 10/3/1977

1: """"""""""""""""""""
2: """"""""""""""""""
3: """"""""""""""""""
4: """"""""""""""""""
5: """"""""""""""""""
6: """"""""""""""""""
7: """"""""""""""""""
8: """"""""""""""""""
9: """"""""""""""""""
10: """"""""""""""""""
11: """"""""""""""""""
12: """"""""""""""""""
13: """"""""""""""""""
14: TITLE "FLOATING POINT ROUTINES"

15: SIGN EPI $F3
16: X2 EPI $F4
17: M2 EPI $F5
18: X1 EPI $F6
19: M1 EPI $F7
20: E EPI $F8
21: OVLOC EQU $F3F
22: ORG $F425

23: ADD CLC CLEAR CARRY.
24: LDX #$2 INUX FOR 3-BYTE ADD.
25: ADD1 LDA M1,X ADD A BYTE OF MAN12 TO MAN11.
26: STA M1, X
27: DEX INUX TO NEXT MORE SIGNIF. BYTE.
28: BPL ADD1 LOOP UNTIL DONE.
29: RTS RETURN
30: 60
31: MUL ASL SIGN CLEAR LSB OF SIGN.
32: JCR ABSWAP ABS VAL OF M1, THEN SWAP WITH MAN11 NEGATIVE?
33: BIT M1 NO, SWAP WITH MAN12 AND RETURN.
34: BPL ABSWAP1 MAN11 NEGATIVE?
35: JSR FCOMPL YES, COMPLEMENT IT.
36: INC SIGN INCR SIGN. COMPLEMENT LSB.
37: ABSWAP1 SEC SET CARRY FOR RETURN TO MUL/DIV.
38: LDX #$4 INDEX FOR 4-BYTE SWAP.
39: SWAP STY E-1,X SWAP A BYTE OF EXP/MAN11 WITH
40: LDA X1-1,X EXP/MAN12 AND LEAVE A COPY OF MAN11 IN E (3 BYTES). E+X US.
41: LDY X2-1,X
42: STY X1-1,X
43: STA X2-1,X
44: CA
45: BNE SWAP1 ADVANCE INUX TO NEXT BYTE.
46: RTS LOOP UNTIL DONE.
47: FLOAT LDA #$8E INIT EXP1 TO 14.
48: STA X1 THEN NORMALIZE TO FLOAT.
49: NORM1 LDA M1 HIGH-ORDER MAN11 BYTE.
50: CMP #$C0 UMPER TWO BITS UNEQUAL?
51: BMI RTS1 YES, RETURN WITH MAN11 NORMAL.
52: DEC X1 DECREMENT EXP1.
53: ASL M1+2
54: POL M1+1 SHIFT MAN1 (3 BYTES) LEFT.
<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>OPERAND</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>61:26 F9</td>
<td>55</td>
<td>ROL M1</td>
</tr>
<tr>
<td>A3: A5 F8</td>
<td>56</td>
<td>NORM</td>
</tr>
<tr>
<td>DO E6</td>
<td>57</td>
<td>LDA X1</td>
</tr>
<tr>
<td>60:60</td>
<td>58</td>
<td>RTS1</td>
</tr>
<tr>
<td>E8: 20 A4 F4</td>
<td>59</td>
<td>FSUB</td>
</tr>
<tr>
<td>B7: 20 7B F4</td>
<td>60</td>
<td>SWPALCN</td>
</tr>
<tr>
<td>61: A5 F4</td>
<td>61</td>
<td>FADD</td>
</tr>
<tr>
<td>70: C5 F8</td>
<td>62</td>
<td>CMP X1</td>
</tr>
<tr>
<td>72: D0 F7</td>
<td>63</td>
<td>BNE SW-PALGN</td>
</tr>
<tr>
<td>74: 20 25 F4</td>
<td>64</td>
<td>JSR ADD</td>
</tr>
<tr>
<td>77: 50 EA</td>
<td>65</td>
<td>ADDEND</td>
</tr>
<tr>
<td>79: 70 05</td>
<td>66</td>
<td>BVS RTLOG</td>
</tr>
<tr>
<td>17B: 90 C4</td>
<td>67</td>
<td>BCC SNAP</td>
</tr>
</tbody>
</table>

* ELSE SHIFT RIGHT ARITH. |

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>OPERAND</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>17D: A5 F9</td>
<td>68</td>
<td>RTAR</td>
</tr>
<tr>
<td>17F: 0A</td>
<td>70</td>
<td>LDA M1</td>
</tr>
<tr>
<td>180: E6 F8</td>
<td>71</td>
<td>ROR1</td>
</tr>
<tr>
<td>182: F0 75</td>
<td>72</td>
<td>INX</td>
</tr>
<tr>
<td>184: A2 FA</td>
<td>73</td>
<td>BNE ROR1</td>
</tr>
<tr>
<td>186: 76 HF</td>
<td>74</td>
<td>ROR E+3.X</td>
</tr>
<tr>
<td>188: E8</td>
<td>75</td>
<td>INX</td>
</tr>
<tr>
<td>189: D0 FB</td>
<td>76</td>
<td>BNL ROR1</td>
</tr>
<tr>
<td>18B: 60</td>
<td>77</td>
<td>RTS</td>
</tr>
<tr>
<td>18C: 20 32 F4</td>
<td>78</td>
<td>F_MUL</td>
</tr>
<tr>
<td>18F: 65 F8</td>
<td>79</td>
<td>JSR MD1</td>
</tr>
<tr>
<td>191: 20 E2 F4</td>
<td>80</td>
<td>ADDC</td>
</tr>
<tr>
<td>194: 18</td>
<td>81</td>
<td>JSR MD2</td>
</tr>
<tr>
<td>196: 20 84 F4</td>
<td>82</td>
<td>MUL1</td>
</tr>
<tr>
<td>199: 90 03</td>
<td>83</td>
<td>BCC MUL2</td>
</tr>
<tr>
<td>19A: 20 25 F4</td>
<td>84</td>
<td>JSR ADD</td>
</tr>
<tr>
<td>19D: 88</td>
<td>85</td>
<td>MUL2</td>
</tr>
<tr>
<td>19E: 10</td>
<td>86</td>
<td>DEY</td>
</tr>
<tr>
<td>1AE: 46 H3</td>
<td>87</td>
<td>MULE1</td>
</tr>
<tr>
<td>1A2: 90 BF</td>
<td>88</td>
<td>JSR MD1</td>
</tr>
<tr>
<td>1A4: 38</td>
<td>89</td>
<td>JSR MD2</td>
</tr>
<tr>
<td>1A5: A2 03</td>
<td>90</td>
<td>RTLOG1</td>
</tr>
<tr>
<td>1A7: A7 00</td>
<td>91</td>
<td>BCC RTLOG1</td>
</tr>
<tr>
<td>1A9: 9F F8</td>
<td>92</td>
<td>JSR ADD</td>
</tr>
<tr>
<td>1AB: 9F F8</td>
<td>93</td>
<td>BCC X1</td>
</tr>
<tr>
<td>1AD: 94</td>
<td>94</td>
<td>DEX</td>
</tr>
<tr>
<td>1AE: D0 F7</td>
<td>95</td>
<td>ENE RTLOG1</td>
</tr>
<tr>
<td>1B0: F0 C5</td>
<td>96</td>
<td>DEQ ADDEND</td>
</tr>
<tr>
<td>1B2: 20 32 F4</td>
<td>97</td>
<td>FDIV</td>
</tr>
<tr>
<td>1B5: E5 F8</td>
<td>98</td>
<td>SBC X1</td>
</tr>
<tr>
<td>1B7: 20 E2 F4</td>
<td>99</td>
<td>JCR MD2</td>
</tr>
<tr>
<td>1C1: 38</td>
<td>100</td>
<td>DIV1</td>
</tr>
<tr>
<td>1C5: 10</td>
<td>101</td>
<td>SEC</td>
</tr>
<tr>
<td>1C9: 38</td>
<td>102</td>
<td>LDX M2.1</td>
</tr>
<tr>
<td>1CE: 48</td>
<td>103</td>
<td>SEC E.1</td>
</tr>
<tr>
<td>1CF: 0A</td>
<td>104</td>
<td>DEX</td>
</tr>
<tr>
<td>1C3: 10</td>
<td>105</td>
<td>EPL DIV2</td>
</tr>
<tr>
<td>1C9: 08</td>
<td>106</td>
<td>INX E.1</td>
</tr>
<tr>
<td>1C9: 02</td>
<td>107</td>
<td>LDX ADD3</td>
</tr>
<tr>
<td>1CE: 0B</td>
<td>108</td>
<td>FLA</td>
</tr>
</tbody>
</table>

