                        B.Y.O.ASSEMBLER
                             -or-
         Build Your Own (Cross-) Assembler....in Forth

                       by Brad Rodriguez

A. INTRODUCTION

    In a previous issue of this journal I described how to
    "bootstrap" yourself into a new processor, with a simple
    debug monitor. But how do you write code for this new CPU,
    when you can't find or can't afford an assembler? Build
    your own!

    Forth is an ideal language for this. I've written cross-
    assemblers in as little as two hours (for the TMS320, over a
    long lunch break). Two days is perhaps more common; and one
    processor (the Zilog Super8) took me five days. But when
    you have more time than money, this is a bargain.

    In part 1 of this article I will describe the basic
    principles of Forth-style assemblers -- structured,
    single-pass, postfix. Much of this will apply to any
    processor, and these concepts are in almost every Forth
    assembler.

    In part 2 I will examine an assembler for a specific CPU:
    the Motorola 6809. This assembler is simple but not
    trivial, occupying 15 screens of source code. Among other
    things, it shows how to handle instructions with multiple
    modes (in this case, addressing modes). By studying this
    example, you can figure out how to handle the peculiarities
    of your own CPU.

B. WHY USE FORTH?

    I believe that Forth is the easiest language in which to
    write assemblers.

    First and foremost, Forth has a "text interpreter" designed
    to look up text strings and perform some related action.
    Turning text strings into bytes is exactly what is needed to
    compile assembler mnemonics! Operands and addressing modes
    can also be handled as Forth "words."

    Forth also includes "defining words," which create large
    sets of words with a common action. This feature is very
    useful when defining assembler mnemonics.

    Since every Forth word is always available, Forth's
    arithmetic and logical functions can be used within the
    assembler environment to perform address and operand
    arithmetic.

    Finally, since the assembler is entirely implemented in
    Forth words, Forth's "colon definitions" provide a
    rudimentary macro facility, with no extra effort.

C. THE SIMPLEST CASE: ASSEMBLING A NOP

    To understand how Forth translates mnemonics to machine
    code, consider the simplest case: the NOP instruction (12
    hex on the 6809).

    A conventional assembler, on encountering a NOP in the
    opcode field, must append a 12H byte to the output file and
    advance the location counter by 1. Operands and comments
    are ignored. (I will ignore labels for the time being.)

    In Forth, the memory-resident dictionary is usually the
    output "file." So, make NOP a Forth word, and give it an
    action, namely, "append 12H to the dictionary and advance

```
    the dictionary pointer."

            HEX
            : NOP,    12 C, ;
```

Assembler opcodes are often given Forth names which include
a trailing comma, as shown above. This is because many
Forth words -- such as AND XOR and OR -- conflict with
assembler mnemonics. The simplest solution is to change the
assembler mnemonics slightly, usually with a trailing comma.
(This comma is a Forth convention, indicating that something
is appended to the dictionary.)

D. THE CLASS OF "INHERENT" OPCODES

Most processors have many instructions, like NOP, which
require no operands. All of these could be defined as Forth
colon definitions, but this duplicates code, and wastes a
lot of space. It's much more efficient to use Forth's
"defining word" mechanism to give all of these words a
common action. In object-oriented parlance, this builds
"instances" of a single "class."

This is done with Forth's CREATE and DOES>. (In fig-Forth,
as used in the 6809 assembler, the words are <BUILDS and
DOES>.)

```
        : INHERENT          ( Defines the name of the class)
            CREATE          ( this will create an instance)
                C,          ( store the parameter for each
                                            instance)
            DOES>           ( this is the class' common action)
                C@          ( get each instance's parameter)
                C,          ( the assembly action, as above)
            ;               ( End of definition)

        HEX
        12  INHERENT NOP,   ( Defines an instance NOP, of class
                              INHERENT, with parameter 12H.)
        3A  INHERENT ABX,   ( Another instance - the ABX instr)
        3D  INHERENT MUL,   ( Another instance - the MUL instr)
```

In this case, the parameter (which is specific to each
instance) is simply the opcode to be assembled for each
instruction.

