# 79-FORTH ROM for Apple II

Dr. C. H. Ting

**The Design Goals**

The main purpose of this project was to implement a FORTH system of the lowest possible cost, and to carry this exciting language to the large population of Apple II users. To lower the system cost, it is necessary to fully utilize all the existing resources inside the Apple II computer, without such expensive peripherals as the floppy disk drives. The design goals were thus set as follows:

- Use a stripped Apple II as the host
- Put the FORTH dictionary in 8K bytes of ROM
- Implement the 79-Standard with editor and assembler extensions
- Build a pseudo disk in RAM with cassette tape as the off-line storage medium

The result will be a FORTH computer in a small box, which can be operated standing-alone and has the capacity of expanding into many educational and professional applications.

**Development Tools**

I did not have very sophisticated tools to develop 6502 based microcomputer systems. The only tool was a HP 65000 Development System, which had a 6502 cross assembler in it. The only way to build a FORTH system was to assemble the 6502 assembly source program on this development system, burn the object codes into a set of PROMs, and insert the PROMs into the Apple for testing and debugging. Any uncovered bug would have to be fixed at the source level. However, I had both the Auto Start Monitor and the Old Monitor in the Apple. The latter was very useful in the debugging process because of its trace capability.

**Approach**

Because of the lack of good development tools, it would be very difficult trying to build a FORTH system from scratch towards the design goals. The best approach was to divide the project into two phases:

- Implement a fig-FORTH system using the 6502 fig-FORTH source listing; and
- Modify the fig-FORTH to meet the design goals.

It was extremely important to build a working fig-FORTH system, because the object codes can be checked out by comparing byte-by-byte with the source. This greatly eased the task of debugging. Once I had the fig-FORTH running, further modifications could be checked and debugged using the FORTH interpreter, which was much more convenient to use and test.

**Implementation**

I first keyed in the 6502 source codes, identical to the 6502 fig-FORTH Source Listing. Both the source codes and the assembled object codes were thoroughly checked out. After changing the terminal I/O routines and offseting the object codes to start at 6000H, the resulting object codes were burnt into 2716's and moved into the Apple on the Apple ROM Card. Using the Apple Monitor, I could move the dictionary from the PROM's into RAM area, starting at 6000H. Debugging in RAM was much easier than doing it in ROM. The tracing aids provided by the fig-FORTH was helpful. However, I found it was more convenient to replace the JMP W-1 instruction by BRK, which returned the system to the monitor. To continue execution, I just keyed in OBOG in the monitor, which jumped over to W-1 (address OBOH) and continued onto the next word.

*The result was four 2716 PROMs sitting neatly on a small PC board.*

Only minor errors were detected and fixed at this stage, because most errors were flushed out by byte comparison of object codes. Since the entire system was in RAM, errors were corrected immediately and more tests could be carried out before a new set of PROMs were burnt.

After the fig-FORTH was thoroughly bug free, I proceeded to the task of modifications. The first thing to do was to trim the fig tree, making room for the editor and the assembler. I deleted the name fields and the link fields of all the run-time codes and some system words which the users are not expected to use. All the disk words were also deleted because the final system would not have a disk. The pseudo disk was implemented by a simple redefinition of **BLOCK**:

```
: BLOCK  ( n —— addr )
  MAXBUF MOD B/BUF * FIRST + ;
```

It returns a RAM address of the desired block, from which data can be referred.