**FLOATING POINT ROUTINES**

---

Page 2


Distributed under the Creative Commons License on page 5

Page 0101 of 0213
### FLOATING POINT ROUTINES

#### 1.4 M., 10/3/1977

<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>F408</td>
<td>90 02 109</td>
<td>BCC DIV4</td>
<td>IF M2&lt;0 THEN DON'T RESTORE M2</td>
</tr>
<tr>
<td>F40A</td>
<td>85 F8 110</td>
<td>STA M2+3, X</td>
<td></td>
</tr>
<tr>
<td>F40C</td>
<td>2A 111</td>
<td>DIV4</td>
<td></td>
</tr>
<tr>
<td>F40D</td>
<td>00 13 112</td>
<td>INX</td>
<td></td>
</tr>
<tr>
<td>F40F</td>
<td>26 FB 113</td>
<td>ROL M2+3</td>
<td></td>
</tr>
<tr>
<td>F411</td>
<td>26 FA 114</td>
<td>ROL M1+1</td>
<td></td>
</tr>
<tr>
<td>F413</td>
<td>26 F9 115</td>
<td>ROL M1</td>
<td></td>
</tr>
<tr>
<td>F415</td>
<td>06 F7 116</td>
<td>ASL M2+2</td>
<td></td>
</tr>
<tr>
<td>F417</td>
<td>26 F6 117</td>
<td>ROL M2+1</td>
<td></td>
</tr>
<tr>
<td>F419</td>
<td>26 F5 118</td>
<td>ROL M2</td>
<td></td>
</tr>
<tr>
<td>F41B</td>
<td>60 1C 119</td>
<td>BCS OVFL</td>
<td></td>
</tr>
<tr>
<td>F41D</td>
<td>88 120</td>
<td>DEY</td>
<td></td>
</tr>
<tr>
<td>F41F</td>
<td>D0 DA 121</td>
<td>BNL DIV1</td>
<td></td>
</tr>
<tr>
<td>F420</td>
<td>F0 BE 122</td>
<td>BEQ MUEND</td>
<td></td>
</tr>
<tr>
<td>F422</td>
<td>86 FB 123 MU2</td>
<td>STX M1+2</td>
<td></td>
</tr>
<tr>
<td>F424</td>
<td>86 FA 124</td>
<td>STX M1</td>
<td></td>
</tr>
<tr>
<td>F426</td>
<td>86 F8 125</td>
<td>STX M1</td>
<td></td>
</tr>
<tr>
<td>F428</td>
<td>60 0D 126</td>
<td>BCS OVCHK</td>
<td></td>
</tr>
<tr>
<td>F42A</td>
<td>30 04 127</td>
<td>BMI M3</td>
<td></td>
</tr>
<tr>
<td>F42C</td>
<td>68 128</td>
<td>PLA</td>
<td></td>
</tr>
<tr>
<td>F42E</td>
<td>68 129</td>
<td>PLA</td>
<td></td>
</tr>
<tr>
<td>F430</td>
<td>90 B2 130</td>
<td>BCC NORMX</td>
<td></td>
</tr>
<tr>
<td>F432</td>
<td>49 30 131 MU3</td>
<td>EOR #80</td>
<td></td>
</tr>
<tr>
<td>F434</td>
<td>85 F8 132</td>
<td>STA X1</td>
<td></td>
</tr>
<tr>
<td>F436</td>
<td>85 F9 133</td>
<td>LDY #17</td>
<td></td>
</tr>
<tr>
<td>F438</td>
<td>A0 17 134</td>
<td>BPL MD3</td>
<td></td>
</tr>
<tr>
<td>F43A</td>
<td>0F 135</td>
<td>OVCHK</td>
<td></td>
</tr>
<tr>
<td>F43C</td>
<td>4C F5 136</td>
<td>JMP OVLOC</td>
<td></td>
</tr>
<tr>
<td>F43E</td>
<td>0F 137</td>
<td>ORG $63D</td>
<td></td>
</tr>
<tr>
<td>F440</td>
<td>A5 F3 139</td>
<td>FIX</td>
<td></td>
</tr>
<tr>
<td>F442</td>
<td>F0 140</td>
<td>BPL UNDFL</td>
<td></td>
</tr>
<tr>
<td>F444</td>
<td>09 BE 141</td>
<td>CMP #8E</td>
<td></td>
</tr>
<tr>
<td>F446</td>
<td>00 142</td>
<td>BNE FIX1</td>
<td></td>
</tr>
<tr>
<td>F448</td>
<td>24 F9 143</td>
<td>BIT M1</td>
<td></td>
</tr>
<tr>
<td>F44A</td>
<td>0A 144</td>
<td>BPL FIXRTS</td>
<td></td>
</tr>
<tr>
<td>F44C</td>
<td>A5 FB 145</td>
<td>LDA M1+2</td>
<td></td>
</tr>
<tr>
<td>F44E</td>
<td>F0 146</td>
<td>BEQ FIXRTS</td>
<td></td>
</tr>
<tr>
<td>F450</td>
<td>E6 1A 147</td>
<td>INC M1+1</td>
<td></td>
</tr>
<tr>
<td>F452</td>
<td>D0 148</td>
<td>BNE FIXRTS</td>
<td></td>
</tr>
<tr>
<td>F454</td>
<td>E6 F9 149</td>
<td>INC M1</td>
<td></td>
</tr>
<tr>
<td>F456</td>
<td>60 150</td>
<td>FIXRTS</td>
<td></td>
</tr>
<tr>
<td>F458</td>
<td>A9 00 151</td>
<td>UNDFL</td>
<td></td>
</tr>
<tr>
<td>F45A</td>
<td>85 F4 152</td>
<td>STA M1</td>
<td></td>
</tr>
<tr>
<td>F45B</td>
<td>85 FA 153</td>
<td>STA M1+1</td>
<td></td>
</tr>
<tr>
<td>F45D</td>
<td>60 154</td>
<td>RTS</td>
<td></td>
</tr>
</tbody>
</table>

---

*R:******SUCCESSFUL ASSEMBLY: NO ERRORS*
CRUSS-REFERENCE: FLOATING POINT ROUTINES

ABS WAP  F437  0032
ADDRWAP  F440  0034
ADD    F425  0064  0084
ADD1   F429  0029
ADDLND F477  0096
ALIGNSWP F47B  0060
COMPL1 F4A7  0095
DIV1    F4EA  0121
DIV2    F4BD  0106
DIV3    F4C7  0112
DIV4    F4CC  0109
L      00FC(Z)  0039  0074  0103
ADD    F46E
ADDPL  F4A4  0035  0059
DIV    F4B2
FIX    F640
FIX1   F6D3  0142
FIXRTS F656  0144  0146  0148
FLOAT  F451
FMUL   F48C
FSUB   F468
M1     00F9(Z)  0025  0027  0033  0049  0053  0054  0055  0069  0113  0114  0115
       0123  0124  0125  0143  0145  0147  0149  0152  0153
M2     00F5(Z)  0026  0102  0110  0116  0117  0118
MD1    F432  0078  0097
MD2    F4E2  0080  0099
MD3    F4F0  0127  0135
MDLND  F4A0  0122
MUL1   F495  0006
MUL2   F49D  0083
NORM   F463  0065  0088
NORM1  F455  0057
NORMX  F4A2  0130
OVCHK  F4F7  0126
OVFL   F4F9  0072  0119
OVLOC  03F5  0136
RVN1   F436  0076
RTAR   F47D  0138
RTLOG  F480  0066
RTLOG1 F434  0082
RTS1   F467  0051
SIN    00F3(Z)  0031  0036  0087
SWAP   F441  0067
SWAP1  F443  0045
SWPULON F46B  0063
UNFL   F557  0140
X1     00F8(Z)  0040  0042  0048  0052  0056  0062  0071  0079  0092  0093  0098
       0132  0139
X2     00F4  0041  0043  0061
This page is not part of the original Wonderbook
The Woz Wonderbook

DOCUMENT

Apple-II

Sweet-16 -- The 6502 Dream Machine

This page is not part of the original Wonderbook
This page is not part of the original Wonderbook
SWEET16 – THE 6502 DREAM MACHINE

While writing APPLE BASIC for a 6502 microprocessor I repeatedly encountered a variant of MURPHY'S LAW. Briefly stated, any routine operating on 16-bit data will require at least twice the code that it should. Programs making extensive use of 16-bit pointers (such as compilers, editors, and assemblers) are included in this category. In my case, even the addition of a few double-byte instructions to the 6502 would have only slightly alleviated the problem. What I really needed was a 6502/RCA 1800 hybrid – a powerful 8-bit data handler complemented by an easy to use processor with an abundance of 16-bit registers and excellent pointer capability. My solution was to implement a non-existent (meta) 16-bit processor in software, interpreter style, which I call SWEET16.

SWEET16 is based around sixteen 16-bit registers (R0-R15), actually 32 memory locations. R0 doubles as the SWEET16 accumulator (ACC), R15 as the program counter (PC), and R14 as the status register. R13 holds compare instruction results and R12 is the subroutine return stack pointer if SWEET16 subroutines are used. All other SWEET16 registers are at the user's unrestricted disposal.

SWEET16 instructions fall into register and non-register categories. The register ops specify one of the sixteen registers to be used as either a data element or a pointer to
data in memory depending on the specific instruction. For example, INR R5 uses R5 as data and ST @R7 uses R7 as a pointer to data in memory. Except for the SET instruction, register ops take 1 byte of code each. The non-register ops are primarily 6502 style branches with the second byte specifying a ±127 byte displacement relative to the address of the following instruction. Providing that the prior register op result meets a specified branch condition, the displacement is added to SWEET16's PC, effecting a branch.

SWEET16 is intended as a 6502 enhancement package, not a stand-alone processor. A 6502 program switches to SWEET16 mode with a subroutine call and subsequent code is interpreted as SWEET16 instructions. The non-register op RTN returns the user program to 6502 mode after restoring the internal register contents (A, X, Y, P, and S). The following example illustrates how to use SWEET16.

300 B9 00 02 LDA IN,Y Get a char.
303 C9 CD CMP "M" "M" for move?
305 D0 09 BNE NOMOVE No, skip move.
307 20 89 F6 JSR SW16 Yes, call SWEET16.
30A 41 M_LOOP LD @R1 R1 holds source address.
30B 52 ST @R2 R2 holds dest. address.
30C F3 DCR R3 Decrement length.
30D 07 FB BNZ M_LOOP Loop until done.
30F 00 RTN Return to 6502 mode.
310 C9 C5 NOMOVE CMP "E" "E" char?
312 D0 13 BEQ EXIT Yes, exit.
314 C8 INY No, continue.

