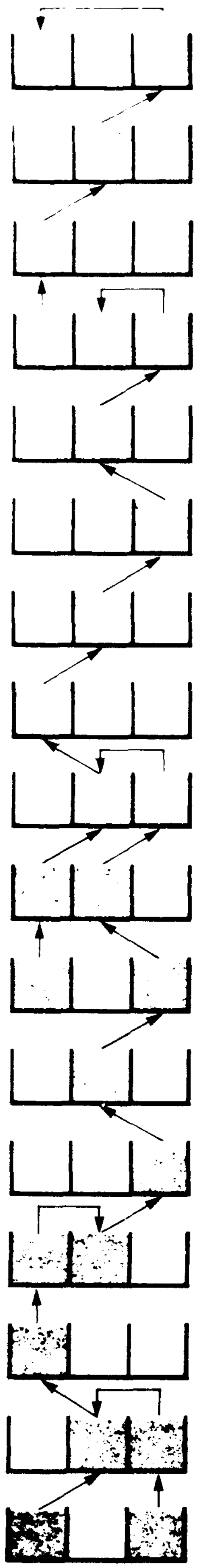
FORTH DIMENSIONS

FORTH INTEREST GROUP
P.O. Box 1105
San Carlos, CA 94070

Volume II
Number 2
Price $2.00



INSIDE

FORTH DIMENSIONS

Published by Forth Interest Group

Volume II No. 2 July/August 1980

Publisher Roy C. Martens

Subscription to FORTH DIMENSIONS is free with membership in the Forth Interest Group at $12.00 per year ($15.00 overseas). For membership, change of address and/or to submit material, the address is:

Forth Interest Group
P.O. Box 1105
San Carlos, CA 94070

HISTORICAL PERSPECTIVE

FORTH was created by Mr. Charles H. Moore in 1969 at the National Radio Astronomy Observatory, Charlottesville, VA. It was created out of dissatisfaction with available programming tools, especially for observatory automation.

Mr. Moore and several associates formed FORTH, Inc. in 1973 for the purpose of licensing and support of the FORTH Operating System and Programming Language, and to supply application programming to meet customers' unique requirements.

The Forth Interest Group is centered in Northern California, although our membership of 1100 is world-wide. It was formed in 1978 by FORTH programmers to encourage use of the language by the interchange of ideas through seminars and publications.

PUBLISHER'S COLUMN

Summer is here. Lazy days should abound but not at FIG and FORTH DIMENSIONS. Individuals and groups are working hard on Standards and Cases. Soon there will be announcements and printing of these efforts. In fact, if everything works out right, the next issue of FORTH DIMENSIONS will be the big one that everyone has been waiting for (have you renewed your subscription and membership?). This doesn't mean that we aren't eager for more articles and letters from members, send them. More issues are in the works.

Now that we have a few issues of the new looking FORTH DIMENSIONS under our belt, we'd like to have your suggestions about improvements and additional information. Do you want more technical material? More beginning input? More new product information? More????? Let us know. Remember your inputs are what make FORTH DIMENSIONS.

Have a nice summer. Renew if you haven't already.

Roy Martens

LYONS' DEN

While listening to the tapes of the FORTH Convention (available from Audio Village, P.O. Box 291, Bloomington, IA 47402, $16.00 for four tapes) I noticed puzzlement over how to communicate concisely the nature of FORTH, that is, what single term—operating system, compiler, interpreter—indentifies the class to which it belongs. How about referring to FORTH as a Meta Interpreter—a program for generating an interpreter (the application) to provide an interactive tool for solving application specific problems (sometimes referred to as JOL's, job-oriented-languages)? Other members of this class are LISP and an obscure IBM system called PLAN, as well as APL. FORTH has unique features distinguishing it from other members of this class, being more optimized for arithmetic than LISP, for example, and being more compact and lower level than APL. Also, its implementation is more like LISP than APL.

Continued on pg. 31

TEMPORAL ASPECTS OF THE FORTH LANGUAGE

A BEGINNER'S STUMBLING BLOCK

John M. Derick
Linda A. Baker
P.O. Box 553
Mountain View, CA 94042

Novice FORTH programmers who have had previous experience with other, more traditional, programming languages almost invariably become confused when first dealing with FORTH. A first time user sitting down at a FORTH terminal soon notices what seem to be time-based inconsistencies. That is, the language seems to require that things be done in the wrong order or that the language itself does things out of time order. The novice, striving to understand these supposed "inconsistencies" detects time as a note of commonality and therefore lumps them all together as one oddity, while in actuality there are three separate areas of difficulty.

The interesting point of this is that the cause of this confusion is so elementary that once the problems are understood, it is difficult to look back and pinpoint why the confusion arose in the first place. This is why these elementary problem areas are not stressed in most existing FORTH literature and are just assumed to be part of the longer than normal learning curve associated with FORTH. Making it clear in the neophyte's mind that there are three separate, but related, factor shortens this learning curve.

Let us examine what situations cause this confusion.

Sitting at a FORTH terminal, you enter a FORTH word, hit a carriage return and the word executes. Other times, though, you enter a line of FORTH words (including the one you just executed previously), hit carriage return and nothing executes. But when used later on this same word executes! As you learn more, you discover that in order to perform some functions you must actually alter the traditional time sequence of programming and modify FORTH's compiler after it already works and is ebugged. Then, to add even more confusion, you find that some words, when added to the compiler, will execute different parts of that same word at different times. Or, when you edit a FORTH program, save it on disk and then compile it; some parts compile as expected but other words execute immediately.

