CHAPTER 25. META-COMPILER

The source code discussed in this chapter is in the file META86.BLK and KERNEL86.BLK, screens 1 to 10.

Meta-compilation is a special feature of Forth to generate a new Forth system by an existing Forth system. It is impossible in other operating systems and languages because of the complexity in the conventional operating systems and language compilers. The most a user can do in those environments is to do a 'sysgen', which allows a user to delete unnecessary or unused features in the full system and build a simpler system tailored to his equipment. The simplicity and conciseness of a Forth system give the user much more freedom in selecting and eliminating features to suit his application. Meta-compilation enables him to create a new system precisely customized to his needs and the new system can be moved to a different computer even with different CPU's.

25.1. CONCEPT OF META-COMPILATION

The theory behind Forth meta-compilation is rather straightforward. Definitions in the Forth dictionary can be compiled or assembled according to the specifications of the target machine. A new dictionary can be created for the target machine containing all the definitions which are necessary for execution on the target machine. This new dictionary can then be transferred to the target machine and executed on the target machine. The new dictionary will, of course, contain the required nucleus to operate on the new host CPU with necessary interpreter, compiler, and applications. A special initialization routine must also be included to power up the new target system. The meta-compilation is the process to generate the new dictionary for the target computer.

Currently F83 has been implemented on 8080, 8086/88, and 68000 microprocessors running under CP/M and MS-DOS operating systems. Meta-compilation was used extensively to transport the F83 model to different CPU's and to different operating systems. The meta-compiler used to create a F83 system is included in the F83 system so that the user can use it to rebuild the system or to generate new systems suitable for his applications. The authors of F83 intended that the users will modify their F83 systems to explore new uses of Forth and to develop commercial products for public utilization.

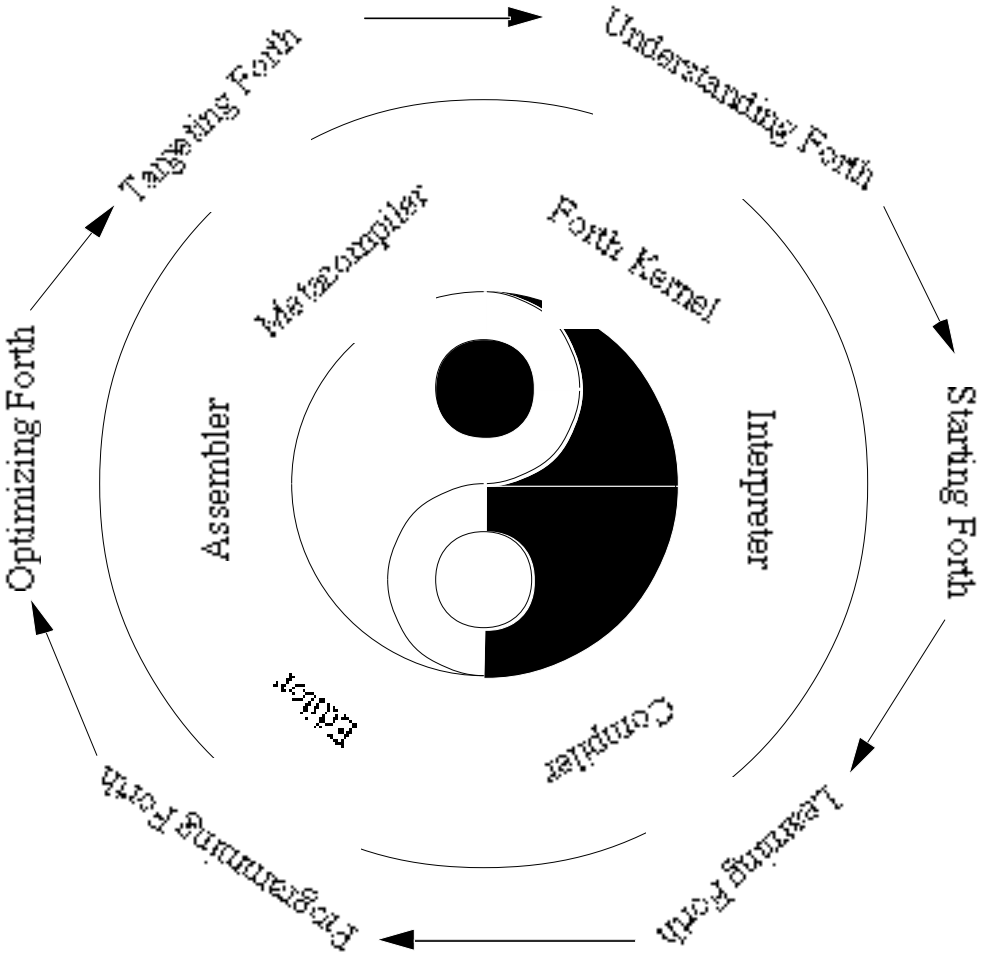
Meta-compilation is considered to be the highest level of extensibility in Forth. The first level of extensibility is to use predefined defining words like : and CODE to add new functions to the existing system. The second level of extensibility is to create new defining words which can be used to compile specialized words or data structures to the dictionary and to interpret them according to user specifications. The third level of extensibility, meta-compilation, is to regenerate the entire Forth system with whatever extensions the user might attach to it. This activity has been the privilege of large corporations and large teams or systems programmers at the expenses of billions of dollars, to build computer operating systems. In Forth, this privilege is accorded to us ordinary souls in the form of a metacompiler.

The most fundamental issue in metacompilation is that the target Forth system occupies an addressing space entirely different from the regular memory space the host Forth system addresses. The virtual address space of the target Forth system must be mapped to the real memory in the host Forth system.