NOTE: Registers A, X, Y, P, and S are not disturbed by SWEET16.
INSTRUCTION DESCRIPTIONS

The SWEET16 opcode list is short and uncomplicated. Excepting relative branch displacements, hand assembly is trivial. All register opcodes are formed by combining two hex digits, one for the opcode and one to specify a register. For example, opcodes 15 and 45 both specify register R5 while codes 23, 27 and 29 are all ST ops. Most register ops are assigned in complementary pairs to facilitate remembering them. Thus LD and ST are opcodes 2n and 3n respectively, while LD @ and ST @ are codes 4n and 5n.

Opcodes 0 to C (hex) are assigned to the thirteen non-register ops. Except for RTN (opcode 0), BK (0A), and RS (B), the non-register ops are 6502 style relative branches. The second byte of a branch instruction contains a ±127 byte displacement value (in two's complement form) relative to the address of the instruction immediately following the branch. If a specified branch condition is met by the prior register op result, the displacement is added to the PC effecting a branch. Except for BR (Branch always) and BS (Branch to Sub-routine), the branch opcodes are assigned in complementary pairs, rendering them easily remembered for hand coding. For example, Branch if Plus and Branch if Minus are opcodes 4 and 5 while Branch if Zero and Branch if NonZero are opcodes 6 and 7.
**SWEET16 Opcode Summary**

<table>
<thead>
<tr>
<th>Register Ops</th>
<th>Non-register Ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>1n SET Rn, Constant (Set)</td>
<td>00 RTN (Return to 6502 mode)</td>
</tr>
<tr>
<td>2n LD Rn (Load)</td>
<td>01 BR ea (Branch always)</td>
</tr>
<tr>
<td>3n ST Rn (Store)</td>
<td>02 BNC ea (Branch if No Carry)</td>
</tr>
<tr>
<td>4n LD @Rn (Load indirect)</td>
<td>03 BC ea (Branch if Carry)</td>
</tr>
<tr>
<td>5n ST @Rn (Store indirect)</td>
<td>04 BP ea (Branch if Plus)</td>
</tr>
<tr>
<td>6n LDD @Rn (Load double indirect)</td>
<td>05 BM ea (Branch if Minus)</td>
</tr>
<tr>
<td>7n STD @Rn (Store double indirect)</td>
<td>06 BZ ea (Branch if Zero)</td>
</tr>
<tr>
<td>8n POP @Rn (Pop indirect)</td>
<td>07 BNZ ea (Branch if NonZero)</td>
</tr>
<tr>
<td>9n STP @Rn (Store pop indirect)</td>
<td>08 BM1 ea (Branch if Minus 1)</td>
</tr>
<tr>
<td>An ADD Rn (Add)</td>
<td>09 BNML ea (Branch if Not Minus 1)</td>
</tr>
<tr>
<td>Bn SUB Rn (Sub)</td>
<td>0A BK (Break)</td>
</tr>
<tr>
<td>Cn POPD @Rn (Pop double indirect)</td>
<td>0B RS (Return from Subroutine)</td>
</tr>
<tr>
<td>Dn CPR Rn (Compare)</td>
<td>0C BS ea (Branch to Subroutine)</td>
</tr>
<tr>
<td>En INR Rn (Increment)</td>
<td>0D (Unassigned)</td>
</tr>
<tr>
<td>Fn DCR Rn (Decrement)</td>
<td>0E (Unassigned)</td>
</tr>
<tr>
<td></td>
<td>0F (Unassigned)</td>
</tr>
</tbody>
</table>
**REGISTER OPS**

**SET Rn, Constant**

\[
\begin{array}{c|c|c}
\text{low} & \text{high} & \text{constant} \\
\end{array}
\]

(Set)

The 2-byte constant is loaded into Rn (n = 0 to F, hex) and branch conditions set accordingly. The carry is cleared.

**Example**

15 34 A0  
SET R5, A034  
R5 now contains A034

**LD Rn**

\[
\begin{array}{c}
\text{2n} \\
\end{array}
\]

(Load)

The ACC (R0) is loaded from Rn and branch conditions set according to the data transferred. The carry is cleared and the contents of Rn are not disturbed.

**Example**

15 34 A0  
SET R5, A034  
24  
LD R5  
ACC now contains A034

**ST Rn**

\[
\begin{array}{c}
\text{3n} \\
\end{array}
\]

(Store)

The ACC is stored into Rn and branch conditions set according to the data transferred. The carry is cleared and the ACC contents are not disturbed.

**Example**

25  
LD R5  
Copy the contents  
36  
ST R6  
of R5 to R6.
LD @Rn

(Load indirect)
The low-order ACC byte is loaded from the memory location
whose address resides in Rn and the high-order ACC byte is
cleared. Branch conditions reflect the final ACC con-
ten's which will always be positive and never minus 1.
The carry is cleared. After the transfer, Rn is incre-
mented by 1.

Example
15 34 A0
45

SET R5,A034
LD @R5

ACC is loaded from
memory location A034
and R5 is incremented
to A035.

ST @Rn

(Store indirect)
The low-order ACC byte is stored into the memory location
whose address resides in Rn. Branch conditions reflect
the 2-byte ACC contents. The carry is cleared. After
the transfer, Rn is incremented by 1.

Example
15 34 A0
16 22 90
45
56

SET R5, A034
SET R6, 9022
LD @R5
ST @R6

Load pointers R5 and R6
with A034 and 9022.
Move a byte from location
A034 to location 9022.
Both pointers are
incremented.
LDD @Rn

\[ 6n \] (Load double-byte indirect)

The low order ACC byte is loaded from the memory location whose address resides in Rn and Rn is then incremented by 1. The high order ACC byte is loaded from the memory location whose address resides in the (incremented) Rn and Rn is again incremented by 1. Branch conditions reflect the final ACC contents. The carry is cleared.

Example

15 34 A0       SET R5, A034
65               LDD @R5

The low-order ACC byte is loaded from location A034, the high-order byte from location A035. R5 is incremented to A036.

STD @Rn

\[ 7n \] (Store double-byte indirect)

The low-order ACC byte is stored into the memory location whose address resides in Rn and Rn is then incremented by 1. The high-order ACC byte is stored into the memory location whose address resides in (the incremented) Rn and Rn is again incremented by 1. Branch conditions reflect the ACC contents which are not disturbed. The carry is cleared.

Example

15 34 A0       SET R5, A034
16 22 90       SET R6, 9022
65               LDD @R5
76               STD @R6

Load pointers R5 and R6 with A034 and 9022. Move double byte from locations A034 and A035 to locations 9022 and 9023. Both pointers are incremented by 2.
POP @Rn

(Pop indirect)

The low order ACC byte is loaded from the memory location whose address resides in Rn after Rn is decremented by 1 and the high order ACC byte is cleared. Branch conditions reflect the final 2-byte ACC contents which will always be positive and never minus 1. The carry is cleared. Because Rn is decremented prior to loading the ACC, single byte stacks may be implemented with the ST @Rn and POP @Rn ops (Rn is the stack pointer).

Example

15 34 A0    SET R5, A034    Init stack pointer.
10 04 00    SET R0, 4      Load 4 into ACC.
35          ST @R5         Push 4 onto stack.
10 05 00    SET R0, 5      Load 5 into ACC.
35          ST @R5         Push 5 onto stack.
10 06 00    SET R0, 6      Load 6 into ACC.
35          ST @R5         Push 6 onto stack.
85          POP @R5        Pop 6 off stack into ACC.
85          POP @R5        Pop 5 off stack.
85          POP @R5        Pop 4 off stack.
STP @Rn

\[ \text{9n} \]

(\text{STORE POP indirect})

The low order ACC byte is stored into the memory location whose address resides in Rn \textit{after} Rn is decremented by 1. Then the high-order ACC byte is stored into the memory location whose address resides in Rn after Rn is again decremented by 1. Branch conditions will reflect the 2-byte ACC contents which are not modified. STP @Rn and POP @Rn are used together to move data blocks beginning at the greatest address and working down. Additionally, single-byte stacks may be implemented with the STP @Rn and LDA @Rn ops.

Example

14 34 A0  \quad \text{SET R4, A034 Init pointers.}
15 22 90  \quad \text{SET R5, 9022}
84        \quad \text{POP @R4 Move byte from A033}
95        \quad \text{STP @R5 to 9021.}
84        \quad \text{POP @R4 Move byte from A032}
95        \quad \text{STP @R5 to 9020.}
ADD Rn

! An !

(Add)

The contents of Rn are added to the contents of the ACC (RO) and the low-order 16 bits of the sum restored in ACC. The 17th sum bit becomes the carry and other branch conditions reflect the final ACC contents.

Example

10 34 76
11 27 42
A1
A0

SET RO, 7634 Init RO (ACC)
SET R1, 4227 and R1.
ADD R1 Add R1 (sum = B85B, carry clear)
ADD RO Double ACC (RO) to 70B6 with carry set.
SUB Rn

Bn

(Subtract)

The contents of Rn are subtracted from the ACC contents by performing a two's complement addition:

\[ \text{ACC} \quad \text{ACC} + \overline{\text{Rn}} + 1 \]

The low order 16 bits of the subtraction are restored in the ACC. The 17th sum bit becomes the carry and other branch conditions reflect the final ACC contents. If the 16-bit unsigned ACC contents are greater than or equal to the 16-bit unsigned Rn contents then the carry is set, otherwise it is cleared. Rn is not disturbed.