The second task was to make the FORTH system ROMmable. All variables were either eliminated or changed to user variables. The only impure words were the vocabulary words like **FORTH**. To make **FORTH**, **EDITOR**, and **ASSEMBLER** stay comfortably in ROM, a new defining word **ROM-VOCABULARY** ought to be used:

*Continued on next page*

## 79-FORTH ROM for Apple II (continuec

```
: ROM-VOCABULARY ( addr —— )
  CREATE , DOES> @ CURRENT ! ;
HEX 1062 ROM-VOCABULARY FORTH
    1068 ROM-VOCABULARY EDITOR
    106E ROM-VOCABULARY ASSEMBLER
```

The addresses specified above point to the RAM locations where the name field addresses of the last words defined in the respective vocabularies are stored. These RAM locations are to be initialized at boot-up to:

```
FORTHLINK: 1060H:  A081
                   EF3D
EDITRLINK: 1066H:  A081
                   E8DF
ASSEMLINK: 106CH:  A081
                   EE01
```

The original **VOCABULARY** remained in the dictionary for the purpose of creating new vocabularies in RAM by the user.

The editor was basically the fig-FORTH editor. However, the command structure was modified to that used by Brodie in his *Starting FORTH*. I hoped to use *Starting FORTH* as an instruction manual for this ROM FORTH system and a compatible editor would not do any harm. The only major departure from the *Starting FORTH* editor was the handling of null strings. Only one string buffer (**PAD**) was used here, while Brodie used two independent buffers for searching and inserting.

Bill Ragsdale's 6502 Assembler was included in this FORTH to let the user experiment at the code level.

Finally, the whole system was updated to the 79-Standard. Many words needed to have their names changed. A few new words were added, and a few words needed to be redefined. Bob Smith's *'79-FORTH Conversion* was most helpful in this phase.

**Result**

The result was four 2716 PROMs sitting neatly on a small PC board. When it was inserted into Slot 0 in an Apple II, it turned the Apple into a very powerful FORTH computer. In an Apple II with 48K bytes RAM, 24K are used as a pseudo disk which holds lots of programs. With some tricks like

```
-32 OFFSET !
```

one could even turn the 16K high graphics memory into a second disk, making the total disk memory 40K. The programs can be dumped to cassette tape for storage. By loading or dumping large chunks of memory from or to tape, the necessity of disk can be avoided while still having all the advantages of FORTH. The main dictionary, securely stored in PROMs, makes the system immune from frequent crashes during program development and testing. □

## A FORTH ASSEMBLER FOR THE 6502

by William F. Ragsdale

### INTRODUCTION

This article should further polarize the attitudes of those outside the growing community of FORTH users. Some will be fascinated by a label-less, macro-assembler whose source code is only 96 lines long! Others will be repelled by reverse Polish syntax and the absence of labels.

The author immodestly claims that this is the best FORTH assembler ever distributed. It is the only such assembler that detects all errors in op-code generation and conditional structuring. It is released to the public domain as a defense mechanism. Three good 6502 assemblers were submitted to the FORTH Interest Group but each had some lack. Rather than merge and edit for publication, I chose to publish mine with all the submitted features plus several more.

Imagine having an assembler in 1300 bytes of object code with:

1. User macros (like IF, UNTIL,) definable at any time.
2. Literal values expressed in any numeric base, alterable at any time.
3. Expressions using any resident computation capability.
4. Nested control structures without labels, with error control.
5. Assembler source itself in a portable high level language.

### OVERVIEW

Forth is provided with a machine language assembler to create execution procedures that would be time inefficient, if written as colon-definitions. It is intended that "code" be written similarly to high level, for clarity of expression. Functions may be written first in high-level, tested, and then re-coded into assembly, with a minimum of restructuring.

### THE ASSEMBLY PROCESS

Code assembly just consists of interpreting with the ASSEMBLER vocabulary as CONTEXT. Thus, each word in the input stream will be matched according the Forth practice of searching CONTEXT first then CURRENT.

```
ASSEMBLER (now CONTEXT)
FORTH     (chained to ASSEMBLER)
user's    (CURRENT if one exits)
FORTH     (chained to user's vocab)
try for literal number
else, do error abort
```

The above sequence is the usual action of Forth's text interpreter, which remains in control during assembly.