This technique provides a substantial memory savings, with
almost no speed penalty. But the real advantage becomes
evident when complex instruction actions -- such as required
for parameters, or addressing modes -- are involved.

E. HANDLING OPERANDS

Most assembler opcodes, it is true, require one or more
operands. As part of the action for these instructions,
Forth routines could be written to parse text from the input
stream, and interpret this text as operand fields. But why?
The Forth envrionment already provides a parse-and-interpret
mechanism!

So, Forth will be used to parse operands. Numbers are
parsed normally (in any base!), and equates can be Forth
CONSTANTs. But, since the operands determine how the opcode
is handled, they will be processed first. The results of
operand parsing will be left on Forth's stack, to be picked
up by the opcode word. This leads to Forth's unique postfix
format for assemblers: operands, followed by opcode.

Take, for example, the 6809's ORCC instruction, which takes
a single numeric parameter:

```
            HEX
            : ORCC,    1A C,  C, ;
```

The exact sequence of actions for ORCC, is: 1) put 1A hex
on the parameter stack; 2) append the top stack item (the
1A) to the dictionary, and drop it from the stack; 3) append
the new top stack item (the operand) to the dictionary, and
drop it from the stack. It is assumed that a numeric value
was already on the stack, for the second C, to use. This
numeric value is the result of the operand parsing, which,
in this case, is simply the parsing of a single integer
value:

                HEX
                0F ORCC,

The advantage here is that all of Forth's power to operate
on stack values, via both built-in operators and
newly-defined functions, can be employed to create and
modify operands. For example:

        HEX
        01 CONSTANT CY-FLAG   (a "named" numeric value)
        02 CONSTANT OV-FLAG
        04 CONSTANT Z-FLAG
          ...
        CY-FLAG Z-FLAG + ORCC,   (add 1 and 4 to get operand)

The extension of operand-passing to the defining words
technique is straightforward.

F. HANDLING ADDRESSING MODES

    Rarely can an operand, or an opcode, be used unmodified.
    Most of the instructions in a modern processor can take
    multiple forms, depending on the programmer's choice of
    addressing mode.

    Forth assemblers have attacked this problem in a number of
    ways, depending on the requirements of the specific
    processor. All of these techniques remain true to the Forth
    methodology: the addressing mode operators are implemented
    as Forth words. When these words are executed, they alter
    the assembly of the current instruction.

    1. Leaving additional parameters on the stack.
        This is most useful when an addressing mode must always
        be specified. The addressing-mode word leaves some
        constant value on the stack, to be picked up by the
        opcode word. Sometimes this value can be a "magic
        number" which can be added to the opcode to modify it
        for the different mode. When this is not feasible, the
        addressing-mode value can activate a CASE statement
        within the opcode, to select one of several actions.
        In this latter case, instructions of different lengths,
        possibly with different operands, can be assembled
        depending on the addressing mode.

    2. Setting flags or values in fixed variables.
        This is most useful when the addressing mode is
        optional. Without knowing whether an addressing mode
        was specified, you don't know if the value on the stack
        is a "magic number" or just an operand value. The
        solution: have the addressing mode put its magic number
        in a predefined variable (often called MODE). This
        variable is initialized to a default value, and reset
        to this default value after each instruction is
        assembled. Thus, this variable can be tested to see if
        an addressing mode was specified (overriding the
        default).

    3. Modifying parameter values already on the stack.
        It is occasionally possible to implement addressing
        mode words that work by modifying an operand value.
        This is rarely seen.

All three of these techniques are used, to some extent,
within the 6809 assembler.

For most processors, register names can simply be Forth
CONSTANTs, which leave a value on the stack. For some
processors it is useful to have register names specify
"register addressing mode" as well. This is easily done by
defining register names with a new defining word, whose
run-time action sets the addressing mode (either on the
stack or in a MODE variable).