Example

10 34 76 \hspace{1em} \text{SET} \hspace{0.5em} \text{R0, 7634} \hspace{1em} \text{Init R0 (ACC)}

11 27 42 \hspace{1em} \text{SET} \hspace{0.5em} \text{R1, 4227} \hspace{1em} \text{and R1.}

A1 \hspace{1em} \text{SUB} \hspace{0.5em} \text{R1} \hspace{1em} \text{Subtract R1 (diff = 340D with carry set)}

A0 \hspace{1em} \text{SUB} \hspace{0.5em} \text{R0} \hspace{1em} \text{Clears ACC (R0)}
POPDRn \[ \text{Cn} \] (POP Double-byte indirect)

Rn is decremented by 1 and the high-order ACC byte is loaded from the memory location whose address now resides in Rn. Then Rn is again decremented by 1 and the low-order ACC byte is loaded from the corresponding memory location. Branch conditions reflect the final ACC contents. The carry is cleared. Because Rn is decremented prior to loading each of the ACC halves, double-byte stacks may be implemented with the STD @Rn and POPD @Rn ops (Rn is the stack pointer).

Example

\begin{align*}
15 & \ 34 & \text{A0} & \text{SET R5, A034} & \text{Init stack pointer.} \\
10 & \ 12 & \text{AA} & \text{SET R0, AA12} & \text{Load AA12 into ACC.} \\
75 & \ & \ & \text{STD @R5} & \text{Push AA12 onto stack.} \\
10 & \ 34 & \text{BB} & \text{SET R0, BB34} & \text{Load BB34 into ACC.} \\
75 & \ & \ & \text{STD @R5} & \text{Push BB34 onto stack.} \\
10 & \ 56 & \text{CC} & \text{SET R0, CC56} & \text{Load CC56 into ACC.} \\
C5 & \ & \ & \text{POPD @R5} & \text{Pop CC56 off stack.} \\
C5 & \ & \ & \text{POPD @R5} & \text{Pop BB34 off stack.} \\
C5 & \ & \ & \text{POPD @R5} & \text{Pop AA12 off stack.}
\end{align*}
CPR Rn

The ACC (R0) contents are compared to Rn by performing the 16-bit binary subtraction ACC-Rn and storing the low order 16 difference bits in R13 for subsequent branch tests. If the 16-bit unsigned ACC contents are greater than or equal to the 16-bit unsigned Rn contents then the carry is set, otherwise it is cleared. No other registers, including ACC and Rn, are disturbed.

Example

15 34 A0  SET  R5, A034  Pointer to memory.
16 BF A0  SET  R6, A0BF  Limit address.
10 00 00  LOOP  SET  R0, 0  Zero data.
75       STD  @R5  Clear 2 locs, incr R5 by 2.
25       LD   R5  Compare pointer R5
d6       CPR  R6  to limit R6.
02 F8     BNC  LOOP  Loop if carry clear.
INR Rn

The contents of Rn are incremented by 1. The carry is cleared and other branch conditions reflect the incremented value.

Example

15 34 A0 SET R5, A034 Init R5 (pointer)
10 00 00 SET R0, 0 Zero to R0.
55 ST @R5 Clears loc A034 and incr R5 to A035.
E5 INR R5 Incr R5 to A036
55 ST @R5 Clears loc A036 (not A035)

DCR Rn

The contents of Rn are decremented by 1. The carry is cleared and other branch conditions reflect the decremented value.

Example (Clear 9 bytes beginning at loc A034)

15 34 A0 SET R5, A034 Init pointer.
14 09 00 SET R4, 9 Init count.
10 00 00 SET R0, 0 Zero ACC.
55 LOOP ST @R5 Clear a mem byte.
F4 DCR R4 Decr. count.
07 FC BNZ LOOP Loop until zero.
NON-REGISTER INSTRUCTIONS

RTN 00
(Return to 6502 mode)
Control is returned to the 6502 and program execution continues at the location immediately following the RTN instruction. The 6502 registers and status conditions are restored to their original contents (prior entering SWEET16 mode)

BR ea 01 d
(Branch Always)
An effective address (ea) is calculated by adding the signed displacement byte (d) to the PC. The PC contains the address of the instruction immediately following the BR, or the address of the BR op plus 2. The displacement is a signed twos complement value from -128 to +127. Branch conditions are not changed. Note that effective address calculation is identical to that for 6502 relative branches. The hex add and subtract features of the APPLE-II monitor may be used to calculate displacements.

\[
\begin{align*}
  d &= \$80 \quad ea = PC + 2 - 128 \\
  d &= \$81 \quad ea = PC + 2 - 127 \\
  d &= \$FF \quad ea = PC + 2 - 1 \\
  d &= \$00 \quad ea = PC + 2 + 0 \\
  d &= \$01 \quad ea = PC + 2 + 1 \\
  d &= \$7E \quad ea = PC + 2 + 126 \\
  d &= \$7F \quad ea = PC + 2 + 127
\end{align*}
\]

Example

\$300: 01 50 \quad BR \ $352
BNC ea

| 02 | d | (Branch if No Carry) |

A branch to the effective address is taken only if the carry is clear, otherwise execution resumes as normal with the next instruction. Branch conditions are not changed.

BC ea

| 03 | d | (Branch if Carry set) |

A branch is effected only if the carry is set. Branch conditions are not changed.

BP ea

| 04 | d | (Branch if Plus) |

A branch is effected only if the prior 'result' (or most recently transferred data) was positive. Branch conditions are not changed.

Example (Clear mem from loc. A034 to A03F)

| 15 34 A0 | SET R5, A034 Init pointer. |
| 14 3F A0 | SET R4, A03F Init limit. |
| 10 00 00 | LOOP SET R0, 0 |
| 55 | ST @R5 Clear mem byte, incr R5. |
| 24 | LD R4 Compare limit to pointer. |
| D5 | CPR R5 |
| 04 F8 | BP LOOP Loop until done. |

BM ea

| 05 | d | (Branch if Minus) |

A branch is effected only if the prior 'result' was minus (negative, MSB = 1). Branch conditions are not changed.
BZ ea  
06 d  
(Branch if Zero)

A branch is effected only if the prior 'result' was zero. Branch conditions are not changed.

BNZ ea  
07 d  
(Branch if NonZero)

A branch is effected only if the prior 'result' was non-zero. Branch conditions are not changed.

BM1 ea  
08 d  
(Branch if Minus 1)

A branch is effected only if the prior 'result' was minus 1 ($FFFF hex). Branch conditions are not changed.

BNM1 ea  
09 d  
(Branch if Not Minus 1)

A branch is effected only if the prior 'result' was not minus 1 ($FFFF hex). Branch conditions are not changed.

BRK  
0A  
(Break)

A 6502 BRK (break) instruction is executed. SWEET16 may be reentered nondestructively at SW16D after correcting the stack pointer to its value prior executing the BRK.
RS

(Return from SWEET16 Subroutine)
RS terminates execution of a SWEET16 subroutine and returns to the SWEET16 calling program which resumes execution (in SWEET16 mode). R12, which is the SWEET16 subroutine return stack pointer, is decremented twice. Branch conditions are not changed.

BS ea

(Branch to SWEET16 Subroutine)
A branch to the effective address \((PC + 2 + d)\) is taken and execution is resumed in SWEET16 mode. The current PC is pushed onto a 'SWEET16 subroutine return address' stack whose pointer is R12, and R12 is incremented by 2. The carry is cleared and branch conditions set to indicate the current ACC contents.

Example (Calling a 'memory move' subroutine to move A034-A03B to 3000-3007)

300: 15 34 A0 SET R5, A034 Init pointer 1.
303: 14 3B A0 SET R4, A03B Init limit 1.
306: 16 00 30 SET R6, 3000 Init pointer 2.
309: 0C 15 BS MOVE Call move subroutine.

320: 45 MOVE LD @R5 Move one
321: 56 ST @R6 byte.
322: 24 LD R4
323: D4 CPR R5 Test if done.
324: 04 FA BP MOVE Return.
326: 0B RS
THEORY OF OPERATION

SWEET16 execution mode begins with a subroutine call to SW16. The user must ensure that the 6502 is in hex mode upon entry. All 6502 registers are saved at this time, to be restored when a SWEET16 RTN instruction returns control to the 6502. If you can tolerate indefinite 6502 register contents upon exit, approximately 30 usec may be saved by entering at SW16 + 3. Because this might cause an inadvertent switch from hex to decimal mode, it is advisable to enter at SW16 the first time through.

After saving the 6502 registers, SWEET16 initializes its PC (R15) with the subroutine return address off the 6502 stack. SWEET16's PC points to the location preceding the next instruction to be executed. Following the subroutine call are 1-, 2-, and 3-byte SWEET16 instructions, stored in ascending memory locations like 6502 instructions. The main loop at SW16B repeatedly calls the 'execute instruction' routine at SW16C which examines one opcode for type and branches to the appropriate subroutine to execute it.

Subroutine SW16C increments the PC (R15) and fetches the next opcode which is either a register op of the form OP REG with OP between 1 and 15 or a non-register op of the form 0 OP with OP between 0 and 13. Assuming a register op, the register specification is doubled to account for the 2-byte SWEET16 registers and placed in the X-Reg for indexing. Then the instruction type is determined. Register ops place the doubled register specification in the high order byte of R14 indicating
the 'prior result register' to subsequent branch instructions.
Non-register ops treat the register specification (right-hand
half-byte) as their opcode, increment the SWEET16 PC to point
at the displacement byte of branch instructions, load the A-Reg
with the 'prior result register' index for branch condition
testing, and clear the Y-Reg.

WHEN IS AN RTS REALLY A JSR?

Each instruction type has a corresponding subroutine.
The subroutine entry points are stored in a table which is
directly indexed into by the opcode. By assigning all the
entries to a common page only a single byte of address need
be stored per routine. The 6502 indirect jump might have been
used as follows to transfer control to the appropriate subroutine.