During assembly of CODE definitions, Forth continues interpretation of each word encountered in the input stream (not in the compile mode). These assembler words specify operands, address modes, and op-codes. At the conclusion of the CODE definition a final error check verifies correct completion by "unsmudging" the definition's name, to make it available for dictionary searches.

### RUN-TIME, ASSEMBLY-TIME

One must be careful to understand at what time a particular word definition executes. During assembly, each assembler word interpreted executes. Its function at that instant is called 'assembling' or 'assembly-time'. This function may involve op-code generation, address calculation, mode selection, etc.

The later execution of the generated code is called 'run-time'. This distinction is particulary important with the conditionals. At assembly time each such word (i.e., IF, UNTIL, BEGIN, etc.) itself 'runs' to produce machine code which will later execute at what is labeled 'run-time' when its named code definition is used.

### AN EXAMPLE

As a practical example, here's a simple call to the system monitor, via the NMI address vector (using the BRK opcode).

```
CODE MON ( exit to monitor )
    BRK, NEXT JMP, END-CODE
```

The word CODE is first encountered, and executed by Forth. CODE builds the following name "MON" into a dictionary header and calls ASSEMBLER as the CONTEXT vocabulary.

The "(" is next found in FORTH and executed to skip til ")". This method skips over comments. Note that the name after CODE and the ")" after "(" must be on the same text line.

### OP-CODES

BRK, is next found in the assembler as the op-code. When BRK, executes, it assembles the byte value 00 into the dictionary as the op-code for "break to monitor via "NMI".

Many assembler words names end in ",". The significance of this is:

1. The comma shows the conclusion of a logical grouping that would be one line of classical assembly source code.
2. "," compiles into the dictionary; thus a comma implies the point at which code is generated.
3. The "," distinguishes op-codes from possible hex numbers ADC and ADD.

### NEXT

Forth executes your word definitions under control of the address interpreter, named NEXT. This short code routine moves execution from one definition, to the next. At the end of your code definition, you must return control to NEXT or else to code which returns to NEXT.

### RETURN OF CONTROL

Most 6502 systems can resume execution after a break, since the monitor saves the CPU register contents. Therefore, we must return control to Forth after a return from the monitor. NEXT is a constant that specifies the machine address of Forth's address interpreter (say $0242). Here it is the operand for JMP,. As JMP, executes, it assembles a machine code jump to the address of NEXT from the assembly time stack value.

### SECURITY

Numerous tests are made within the assembler for user errors:

1. All parameters used in CODE definitions must be removed.
2. Conditionals must be properly nested and paired.
3. Address modes and operands must be allowable for the op-codes

These tests are accomplished by checking the stack position (in CSP) at the creation of the definition name and comparing it with the position at END-CODE. Legality of address modes and operands is insured by means of a bit mask associated with each operand.

Remember that if an error occurs during assembly, END-CODE never executes. The result is that the "smudged" condition of the definition name remains in the "smudged" condition and will not be found during dictionary searches.

The user should be aware that one error not trapped is referencing a definition in the wrong vocabulary:

i.e., 0= of ASSEMBLER when you want 0= of FORTH

(Editor's note: the listing assumes that the figFORTH error messages are already available in the system, as follows:

?CSP issues the error message "DEFINITION NOT FINISHED" if the stack position differs from the value saved in the user variable CSP, which is set at the creation of teh definition name.

?PAIRS issues the error message "CONDITIONALS NOT IMPAIRED" if its two arguments do not match.

3 ERROR prints the error message "HAS INCORRECT ADDRESS MODE".)

SUMMARY

The object code of our example is:

```
305  983 4D 4F CE     CODE MON
305D 4D 30            link field
305F 61 30            code field
3061 00               BRK
3062 4C 42 02         JMP NEXT
```

OP-CODES, revisited

The bulk of the assembler consists of dictionary entries for each op-code. The 6502 one mode op-codes are:

```
BRK,  CLC,  CLD,  CLI,  CLV,
DEX,  DEY,  INX,  INY,  NOP,
PHA,  PHP,  PLA,  PLP,  RTI,
RTS,  SEC,  SED,  SEI,  TAX,
TAY,  TSX,  TXS,  TXA,  TYA,
```

When any of these are executed, the corresponding op-code byte is assembled into the dictionary.