Some processors allow multiple addressing modes in a single
instruction. If the number of addressing modes is fixed by
the instruction, they can be left on the stack. If the
number of addressing modes is variable, and it is desired to
know how many have been specified, multiple MODE variables
can be used for the first, second, etc. (In one case -- the
Super8 -- I had to keep track of not only how many
addressing modes were specified, but also where among the
operands they were specified. I did this by saving the
stack position along with each addressing mode.)

Consider the 6809 ADD instruction. To simplify things,
ignore the Indexed addressing modes for now, and just
consider the remaining three addressing modes: Immediate,
Direct, and Extended. These will be specified as follows:

                     source code   assembles as
        Immediate: number # ADD,   8B nn
        Direct: address <> ADD,   9B aa
        Extended: address ADD,   BB aa aa

Since Extended has no addressing mode operator, the
mode-variable approach seems to be indicated. The Forth
words # and <> will set MODE.

Observe the regularity in the 6809 opcodes. If the
Immediate opcode is the "base" value, then the Direct opcode
is this value plus 10 hex, and the Extended opcode is this
value plus 30 hex. (And the Indexed opcode, incidentally,
is this value plus 20 hex.) This applies uniformly across
almost all 6809 instructions which use these addressing
modes. (The exceptions are those opcodes whose Direct
opcodes are of the form 0x hex.)

Regularities like this are made to be exploited! This is a
general rule for writing assemblers: find or make an opcode
chart, and look for regularities -- especially those
applying to addressing modes or other instruction modifiers
(like condition codes).

In this case, appropriate MODE values are suggested:

        VARIABLE MODE   HEX
        : #   0 MODE ! ;
        : <>   10 MODE ! ;
        : RESET   30 MODE ! ;

The default MODE value is 30 hex (for Extended mode), so a
Forth word RESET is added to restore this value. RESET will
be used after every instruction is assembled.

The ADD, routine can now be written. Let's go ahead and
write it using a defining word:

        HEX
        : GENERAL-OP   \ base-opcode --
            CREATE C,
            DOES>   \ operand --
            C@   \ get the base opcode
            MODE @ +   \ add the "magic number"
            C,   \ assemble the opcode
            MODE @ CASE
                0 OF C, ENDOF   \ byte operand

```
            10 OF C, ENDOF     \ byte operand
            30 OF , ENDOF      \ word operand
          ENDCASE
          RESET ;

      8B GENERAL-OP ADD,
```

Each "instance" of GENERAL-OP will have a different base
opcode. When ADD, executes, it will fetch this base opcode,
add the MODE value to it, and assemble that byte. Then it
will take the operand which was passed on the stack, and
assemble it either as a byte or word operand, depending on
the selected mode. Finally, it will reset MODE.

Note that all of the code is now defined to create
instructions in the same family as ADD:

```
          HEX 89 GENERAL-OP ADC,
              84 GENERAL-OP AND,
              85 GENERAL-OP BIT,
                      etc.
```

The memory savings from defining words really become evident
now. Each new opcode word executes the lengthy bit of DOES>
code given above; but each word is only a one-byte Forth
definition (plus header and code field, of course).

This is not the actual code from the 6809 assembler -- there
are additional special cases which need to be handled. But
it demonstrates that, by storing enough mode information,
and by making liberal use of CASE statements, the most
ludicrous instruction sets can be assembled.

G. HANDLING CONTROL STRUCTURES

   The virtues of structured programming, have long been sung
   -- and there are countless "structured assembly" macro
   packages for conventional assemblers. But Forth assemblers
   favor label-free, structured assembly code for a pragmatic
   reason: in Forth, it's simpler to create assembler
   structures than labels!

   The structures commonly included in Forth assemblers are
   intended to resemble the programming structures of
   high-level Forth. (Again, the assembler structures are
   usually distinguished by a trailing comma.)