LDA #ADRHi High-order address byte.
STA IND+1
LDA OPTBL,X Low-order byte.
STA IND
JMP (IND)

To save code the subroutine entry address (minus:1)
is pushed onto the stack, high-order byte first. A 6502 RTS
(ReTurn from Subroutine) is used to pop the address off the
stack and into the 6502 PC (after incrementing by 1). The
net result is that the desired subroutine is reached by executing
a subroutine return instruction!
OPCODE SUBROUTINES

The register op routines make use of the 6502 'zero page indexed by X' and 'indexed by X indirect' addressing modes to access the specified registers and indirect data. The 'result' of most register ops is left in the specified register and can be sensed by subsequent branch instructions since the register specification is saved in the high-order byte of R14. This specification is changed to indicate R0 (ACC) for ADD and SUB instructions and R13 for the CPR (compare) instruction.

Normally the high-order R14 byte holds the 'prior result register' index times 2 to account for the 2-byte SWEET16 registers and thus the LSB is zero. If ADD, SUB, or CPR instructions generate carries, then this index is incremented, setting the LSB.

The SET instruction increments the PC twice, picking up data bytes in the specified register. In accordance with 6502 convention, the low-order data byte precedes the high-order byte.

Most SWEET16 nonregister ops are relative branches. The corresponding subroutines determine whether or not the 'prior result' meets the specified branch condition and if so update the SWEET16 PC by adding the displacement value (-128 to +127 bytes).
The RTN op restores the 6502 register contents, pops the subroutine return stack and jumps indirect through the SWEET16 PC. This transfers control to the 6502 at the instruction immediately following the RTN instruction.

The BK op actually executes a 6502 break instruction (BRK), transferring control to the interrupt handler.

Any number of subroutine levels may be implemented within SWEET16 code via the BS (Branch to Subroutine) and RS (Return from Subroutine) instructions. The user must initialize and otherwise not disturb R12 if the SWEET16 subroutine capability is used since it is utilized as the automatic subroutine return stack pointer.

MEMORY ALLOCATION

The only storage that must be allocated for SWEET16 variables are 32 consecutive locations in page zero for the SWEET16 registers, four locations to save the 6502 register contents, and a few levels of the 6502 subroutine return address stack. If you don’t need to preserve the 6502 register contents, delete the SAVE and RESTORE subroutines and the corresponding subroutine calls. This will free the four page zero locations ASAV, XSAV, YSAV, and PSAV.