To an experienced FORTH programmer it is quite obvious that there are actually three separate (but releated) aspects of FORTH represented in this example. To a beginner all of these attributes are lumped together in one tangled question of "who's on first????" and "when did he get there????"

With the exception of different parts of a word executing at different times, these are very trivial problems to an experienced FORTH programmer. To the beginner they are totally new concepts that must be sorted out and grasped--even though once understood they really are trivial concepts.

Let us first address the most basic of these three time related stumbling blocks; that of modifying the compiler.

Before we continue it is important to point out that there are several steps (one may almost say laws) that always must be followed to generate object code from source code. Traditional programming languages take these steps in a straight line one-pass manner. FORTH also takes these same steps (i.e., a compiler has been written and installed). The difference with FORTH however, is that the act of writing the compiler is not intended to be a one-pass step. Instead it is a recursive procedure where the compiler is constantly modified and tailored to the users needs over and over again. This alters the time sequence of things and is a slightly shocking concept but the basic rules are still the same.

In traditional languages a programmer goes through several temporally separated steps to generate a user program: A compiler (or assembler), an editor, a link editor and loader are all separately created and installed on the user's system. Then the user edits a program, compiles it, links it, then loads and tests it. Everything is done in such absolutely clear cut steps that one is subtly led to believe that this is the absolute nature of the world.

FORTH on the other hand is a highly interactive, dictionary-based language where new additions to the language (i.e., user added words) are simply added to the end of the dictionary thereby "extending" it. FORTH's compiler is part of this dictionary and therefore words added to the dictionary can actually affect or be used in the compiler. In FORTH, this is not only possible, it is required if one is to fully use the power of the language.

A simple concept? Yes. But it is so contrary to traditional practice that it is hard for a neophyte to believe advanced documentation which tells how to build compiler directives such as "creating" or "defining" words while only alluding to the fact that the compiler can and should be modified.

Therefore, let us emphasize this fact: The compiler in FORTH is not sacred. The traditional sequential steps of writing a compiler and forever using that particular product do not apply in FORTH. FORTH's compiler may be modified at any time. All, or part of it may be executed at any time. As a matter of fact "creating" or "defining" words used in the compiler are actually tiny standalone compilers in themselves and can be used to perform mini-compilations whenever they are referenced.

Now that this compiler modification aspect has been "factored" out of the jumble of time related "confusions", the beginner is still left with the second point of confusion: Namely why words sometimes execute immediately and sometimes do not.

The technical reason why words execute immediately is that the "precedence" bit associated with that word is set on; but it is the philosophical reasoning for the existence of the precedence bit that is of importance to the neophyte.

Again all of this is tied in with the fact that FORTH's compiler is an integral interactive part of the language. It is an integral part of the language because it is composed of common FORTH words used not only in the compiler but in every other FORTH application as well.

Entering a FORTH word or words on a terminal, and hitting carriage return causes that word or words to be immediately executed and is similar to executing an already compiled and linked object module. The dictionary is searching until the word is found and the definition is executed. To do this, the word is preceded by a colon, FORTH is put into the compiler state, and all words up until a semicolon will be compiled (i.e., placed into the dictionary for future execution). This is similar to inputting source code to a FORTRAN compiler and getting object code out.

The point being made here is that FORTH continuously changes between "compiler" state and "execution" state. When in compiler state, most input words are compiled, not executed. Notice the word "most". Some words are executed while in compiler state. These are naturally called compiler words.

These compiler words are identical in appearance to any other FORTH word. Indeed they actually are simply FORTH words with the exception that their precedence bit is set. They are analogous to assembly language pseudo ops or compiler directives. A pseudo op (like ORG) in assembly language gives direction to or is "executed" by the assembler; not the object code. It is never executed by the user program.

Thus, words in FORTH may be "flagged" to operate as pseudo ops. That is, they may be chosen to execute immediately and thereby perform some act of compilation upon other words in the definition; (even if they are imbedded inside of a string of source code--just as a pseudo op would do in assembly language). This "flag" is the precedence bit. When the FORTH interpreter detects that this bit is set, it will cause it associated word to be executed immediately, even while in compiler mode. Using the word IMMEDIATE just after a definition is the method used to set the precedence bit.

This is a very powerful feature of the FORTH language. It allows definitions to execute while in compile mode and since FORTH makes no distinction between "supplied" words and user written words the compiler itself can be added to and improved. This feature is called "extendability".

There are certain defining words in FORTH that take the trait of "when a word is executed" one step further. Conceptionally advanced word such as BUILDS and DOES allow a definition to be constructed so that the first half of the word will be used at compile time but the second half will execute at execution time.

While it is beyond the scope of this paper to go into the usage of BUILDS and DOES type words, it should be noted that they exist and really do have two separate times of execution.

The last point of confusion is: When words contained in a "loaded" block execute immediately instead of compiling (or visa versa). When FORTH loads a block, it treats the incoming data almost as if it were being read from a keyboard. Definitions are compiled and put into the dictionary as they are encountered in the data stream. But, if a word is encountered that is not contained inside of a definition (whether intentionally or not!) that word is executed immediately, just as if it was entered from the keyboard. This is a quite straight forward, and quite understandable effect once it is pointed out. The rule here is to put words to be compiled inside of definitions. Leave words to be executed immediately outside of definitions.

A good example of word purposely left outside of a definition is DECIMAL. This word is normally used as the last word of a loaded block to insure that after compilation the system is left in its standard base ten state.

In summary, the temporal confusion that occurs when first using FORTH is all quite elementary and understandable--at least in principle. And at a beginners stage, principle is very important.