The meta-compiler must be able to build the new system in the virtual memory space and resolve all the addresses and linkages accordingly. For one thing, the words in the new system cannot be executed and the new dictionary cannot be searched like normal Forth words. Searching must be done through one or more symbol tables, and compiler directives must be defined in a special vocabulary to help building structures in the words belonging to the target system. These are non-trivial tasks.

Forth system generated from meta-compilation seems to be very mysterious to people unfamiliar with the art, because the kernel is written in terms of the meta-compiler. If you do not understand the Forth meta-compiler, it is very difficult to read and comprehend the source code in the kernel. The great puzzle is that if you do not understand the Forth kernel, how can you understand the Forth meta-compiler? Does the chicken come before the egg, or the other way around? This puzzle is depicted in Fig. 25.1. The solution to this puzzle is that you pick one point to enter the circle. It does not matter where you enter. You must go around the circle many times, each time picking up something you didn't understand before. When you close the loop around the circle leaving no gap on the trace, you have attained the 'Tao' of Forth.

Figure 25.1 The Tao of Forth



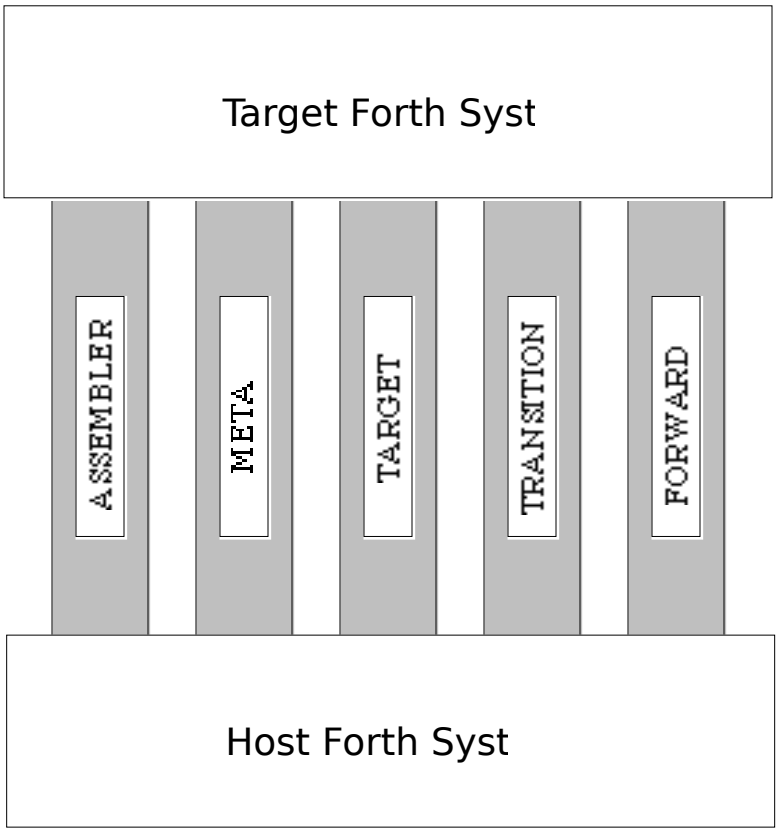
25.2. VOCABULARIES FOR META-COMPILATION

The main purpose of meta-compilation is to build new definitions in the new target system using the same Forth words. It is thus necessary that a Forth word should execute differently depending upon when and where it is invoked. Multiple definitions of the same word is a bad practice in normal Forth programming, but necessary in meta-compilation. It is accomplished by using many vocabularies to house different definitions of the same words. Any of these definitions can be invoked by selecting a specific vocabulary searching order.

```
ONLY FORTH ALSO                 Start with the normal FORTH and ROOT vocabularies.
VOCABULARY META                 Define META vocabulary which will contain all the words to
                                effect meta-compilation. Many of them are re-definitions.
META ALSO                       META vocabulary will be searched before FORTH
                                vocabulary and ONLY vocabulary.
META DEFINITIONS                Let META also be the current vocabulary so that new words
                                will be added to META.

: [FORTH]                       An immediate version of FORTH.
        FORTH
        ;       IMMEDIATE       Declare [FORTH] as immediate so that it will be executed
                                during compiling.

: [META]                        An immediate version of META.
        META ;  IMMEDIATE

: SWITCH            ( --- )     Exchange the saved values of CONTEXT and CURRENT
                                with themselves. It should always be used in pair to save
                                and restore the CONTEXT and CURRENT vocabularies.
                                Between the pair one can change vocabulary and select new
                                vocabulary definitions.
        NOOP NOOP               Two cells to save the current CONTEXT and CURRENT
                                vocabularies.
        DOES>       ( --- )
        DUP @                   Contents of the first cell of NOOP.
        CONTEXT @               Context vocabulary.
        SWAP CONTEXT !          Copy save context to CONTEXT.
        OVER !                  Save CONTEXT to NOOP cell.
        2+                      Address of second NOOP cell.
        DUP @                   Fetch saved current vocabulary.
        CURRENT @               Current CURRENT vocabulary.
        SWAP CURRENT !          Save it in second NOOP cell.
        SWAP !                  Restore CURRENT.
        ;

VOCABULARY TARGET               A vocabulary to hold the symbol table for all definitions in
                                the target system.
VOCABULARY TRANSITION           A vocabulary holding special case compiling words like ." and [ .
VOCABULARY FORWARD              A vocabulary holding all forward references as deferred words.
```

```
VOCABULARY USER          A vocabulary holding the USER version of defining words.
ONLY DEFINITIONS         Add all the vocabulary names to the ONLY vocabulary so
                         hat they are always accessible and that all words in every
                         vocabulary are accessible.
FORTH ALSO META ALSO Collect vocabulary names from both FORTH and META vocabularies.

: META   META   ;        META in ONLY calls META in FORTH.
: TARGET   TARGET   ;    And so forth.
: TRANSITION   TRANSITION   ;
: ASSEMBLER   ASSEMBLER   ;
: FORWARD   FORWARD   ;
: USER   USER   ;
```