USER MODIFICATIONS

You may wish to add some of your own instructions to this implementation of SWEET16. If you use the unassigned opcodes $0E and $0F, remember that SWEET16 treats these as 2-byte instructions. You may wish to handle the break instruction as a SWEET16 call, saving two bytes of code each time you transfer into SWEET16.
mode. Or you may wish to use the SWEET16 BK (Break) op as a 'CHAROUT' call in the interrupt handler. You can perform absolute jumps within SWEET16 by loading teh ACC (R0) with the address you wish to jump to (minus 1) and executing a ST R15 instruction.
**SWEET16 INTERPRETER**

```
45 P.M., 10/3/1977

1  ***************************************************************************
2  * APPLE-II PSEUDO  *
3  * MACHINE INTERPRETER  *
4  * Copyright 1977  *
5  * APPLE COMPUTER INC  *
6  * All Rights Reserved  *
7  * S. WOZNIAK  *
8  *
9  ***************************************************************************
10  TITLE "SWEET16 INTERPRETER"
11
12
13  ROL  EPIZ  $0
14  ROH  EPIZ  $1
15  R14H  EPIZ  $1D
16  R15L  EPIZ  $1E
17  R15H  EPIZ  $1F
18  S16PAG  EQU  $F7
19  SAVE  EQU  $FF4A
20  RESTORE  EQU  $FF3F
21  ORG  $F889
22
23  SW16  JSR  SAVE
24
25  PLA
26  STA  R15L
27
28  PLA
29  STA  R15H
30
31  SW16B  JSR  SW16C
32  JMP  SW16B
33
34  INC  R15L
35
36  BNE  SW16D
37
38  INC  R15H
39
40  LDA  #516PAG
41
42  PHA
43
44  LDA  (#R15L), Y
45
46  INCR SWEET16 PC FOR FETCH
47
48  PUSH ON STACK FOR RTS
49
50  FEIX INSTR
51  MASK REG SPECIFICATION
52  DOUBLE FOR 2-BYTE REG'S
53  TO X-REG FOR INDEXING
54
55  NOW HAVE OPCODE
56  IF ZERO THEN NON-REG OP
57  INUCATE PRIOR RESULT REG'
58
59  OPCODE*2 TO LSB'S
60
61  TO Y-REG FOR INDEXING
62  LOW-ORDER ADDR BYTE
63
64  ONTO STACK
65
66  GOTO REG-OP ROUTINE
67
68  INCR PC
```
<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F86F</td>
<td>BD E4 F6 55 T00R2 LDA ERTBL X</td>
<td>LOW-ORDER ADDR BYTE</td>
<td></td>
</tr>
<tr>
<td>F86C</td>
<td>43 56</td>
<td>PHA</td>
<td>ON TOP STACK FOR NON-REG OP</td>
</tr>
<tr>
<td>F86C3</td>
<td>A5 0D 57</td>
<td>LDA R14H</td>
<td>'PRIOR RESULT REG' INDEX</td>
</tr>
<tr>
<td>F86C5</td>
<td>4A 58</td>
<td>LSR A</td>
<td>PREPARE CARRY FOR BC, ENC.</td>
</tr>
<tr>
<td>F86C6</td>
<td>60 59</td>
<td>RTS</td>
<td>GOTO NON-REG OP KOU1NL</td>
</tr>
<tr>
<td>F86C7</td>
<td>63 60</td>
<td>RTNZ PLA</td>
<td>POP RETURN ADDRESS</td>
</tr>
<tr>
<td>F86C8</td>
<td>63 61</td>
<td>PLA</td>
<td></td>
</tr>
<tr>
<td>F86C9</td>
<td>30 3F FF 62</td>
<td>USR RESTORE</td>
<td>RESTORE $502 REG CONTENTS</td>
</tr>
<tr>
<td>F86CC</td>
<td>6A 00 63</td>
<td>JMP (R15L)</td>
<td>RETURN TO $502 CODE VIA PC</td>
</tr>
<tr>
<td>F86CF</td>
<td>B1 1E 64</td>
<td>SETZ LDA (R15L), Y</td>
<td>HIGH-ORDER BYTE OF CONST</td>
</tr>
<tr>
<td>F86D1</td>
<td>95 05 65</td>
<td>STA RCH X</td>
<td></td>
</tr>
<tr>
<td>F86D3</td>
<td>88 66</td>
<td>DEY</td>
<td></td>
</tr>
<tr>
<td>F86D4</td>
<td>B1 1E 67</td>
<td>LDA (R15L), Y</td>
<td>LOW-ORDER BYTE OF CONSTANT</td>
</tr>
<tr>
<td>F86D6</td>
<td>95 00 68</td>
<td>STA ROL X</td>
<td></td>
</tr>
<tr>
<td>F86D8</td>
<td>93 69</td>
<td>TYA</td>
<td>Y-REG CONTAINS 1</td>
</tr>
<tr>
<td>F86D9</td>
<td>38 70</td>
<td>SEC</td>
<td></td>
</tr>
<tr>
<td>F86DA</td>
<td>65 1E 71</td>
<td>ADC R15L</td>
<td>ADD 2 TO PC</td>
</tr>
<tr>
<td>F86DC</td>
<td>85 1E 72</td>
<td>STA R15L</td>
<td></td>
</tr>
<tr>
<td>F86DE</td>
<td>90 02 73</td>
<td>BCC SET2</td>
<td></td>
</tr>
<tr>
<td>F86D0</td>
<td>E6 1F 74</td>
<td>INC R15H</td>
<td></td>
</tr>
<tr>
<td>F86D2</td>
<td>60 75</td>
<td>SET2 RTS</td>
<td></td>
</tr>
<tr>
<td>F86D3</td>
<td>02 76</td>
<td>OPTBL DFB SET-1</td>
<td>(1X)</td>
</tr>
<tr>
<td>F86D4</td>
<td>F9 77</td>
<td>ERTBL DFB RTN-1</td>
<td>(0)</td>
</tr>
<tr>
<td>F86D5</td>
<td>C4 78</td>
<td>DFB LD-1 DFB BR-1</td>
<td>(2X)</td>
</tr>
<tr>
<td>F86D6</td>
<td>9D 79</td>
<td>DFB BR-1 DFB ST-1</td>
<td>(1)</td>
</tr>
<tr>
<td>F86D7</td>
<td>01 80</td>
<td>DFB ST-1 DFB SC-1</td>
<td>(3X)</td>
</tr>
<tr>
<td>F86D8</td>
<td>9E 81</td>
<td>DFB SC-1 DFB D-1</td>
<td>(2)</td>
</tr>
<tr>
<td>F86D9</td>
<td>25 82</td>
<td>DFB D-1 DFB LDA-1</td>
<td>(4X)</td>
</tr>
<tr>
<td>F86DA</td>
<td>AF 83</td>
<td>DFB LDA-1 DFB BC-1</td>
<td>(3)</td>
</tr>
<tr>
<td>F86DB</td>
<td>16 84</td>
<td>DFB BC-1 DFB LDT-1</td>
<td>(4)</td>
</tr>
<tr>
<td>F86DC</td>
<td>B2 85</td>
<td>DFB LDT-1 DFB ER-1</td>
<td>(5X)</td>
</tr>
<tr>
<td>F86DD</td>
<td>47 86</td>
<td>DFB ER-1 DFB RT-1</td>
<td>(6X)</td>
</tr>
<tr>
<td>F86DE</td>
<td>B9 87</td>
<td>DFB RT-1 DFB ET-1</td>
<td>(7X)</td>
</tr>
<tr>
<td>F86DF</td>
<td>51 88</td>
<td>DFB ET-1 DFB ER-1</td>
<td>(5)</td>
</tr>
<tr>
<td>F86F0</td>
<td>C0 89</td>
<td>DFB ER-1 DFB DB-1</td>
<td>(6)</td>
</tr>
<tr>
<td>F86F1</td>
<td>2F 90</td>
<td>DFB DB-1 DFB DB-1</td>
<td>(7)</td>
</tr>
<tr>
<td>F86F2</td>
<td>C9 91</td>
<td>DFB DB-1 DFB DB-1</td>
<td>(8)</td>
</tr>
<tr>
<td>F86F3</td>
<td>5B 92</td>
<td>DFB DB-1 DFB DB-1</td>
<td>(9X)</td>
</tr>
<tr>
<td>F86F4</td>
<td>D2 93</td>
<td>DFB DB-1 DFB DB-1</td>
<td>(8)</td>
</tr>
<tr>
<td>F86F5</td>
<td>85 94</td>
<td>DFB DB-1 DFB DB-1</td>
<td>(AX)</td>
</tr>
<tr>
<td>F86F6</td>
<td>DD 95</td>
<td>DFB DB-1 DFB DB-1</td>
<td>(D)</td>
</tr>
<tr>
<td>F86F7</td>
<td>6E 96</td>
<td>DFB DB-1 DFB DB-1</td>
<td>(EX)</td>
</tr>
<tr>
<td>F86F8</td>
<td>05 97</td>
<td>DFB DB-1 DFB DB-1</td>
<td>(C)</td>
</tr>
<tr>
<td>F86F9</td>
<td>33 98</td>
<td>DFB DB-1 DFB DB-1</td>
<td>(CX)</td>
</tr>
<tr>
<td>F86FA</td>
<td>E8 99</td>
<td>DFB DB-1 DFB DB-1</td>
<td>(B)</td>
</tr>
<tr>
<td>F86FB</td>
<td>70 100</td>
<td>DFB DB-1 DFB DB-1</td>
<td>(DX)</td>
</tr>
<tr>
<td>F86FC</td>
<td>93 101</td>
<td>DFB DB-1 DFB DB-1</td>
<td>(C)</td>
</tr>
<tr>
<td>F86FD</td>
<td>1E 102</td>
<td>DFB DB-1 DFB DB-1</td>
<td>(EX)</td>
</tr>
<tr>
<td>F86FE</td>
<td>E7 103</td>
<td>DFB DB-1 DFB DB-1</td>
<td>(D)</td>
</tr>
<tr>
<td>F86FF</td>
<td>65 104</td>
<td>DFB DB-1 DFB DB-1</td>
<td>(FX)</td>
</tr>
<tr>
<td>F8700</td>
<td>E7 105</td>
<td>DFB DB-1 DFB DB-1</td>
<td>(E)</td>
</tr>
<tr>
<td>F8701</td>
<td>E7 106</td>
<td>DFB DB-1 DFB DB-1</td>
<td>(UNUSED)</td>
</tr>
<tr>
<td>F8702</td>
<td>E7 107</td>
<td>DFB DB-1 DFB DB-1</td>
<td>(F)</td>
</tr>
<tr>
<td>F8703</td>
<td>10 CA 108 SET 68L SETZ ALWAYS TAKEN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>Opcode</td>
<td>Decoded Instruction</td>
<td>Notes</td>
</tr>
<tr>
<td>---------</td>
<td>--------</td>
<td>---------------------</td>
<td>-------</td>
</tr>
<tr>
<td>F/05:</td>
<td>E5 00</td>
<td>109 LD</td>
<td></td>
</tr>
<tr>
<td>F/07:</td>
<td>85 00</td>
<td>110 EK</td>
<td>EQU -1</td>
</tr>
<tr>
<td>F/09:</td>
<td>85 01</td>
<td>112 STA ROL</td>
<td></td>
</tr>
<tr>
<td>F/0B:</td>
<td>85 01</td>
<td>113 STA RCH</td>
<td></td>
</tr>
<tr>
<td>F/0D:</td>
<td>60</td>
<td>114 RTS</td>
<td></td>
</tr>
<tr>
<td>F/0E:</td>
<td>A5 00</td>
<td>115 ST</td>
<td></td>
</tr>
<tr>
<td>F/10:</td>
<td>85 00</td>
<td>116 STA ROL</td>
<td></td>
</tr>
<tr>
<td>F/12:</td>
<td>A5 01</td>
<td>117 LDA RCH, X</td>
<td>MOVE RX TO RO</td>
</tr>
<tr>
<td>F/14:</td>
<td>85 01</td>
<td>118 STA RCH, X</td>
<td></td>
</tr>
<tr>
<td>F/16:</td>
<td>60</td>
<td>119 RTS</td>
<td></td>
</tr>
<tr>
<td>F/17:</td>
<td>A5 00</td>
<td>120 STAT</td>
<td></td>
</tr>
<tr>
<td>F/19:</td>
<td>81 00</td>
<td>121 STAT2</td>
<td></td>
</tr>
<tr>
<td>F/1B:</td>
<td>A0 00</td>
<td>122 LDX ROL (R, X)</td>
<td></td>
</tr>
<tr>
<td>F/1D:</td>
<td>84 1D</td>
<td>123 STAT3</td>
<td></td>
</tr>
<tr>
<td>F/1F:</td>
<td>F6 00</td>
<td>124 INK</td>
<td>INC ROL, X</td>
</tr>
<tr>
<td>F/21:</td>
<td>D0 02</td>
<td>125 BNE INK2</td>
<td></td>
</tr>
<tr>
<td>F/23:</td>
<td>F6 01</td>
<td>126 INC ROL, X</td>
<td></td>
</tr>
<tr>
<td>F/25:</td>
<td>60</td>
<td>127 RTS</td>
<td></td>
</tr>
<tr>
<td>F/26:</td>
<td>A1 00</td>
<td>128 LDAT</td>
<td></td>
</tr>
<tr>
<td>F/28:</td>
<td>85 00</td>
<td>129 STA ROL</td>
<td>LOAD INDIRECT (RX)</td>
</tr>
<tr>
<td>F/2A:</td>
<td>A0 00</td>
<td>130 LDI ROL</td>
<td>TO RO</td>
</tr>
<tr>
<td>F/2C:</td>
<td>84 01</td>
<td>131 SYT ROH</td>
<td></td>
</tr>
<tr>
<td>F/2E:</td>
<td>80 00</td>
<td>132 BEQ STAT3</td>
<td></td>
</tr>
<tr>
<td>F/30:</td>
<td>80 00</td>
<td>133 POP</td>
<td></td>
</tr>
<tr>
<td>F/32:</td>
<td>F0 06</td>
<td>134 BEQ POP2</td>
<td></td>
</tr>
<tr>
<td>F/34:</td>
<td>20 66 F7</td>
<td>135 POPD</td>
<td>JSR DCR</td>
</tr>
<tr>
<td>F/37:</td>
<td>A1 00</td>
<td>136 LDA ROL (R, X)</td>
<td>POP HIGH-ORDER BYTE @X</td>
</tr>
<tr>
<td>F/39:</td>
<td>A3</td>
<td>137 TAY</td>
<td>SAVE IN Y-REG</td>
</tr>
<tr>
<td>F/3A:</td>
<td>20 66 F7</td>
<td>138 POP3</td>
<td>JSR DCR</td>
</tr>
<tr>
<td>F/3C:</td>
<td>A1 00</td>
<td>139 LDA ROL (R, X)</td>
<td>INCR RX</td>
</tr>
<tr>
<td>F/3F:</td>
<td>85 00</td>
<td>140 STA ROL</td>
<td>LOW-ORDER BYTE</td>
</tr>
<tr>
<td>F/41:</td>
<td>84 01</td>
<td>141 STY ROH</td>
<td>TO RO</td>
</tr>
<tr>
<td>F/43:</td>
<td>A0 00</td>
<td>142 POP3</td>
<td></td>
</tr>
<tr>
<td>F/45:</td>
<td>84 1D</td>
<td>143 STY R14H</td>
<td></td>
</tr>
<tr>
<td>F/47:</td>
<td>60</td>
<td>144 RTS</td>
<td></td>
</tr>
<tr>
<td>F/48:</td>
<td>20 26 F7</td>
<td>145 LDDAT</td>
<td>JSR LEAT</td>
</tr>
<tr>
<td>F/4A:</td>
<td>A1 00</td>
<td>146 LDA ROL (R, X)</td>
<td>LOW BYTE TO RO, INCR RX</td>
</tr>
<tr>
<td>F/4C:</td>
<td>85 01</td>
<td>147 STA ROH</td>
<td>HIGH-ORDER BYTE TO RO</td>
</tr>
<tr>
<td>F/4F:</td>
<td>4C 1F F7</td>
<td>148 JMP INR</td>
<td></td>
</tr>
<tr>
<td>F/52:</td>
<td>20 17 F/</td>
<td>149 STUAT</td>
<td>JSR STAT</td>
</tr>
<tr>
<td>F/55:</td>
<td>A5 01</td>
<td>150 LDA ROL (R, X)</td>
<td>STORE INDIRECT LOW-ORDER</td>
</tr>
<tr>
<td>F/57:</td>
<td>81 00</td>
<td>151 LDA ROL (R, X)</td>
<td>STORE HIGH-ORDER BYTE.</td>
</tr>
<tr>
<td>F/59:</td>
<td>4C 1F F7</td>
<td>152 JMP INR</td>
<td>INC RX AND RETURN</td>
</tr>
<tr>
<td>F/5C:</td>
<td>20 66 F/</td>
<td>153 STPAT</td>
<td>JSR DCR</td>
</tr>
<tr>
<td>F/5F:</td>
<td>A5 00</td>
<td>154 LDA ROL</td>
<td>DEC RX</td>
</tr>
<tr>
<td>F/61:</td>
<td>81 00</td>
<td>155 STA ROL (R, X)</td>
<td>STORE RO LOW BYTE @RX</td>
</tr>
<tr>
<td>F/63:</td>
<td>4C 1F F7</td>
<td>156 JMP POP3</td>
<td></td>
</tr>
<tr>
<td>F/66:</td>
<td>E5 00</td>
<td>157 DCR</td>
<td>JMP ROL, X</td>
</tr>
<tr>
<td>F/68:</td>
<td>D0 02</td>
<td>158 BNE DCR2</td>
<td></td>
</tr>
<tr>
<td>F/6A:</td>
<td>D6 01</td>
<td>159 DEC ROH, X</td>
<td></td>
</tr>
<tr>
<td>F/6C:</td>
<td>D6 00</td>
<td>160 DCR2</td>
<td></td>
</tr>
<tr>
<td>F/6E:</td>
<td>60</td>
<td>161 RTS</td>
<td></td>
</tr>
<tr>
<td>F/6F:</td>
<td>A0 00</td>
<td>162 SUB</td>
<td></td>
</tr>
</tbody>
</table>
SWEET16 INTEGRATED