The three general categories; modifying the compiler, compiler directives using the precedence bit, and loading and compiling blocks, all perform execution at predictable times and really do have a direct correspondence with traditional programming sequences.

A GENERALIZED LOOP CONSTRUCT FOR FORTH

For some time, I have been building my own version of a FORTH-like language with direct rather than indirect threaded code, running on the 8080. Last year I learned that my approach is almost identical to that of URTH; this is not surprising since the design criterion of highest possible execution speed was the same. To this end, the inner interpreter has one level of indirection removed (compared to FORTH) and jumps (as for IF , ELSE , LOOP , WHILE , UNTIL , etc.) are compiled to their 16 bit absolute value, rather than a 16 bit offset. All this by way of preface that although my "home base" is isolated from the West Coast and my implementation of the following words may not be exactly FORTH compatible, yet I feel that the concepts presented are new and useful in the FORTH environment.

The article 'FORTH-85 "CASE" STATEMENT' by Richard B. Main in FORTH DIMENSIONS, Volume 1, Number 5 had a catalytic effect in the development of these ideas, specifically the technique of saving an unknown number of addresses on the stack and using zero as a marker for the last address. It seemed to me that one area to apply this scheme with good effect is in the BEGIN ... UNTIL and BEGIN ... WHILE ... REPEAT loop constructs which currently permit only one exit test. This sometimes forces awkward stack manipulations to "or" conditions when two or more conditions must be tested, any one of which is sufficient to terminate the loop. The proposed constructs solve this problem, require no more lower level CODE words than already exist, and add to the elegance of the language by removing the word REPEAT.

The generalized loop is constructed one of two ways:

```
      BEGIN ... WHILE ... WHILE ... WHILE ... UNTIL
or
      BEGIN ... WHILE ... WHILE ... WHILE ... AGAIN
```

There can be any number of WHILE words in each loop, including none. The meaning of the words BEGIN , WHILE , UNTIL , and AGAIN is exactly the same as currently understood; no new concepts need be learned. For newcomers to the language (of which we all hope for, and in large numbers) the learning task is easier because we have reduced the number of FORTH basic words while at the same time increasing the power of the language by permitting more powerful combinations of these words. This is surely a good direction since the human (programmer) mind is unsurpassed at manipulating symbols, but not in remembering them.

The Words

The following definitions work in my system. In FORTH, where XELSE and XIF require a compiled offset rather than an absolute address, the words WHILE , COMPADDS , AGAIN , and UNTIL must be changed slightly.

```
( GENERALIZED LOOP WORDS - BEGIN WHILE UNTIL AGAIN )

: BEGIN      HERE 0                         ; IMMEDIATE
: WHILE      LIT XIF , HERE 0               ; IMMEDIATE
: UNTIL      DROP LIT XIF , ,               ; IMMEDIATE ( TEMPORARY
: COMPADDS   BEING DUP IF HERE 1+ 1+ SWAP ! ENDIF ( = UNTIL ) ;
: AGAIN      LIT XELSE , COMPADDS ,         ; IMMEDIATE
: UNTIL      LIT XIF , COMPADDS ,           ; IMMEDIATE
```

How They Work, Compile Time

BEGIN Pushes onto the stack the address to which the loop should jump, followed by a zero. The zero is used as a market by the COMPADDS word.

WHILE (if used) Compiles a conditional jump to the temporary address of zero, and also pushes the address of the temporary address to the stack. The temporary address, which can never be zero, will later be overwritten by COMPADDS with the address of the next word immediately after the loop structure; this is how WHILE effects a loop exit.

UNTIL	(temporary) Allows correct compilation of the COMPADDS word's BEGIN ... UNTIL structure. It will shortly be replaced with the generalized UNTIL .
COMPADDS	Overwrites the address of all previous WHILE words until the last BEGIN . Each address on the stack (there may be none) is overwritten with the vale HERE+2. The zero placed on the stack by the last BEGIN terminates the overwriting and leaves the address of the first word in the loop on the top of the stack.
AGAIN	Compiles an unconditional jump, completes all previous WHILE words, and then compiles the address of the unconditional jump, pointing to the top of the loop.
UNTIL	Identical to AGAIN , except a conditional jump is compiled, allowing a conditional loop exit.

How They Work, Run Time

They work the same as the previously known BEGIN , WHILE , UNTIL , and AGAIN .

Error Procedures

Error checks can easily be added to these words. This is done as below:

```
( GENERALIZED LOOP WORDS - BEGIN WHILE UNTIL AGAIN )
( WITH ERROR PROCEDURES AS PER RULL-HOLLAND )
: BEGIN      HERE 0 1                            ; IMMEDIATE
: WHILE      1 ?PAIRS LIT XIF, HERE 0 , 1        ; IMMEDIATE
: UNTIL      DROP DROP LIT XIF , ,               ; IMMEDIATE ( TEMPORARY )
: COMPADDS   BEGIN DUP IF HERE 1+ 1+ SWAP ! ENDIF 0 UNTIL ;
: AGAIN      1 ?PAIRS LIT XELSE , COMPADDS ,   ; IMMEDIATE
: UNTIL      1 ?PAIRS LIT XIF , COMPADDS ,      ; IMMEDIATE
```

The operation is self-evident.

Conclusion

Generalized loop words BEGIN , WHILE , UNTIL , and AGAIN have been proposed. Their use provides, as a subset, the well known actions of BEGIN ... AGAIN , BEGIN ... UNTIL , and BEGIN ... WHILE ... REPEAT (with the word REPEAT replaced by AGAIN). When used in this manner the new words impose no more run time overhead in time or space than the words they replace. If the new words did nothing more, they would still be desirable because they "orthogonalize" the unconditional loop termination word, making it AGAIN regardless of the presence or absence of the WHILE word.