Figure 25.2 Supporting vocabularies for meta-compilation



```
ONLY FORTH ALSO META ALSO DEFINITIONS     Restore the search order as META,
                                          FORTH and ONLY.
```

A few words are defined to re-order the vocabularies in the searching sequence to locate specific words in a specific vocabulary.

```
: IN-TARGET                   Search only the symbol table.
        ONLY TARGET DEFINITIONS
        ;
: IN-TRANSITION               Search TRANSITION, TARGET, and FORWARD in that order.
        ONLY FORWARD ALSO
        TARGET DEFINITIONS ALSO
        TRANSITION        ;

: IN-META                     The normal environment in doing meta-compilation.
        ONLY FORTH ALSO
        META DEFINITIONS ALSO
        ;

: IN-FORWARD                  Used when a word is undefined and must be compiled on the fly.
        FORWARD DEFINITIONS   ;
```

25.3. ACCESSING MEMORY IN THE TARGET SYSTEM

During meta-compilation, the dictionary of the target system is in a virtual memory space to which new definitions can be added but are not accessible for other purposes. These new definitions can never be executed because the target system will only be useful in the target computer which may be a totally different machine from the host computer performing the meta-compilation. The target virtual memory is mapped onto the host memory space by a constant offset and a variable dictionary pointer:

```
0 CONSTANT TARGET-ORIGIN          The offset address in the host memory where the target
                                  dictionary begins.  The value of TARGET ORIGIN Must
                                  be assigned before meta-compiling.

VARIABLE DP-T      ( --- addr )   The dictionary pointer for the target system during
                                  meta-compilation.
```

All memory accessing words in FORTH must be redefined for meta- compilation to access the memory of the target system.

```
: THERE            ( taddr --- addr )   Map a target address to a host address.
        TARGET-ORIGIN +                 Add offset.
        ;

: C@-T             ( taddr --- char )   Fetch a byte from given target address.
        THERE C@   ;
```

```
: @-T               ( taddr --- n )     Fetch a word from given target address.
        THERE @     ;

: C!-T              ( char taddr --- )  Store a byte at the target address.
        THERE C!    ;

: !-T               ( n taddr --- )     Store a word at the target address.
        THERE !     ;

: HERE-T            ( --- taddr )       Return target address of the next available dictionary byte.
        DP-T @      ;

: ALLOT-T           ( n --- )           Allocate more space in the target dictionary.
        DP-T +!     ;

: C,-T              ( char --- )        Add a byte to the target dictionary.
        HERE-T C!-T                     Compile one byte.
        1 ALLOT-T                       Move target dictionary pointer.
        ;

: ,-T               ( n --- )           Add a word to the target dictionary.
        HERE-T !-T                      Store one word.
        2 ALLOT-T                       Move pointer.
        ;

: S,-T              ( addr n --- )      Add a string to the target dictionary.
        0 ?DO                           Scan the length of string.
                DUP C@                  Fetch one character.
                C,-T                    Compile one byte.
                1+                      Increment addr.
        LOOP  DROP                      Discard addr still on stack.
        ;
```

25.4. BRANCHING CONSTRUCTS

Two sets of words setting up and resolving branches are needed in the meta-compiler: one for high level definitions and one for code definitions.

```
: ?>MARK            ( --- f addr )      Set up a forward branch in colon definition.
        TRUE                            Flag.
        HERE-T                          Address for the forward branch.
        0 ,-T                           Reserved for forward branch address.
        ;

: ?>RESOLVE( f addr --- )               Resolve a forward branch.
        HERE-T                          Address to branch to.
```

```
        SWAP !                          Store at the from address.
        ?CONDITION                      Error checking.
        ;

: ?<MARK            ( --- f addr )      Set up a backward branch.
        TRUE                            Put the flag on the stack,
        HERE-T                          with the current dictionary address.
        ;

: ?<RESOLVE( f addr --- )               Resolve a backward branch.
        ,-T                             Store the address to be branched to.
        ?CONDITION                      Error checking.
        ;
```

The following branching words are to be used in the assembler to set up branches in code definitions.

```
: M?>MARK           ( f addr --- )      Set up a forward branch in code definition.
        TRUE  HERE-T                    Leave current address.
        0 C,-T                          Reserve one byte for branching offset.
        ;

: M?>RESOLVE        ( f addr --- )      Resolve a forward branch in code definition.
        HERE-T                          Current address.
        OVER 1+ -                       Offset to the mark.
        SWAP C!-T                       Store into the reserved space.
        ?CONDITION                      Error checking.
        ;

: M?<MARK           ( --- f addr )      Set up a backward branch in code definition.
        TRUE                            Push flag on the stack,
        HERE-T                          with the dictionary address.
        ;

: M?<RESOLVE        ( f --- addr )      Resolve a backward branch in code definition.
        HERE-T 1+ -                     Offset from addr.
        C,-T                            Assemble offset after the branching instruction.
        ?CONDITION                      Error checking.
        ;
```

These assembler branching words are to be used to build code definitions in the target system. The regular FORTH assembler can be used for target meta-compilation if the branching words are smart enough to assemble structures in the virtual memory space of the target system. They are made smart by patching the executing addresses of the above definitions into the corresponding deferred words in the regular assembler:

```
' C,-T   ASSEMBLER IS C,
' ,-T    ASSEMBLER IS ,
' HERE-T   ASSEMBLER IS HERE
```

```
' M?>MARK   ASSEMBLER IS ?>MARK
' M?>RESOLVE   ASSEMBLER IS ?>RESOLVE
' M?<MARK   ASSEMBLER IS ?<MARK
' M?<RESOLVE   ASSEMBLER IS ?<RESOLVE
```

All the tools provided in the assembler are now available for the meta-compiler to build the nucleus portion of the target system.