1 45 P.M., 10/5/77

F7/1: 38 163 SEC
F7/2: A5 00 164 LDA ROL
F7/4: F5 00 165 SBC ROL, X
F7/6: 99 00 00 166 STA ROL, Y
F7/7: A5 01 167 LDA RCH
F7/8: F5 01 168 SBC RCH, X
F7/9: 99 01 00 169 SUB2 STA RCH, Y
F7/10: 98 170 TYA
F7/11: 69 00 171 ADC #$0
F7/12: 85 1D 172 STA R14H
F7/13: 60 173 RTS
F7/14: A5 00 174 ADD LDA ROL
F7/15: 75 00 175 ADC ROL, X
F7/16: A5 01 176 STA ROL, X
F7/17: 75 01 177 1LDA RCH
F7/18: 75 01 178 ADC RCH, X
F7/19: A0 00 179 LDY #$0
F7/20: 10 01 180 BEQ SUB2
F7/21: 94 1E 181 BS LDA R15L
F7/22: 20 19 F7 182 JSR STAT2
F7/23: 99 1E 183 STA R15H
F7/24: 20 19 F7 184 JSR STAT2
F7/25: 18 185 BR CLC
F7/26: 80 0E 186 BNC ECS ENC2
F7/27: B1 01 187 BR1 LDA (R15L), Y
F7/28: 10 01 188 BR1 DISPLACEMENT BYTE
F7/29: A5 00 189 DEY
F7/30: A5 01 190 BR2 ADC R15L
F7/31: 99 01 191 STA R15L
F7/32: 98 192 TYA
F7/33: 65 1F 193 ADC R15H
F7/34: 85 1F 194 STA R15H
F7/35: 60 195 BNC2 RTS
F7/36: 80 EC 196 BC ECS
F7/37: 60 197 RTS
F7/38: OA 198 BP ASL A DOUBLE RESULT-REG INDEX
F7/39: AA 199 TAX TO X-REG FOR INDEXING
F7/40: B5 01 200 LDA RCH, X TEST FOR PLUS
F7/41: 10 05 201 EPL R1 BR1 BRANCH IF SO
F7/42: 60 202 RTS
F7/43: 02 203 EM R1 DOUBLE RESULT-REG INDEX
F7/44: AA 204 TAX
F7/45: B5 01 205 LDA RCH, X TEST FOR MINUS
F7/46: 30 E1 206 EMI R1
F7/47: 60 207 RTS
F7/48: 0A 208 BZ ASL A DOUBLE RESULT-REG INDEX
F7/49: AA 209 TAX
F7/50: B5 00 210 LDA ROL, X TEST FOR ZERO
F7/51: 55 01 211 ORA RCH, X (BOOTH BYTES)
F7/52: 0F D3 212 BEQ R1 BR1 BRANCH IF SO
F7/53: 60 213 RTS
F7/54: 0A 214 BNZ ASL A DOUBLE RESULT-REG INDEX
F7/55: AA 215 TAX
F7/56: B5 00 216 LDA ROL, X TEST FOR NONZERO

NOTE: REG = 13 H FOR CFR
LAST RESULT REG*2
CARRY TO LSB
RO:RX TO RO
RO:RX TO RO
RO FOR RESULT
NOTE: X-REG IS 12x2!
PUSH LOW PC BYTE VIA R12
PUSH HIGH-ORDER PC BYTE
NO CARRY TEST
DISPLACEMENT BYTE
ADD TO PC
DOUBLE RESULT-REG INDEX
DOUBLE RESULT-REG INDEX
DOUBLE RESULT-REG INDEX
DOUBLE RESULT-REG INDEX
DOUBLE RESULT-REG INDEX

Distributed under the Creative Commons License on page 5
1:45 P.M., 10/3/1977

1. SWEETIE INTERPRETER

2. DISTRIBUTED UNDER THE CREATIVE COMMONS LICENSE ON PAGE 5

PAGE: 5

F/VCE: 15 01 217
F/V0: D0 LF 218
F/V2: 60 219
F/V3: 0A 220
F/04: AA 221
F/V5: B5 00 222
F/V7: 35 01 223
F/V9: 49 FF 224
F/VB: F0 C4 225
F/VD: 60 226
F/VC: 0A 227
F/VF: AA 228
F/00: B5 00 229
F/02: 35 01 230
F/04: 49 FF 231
F/06: D0 89 232
F/08: 60 233
F/09: A2 18 234
F/0B: 20 66 F7 235
F/0E: A1 00 236
F/0F: 85 1F 237
F/12: 20 66 F7 238
F/15: A1 00 239
F/17: 85 1E 240
F/19: 60 241
F/1A: 4C C7 F6 242

RTN
JMP RTNZ

*******SUCCESSFUL ASSEMBLY: NO ERRORS
<table>
<thead>
<tr>
<th>CROSS-REFERENCE</th>
<th>SWEETIE INTERPRETER</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDR</td>
<td>F/36</td>
</tr>
<tr>
<td>BR</td>
<td>F/780</td>
</tr>
<tr>
<td>B'k</td>
<td>F/704</td>
</tr>
<tr>
<td>BM</td>
<td>F/75A</td>
</tr>
<tr>
<td>BM1</td>
<td>F/7D3</td>
</tr>
<tr>
<td>ENC</td>
<td>F/79F</td>
</tr>
<tr>
<td>ENC2</td>
<td>F/79F</td>
</tr>
<tr>
<td>BMN1</td>
<td>F/7DE</td>
</tr>
<tr>
<td>BMNZ</td>
<td>F/CA</td>
</tr>
<tr>
<td>BP</td>
<td>F/763</td>
</tr>
<tr>
<td>BR</td>
<td>F/79E</td>
</tr>
<tr>
<td>BR1</td>
<td>F/7A1</td>
</tr>
<tr>
<td>BR2</td>
<td>F/7A6</td>
</tr>
<tr>
<td>BRNBL</td>
<td>F/6E4</td>
</tr>
<tr>
<td>BS</td>
<td>F/794</td>
</tr>
<tr>
<td>B7</td>
<td>F/7C1</td>
</tr>
<tr>
<td>CMR</td>
<td>F/771</td>
</tr>
<tr>
<td>DCR</td>
<td>F/766</td>
</tr>
<tr>
<td>DCA</td>
<td>F/766</td>
</tr>
<tr>
<td>I1R</td>
<td>F/71F</td>
</tr>
<tr>
<td>I1K2</td>
<td>F/725</td>
</tr>
<tr>
<td>I2U</td>
<td>F/705</td>
</tr>
<tr>
<td>I2DAT</td>
<td>F/726</td>
</tr>
<tr>
<td>I2DAT</td>
<td>F/748</td>
</tr>
<tr>
<td>NUL3</td>
<td>F/7E8</td>
</tr>
<tr>
<td>OPT3</td>
<td>F/743</td>
</tr>
<tr>
<td>PUP3</td>
<td>F/7E7</td>
</tr>
<tr>
<td>PUPD</td>
<td>F/7E7</td>
</tr>
<tr>
<td>ROL1</td>
<td>0001(2)</td>
</tr>
<tr>
<td>ROL2</td>
<td>0000(2)</td>
</tr>
<tr>
<td>R14H</td>
<td>001D(2)</td>
</tr>
<tr>
<td>R15L1</td>
<td>001E</td>
</tr>
<tr>
<td>R15L2</td>
<td>001E</td>
</tr>
<tr>
<td>R15L3</td>
<td>001E</td>
</tr>
<tr>
<td>RESTORE</td>
<td>F/3F</td>
</tr>
<tr>
<td>RC</td>
<td>F/9E</td>
</tr>
<tr>
<td>RTN</td>
<td>F/FA</td>
</tr>
<tr>
<td>RTNZ</td>
<td>F/6C7</td>
</tr>
<tr>
<td>S16PA6</td>
<td>00F7</td>
</tr>
<tr>
<td>SAVE</td>
<td>F/4A</td>
</tr>
<tr>
<td>SET</td>
<td>F/703</td>
</tr>
<tr>
<td>SLT2</td>
<td>F/6E2</td>
</tr>
<tr>
<td>SEL2</td>
<td>F/6CF</td>
</tr>
<tr>
<td>S1</td>
<td>F/70E</td>
</tr>
<tr>
<td>STAT</td>
<td>F/717</td>
</tr>
<tr>
<td>STAT2</td>
<td>F/719</td>
</tr>
<tr>
<td>STAT3</td>
<td>F/71D</td>
</tr>
<tr>
<td>STAT4</td>
<td>F/71D</td>
</tr>
<tr>
<td>SHTAT</td>
<td>F/752</td>
</tr>
<tr>
<td>SHTAT</td>
<td>F/752</td>
</tr>
<tr>
<td>SUB</td>
<td>F/76F</td>
</tr>
<tr>
<td>SUB2</td>
<td>F/77D</td>
</tr>
<tr>
<td>SW16</td>
<td>F/639</td>
</tr>
<tr>
<td>SW16B</td>
<td>F/639</td>
</tr>
<tr>
<td>SW16C</td>
<td>F/639</td>
</tr>
<tr>
<td>SW16D</td>
<td>F/639</td>
</tr>
</tbody>
</table>