But, as an added benefit of the new words, more powerful constructs such as BEGIN ... WHILE ... UNTIL or BEGIN ... WHILE ... WHILE ... AGAIN are possible. Thus multiple tests and exits from a loop can be arranged in the most natural order, without the need to "or" the results of the tests. These multiple loop exits do not violate the principles of structured programming since they all lead to a common point; in other words, the loop, as a structure, has one entry and one exit.

Future Research

After much thought about the implications of the proposed words in relation to the FORTH philosophy of programming, I must say that of the two changes wrought by these words, viz.

and orthogonalization of the loop construct, and the ability to have multiple loop exits, I believe that orthogonalization is by far the most important result. In FORTH, while the very act of programming consists of extending the language by creating many new words useful in the application environment, even so, I believe that the initial basic words, especially the structured programming constructs such as IF ... ELSE ... ENDIF , BEGIN ... UNTIL , and DO ... LOOP should be as few and as general purpose as possible.

In addition, they should be carefully names so as to convey their action to programmers new to FORTH, but familiar with similar structures on other, "industry standard" languages such as ALGOL, PASCAL, and C. The construct IF ... ELSE ... THEN is poor in this respect; the word THEN confuses novices to FORTH since it usually implies selection, while in this case it is really a construct terminator. I assume that this is the reason why the change from THEN to ENIF was specified in FORTH-79. Similarly, BEGIN ... END is confusing since it does not imply repetition to the average programmer. FORTH-79 partially corrects this confusion with BEGIN .. UNTIL , but I believe some word signifying repetition should replace BEGIN , such as REPEAT ... UNTIL , REPEAT ... AGAIN , and REPEAT ... WHILE ... AGAIN .

As for DO ... LOOP , this construct cries out for a convenient way to prematurely exit the loop. LEAVE seems weird - at odds with commonly accepted practice - since it has a deferred effect, taking place only at the end of the loop. Although I won't remove it from the language, I suggest an alternative: Do ... WHILE ... LOOP . At the execution of the optional WHILE , if the stack is zero the loop is exitted. Not possible because WHILE is already used for the REPEAT ... WHILE ... AGAIN loop, you say? But it is possible! A very useful by-product of the Error Procedures of University at Ulrecht, Netherlands is that they always leave at the top of the stack (during compile time) a flag indicating the identity of the innermost construct, different for REPEAT ... and DO ...; it is then a simple matter to arrange WHILE to have different actions and to compile entirely different CODE words depending on this value. Of course, we would not limit the number of WHILE words between DO and LOOP . LOOP must be modified, as was described above for AGAIN , to permit this.

Bruce Komusin
Ontel Corp.
250 Crossways Park Dr.
Woodbury, NY 11797

New Product

OmniForth, from Interactive Computer Systems, is now available for the North Star computer. FORTH combines structured programming, stack organization, virtual memory, compiler, assembler, and file system into an extensible macrolanguage. Organized as a dictionary of words, FORTH allows defining new words that extend the vocabulary to suit any application. Words are compiled on entry into code ready for immediate test, and execute ten times faster than Basic. FORTH supports coding time-critical routines in assembler for the fastest response. OmniForth contains the interactive FORTH compiler (modeled on Fig-FORTH), assembler for the 8080 and Z-80, file system, and text editor. Omni-Forth requires 24K memory and North Star DOS, and costs $49.95; an optional Introduction to FORTH manual is available for $15.00. Interactive Computer Systems, Inc., 6403 DiMarco Road, Tampa, FL 33614.

FILE NAMING SYSTEM

Peter H. Helmers
University of Rochester

This particular FORTH file naming system is set up to use a disk based directory to name files which are comprised of a series of disk blocks. The system does not include any specific file formats, but instead is used to translate a filename to a block number. This block number can be a traditional "load block", a directory block for a linked set of random data blocks, or perhaps the initial block in a multi-block text file. Routines are available to control a disk's bit map of allocated blocks so that already utilized blocks are not overwritten. Additional routines allow creation of filename/block entries at either fixed block locations or at random locations, or deletion of file entries, directory listings, etc.

The philosophy in writing this package was that file formats should be user definable although several standard uses are being brought up for text files, and data arrays stored in consecutive blocks. By using the words available, additional file formats can be easily added.

The file naming system presently uses three blocks at the end of each disk. The first block contains two data arrays: a bit map of block usage on the disk, and a list of block-pointers for each defined filename. The bitmap uses one bit per disk block to define whether the block is used or not; the bit is a "1" if the block is used. The block pointer array consists of 64 integers which point to the filename's starting block number. A value of -1 means that the filename is undefined.

The second two blocks contain 64 filename strings of up to 32 characters each. Each name string is actually stored as a fixed length 32 byte string with any extra characters being padded blanks. A non-valid file is flagged by a -1 value for the block pointer, not by a null of special string.

The following is a list of the primary user oriented words in this file naming package:

("STR") FIND-NAME (INDX)

FIND-NAME searches for the STR in the directory and returns its directory index if found, or a -1 if not found. Thus a user can test for a -1 to see if a filename exists.

INIT-DIRECTORY

INIT-DIRECTORY is used to set all block pointers to -1's so that no files will be considered to be in existence.

INIT-BIT-MAP

INIT-BIT-MAP is used to set all bit map bits to 0's, thus indicating that no disk blocks are being used.

(BLK#) FREE-BLK

FREE-BLK is used to reset a given block's bit map bit, thus indicating that it is not in use.