25.5. FORWARD REFERENCE

FORTH normally does not allow forward referencing to a word that is not defined in its dictionary. This is a good practice to ensure that any defined word is immediately available for execution, testing and compiling. However, it presents a problem in meta-compilation because we have to have defining words to compile new definitions into the target system, while these defining words can only be defined much later in the meta-compiling process. The defining words must be made available at the very beginning of compiling the target system. F83 allows forward referencing to words not yet defined by creating deferred words which will be resolved and made executable at a later stage when the tools are available. The deferred words will be linked together in a list stored in the FORWARD vocabulary. At the end of meta-compilation, the list of forward references will be examined and vectored to executable words.

```
: MAKE-CODE         ( pfa --- )     Take the code field address pointed to by pfa and compile it
                                    in the target system.
        @                           Fetch cfa from pfa.
        ,-T                         Compile to target dictionary.
        ;

: LINK-BACKWARDS        ( pfa --- )     Extend the linked list of unresolved forward references.
        HERE-T                      Current dictionary address.
        OVER @ ,-T                  Store the address pointed to by pfa into current dictionary.
        SWAP !                      Store current dictionary address into pfa, thus extending the
                                    linked list.
        ;

: RESOLVED?         ( pfa --- f )   Return a true flag if the word whose pfa is on the stack is
                                    already resolve.
        2+                          Flag indicating that the word is resolved.
        C@                          Get it on stack as the flag.
        ;

: FORWARD-CODE  ( pfa --- )         If a forward reference is resolved, compile the code.
                                    Otherwise link it to the forward reference list.
        DUP RESOLVED?               A resolved forward reference?
        IF MAKE-CODE                If so, compile.
        ELSE LINK-FORWARD           Else link.
        THEN ;
```

```
: FORWARD:          ( --- )          Define a forward reference word and initialize it to be unresolved.
        SWITCH                       Save the current vocabularies.
        FORWARD DEFINITIONS          Make FORWARD the current and context vocabulary to
                                     build the forward reference word.
        CREATE                       Make the header.
        SWITCH                       Revert back to the original environment.
        0 ,                          Dummy execution address.
        0 C,                         Unresolved flag.
        DOES>        ( --- )
        FORWARD-CODE                 When a forward reference word is executed, either compile it
                                     to dictionary or link to the list of unresolved references.
        ;
```

25.6. COMPILING NEW WORDS TO TARGET SYSTEM

```
VARIABLE WIDTH                       The maximum length of the names in target definitions.
31 WIDTH !                           It is initialized to allow 31 characters in names.
VARIABLE LAST-T                      A variable pointing to the name field of the most recently
                                     defined word in target.
VARIABLE CONTEXT-T                   Pointer to the array of context and resident vocabularies in
                                     the target.
VARIABLE CURRENT-T                   Pointer to the vocabulary where new definitions are to be
                                     linked.

: HASH              ( str-addr voc-addr --- thread )     From the name of a definition and the
                                     address of the current vocabulary, return the thread address
                                     to link the new definition.
        SWAP 1+ C@                   Get the first character in the name.
        3 AND                        Only four threads are implemented.
        2* +                         Return the address of the thread in the body of the vocabulary.
        ;

: HEADER            ( --- )          Create a header in the target dictionary.  It makes a header
                                     out of the next word in the input stream and fixes up all the
                                     appropriate pointers to link it into the target dictionary.
        BL WORD                      Get the name.
        C@ 1+ WIDTH @ MIN            The length of the name field.
        ?DUP IF                      If length is not zero, make the header.
                ALIGN                Align new header to word boundary.
                BLK @                Current block number.
                4096 +               The view field with the block number and the file number,
                                     assumed to be 1.
                ,-T                  Compile the view field.
                HERE CURRENT-T @ HASH        Find the thread to link the new word.
                DUP @-T ,-T          Compile the link field.
                HERE-T 2-            The link field address of the new word in the target dictionary.
                SWAP !-T             Update the top of linking thread in the current vocabulary.
```

```
        HERE-T                  Save a copy of the name field address.
            HERE ROT S,-T       Move the name from host to target.
            ALIGN               Make sure the code field fall on even word boundary.
        DUP LAST-T !            Update the LAST-T with new name field address.
            128 SWAP THERE CSET     Set the delimiting bit in the first (count) byte of the
                                    name field.
            128 HERE-T 1- THERE CST     Set the delimiting bit in the last byte of name field.
        THEN
        ;                       No header will be created if WIDTH is set to zero.

: TARGET-CREATE  ( --- )        Create a header in target and an entry in the symbol table.
                                The new word is initialized as resolved so that it will be
                                compiled to the target when invoked.
        >IN @                   Save the input stream pointer.
        HEADER                  Create the target header.
        >IN !                   Restore the input pointer to reuse the name of the new definition.
        IN-TARGET CREATE        Create an entry in the symbol table, TARGET vocabulary.
        IN-META                 Return to meta-compiler.
        HERE-T ,                Compile the execution address of the new target word to pfa
                                of the symbol table entry.
        1 C,                    Compile the resolved flag.
        DOES>       ( --- )     When the entry in the symbol table is executed, the execution
                                address of the target word will be compiled to the target dictionary.
        MAKE-CODE               Compile the contents of the pfa to target dictionary.
        ;

: RECREATE       ( --- )        Same as TARGET-CREATE, but don't advance the input
                                stream pointer so that the name of word can be used again.
        >IN @                   Save the input stream pointer.
        TARGET-CREATE           Create headers in target and symbol table.
        >IN !                   Restore input stream pointer.
        ;

: CODE           ( --- )        Set up to assemble a new code definition to the target
                                dictionary. The target cfa is set to target pfa.
        TARGET-CREATE           Make the headers.
        HERE-T 2+               Parameter field address of the new code word.
        ,-T                     Compile code field in the target code word.
        ASSEMBLER !CSP          Set trap for error checking.
        ;

: LABEL          ( --- )        Remember the current target dictionary address and assign it a
                                name so that a subroutine can be called from a code definition.
        ASSEMBLER DEFINITIONS
        HERE-T CONSTANT         Assign a name to the target address.
        ;

ASSEMBLER DEFINITIONS           Go back to the FORTH assembler.
```

```
: END-CODE              Redefine the code word terminator for the target compiler.
      IN-META           Specify meta-compiling environment.
      ?CSP              Do error checking by comparing stack depth at the beginning
                        and end of a code definition.
      ;

META                    Return to meta-compiler.
IN-META                 And reorder the vocabularies as needed by the
                        meta-compilation.
```