Distributed under the Creative Commons License on page 5
null
The Woz Wonderbook

DOCUMENT

Apple-II

6502 Code Relocation Program

14 November 1977

This page is not part of the original Wonderbook
A 6502

CODE RELLOCATION

PROGRAM

for the

APPLE - I I COMPUTER

S. Wozniak (WOZ)

November 14, 1977
APPLE-II MACHINE CODE RELOCATION PROGRAM

Quite frequently I have encountered situations calling for relocation of machine language (not BASIC) programs on my 6502-based APPLE-II computer. Relocation means that the new version must run properly from different memory locations than the original. Because of the relative branch instruction, certain small 6502 programs need simply be moved and not altered. Others require only minor hand modification, which is simplified on the APPLE-II by the built-in disassembler which pinpoints absolute memory reference instructions such as JMPs and JSRs. However, most of the situations which I have encountered involved rather lengthy programs containing multiple data segments interspersed with code. For example, I once spent over an hour to hand-relocate the 8K byte APPLE8II monitor and BASIC to run from RAM addresses and at least one error probably went by undetected. That relocation can now be accomplished in a couple minutes using the relocation program described herein.
The following situations call for program relocation:

(1) Two programs which were written to run in identical locations must now reside and run in memory concurrently.

(2) A program currently runs from ROM. In order to modify its operation experimentally, a version must be generated which runs from RAM (different addresses).

(3) A program currently running in RAM must be converted to run from EPROM or ROM addresses.

(4) A program currently running on a 16K machine must be relocated in order to run on a 4K machine. Furthermore, the relocation may have to be performed on the smaller machine.

(5) Due to memory mapping differences, a program running on an APPLE-I (or other 6502 based) computer falls into unusable address space on an APPLE-II (or other) computer.

(6) Due to operating system variable assignment differences either the page-zero or non-page-zero variable allocation for a specific program may have to be modified when moving the program from one make of computer to another.

(7) A program exists as several chunks strewn about memory which must be combined in a single, contiguous block.
(8) A program has outgrown the available memory space and must be relocated to a larger 'free' space.

(9) A program insertion or deletion requires a chunk of the program to move a few bytes up or down.

(10) On a whim, the user wishes to move a program.
PROGRAM MODEL

It is easy to visualize relocation as taking a program which resides and runs in a 'source block' of memory and creating a modified version in a 'destination block' which runs properly. This model dictates that the relocation must be performed in an environment in which the program may in fact reside in both blocks. In many cases, the relocation is being performed because this is impossible. For example, a program written to begin at location $400 on an APPLE-I ($ stands for hex) falls in the APPLE-II screen memory range. It must be loaded elsewhere on the APPLE-II prior to relocation.

A more versatile program model is as follows. A program or section of a program runs in a memory range termed the 'source block' and resides in a range termed the 'source segments'. Thus a program written to run at location $400 may reside at location $800. The program is to be relocated so that it will run in a range termed the 'destination block' although it will reside in a range termed 'destination segments' (not necessarily the same). Thus a program may be relocated such that it will run from location $D000 (a ROM address) yet reside beginning at location $C00 prior to being saved on tape or used to burn EPROMs (obviously, the relocated program cannot immediately reside at locations reserved for ROM). In some cases the source and destination segments may overlap.
BLOCKS AND SEGMENTS EXAMPLE

Location during Relocation

$800

Program runs from location $400 on APPLE-I

$B87

Relocation

$C00

Relocated version runs from location $D000 on APPLE-II

$F87

SOURCE BLOCK: $400-$787 DEST BLOCK: $D000-$D387
SOURCE SEGMENTS: $800-$B87 DEST SEGMENTS: $C00-$F87
THE RELOCATION ALGORITHM

(1) Set SOURCE PTR to beginning of source segment and DEST PTR to beginning of destination segment.

(2) Copy 3 bytes from source segment (using SOURCE PTR) to temp INST area.

(3) Determine instruction length from opcode (1, 2, or 3 byte).

(4) If two byte instruction with non-zero-page addressing mode (immediate or relative) then go to (7).

(5) If two byte instruction then clear 3rd byte so address field is 0-255 (zero page).

(6) If address field (2nd and 3rd bytes of INST area) falls within source block, then substitute
   ADR - SOURCE BLOCK BEGIN + DEST BLOCK BEGIN

(7) Move 'length' bytes from INST area to dest segment (using DEST PTR). Update SOURCE and DEST PTRs by length.

(8) If SOURCE PTR is less than or equal to SOURCE SEGMENT END then goto (2), else done.
DATA SEGMENTS

The problem with relocating a large program all at once is that data (tables, text, etc.) may be interspersed throughout the code. Thus data may be 'relocated' as though it were code or might cause some code not to be relocated due to boundary uncertainty introduced when the data takes on the multi-byte attribute of code. This problem is circumvented by considering the 'source segments' and 'destination segments' sections to contain both code and data segments.

CODE AND DATA SEGMENTS EXAMPLE

```
$800  Code Segment $800-$892
       Data Segment $893-$992
       Code Segment $993-$ABF
       Data Segment $AC0-$ACF
       Code Segment $ACF-$B87

$B87  
```

The source code segments are relocated to the 'destination segments' area and the source data segments are moved. Note that several commands will be necessary to accomplish the complete relocation.
USAGE

1. Load RELOC by hand or off tape into memory locations $3A6-$3FA. Note that locations $3FB-$3FF are not disturbed by tape load versions to insure that the APPLE-II interrupt vectors are not clobbered. The monitor user function YC (Control-Y) will now call RELOC as a subroutine at location $3F8.

2. Load the source program into the 'source segments' area of memory if it is not already there. Note that this need not be where the program normally runs.

3. Specify the source and destination block parameters, remembering that the blocks are the locations that the program normally runs from, not the locations occupied by the source and destination segments during the relocation. If only a portion of a program is to be relocated then that portion alone is specified as the block.

    * DEST BLOCK BEG < SOURCE BLOCK BEG . END YC *

Note that the syntax of this command closely resembles that of the MONITOR 'MOVE' command. The initial '*' is generated by the MONITOR, not typed by the user.
4. Move all data segments and relocate all code segments in sequential (increasing address) order.

First Segment (if CODE)
* DEST SEGMENT BEG < SOURCE SEGMENT BEG . END YC

First Segment (if DATA)
* DEST SEGMENT BEG < SOURCE SEGMENT BEG . END M

Subsequent segments (if CODE)
  *. SOURCE SEGMENT END YC  (Relocation)

Subsequent segments (if DATA)
  *. SOURCE SEGMENT END M  (Move)

Note that it is wise to prepare a list of segments (code and data) prior to relocation.

If the relocation is performed 'in place' (SOURCE and DEST SEGMENTS reside in identical locations) then the SOURCE SEGMENT BEG parameter may be ommitted from the first segment relocate (or move).
EXAMPLES

1. Straightforward Relocation

Program A resides and runs in locations $800-$97F. The relocated version will reside and run in locations $A00-$B7F.

<table>
<thead>
<tr>
<th>SOURCE SEGMENTS</th>
<th>DEST SEGMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$800→$800-$88F</td>
<td>$A00→$A00-$A8F</td>
</tr>
<tr>
<td>DATA $890-$8AF</td>
<td>DATA $A90-$AAAF</td>
</tr>
<tr>
<td>CODE $8B0-$90F</td>
<td>CODE $AB0-$B0F</td>
</tr>
<tr>
<td>DATA $910-$93F</td>
<td>DATA $B10-$B3F</td>
</tr>
<tr>
<td>CODE $940-$97F</td>
<td>CODE $B40-$B7F</td>
</tr>
</tbody>
</table>

SOURCE BLOCK $800-$97F
DEST BLOCK $A00-$B7F

SOURCE SEGMENTS $800-$97F
DEST SEGMENTS $A00-$B7F

(a) Load RELOC
(b) Define blocks
* A00 < 800 . 97F YC *
(c) Relocate first segment (code).
* A00 < 800 . 88F YC
(d) Move and relocate subsequent segments in order.

* 8AF M      (data)
* 90F YC     (code)
* 93F M      (data)
* 97F YC     (code)

Note that step (d) illustrates abbreviated versions of
the following commands:

* A90 < 890  . 8AF M      (data)
* ABO < 8B0  . 90F YC    (code)
* B10 < 910  . 93F M      (data)
* B40 < 940  . 97F YC    (code)