(BLK#) RESERVE BLK

RESERVE-BLK is used to set a given block's bit map to indicate that it is in use.

FIND-FREE-BLK (BLK#)

FIND-FREE-BLK is used to find the first free block encountered in the bit map. It returns a "free" block number if one can be found, or a -1 if the disk is full.

("NAME") NEW

NEW is used to create a new filename entry with a block pointer found from the first free block encountered in the bit map.

("NAME"), (BLK#) NEW FIXED

NEW-FIXED is used to define a new filename with a specific block pointer (for example, a traditional "load block").

("NAME") FILE (BLK#)

FILE is used to translate a filename string to a specific block number.

("NAME") ERASE

ERASE is used to erase the given filename from the directory.

DIRECTORY

DIRECTORY is used to print a listing on the console of all defined filenames.

```
(FILE NAMING SYSTEM - PHH - 12 3 79 ) BASE @ HEX
: FILE-ERROR
   DOCASE
            DUP 1 = WHEN T" ALL BLOCKS USED "
      CASE DUP 2 = WHEN T" FILE ALREADY EXISTS "
      CASE DUP 3 = WHEN T" DIRECTORY FULL "
      CASE DUP 4 = WHEN T" NAME TOO LONG "
      CASE DUP 5 = WHEN T" FILE NOT FOUND "
   ENDCASE
   CR
   RESTART
;
2DROP ( DO CASE BUG )
BASE !    ;S
```

```
( FILE NAMING SYSTEM - PHH - 12 4 79 ) BASE @ HEX
: PB O DO T"  " LOOP ;
: "SPACES O DO "  " "+ LOOP ; ( ADD [TOS] SPACES TO STRNG )
: "GET 20 SWAP "@F ;  ( GET 32 BYTE STRNG FROM ADDR ON TOS )
: "PUT 20 SWAP "!F ; ( PUT 32 BYTE STRNG TO ADDR ON TOS )
: FILE-NAME-FIX                ( MAKE NAME 32 CHARS LONG )
   "LEN SUP 1E >               ( CHECK THAT NAME <= 30 CHARS )
   IF
      4 FILE-ERROR             ( NPOE, SO GIVE ERROR )
   THEN
   20 -- "SPACES               ( PAD W/BLNKS TO 32 CHARS )
;
BASE !    ;S
```

```
( FILE NAMING SYSTEM - PHH - 30 NOV 79 ) BASE @ HEX
OF8 CONSTANT DIR ( FILE DIRECTORY BLOCKS START HERE )
: INDX->STR-ADDR             ( INDX ON TOS ON ENTRY )
   20 /MOD                   ( 32 FILENAMES/BLOCK )
   DIR + 1+                  ( NAMES IN BLKS DIR+1,DIR+2 )
   BLOCK                     ( ADDR OF BLOCK W/ NAME IN IT )
   SWAP 5 <-L                ( BYTE OFFSET INTO BLOCK
   +                         ( RTRN ADDR OF NAME STRING ON TOS )
;
: INDX->BLK-PTR-ADDR         ( INDX ON TOS )
   1 <-L                     ( CREATE BYTE OFFSET INTO BLOCK )
   DIR BLOCK                 ( ADDR OF BLOCK WITH FILE POINTERS )
   +                         ( RTRN ADDR OF FILE'S BLOCK PNTR )
;
BASE !    ;S
```

```
( FILE NAMING SYSTEM - PHH - 11 30 79 ) BASE @ HEX
-1 VARIABLE FILE-INDX   -1 VARIABLE FILE-BLK
: FIND-NAME -1 FILE-INDX !  ( SET INDX FOR NO MATCH )
   40 0 DO                  ( CHECK ALL POSSIBLE NAMES )
      I INDX->BLK-PTR-ADDR @ -1 =
                            ( VALID FILE - SO CHECK NAME MATCH )
      IF
         "DUP I INDX->STR-ADDR "GET "=
                               ( NAME MATCH FOUND )
         IF
            I FILE-INDX ! EXIT ( SET INDX AND ESCAPE )
         THEN                  ( OTHERWISE )
      THEN
   LOOP                        ( TRY NEXT NAME ENTRY IF NOT DONE )
   "DROP FILE-INDX @           ( REMOVF TARGET STRING AND ... )
                               ( RETURN THE INDX OF THE STRING )
;
BASE !    ;S
```

```
( FILE NAMING SYSTEM - PHH - 11 30 79 ) BASE @ HEX
: CREATE-NAME                  ( FILE-NAME STRING ON TOS )
   -1 FILE-INDX !              ( SET TO INDICATE NO ROOM AVAIL )
   40 0 DO                     ( SEARCH DIRECTORY FOR NULL FILE )
      I INDX->BLK-PTR-ADDR @ -1 =
      IF                       ( NULL, SO PLACE NAME HERE )
         I FILE-INDX !         ( SAVE INDX WHERE NAME IS SAVED )
         I INDX->STR-ADDR "PUT  ( SAVE FILE'S NAME IN DIR )
         "" UPDATE EXIT        ( NULL STR TO TOSS, AND EXIT )
      THEN
   LOOP                        ( UNTIL MATCH OR END OF DIR )
   "DROP FILE-INDX @           ( DROP TARGET OR NULL STRING, & )
                               ( RTRN INDX OF NEW FILE )
;
BASE !    ;S
```

```
( FILE NAMING SYSTEM - PHH - 11 30 79 ) BASE @ HEX
: DELETE-FILE                  ( DELETE FILE GIVEN BY INDX ON TOS )
   INDX->BLK-PTR-ADDR          ( FIND ADDR OF BLK'S POINTER )
   -1 SWAP !                   ( FLAG DELETION BY -1 BLK PTR )
   UPDATE                      ( FORCE DISK UPDATE )
;                              ( -DELETE ALL DIR ENTRIES )
: INIT-DIRECTORY               ( INDX ALL 64 DIR ENTRIES )
   40 0 DO                     ( DELETE EACH BY INDX )
      I DELETE-FILE
   LOOP
;
BASE !    ;S
```