25.7. TRANSITION COMPILER DIRECTIVES

Compiler directives, which build structures in the target words or performing special actions other than compiling execution addresses, cannot be executed immediately within the target compilation environment. The compiler directives in the normal FORTH cannot be used either, because structures and special conditions must be built in the target system. These meta-compiler directives are all put in the TRANSITION vocabulary and are executed from there. When these compiler words are encountered in the definitions of words to be put into the target dictionary, the corresponding words in the TRANSITION vocabulary are executed so that the special condition in the target definition can be dealt with immediately.

```
: 'T                ( --- cfa)  Look up the next word in the input stream only in the target
                                vocabulary. Preserve original context.
      CONTEXT @                 Save context vocabulary.
      TARGET DEFINED            Look up next word in the TARGET vocabulary.
                                ( context cfa f --- )
      ROT CONTEXT !             Restore context.
      0= ?MISSING               Abort if the word cannot be found.
      ;

: [TARGET]          ( --- )     Force the compilation of a TARGET word regardless of the
                                current CONTEXT vocabulary.
      'T                        Find the word in TARGET vocabulary.
      ,                         Compile its execution address.
      ;

: 'F                ( --- cfa ) Look up the next word in the input stream only in the
                                FORWARD vocabulary. Preserve current context.
      CONTEXT @                 Save context on stack.
      FORWARD DEFINED           Search FORWARD vocabulary for the next input word.
      ROT CONTEXT !             Restore context.
      0= ?MISSING               Abort if word cannot be found in the FORWARD vocabulary.
      ;

: T:                ( --- )     Define a new compiler directive word in the TRANSITION
                                vocabulary. It is otherwise the same as : .
```

```
        SWITCH                      Save the current CONTEXT and CURRENT vocabularies
.       TRANSITION DEFINITIONS      Make TRANSITION the current vocabulary.
        CREATE                      Define the following word in the TRANSITION vocabulary.
        SWITCH                      Restore original context.
        ]                           Compile the body of the new word.
        DOES>                       This is how the new word defined by T: should be executed:
        >R                          Push the parameter address of the new word on the return
                                    stack. The list of execution addresses compiled in the
                                    parameter field will be executed in sequence. Similar to what
                                    NEST might have done.
        ;

: T;                ( --- )         Terminate a word defined by T:.
        SWITCH                      Save context.
        TRANSITION DEFINITIONS      Change context to TRANSITION.
        [COMPILE] ;                 Compile end of word definition.
        SWITCH                      Restore context.
        ;           IMMEDIATE
```

Following are the string operators for in-line comments and documentation:

```
T: (                ( --- )
        [COMPILE] (                 Inherit ( from host to TRANSITION.
        T;

T: (S               ( --- )
        [COMPILE] (S                Inherit (S from host.
        T;

T: \                ( --- )
        [COMPILE] \                 Inherit \ from host.
        T;
```