The multi-mode op-codes are:

```
ADC,  AND,  CMP,  EOR,  LDA,
ORA,  SBC,  STA,  ASL,  DEC,
INC,  LSR,  ROL,  ROR,  STX,
CPX,  CPY,  LDX,  LDY,  STY,
JSR,  JMP,  BIT,
```

These usually take an operand, which must already be on the stack. An address mode may also be specified. If none is given, the op-code uses z-page or absolute addressing. The address modes are determined by:

| Symbol | Mode | Operand |
|---|---|---|
| .A | accumulator | none |
| # | immediate | 8 bits only |
| ,X | indexed X | z-page or absolute |
| ,Y | indexed Y | z-page or absolute |
| X) | indexed indirect X | z-page only |
| )Y | indirect indexed Y | z-page only |
| ) | indirect | absolute only |
| none | memory | z-page or absolute |

EXAMPLES

Here are examples of Forth vs. conventional assembler. Note that the operand comes first, followed by any mode modifier, and then the op-code mnemonic. This makes best use of the stack at assembly time. Also, each assembler word is set off by blanks, as is required for all Forth source text.

```
.A ROL,        ROL A
1 # LDY,       LDY #1
DATA ,X STA,   STA DATA,X
DATA ,Y CMP,   CMP DATA,Y
6 X) ADC,      ADC (06,X)
POINT )Y STA,  STA (POINT),Y
VECTOR ) JMP,  JMP (VECTOR)
```

(.A distinguishes from hex number 0A)

The words DATA and VECTOR specify machine addresses. In the case of "6 )X ADC," the operand memory address $0006 was given directly. This is occasionally done if the usage of a value doesn't justify devoting the dictionary space to a symbolic value.

6502 CONVENTIONS

Stack Addressing

The data stack is located in z-page, usually addressed by "Z-PAGE,X". The stack starts near $009E and grows downward. The X index register is the data stack pointer. Thus, incrementing X by two removes a data stack value; decrementing X twice makes room for one new data stack value.

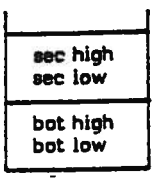
Sixteen bit values are placed on the stack according to the 6502 convention; the low byte is at low memory, with the high byte following. This allows "indexed, indirect X" directly off a stack value.

The bottom and second stack values are referenced often enough that the support words BOT and SEC are included. Using

BOT LDA, assembles LDA (0,X) and
SEC ADC, assembles ADC (2,X)

BOT leaves 0 on the stack and sets the address mode to ,X. SEC leaves 2 on the stack also setting the address mode to ,X.

Here is a pictorial representation of the stack in z-page.




<==X offset above $0000

Here is an examples of code to "or" to the accumulator four bytes on the stack:

```
BOT  LDA,    LDA (0,X)
BOT 1+ ORA,  ORA (1,X)
SEC  ORA,    ORA (2,X)
SEC 1+ ORA,  ORA (3,X)
```

To obtain the 14-th byte on the stack: BOT 13 + LDA,

RETURN STACK

The Forth Return Stack is located in the 6502 machine stack in Page 1. It starts at $01FE and builds downward. No lower bound is set or checked as Page 1 has sufficient capacity for all (non-recursive) applications.

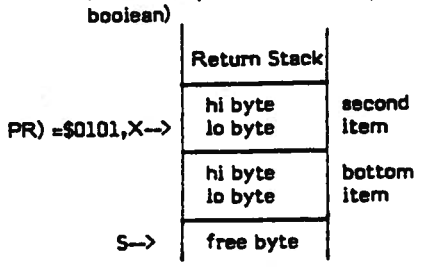
By 6502 convention the CPU's register points to the next free byte below the bottom of the Return Stack. The byte order follows the convention of low significance byte at the lower address.