A Riddle

FORTH Supervisor:	What's the difference between 'ignorance' and 'indifference'?
FORTH Programmer:	I don't know and I don't care.

```
( FILE NAMING SYSTEM - PHH - 11 30 79 ) BASE @ HEX
: GET-BIT-MASK                ( GET BIT MAP INFO FOR BLK# ON TOS )
   DUP
   7 & 1 SWAP <-L             ( GENERATE BIT#, THEN BIT MASK )
   SWAP 3 ->L                 ( GEN. BYTE OFFSET IN BIT MAP )
   300 +
   DIR BLOCK +                ( ADD BIT MAP OFFSET W/IN DIR BLK )
   DUP C@                     ( DUP IT, AND GET ITS VALUE )
   ROT                        ( RTRN BIT MAP ADDR, OLD BIT MAP )
;                             ( BYTE, & BIT MASK ON TOS )
: FREE-BLK                    ( BLK# ON TOS TO BE FREE'D )
   GET-BIT-MASK
   -1 XOR & SWAP              ( MASK BLK'S BIT MAP BIT TO 0 )
   C! UPDATE                  ( STORE BACK IN BIT MAP & TO DISK )
;
BASE !   ;S
```

```
( FILE NAMING SYSTEM - PHH - 12 3 79 ) BASE @ HEX
: RESERVE-BLK                 ( MARK BLK ON TOS AS USED )
   GET-BIT-MASK
   OR SWAP C! UPDATE          ( SET BIT IN BIT MASK )
;
: INIT-BIT-MAP                ( FREE ALL BLKS, THEN RESERVE )
                              ( THE RANGE OF BLKS GIVEN ON TOS )
   DIR 0 DO                   ( FREE ALL BLKS IN DISK )
      I FREE-BLK
   LOOP
   SWAP 1+ SWAP DO            ( RANGE OF BLKS ON TOS )
      I RESERVE-BLK           ( RESERVE ALL BLKS IN THE RANGE )
   LOOP
;
BASE !   ;S
```

```
( FILE NAMING SYSTEM - PHH - 12 5 79 ) BASE @ HEX
: FIND-FREE-BLK               ( SEARCH BIT MAP FOR FREE BLOCK )
   -1 FILE BLK !              ( FLAG RESULT FOR NO BLKS FOUND )
   DIR 0 DO                   ( NOW SEARCH ENTIRE BIT MAP )
      I GET-BIT-MASK & 0=     ( IS BLK IN USE? )
      IF                      ( NO, ... )
         I FILE-BLK ! EXIT    ( SO SAVE BLK#, AND EXIT LOOP )
      THEN
      DROP                    ( BIT MAP ADDR )
  LOOP                        ( TRY THE NEXT BLOCK )
  FILE-BLK @                  ( DONE, SO RETURN THE FOUND BLK )
;                             ( NOTE, -1 => NO BLKS FREE )
BASE !  ;S
```

```
( FILE NAMING SYSTEM - PHH - 12 3 79 ) BASE @ HEX
: NEW                         ( SET UP NEW FILE W/ NAME ON TOSS )
  FILE-NAME-FIX               ( FIRST, FORCE VALID NAME LEN )
  FIND-FREE-BLK DUP -1 =      ( MORE ROOM ON DISK? )
  IF 1 FILE-ERROR THEN        ( NO, ALL BLKS RESERVED )
  "DUP FIND-NAME -1 =         ( NAME ALREADY USED? )
  IF 2 FILE-ERROR THEN        ( YES, GIVE ERROR MESSAGE )
  CREATE-NAME DUP -1 =        ( PUT NAME IN DIR, IF NOT FULL )
  IF 3 FILE-ERROR THEN        ( DIR FULL ERROR )
  SWAP DUP RESERVE-BLK        ( SET NEW BLK, FOUND BY )
                              ( FIND-FREE-BLK, AS RESERVED )
  SWAP INDX->BLK-PTR-ADDR !   ( STORE FILE'S BLK POINTER )
  UPDATE                      ( GO TELL IT TO THE DISK, TOO ! )
;
BASE !   ;S
```

```
( FILE NAMING SYSTEM - PHH - 12 3 79 ) BASE @ HEX
: NEW-FIXED                   ( LIKE 'NEW' EXCEPT BLK POINTER )
                              ( GIVEN BY # ON TOS )
  FILE-NAME-FIX               ( FORCE 32 CHAR LENGTH )
  "DUP FIND-NAME -1 =         ( NAME ALREADY EXIST? )
  IF 2 FILE-ERROR THEN        ( YES, SO GIVE ERROR MESSAGE )
  CREATE-NAME DUP -1 =        ( PUT NAME IN DIR, IF DIR NOT FULL )
  IF 3 FILE-ERROR THEN        ( DIR FULL, SO GIVE ERROR )
  SWAP DUP RESERVE-BLK        ( RESERVE BLK, GIVEN BY # ON TOS )
                              ( ON ENTRY TO 'NEW-FIXED' )
  SWAP INDX->BLK-PTR-ADDR !   ( AND STORE BLK# AS FILE'S PTR )
  UPDATE                      ( GO TELL IT TO THE DISK ! )
;
BASE !   ;S
```

```
( FILE NAMING SYSTEM - PHH - 12 3 79 ) BASE @ HEX
: FILE                        ( XLATE FILE NAME ON TOSS TO BLK# )
   FILE-NAME-FIX              ( FORCE 32 CHAR STRNG LEN )
   FIND-NAME DUP -1 =         ( FIND NAME'S DIR INDX )
   IF 5 FILE-ERROR THEN       ( NAME NOT FOUND IN DIR )
   INDX->BLK<PTR-ADDR @       ( GET NAME'S BLOCK # )
;                             ( AND RETURN ON TOS )
: ERASE                       ( ERASE NAME ON TOSS FROM DIR )
   FILE-NAME-FIX              ( FORCE 32 CHAR STRING LENGTH )
   FIND-NAME DUP -1 =         ( GET NAME'S DIR INDX, IF ANY )
   IF 5 FILE-ERROR THEN       ( NAME NOT FOUND IN DIR )
   DUP DELETE-FILE            ( DELETE FILE GIVEN BY INDX# )
   INDX->BLK-PTR-ADDR @       ( GET THE OLD BLK POINTER )
   FREE-BLK                   ( ...AND FREE IT IN THE BIT MAP )
;
BASE !   ;S
```

```
( FILE NAMING SYSTEM - PHH - 12 3 79 ) BASE @ HEX
: DIRECTORY                   ( PRINT ENTIRE DIRECTORY )
   40 0 DO                    ( CHECK EACH DIR ENTRY )
      I INDX->BLK-PTR-ADDR @ (GET BLK PNTR )
        DUP -1 =              ( IS IT AN EXISTANT FILE? )
        IF                    ( YES, SO PRINT ITS CONTENTS )
           I INDX->STR-ADDR   ( FIRST, GET THE ADDR OF THE NAME )
           "GET ".            ( PUT IT ON TOSS, AND PRINT IT )
           5 PB . CR          ( PRINT 5 BLNKS, AND THE BLK # )
        ELSE DROP             ( BLK NUMBER )
        THEN
   LOOP                       ( CONTINUE FOR ALL POSSIBLE FILES )
;
BASE !   ;S
```