Special words are needed to compile string literals into the target dictionary:

```
: STRING,-T         ( --- )         Scan the input stream for a " as the string delimiter and
                                    compile the string into target dictionary.
        ASCII " PARSE               Parse input text to ".
        DUP C@ 1+                   Length of string just parsed.
        S,-T                        Move the string and compile into the target dictionary.
        ALIGN                       Align to cell boundary.
        ;

FORWARD: <(.")>                     Runtime forward reference for the code compiled by ." .

T: ."                               Compile the runtime code <(.")> and a string literal in the
                                    arget dictionary.
        [FORWARD] <(.")>            Compile the forward reference.
```

```
        STRING,-T                   Compile the string literal.
        ;

FORWARD: <(")>                      Runtime forward reference for the code compiled by " .

T: "                                Compile the unknown runtime code <(")> with a string
                                    literal.
        [FORWARD] <(")>             Compile the forward reference word <(")>.
        STRING,-T                   Compile string literal from input stream.
        ;

FORWARD: <(ABORT")>                 Runtime forward reference for ABORT".

T: ABORT"                           Compile the unknown runtime code for abort, followed by a
                                    string.
        [FORWARD] <(ABORT")>        Compile the abort code.
        STRING,-T                   With the string literal.
        ;

FORWARD: <(;USES)>                  Forward reference for code routine compiled by ;USES.

FORTH VARIABLE STATE-T              True if in the compiling state inside a colon definition.  False
                                    if outside or in the interpreting state.

T: ;USES            ( --- )         Compile a code field whose runtime routine already exists.
                                    It is similar to ;CODE otherwise.
        [FORWARD] <(;USES)>         Compile the code field using the address of <(;USES)>.
        IN-META                     Force the context of meta-compiler.
        ASSEMBLER                   Invoke assembler to start assembling code routine.
        !CSP                        Install error checking.
        STATE-T OFF                 Assembler words are interpreted, not compiled.
        T;

T: [COMPILE]        ( --- )         Compile a TARGET word rather than execute its TRANSITION
                                     counterpart.
        'T                          Find the next word and return its execution address.
        EXECUTE                     Execute the word in the symbol table vocabulary TARGET.
                                    The effect is to compile the word into the target dictionary.
        ;

FORWARD: <(IS)>                     Forward reference to the runtime routine of IS.

T: IS               ( --- )         Compile the unknown address of <(IS)>.
        [FORWARD] <(IS)>  T;

: IS                ( cfa --- )     This is the version of IS in the meta-compiler which actually
                                    matches the forward reference.
        'T                          Find the cfa of the next word to be patched.
```

```
        >BODY @                 The execution address pointing to execution routine.
        >BODY !-T               Patch in with the cfa on stack. Thus resolve the forward reference.
        ;

T: ALIGN                        Align the dictionary pointer to word boundary.
        T;                      This is not needed in 8086, which is a true byte machine.

T: EVEN         ( n --- n' )    Make the number n even.
        T;                      Noop in 8086 or 8080.
```

25.8. DEFINING WORDS IN META-COMPILER

```
FORWARD: <VARIABLE>             Forward reference for runtime routine of CREATE and
                                VARIABLE.

: CREATE                        Create a target word using the runtime routine for
                                VARIABLE and a host word to return the HERE address in
                                target.
        RECREATE                Create target word and an entry in the symbol table
                                vocabulary TARGET. The input pointer is not advance.
        [FORWARD] <VARIABLE>    Compile code field in target.
        HERE-T CONSTANT         Define the pfa as a constant in the host.
        ;
: VARIABLE                      Define a variable in the target.
        CREATE                  Use the above CREATE.
        0 ,-T                   initialize the parameter field.
        ;

FORWARD: <DEFER>                Forward reference for the runtime routine of DEFER.

: DEFER                         Define a deferred word or an execution vector in the target.
        TARGET-CREATE           Create a target word and a symbol table entry.
        [FORWARD] <DEFER>       Compile code field.
        0 ,-T                   Compile a dummy parameter to hold execution address.
        ;

: DIGIT?        ( char --- f )  Return true if the character is a digit in current base.
        BASE @ DIGIT            Convert the char using current BASE.
        NIP                     Only the flag is needed.  Discard the result of conversion.
        ;

: PUNCT?        ( char --- f )  Return true if the character is a valid punctuation character
                                for numbers such as leading - or decimal point.
        ASCII . OVER = SWAP     A period?
        ASCII - OVER = SWAP     Or a minus sign?
        ASCII / OVER = SWAP     Or a / ?
        DROP OR OR              Return the flag.
```

```
;

: NUMERIC?          ( addr len --- f )      Return true if the string is a valid number in the
                                            current base. At least one valid digit should be present in the
                                            string.
        DUP 1 =                             Only one character?
        IF                                  Yes.
                DROP                        Discard len .
                C@ DIGIT?                   Is it a valid digit?
                EXIT                        No other action needed.
        THEN
        1 -ROT                              Initial flag.
        0 ?DO                               Scan the length of the string.
                DUP C@                      Get one character.
                DUP DIGIT?                  Is it a digit?
                SWAP PUNCT?                 Or a punctuation?
                OR ROT AND                  AND the test results to flag.
                SWAP 1+                     Increment addr.
        LOOP  DROP                          Discard addr.
        ;
```

25.9. USER VARIABLES

User variables are collected in a table called user area for multitasking context switching. All the variables pertinent to the independent operation of a task have to be preserved for each user or task when it relinquishes control of CPU to other tasks so that when it regains the control of CPU the task can continue from where was left off. The managing of the user area requires redefining many dictionary accessing words. These redefinitions are collected in a separated vocabulary USER. The target compiler must have its own versions of these words to compile user variables in the target system.

```
FORTH VARIABLE #USER-T      A variable in FORTH to count the number of user variables
                            defined in the target system.
META ALSO                   Revert to meta-compiler.
USER DEFINITIONS            Following words are added to the USER vocabulary.

: ALLOT             ( n --- )       Allocate space in the user area.
        #USER-T +!                  Add n to the user area counter.
        ;

FORWARD: <USER-VARIABLE>    Forward reference for the runtime routine of USER-VARIABLE.

: VARIABLE          ( --- )         Create a user variable in the user area.
        SWITCH                      Save context.
        RECREATE                    Create headers in target and symbol table.
        [FORWARD] <USER-VARIABLE>   Compile code field pointing to <USER-VARIABLE>.
        #USER-T @                   Current user area pointer.
        DUP ,-T                     Compile the pointer in parameter field.
```

```
        2 ALLOT                 Move user area pointer.
        META DEFINITIONS        Change current vocabulary to META.
        CONSTANT                Create a constant in META holding the user area pointer.
        SWITCH                  Restore context.
        ;

FORWARD: <USER-DEFER>           Forward reference for runtime routine of user deferred
                                words.

: DEFER             ( --- )     Create a user deferred word or a task local execution vector.
        SWITCH                  Save context.
        TARGET-CREATE           Create target and symbol table entries.
        [FORWARD] <USER-DEFER>  Compile code field pointing to <USER-DEFER>.
        SWITCH                  Restore context.
        #USER-T @ ,-T           Compile the user area pointer.
        2 ALLOT                 Move user area pointer.
        ;

ONLY FORTH ALSO META ALSO DEFINITIONS       Restore the meta-compiling environment.
```