Return stack values may be obtained by: PLA, PLA, which will pull the low byte, then the high byte from the return stack. To operate on arbitrary bytes, the method is:

1) save X in XSAVE
2) execute TSX, to bring the S register to X.
3) use RP) to address the lowest byte of the return stack. Offset the value to address higher bytes. (Address mode is automatically set to ,X.)
4) Restore X from XSAVE.

As an example, this definition non-destructively tests that the second item on the return stack (also the machine stack) is zero.

```
CODE IS-IT  ( zero ? )
    XSAVE STX, TSX,  (setup for
    return stack)
    RP) 2+ LDA, RP) 3 + ORA,
    ( or 2nd item's two bytes
    together)
0= IF, INY, THEN, ( if zero, bump
Y to one)
    TYA, PHA, XSAVE LDX, (save
    low byte, rstore data stack)
    PUSH JMP, END-CODE ( push
    boolean)
```



## FORTH REGISTERS

Several Forth registers are available only at the assembly level and have been given names that return their memory addresses. These are:

IP address of the Interpretive Pointer, specifying the next Forth address which will be interpreted by NEXT.

W address of the pointer to the code field of the dictionary definition just interpreted by NEXT. W-1 contains $6C, the op-code for indirect jump. Therefore, jumping to W-1 will indirectly jump via W to the machine code for the definition.

UP User Pointer containing address of the base of the user area.

N a utility area in z-page from N-1 thru N+7.

### CPU Registers

When Forth execution leaves NEXT to execute a CODE definition, the following conventions apply:

1. The Y index register is zero. It may be freely used.
2. The X index register defines the low byte of the bottom data stack item relative to machine address $0000.
3. The CPU stack pointer S points one byte below the low byte of the bottom return stack item. Executing PLA, will pull this byte to the accumulator.
4. The accumulator may be freely used.
5. The processor is in the binary mode and must be returned in that mode.

## XSAVE

XSAVE is a byte buffer in z-page, for temporary storage of the X register. Typical usage, with a call which will change X, is:

```
CODE DEMO
    XSAVE STX, USER'S JSR,
    ( which will change X )
    XSAVE LDX, NEXT JMP,
    END-CODE
```

## N Area

When absolute memory registers are required, use the 'N Area' in the base page. These registers may be used as pointers for indexed/indirect addressing or for temporary values. As an example of use, see CMOVE in the system source code.

The assembler word N returns the base address (usually $00D1). The N Area spans 9 bytes, from N-1 thru N+7. Conventionally, N-1 holds one byte and N, N+2, N+4, N+6 are pairs which may hold 16-bit values. See SETUP for help on moving values to the N Area.

It is very important to note that many Forth procedures use N. Thus, N may only be used within a single code definition. Never expect that a value will remain there, outside a single definition!

```
CODE DEMO  HEX
   6 # LDA, N 1 - STA,
     (setup a counter)

BEGIN, 8001 BIT,
     (tickle a port)

     N 1 - DEC,
       (decrement the counter)

   0= UNTIL,  NEXT JMP, END-CODE
     (loop till negative)
```

## SETUP

Often we wish to move stack values to the N area. The sub-routine SETUP has been provided for this purpose. Upon entering SETUP the accumulator specifies the quantity of 16-bit stack values to be moved to the N area. That is, A may be 1, 2, 3, or 4 only:

```
3 # LDA,  SETUP JSR,

 stack before   N after    stack after
      H high                    H
      G_low          bot--> G_
      F          F
      E_         E_
      D          D
sec--> C_        C_
      B          B
bot--> A_   N--> A_
```

## CONTROL FLOW

Forth discards the usual convention of assembler labels. Instead, two replacements are used. First, each Forth definition name is permanently included in the dictionary. This allows procedures to be located and executed by name at any time as well as be compiled within other definitions.