LYONS' DEN

(Continued from pg. 22)

Regarding FORTH this way captures some of the reasons why FORTH should not be used as merely a low level pseudo-machine in the way Wirth used P-Code to implement PASCAL, or as how meta compilers, as opposed to how a meta interpreter works. Of course, any language can be used to write an interpreter, but FORTH provides tools for this purpose built in and is thus pre-structured for that kind of application. This may also suggest—as just a possibility—why there has been observed markedly less use of conditional branches in FORTH programs relative to FORTRAN; perhaps many of the conditionals that would be explicit in FORTRAN are simply performed as executions of the interpreter functions which perform a complex set of conditional branches automatically without having to identify them as such. I will wager LISP is the same way.

George B. Lyons
Jersey City, NJ

TOWERS OF HANOI

by Peter Midnight

Here are the listings of a graphic representation of the ancient Towers of Hanoi puzzle which is adjustable for any CRT terminal with curser addressing.

Recently, when I got fig FORTH running on my system under North Star DOS, I decided to translate this program into FORTH as an exercise and as a comparison between FORTH and PASCAL. In the process I noticed some inefficiencies but chose to translate them more or less directly, for the sake of comparison.

The UCSP PASCAL program is available by requesting the Jan/Feb 1980 News-letter from Homebrew Computer Club, P.O. Box 626, Mountain View, CA 94042.

Forth Program

```
SCR # 12
( TOWERS OF HANOI      Copyright, 1979, Peter Midnight )
( Translated for speed comparison ) FORTH DEFINITIONS DECIMAL
( First extend Forth to include a few features of Pascal )
: MYSELF  ( In definition, this is a recursive use of new
   LATEST PFA CFA , ; IMMEDIATE                          word )
: GOTOXY    ( X Y GOTOXY )    27 EMIT  61 EMIT
  0 MAX  15 MIN  32 +  EMIT   0 MAX  53 MIN  32 +  EMIT  ;
: CLEARSCREEN  12 EMIT ;
: 2DROP  DROP DROP ;
: PICK  SP@  SWAP  2 *  +  @  ;
: 4DUP  4 PICK  4 PICK  4 PICK  4 PICK ;
   10 CONSTANT NMAX  ( maximum permisable number of rings )
NMAX VARIABLE (N)  : N (N) @ ; ( formerly a constant )
   0 CONSTANT HELL_FREEZES_OVER   43 CONSTANT COLOR ( + )
   0 VARIABLE RING   N 2 -  ALLOT  ( array [1..N] of bytes )
  -->
```

```
SCR # 13
( TOWERS OF HANOI      Copyright, 1979, Peter Midnight )
: DELAY   ( centiseconds DELAY )
  0 DO  17 0 DO  127 127 * DROP  LOOP  LOOP  ;
: POS     ( location POS -> coordinate )
  2 N *  1+  *  N +  ;
: HALFDISPLAY   ( color size HALFDISPLAY )
  0 DO   DUP EMIT   LOOP   DROP  ;
: <DISPLAY>   ( line color size <DISPLAY> )
  2DUP HALFDISPLAY   ROT 3 <  IF BL  ELSE 124 ( | )
  THEN EMIT   HALFDISPLAY  ;
: DISPLAY   ( size pos line color DISPLAY )
  SWAP >R ROT ROT OVER - R ( color size pos-size line )
  GOTOXY   R> ( color size line )  ROT ROT <DISPLAY> ;
-->