25.10. VOCABULARY

The defining word VOCABULARY creates new vocabularies when the target system is brought up and running. In the parameter field of the vocabulary definition, four cells are used to store the addresses of the ends of four dictionary threads for the hashing algorithm. The last cell stores the VOC-LINK address, which points to the vocabulary defined immediately before the currently defined vocabulary. This way, all the vocabularies defined in the running system are linked together themselves. This vocabulary linkage is necessary when vocabularies have to be eliminated by FORGET.

```
FORTH VARIABLE VOC-LINK-T   A variable linking all defined vocabularies together.

FORWARD: <VOCABULARY>       The forward reference for the runtime routine of VOCABULARY.

: VOCABULARY    ( --- )     Create a vocabulary in the target system.
        RECREATE            Create headers.
        [FORWARD] <VOCABULARY>  Compile code field.
        HERE-T              Save the parameter field address of the defined vocabulary.
        #THREADS 0 DO 0 ,-T LOOP    Initialize four threads in the vocabulary.
        HERE-T VOC-LINK-T @ ,-T     Store previous vocabulary pfa after the thread field.
        VOC-LINK-T !        Store pfa of this vocabulary into VOC-LINK-T and extend
                            the vocabulary linkage.
        CONSTANT            Define a constant in the host.
        DOES>       ( --- )
        @                   Fetch the starting address of the thread field in the vocabulary.
        CONTEXT-T !         Store it in the context variable to make it the context vocabulary.
        ;
```

```
: IMMEDIATE         ( --- )      If heads are compiled in target, set the precedent or
                                 immediate bit in the name field.
        WIDTH @ IF               If heads are compiled,
                64               Precedent bit.
                LAST-T @ THERE   Address of the name field.
                CTOGGLE          Flip the precedent bit.
        THEN ;
```

25.11. RESOLVING FORWARD REFERENCES

```
: FIND-UNRESOLVED       ( --- cfa f )   Search for a word in the FORWARD vocabulary and
                                        return the status.
        'F                              Find the next word in the input stream in the FORWARD
                                        vocabulary.
        DUP >BODY                       Get the parameter field address.
        RESOLVED?                       Return the a true flag if the word is resolved.
        ;

: RESOLVE    ( taddr cfa --- )          Run through the linked list of forward reference and resolve
                                        each of them with the given address.
        >BODY                           The parameter field address from cfa.
        2DUP  TRUE OVER 2+ C!           Store the 'resolved' flag in the third byte of the forward
                                        reference word in FORWARD.
        @                               Address of the last member in the linked list of unresolved
                                        reference.
        BEGIN                           Run down the list.
                DUP                     If address is not 0, go do the resolving.
        WHILE
                2DUP @-T                Get the next unresolved reference.
                -ROT                    Replace the old one.
                SWAP !-T                Resolve the old reference.
        REPEAT
        2DROP                           Clear stack.
        ;

: RESOLVES          ( taddr --- )       The command used by user to resolve forward reference
.       FIND-RESOLVED                   Search the next word in the FORWARD vocabulary and
                                        determine if it is resolved.
        IF                              Yes.  Resolved.
                >NAME .ID               Print its name.
                ." Already resolved."   And a message.
                DROP                    No need of taddr.
        ELSE                            Not resolved.
                RESOLVE                 Then resolve the references.
        THEN ;
```

At the end of meta-compilation, all the forward references must be resolved before the target system is

saved. Otherwise, the target system will surely crash when it is executed. There is a long list of reference to be resolved. Following is a short list of examples for illustration. You should consult the source listing for the complete list.

```
' (.") RESOLVES <(.")> ' (") RESOLVES <(")>
' (;CODE) RESOLVES <(;CODE)> ' (;USES) RESOLVES <(;USES)>
[FORTH] ASSEMBLER DOCREATE META RESOLVES <VARIABLE>
[FORTH] ASSEMBLER DOUSER-DEFER META RESOLVES <USER-DEFER>
etc., etc.
```

Deferred words and many system variables need also be initialized:

```
' (LOAD) IS LOAD   ' CRLF IS CR   ' (KEY?) IS KEY?
etc., etc.

' FORTH >BODY CURRENT !-T    ' FORTH >BODY CONTEXT !-T
HERE-T DP UP @-T + !-T
etc., etc.
```

25.12. REDEFINING HOST WORDS

Many important words in the host or FORTH vocabulary are redefined to be used in meta-compilation. To use the host versions of them explicitly, they are redefined as host words prefixed with an 'H' character before the regular name.

```
: H:   [COMPILE] :   ;

H: '   'T >BODY @    ;

H: ,   ,-T   ;

H: C,  C,-T  ;

H: HERE   HERE-T   ;

H: ALLOT  ALLOT-T  ;

H: DEFINITIONS   DEFINITIONS   CONTEXT-T @ CURRENT-T !   ;

: ]]   ]  ;                          Alias of ], which will be used by the target compiler.

: [[   [COMPILE] [   ;   FORTH IMMEDIATE META ]] has to be stopped by the FORTH [, which takes
an alias of [[.

FORWARD: DEFINITIONS          Making both [ and DEFINITIONS forward references so
                              that the target compiler can assign compiler functions to them.
```

FORWARD: [

25.13. RUNNING THE META-COMPILER

Meta-compilation is a complicated process and should be used only when you have to tailor the Forth system to a very specific application. Since F83 systems were generated using meta-compilation and the authors were kind enough to provide us with the complete source of the meta-compiler and the actual loading commands to generate the F83 system, we have an excellent guide and example to follow. It is worthwhile to review the loading sequence in generating the F83 system. When you do meta-compiling of your own system, you probably should follow this sequence as closely as possible, making minimal changes and modifications in the kernel and adding your applications on top of the kernel. After you have gone through this process several times and obtain working systems, then you can start re-working the kernel.