1. BEGIN, ... UNTIL,

   The BEGIN, ... UNTIL, construct is the simplest assembler
   structure to understand. The assembler code is to loop back
   to the BEGIN point, until some condition is satisfied. The
   Forth assembler syntax is

            BEGIN,   more code   cc UNTIL,

   where 'cc' is a condition code, which has presumably been
   defined -- either as an operand or an addressing mode -- for
   the jump instructions.

   Obviously, the UNTIL, will assemble a conditional jump. The
   sense of the jump must be "inverted" so that if 'cc' is
   satisfied, the jump does NOT take place, but instead the
   code "falls through" the jump. The conventional assembler
   equivalent would be:

            xxx: ...
                  ...
                  ...
                  JR ~cc,xxx

   (where ~cc is the logical inverse of cc.)

```

```
        Forth offers two aids to implementing BEGIN, and UNTIL,.
        The word HERE will return the current location counter
        value. And values may be kept deep in the stack, with no
        effect on Forth processing, then "elevated" when required.

        So: BEGIN, will "remember" a location counter, by placing
        its value on the stack. UNTIL, will assemble a conditional
        jump to the "remembered" location.

                : BEGIN, (- a)   HERE ;
                : UNTIL, (a cc -) NOTCC JR, ;

        This introduces the common Forth stack notation, to indicate
        that BEGIN, leaves one value (an address) on the stack.
        UNTIL, consumes two values (an address and a condition code)
        from the stack, with the condition code on top. It is
        presumed that a word NOTCC has been defined, which will
        convert a condition code to its logical inverse. It is also
        presumed that the opcode word JR, has been defined, which
        will expect an address and a condition code as operands.
        (JR, is a more general example than the branch instructions
        used in the 6809 assembler.)

        The use of the stack for storage of the loop address allows
        BEGIN, ... UNTIL, constructs to be nested, as:

            BEGIN, ... BEGIN, ... cc UNTIL, ... cc UNTIL,

        The "inner" UNTIL, resolves the "inner" BEGIN, forming a
        loop wholly contained within the outer BEGIN, ... UNTIL,
        loop.

   2. BEGIN, ... AGAIN,

        Forth commonly provides an "infinite loop" construct,
        BEGIN ... AGAIN , which never exits. For the sake of
        completeness, this is usually implemented in the assembler
        as well.

        Obviously, this is implemented in the same manner as BEGIN,
        ... UNTIL, except that the jump which is assembled by AGAIN,
        is an unconditional jump.

   3. DO, ... LOOP,

        Many processors offer some kind of looping instruction.
        Since the 6809 does not, let's consider the Zilog Super8;
        its Decrement-and-Jump-Non-Zero (DJNZ) instruction can use
        any of 16 registers as the loop counter. This can be
        written in structured assembler:

                DO,   more code   r LOOP,

        where r is the register used as the loop counter. Once
        again, the intent is to make the assembler construct
        resemble the high-level Forth construct.

                : DO, (- a)   HERE ;
                : LOOP, (a r -)   DJNZ, ;

        Some Forth assemblers go so far as to make DO, assemble a
        load-immediate instruction for the loop counter -- but this
        loses flexibility. Sometimes the loop count isn't a
        constant. So I prefer the above definition of DO, .

   4. IF, ... THEN,

        The IF, ... THEN, construct is the simplest forward-
        referencing construct. If a condition is satisfied, the
        code within the IF,...THEN, is to be executed; otherwise,
        control is transferred to the first instruction after THEN,.

        (Note that Forth normally employs THEN, where other
        languages use "endif." You can have both in your
```

```
       assembler.)
```

       The Forth syntax is

```
               cc IF,  ... ... ...   THEN,
```

       for which the "conventional" equivalent is

```
                   JP    ~cc,xxx
                   ...
                   ...
                   ...
               xxx:
```

       Note that, once again, the condition code must be inverted
       to produce the expected logical sense for IF, .

       In a single pass assembler, the requisite forward jump
       cannot be directly assembled, since the destination address
       of the jump is not known when IF, is encountered. This
       problem is solved by causing IF, to assemble a "dummy" jump,
       and stack the address of the jump's operand field. Later,
       the word THEN, (which will provide the destination address)
       can remove this stacked address and "patch" the jump
       instruction accordingly.