Secondly, within a code definition, execution flow is controlled by label-less branching according to "structured programming". This method is identical to the form used in colon-definitions. Branch calculations are done at assembly time by temporary stack values placed by the control words:

BEGIN, UNTIL, IF, ELSE, THEN,

Here again, the assembler words end with a comma, to indicate that code is being produced and to clearly differentiate from the high-level form.

One major difference occurs! High-level flow is controlled by run-time boolean values on the data stack. Assembly flow is instead controlled by processor status bits. The programmer must indicate which status bit to test, just before a conditional branching word (IF, and UNTIL,).

Examples are:

```
PORT LDA, 0= IF, <a> THEN,
    (read port, if equal to zero do <a> )

PORT LDA, 0= NOT IF, <a> THEN,
    (read port, if not equal to zero
    do <a> )
```

The conditional specifiers for 6502 are:

| | | |
|---|---|---|
| CS | test carry set | C=1 in processor status |
| 0< | byte less than zero | N=1 |
| 0= | equal to zero | Z=1 |
| CS NOT | test carry clear | C=0 |
| 0 <NOT | test positive | N=0 |
| 0= NOT | test not equal zero | Z=0 |

The overflow status bit is so rarely used, that it is not included. If it is desired, compile:

```
ASSEMBLER DEFINITIONS HEX
50 CONSTANT VS     (test overflow
set)
```

## CONDITIONAL LOOPING

A conditional loop is formed at assembler level by placing the portion to be repeated between BEGIN, and UNTIL,:

```
   6 # LDA,  N STA,
  (define loop counter in N)
BEGIN,  PORT DEC,
  (repeated action)
    N DEC,      0= UNTIL,
  (N reaches zero)
```

First, the byte at address N is loaded with the value 6. The beginning of the loop is marked (at assembly time) by BEGIN,. Memory at PORT is decremented, then the loop counter in N is decremented. Of course, the CPU updates its status register as N is decremented. Finally, a test for Z=1 is made; if N hasn't reached zero, execution returns to BEGIN,. When N reaches zero (after executing PORT DEC, 6 times) execution continues ahead after UNTIL,. Note that

BEGIN, generates no machine code, but is only an assembly time locator.

CONDITIONAL EXECUTION

Paths of execution may be chosen at assembly in a similar fashion and done in colon-definitions. In this case, the branch is chosen based on a processor status condition code.

```
PORT LDA,   0= IF,   (for zero set)
THEN, (continuing code)
```

In this example, the accumulator is loaded from PORT. The zero status is tested if set (Z=1). If so, the code (for zero set) is executed. Whether the zero status is set or not, execution will resume at THEN,.

The conditional branching also allows a specific action for the false case. Here we see the addition of the ELSE, part.

```
PORT LDA, 0= IF,  < for zero set>
          ELSE, <for zero clear>
          THEN, <continuing code>
```

The test of PORT will select one of two execution paths, before resuming execution after THEN,. The next example increments N based on bit D7 of a port:

```
PORT LDA,            ( fetch one byte )
  0< IF, N DEC,      ( if D7=1, decrement
                       N )
     ELSE, N INC,    ( if D7=0, increment
                       N )
     THEN,           ( continue ahead )
```

CONDITIONAL NESTING

Conditionals may be nested, according to the conventions of structured programming. That is, each conditional sequence begun (IF, BEGIN,) must be terminated (THEN, UNTIL,) before the next earlier conditional is terminated. An ELSE, must pair with the immediately preceding IF,.

```
BEGIN, < code always executed>
   CS IF, <code if carry set>
       ELSE,  <code if carry clear>
       THEN,
   0= NOT UNTIL,  ( loop till condition
                    flag is non-zero)
       <code that continues onward>
```

Next is an error that the assembler security will reveal.

```
BEGIN, PORT LDA,
   0= IF, BOT INC,
       0= UNTIL,  ENDIF,
```

The UNTIL, will not complete the pending BEGIN, since the immediately preceding IF, is not completed. An error trap will occur at UNTIL, saying "conditionals not paired".