To fire up the meta-compiler, you must first prepare a disk with F83.COM, META86.BLK, and KERNEL86.BLK files on it. Type

```
        F83  META86.BLK
and     1  LOAD
```

to bring up the F83 system, which in turn will load the first screen in the META86.BLK, the load screen of the meta-compiler. The loading command in screen 1:

```
        3 21 THRU
```

loads the meta-compiler, containing all the words discussed in this chapter. After the meta-compiler is loaded, we are ready to generate the kernel Forth or the minimal Forth operating system. The following command in screen 1 of META86.BLK opens the KERNEL86.BLK file and compiles the F83 kernel:

```
        ONLY FORTH DEFINITIONS ALSO
        FROM KERNEL86.BLK  1 LOAD
```

Since it will need the KERNEL86.BLK file, this file must also be on the disk. If your disk is not big enough to hold all these files, you should delete the FROM ... line from the screen 1 in META86.BLK. After the meta-compiler is loaded, you can change disk and type it in on the keyboard to load the kernel.

Screen 1 of the KERNEL86.BLK file is the loading screen of the kernel. The commands:

```
        ONLY FORTH    META ALSO FORTH
```

include the META vocabulary in the search order and we are ready to compile the kernel. However, we have to first allocate memory space to store the target kernel Forth system. This is done by:

```
256 DP-T !                      Initialize the dictionary pointer and leave 256 bytes at the
                                bottom of the dictionary for interrupt vectors.
HERE 12000 +                    The physical address of the target dictionary.
' TARGET-ORIGIN >BODY !         Store it in the constant, the address offset into the target dictionary.
```

IN-META	Establish the meta-compiling environment.
2 92 THRU	Load the entire kernel Forth system.

We should pay special attention to the last screens in the KERNEL86.BLK file, where all the forward references are resolved, all the deferred words are vectored to proper executable words, and all the system variables are initialized. To make a target system run properly, these things have to be done correctly.

After the kernel Forth is meta-compiled, it must be saved on the disk as an executable object file. It is saved and given the name KERNEL.COM:

META 256 THERE	The physical address where the target dictionary starts.
HERE-T	The logical address of the end of the target dictionary, which is the length of the target dictionary in bytes.
ONLY FORTH ALSO DOS	Switch to DOS vocabulary to access the SAVE command.
SAVE A:KERNEL.COM	Copy the target dictionary into KERNEL.COM, which is executable.
FORTH	

At this point, we have generated a minimal Forth system and put it in an object file KERNEL.COM. This is a usable Forth system containing the text interpreter and colon compiler. However, its function is limited and not quite usable as a system to do programming and development work. If you wanted to develop a Forth application, this kernel serves well as the foundation to support your application. You can load the application program on top of the kernel and it will become a product you can sell. As a product, F83 system has lots of bells and whistles to add to the kernel. The sequence to add applications to the kernel Forth is as following.

BYE	Exit F83 and return to the DOS environment.

Copy the KERNEL.COM file to a disk which contains a file with all the application programs. In this file, screen 1 must be a load screen which will load all the application programs. In the case of F83 system, this file is EXTEND86.BLK. Moreover, EXTEND86.BLK will load programs in UTILITY.BLK and CPU8086.BLK. Therefore, you will have to copy these two files to the disk. If your disk does not have enough room for all these files, you can delete the loading commands in screen 1 of the EXTEND86.BLK file and type them on the keyboard after switching disks.

To load the application on top of the kernel, type:

```
        KERNEL  EXTEND86.BLK
and     1  LOAD
```

The object file KERNEL will be loaded into the memory and the kernel Forth will be booted. It then loads the first screen in EXTEND86.BLK which loads in all the utility making up the entire F83 system. In this screen you will find the following loading commands:

3 LOAD	Load basic utility words and the ONLY-ALSO vocabulary mechanism.

6 LOAD	Load DOS file management words.
FROM CPU8086.BLK 1 LOAD	Load the 8086 assembler, and some CPU specific words to support I/O, debugger, and multitasker.
FROM UTILITY.BLK 1 LOAD	Load all the utility we discussed in Part III.

The F83 system is now complete and it is also saved on the disk in a file named F83.COM

```
SAVE A:F83.COM
```

This process is what was needed to build the F83 system. You have to follow it closely in building your own Forth system.

If you get this far, please take a moment congratulating yourself. If at this point you are still not quite comfortable with the meta-compilation, do not despair. Remember the circle of Tao? There is no end in that circle. Go back to the kernel and retrace the Forth thoughts again. Or better yet, find your favorite Forth code, add it to F83 and try to recompile the whole system. Change the boot-up code so that when the new system is loaded it will start executing your code directly. You can easily succeed in recompilation, if you only added new code without modifying the F83 system. Doing the meta-compilation a few times will greatly enhance your confident in yourself and in F83. Consequently, you will understand the whole system much sooner than what you have allowed yourself.