```
           : IF, ( cc - a)   NOT 0 SWAP JP,  ( conditional jump
                           HERE 2 - ;     with 2-byte operand)
           : THEN, ( a)   HERE SWAP ! ;   ( store HERE at the
                                             stacked address)
```

       IF, inverts the condition code, assembles a conditional jump
       to address zero, and then puts on the stack the address of
       the jump address field. (After JP, is assembled, the
       location counter HERE points past the jump instruction, so
       we need to subtract two to get the location of the address
       field.)   THEN, will patch the current location into the
       operand field of that jump.

       If relative jumps are used, additional code must be added to
       THEN, to calculate the relative offset.

    5. IF, ... ELSE, ... THEN,

       A refinement of the IF,...THEN, construct allows code to be
       executed if the condition is NOT satisfied. The Forth
       syntax is

```
               cc IF,  ... ...  ELSE,  ... ...   THEN,
```

       ELSE, has the expected meaning: if the first part of this
       statement is not executed, then the second part is.

       The assembler code necessary to create this construct is:

```
                   JP    ~cc,xxx
                   ...              ( the "if" code)
                   ...
                   JP    yyy
               xxx: ...             ( the "else" code)
                   ...
               yyy:
```

       ELSE, must modify the actions of IF, and THEN, as follows:
       a) the forward jump from IF, must be patched to the start of
       the "else" code ("xxx"); and b) the address supplied by
       THEN, must be patched into the unconditional jump
       instruction at the end of the "if" code ("JP yyy"). ELSE,
       must also assemble the unconditional jump. This is done
       thus:

```
           : ELSE ( a - a)   0 T JP,    ( unconditional jump)
                       HERE 2 -     ( stack its address
                                       for THEN, to patch)
```

```
                    SWAP              ( get the patch address
                                          of the IF, jump)
                    HERE SWAP !  ( patch it to the current
                                          location, i.e., the
                    ;                     next instruction)
```

   Note that the jump condition 'T' assembles a "jump always"
   instruction. The code from IF, and THEN, can be "re-used"
   if the condition 'F' is defined as the condition-code
   inverse of 'T':

```
     : ELSE ( a - a)   F IF,  SWAP  THEN, ;
```

   The SWAP of the stacked addresses reverses the patch order,
   so that the THEN, inside ELSE, patches the original IF; and
   the final THEN, patches the IF, inside ELSE,. Graphically,
   this becomes:

```
     IF,(1)   ...   IF,(2)  THEN,(1)   ...   THEN,(2)
                     _____/
                        inside ELSE,
```

   IF,...THEN, and IF,...ELSE,...THEN, structures can be
   nested. This freedom of nesting also extends to mixtures of
   these and BEGIN,...UNTIL, structures.

6. BEGIN, ... WHILE, ... REPEAT,

   The final, and most complex, assembler control structure is
   the "while" loop in which the condition is tested at the
   beginning of the loop, rather than at the end.

   In Forth the accepted syntax for this structure is

```
          BEGIN,  evaluate  cc WHILE,  loop code  REPEAT,
```

   In practice, any code -- not just condition evaluations --
   may be inserted between BEGIN, and WHILE,.

   What needs to be assembled is this: WHILE, will assemble a
   conditional jump, on the inverse of cc, to the code
   following the REPEAT,. (If the condition code cc is
   satisfied, we should "fall through" WHILE, to execute the
   loop code.) REPEAT, will assemble an unconditional jump
   back to BEGIN. Or, in terms of existing constructs:

```
     BEGIN,(1)  ...  cc IF,(2)  ...  AGAIN,(1)  THEN,(2)
```

   Once again, this can be implemented with existing words, by
   means of a stack manipulation inside WHILE, to re-arrange
   what jumps are patched by whom:

```
          : WHILE, ( a cc - a a)   IF, SWAP ;
          : REPEAT, ( a a - )   AGAIN,  THEN, ;
```

   Again, nesting is freely permitted.

H. THE FORTH DEFINITION HEADER

   In most applications, machine code created by a Forth
   assembler will be put in a CODE word in the Forth
   dictionary. This requires giving it an identifying text
   "name," and linking it into the dictionary list.

   The Forth word CREATE performs these functions for the
   programmer. CREATE will parse a word from the input stream,
   build a new entry in the dictionary with that name, and
   adjust the dictionary pointer to the start of the
   "definition field" for this word.

   Standard Forth uses the word CODE to distinguish the start
   of an assembler definition in the Forth dictionary. In
   addition to performing CREATE, the word CODE may set the

```
assembler environment (vocabulary), and may reset variables
(such as MODE) in the assembler.  Some Forths may also
require a "code address" field; this is set by CREATE in
some systems, while others expect CODE to do this.
```

I. SPECIAL CASES

1. Resident vs. cross-compilation

```
Up to now, it has been assumed that the machine code is to
be assembled into the dictionary of the machine running the
assembler.
```

```
For cross-assembly and cross-compilation, code is usually
assembled for the "target" machine into a different area of
memory.  This area may or may not have its own dictionary
structure, but it is separate from the "host" machine's
dictionary.
```

```
The most common and straightforward solution is to provide
the host machine with a set of Forth operators to access the
"target" memory space.  These are made deliberately
analogous to the normal Forth memory and dictionary
operators, and are usually distinguished by the prefix "T".
The basic set of operators required is:
```

```
         TDP            target dictionary pointer DP
         THERE          analogous to HERE, returns TDP
         TC,            target byte append C,
         TC@            target byte fetch C@
         TC!            target byte store C!
         T@             target word fetch @
         T!             target word store !
```

```
Sometimes, instead of using the "T" prefix, these words will
be given identical names but in a different Forth
vocabulary.  (The vocabulary structure in Forth allows
unambiguous use of the same word name in multiple contexts.)
The 6809 assembler in Part 2 assumes this.
```

2. Compiling to disk

```
Assembler output can be directed to disk, rather than to
memory.  This, too, can be handled by defining a new set of
dictionary, fetch, and store operators.  They can be
distinguished with a different prefix (such as "T" again),
or put in a distinct vocabulary.
```

```
Note that the "patching" manipulations used in the
single-pass control structures require a randomly-
accessible output medium.  This is not a problem with disk,
although heavy use of control structures may result in some
inefficient disk access.
```

3. Compiler Security

```
Some Forth implementations include a feature known as
"compiler security," which attempts to catch mismatches of
control structures.  For example, the structure
```

```
        IF, ... cc UNTIL,
```

```
would leave the stack balanced (UNTIL, consumes the address
left by IF,), but would result in nonsense code.
```

```
The usual method for checking the match of control
structures is to require the "leading" control word to leave
a code value on the stack, and the "trailing" word to check
the stack for the correct value.  For example:
```

```
        IF,   leaves a 1;
        THEN,  checks for a 1;
        ELSE,  checks for a 1 and leaves a 1;
```

```
              BEGIN,   leaves a 2;
              UNTIL,   checks for a 2;
              AGAIN,   checks for a 2;
              WHILE,   checks for a 2 and leaves a 3;
              REPEAT,  checks for a 3.
```

This will detect most mismatches. Additional checks may be
included for the stack imbalance caused by "unmatched"
control words. (The 6809 assembler uses both of these error
checks.)

The cost of compiler security is the increased complexity of
the stack manipulations in such words as ELSE, and WHILE,.
Also, the programmer may wish to alter the order in which
control structures are resolved, by manually re-arranging
the stack; compiler security makes this more difficult.

4. Labels

Even in the era of structured programming, some programmers
will insist on labels in their assembler code.

The principal problem with named labels in a Forth assembler
definition is that the labels themselves are Forth words.
They are compiled into the dictionary -- usually at an
inconvenient point, such as inside the machine code. For
example:

```
              CODE TEST  ...  machine code  ...
                  HERE CONSTANT LABEL1
                  ...  machine code  ...
                  LABEL1 NZ JP,
```

will cause the dictionary header for LABEL1 -- text, links,
and all -- to be inserted in the middle of CODE. Several
solutions have been proposed:

  a) define labels only "outside" machine code.
     Occasionally useful, but very restricted.

  b) use some predefined storage locations (variables) to
     provide "temporary," or local, labels.

  c) use a separate dictionary space for the labels, e.g.,
     as provided by the TRANSIENT scheme [3].

  d) use a separate dictionary space for the machine code.
     This is common practice for meta-compilation; most
     Forth meta- compilers support labels with little
     difficulty.

5. Table Driven Assemblers

Most Forth assemblers can handle the profusion of addressing
modes and instruction opcodes by CASE statements and other
flow-of-control constructs. These may be referred to as
"procedural" assemblers.

Some processors, notably the Motorola 68000, have
instruction and addressing sets so complex as to render the
decision trees immense. In such cases, a more "table-
driven" approach may save substantial memory and processor
time.

(I avoid such processors. Table driven assemblers are much
more complex to write.)

6. Prefix Assemblers

Sometimes a prefix assembler is unavoidable. (One example:
I recently translated many K of Super8 assembler code from
the Zilog assembler to a Forth assembler.) There is a
programming "trick" which simulates a prefix assembler,
while using the assembler techniques described in this

article.

Basically, this trick is to "postpone" execution of the
opcode word, until after the operands have been evaluated.
How can the assembler determine when the operands are
finished? Easy: when the next opcode word is encountered.

So, every opcode word is modified to a) save its own
execution address somewhere, and b) execute the "saved"
action of the previous opcode word. For example:

          ... JP operand   ADD operands ...

JP stores its execution address (and the address of its
"instance" parameters) in a variable somewhere. Then, the
operands are evaluated. ADD will fetch the information
saved by JP, and execute the run-time action of JP. The JP
action will pick up whatever the operands left on the stack.
When the JP action returns, ADD will save its own execution
address and instance parameters, and the process continues.
(Of course, JP would have executed its previous opcode.)

This is confusing. Special care must be taken for the first
and last opcodes in the assembler code. If mode variables
are used, the problem of properly saving and restoring them
becomes nightmarish. I leave this subject as an exercise
for the advanced student...or for an article of its own.

J. CONCLUSION

I've touched upon the common techniques used in Forth
assemblers. Since I believe the second-best way to learn is
by example, in part 2 I will present the full code for the
6809 assembler. Studying a working assembler may give you
hints on writing an assembler of your own.

The BEST way to learn is by doing!

K. REFERENCES

1. Curley, Charles, Advancing Forth. Unpublished manuscript
   (1985).

2. Wasson, Philip, "Transient Definitions," Forth Dimensions
   III/6 (Mar-Apr 1982), p.171.

L. ADDITIONAL SOURCES

1. Cassady, John J., "8080 Assembler," Forth Dimensions III/6
   (Mar-Apr 1982), pp. 180-181. Noteworthy in that the entire
   assembler fits in less than 48 lines of code.

2. Ragsdale, William F., "A FORTH Assembler for the 6502," Dr.
   Dobb's Journal #59 (September 1981), pp. 12-24. A simple
   illustration of addressing modes.

3. Duncan, Ray, "FORTH 8086 Assembler," Dr. Dobb's Journal #64
   (February 1982), pp. 14-18 and 33-46.

4. Perry, Michael A., "A 68000 Forth Assembler," Dr. Dobb's
   Journal #83 (September 1983), pp. 28-42.

5. Assemblers for the 8080, 8051, 6502, 68HC11, 8086, 80386,
   68000, SC32, and Transputer can be downloaded from the Forth
   Interest Group (FORTH) conference on GEnie.