RETURN OF CONTROL, revisited

When concluding a code definition, several common stack manipulations often are needed. These functions are already in the nucleus, so we may share their use just by knowing their return points. Each of these returns control to NEXT.

| | |
|---|---|
| POP POPTWO | remove one 16-bit stack values. |
| POPTWO | remove two 16-bit stack values. |
| PUSH | add two bytes to the data stack. |
| PUT | write two bytes to the data stack, over the present bottom of the stack. |

Our next example complements a byte in memory. The bytes' address is on the stack when INVERT is executed.

```
CODE INVERT  ( a memory byte )  HEX
  BOT X) LDA,    (fetch byte addressed
                  by stack)
    FF # EOR,    (complement accumu-
                  lator)
  BOT X) STA,  ( replace in memory )
  POP JMP,  END-CODE  ( discard
                  pointer from stack,
                  return to NEXT )
```

A new stack value may result from a code definition. We could program placing it on the stack by:

```
CODE ONE ( put 1 on the stack )
     DEX, DEX,  ( make room on the
                  data stack)
     1 # LDA, BOT STA,  (store low byte)
     BOT 1+ STY, ( hi byte stored from Y
                  since = zero)
     NEXT JMP,  END-CODE
```

A simpler version could use PUSH:

```
CODE ONE
   1 # LDA,  PHA,  ( push low byte to
   machine stack )
   TYA,  PUSH JMP,  ( high byte to
   accumulator, push to data stack )
   END-CODE
```

The convention for PUSH and PUT is:

1. push the low byte onto the machine stack.
2. leave the high byte in the accumulator.
3. jump to PUSH or PUT.

PUSH will place the two bytes as the new bottom of the data stack. PUT will over-write the present bottom of the stack with the two bytes. Failure to push exactly one byte on the machine stack will disrupt execution upon usage!

FOOLING SECURITY

Occasionally we wish to generate unstructured code. To accomplish this, we can control the assembly time security checks, to our purpose. First, we must note the parameters utilized by the control structures at assembly time. The notation below is taken from the assembler glossary. The --- indicates assembly time execution, and separate input stack values from the output stack values of the words execution.

```
BEGIN, ==>                      --- addrB 1
UNTIL, ==>  addrB 1   cc        ---

IF,    ==>            cc        --- addrI 2
ELSE,  ==>  addrI 2              --- addrE 2
THEN,  ==>  addrI 2              ---
       or   addrE 2              ---
```

The address values indicate the machine location of the corresponding 'B'EGIN, 'I'F, or 'E'LSE,. cc represents the condition code to select the processor status bit referenced. The digit 1 or 2 is tested for conditional pairing.

The general method of security control is to drop off the check digit and manipulate the addresses at assembly time. The security against errors is less, but the programmer is usually paying intense attention to detail during this effort.

To generate the equivalent of the high level:

```
BEGIN <a> WHILE <b> REPEAT
```

we write in assembly:

```
BEGIN, DROP ( the check digit
                1, leaving addrB)
       <a>
  CS IF,  ( leaves addrI and digit
            2)
          <b>
    ROT ( bring addrB to bottom)
    JMP, (to addrB of BEGIN, )
  ENDIF,  ( complete false for-
            ward branch from IF, )
```

It is essential to write the assembly time stack on paper, and run through the assembly steps, to be sure that the check digits are dropped and re-inserted at the correct points and addresses are correctly available.

ASSEMBLER GLOSSARY

| | |
|---|---|
| # | Specify 'immediate' addressing mode for the next op-code generated. |
| )Y | Specify 'indirect indexed Y' addressing mode for the next op-code generated. |