```

```
SCR # 14
( TOWERS OF HANOI      Copyright, 1979, Peter Midnight )
: PRESENCE   ( tower ring PRESENCE -> boolean )
  RING +  C@  =  ;
: LINE        ( tower LINE -> display_line_of_top )
  4 SWAP  N 0 DO  DUP I PRESENCE 0=  ROT + SWAP  LOOP DROP ;
: 1-   1 - ;

: RAISE   ( size tower RAISE )
  DUP   POS  SWAP  LINE   1  SWAP  DO
  2DUP I BL DISPLAY   2DUP  I 1-  COLOR DISPLAY
  -1 +LOOP   2DROP  ;
: LOWER   ( size tower LOWER )
  DUP   POS  SWAP  LINE 1+  2 DO
  2DUP  I 1-  BL DISPLAY   2DUP I COLOR DISPLAY
  LOOP  2DROP  ;
-->

MSG # 15
```

```
SCR # 15
( TOWERS OF HANOI      Copyright, 1979, Peter Midnight )
: MOVELEFT  ( size source_tower destiny_tower MOVELEFT )
  POS 1-  SWAP POS 1- DO  DUP  R 1+  1  BL  DISPLAY
  DUP R 1 COLOR DISPLAY   -1 +LOOP   DROP  ;
: MOVERIGHT ( size source_tower destiny_tower MOVERIGHT )
  POS 1+  SWAP POS 1+ DO  DUP  R 1-  1  BL  DISPLAY
  DUP R 1 COLOR DISPLAY     LOOP     DROP  ;
: TRAVERSE  ( size source_tower destiny_tower TRAVERSE )
  2DUP >    IF MOVELEFT  ELSE MOVERIGHT   THEN  ;
: MOVE    ( size source_tower destiny_tower MOVE )
      ?TERMINAL IF  0 N 4 + GOTOXY  ABORT  THEN
  ROT ROT 2DUP RAISE   >R 2DUP R> ROT TRAVERSE
  2DUP  RING + 1-  C!    SWAP LOWER  ;
-->


```

```
SCR # 16
( TOWERS OF HANOI      Copyright, 1979, Peter Midnight )
: MULTIMOV  ( size source destiny spare MULTIMOV )
  4 PICK  1 =  IF DROP MOVE     ELSE
  >R >R SWAP 1- SWAP R> R>      4DUP SWAP MYSELF
  4DUP DROP   ROT 1+ ROT ROT   MOVE
  ROT ROT SWAP   MYSELF    THEN  ;

: MAKETOWER   ( tower MAKETOWER )
  POS  4 N +  3 DO  DUP I GOTOXY  124 EMIT ( | )  LOOP DROP ;
: MAKEBASE   ( no arguments )
  0 N 4 + GOTOXY  N 6 * 3 +  0 DO  45 EMIT ( - )  LOOP ;
: MAKERING   ( tower size MAKERING )
  2DUP RING + 1- C!   SWAP LOWER ;
: SETUP   ( no arguments )   CLEARSCREEN
  N 1+ 0 DO  1 RING I + C!  LOOP   3 0 DO   I MAKETOWER  LOOP
  MAKEBASE   0 N DO  0 I MAKERING  -1 +LOOP  ;   -->
```

```
SCR # 17
( TOWERS OF HANOI      Copyright, 1979, Peter Midnight )
: TOWERS   ( quantity TOWERS )
  1 MAX  NMAX MIN  (N) !
  SETUP   N 2 0 1    BEGIN
  OVER POS  N 4 +  GOTOXY   N 0 DO  7 EMIT  50 DELAY  LOOP
  ROT   4DUP   MULTIMOV
  HELL_FREEZES_OVER UNTIL  ;

;S

( Results:  DELAY runs much slower in Forth than in Pascal.
  But the rest of the program is over twice as fast in Forth!

  Note that CLEARSCREEN and GOTOXY are terminal dependant.
  NMAX should be 10 for 16x64 or 12 for 24x80 screens. )


MSG # 15
```

Thanks to "THE I/O PORT", the Official Newsletter of the Tulsa Computer Society, for the feature article on FORTH by Art Sorski in their April 1980 issue. Address: The Tulsa Computer Society, P.O. Box 1133, Tulsa, OK 74101.

LETTERS

I'd like to take this chance to accomplish several aims. First, let me congratulate Roy Martens and the entire editorial staff for a fine publication in FORTH DIMENSIONS.

My interest in FORTH is far from passive; I have been using the University of Rochester's (my employer, by the way) URTH dialect for several years now. While at first I used it mainly at home for a private music synthesizer research project, I have more recently been applying it with success to several laboratories within the University's Medical Center. The applications have primarily been concerned with slow speed (10 to 100 samples per second) analog data acquisition and analysis - the latter involving the use of the AMD 9511 IC for number crunching (and it is fast ...!). These data acquisition systems have been described in an article which I just recently submitted to BYTE for publication (I hope).

While using FORTH in these applications, I have developed a set of goals for the elimination of some of the limitations of FORTH (there are some, you know ...). One of the major problems has been saving only three characters plus the length for identifiers; I have just recently implemented changes to adopt (in URTH) the FIG standard. Using primarily S-100 hardware, I am also now implementing a hardward debug facility for FORTH which allows easier program development. The design is very simple, but allows traps at instructions, memory references, and/or I/O references. I consider this method of debugging immeasurably more useful than just software trapping at each pass through NEXT.

Additional FORTH changes planned are the implementation of a random block text file system with variable record length and blanks compaction. I feel that this system will make it easy to write programs in a more readable format since this better formatted text will use less space than the current block oriented text editors. Thus there will be less of a temptation to use a short, cryptic coding style. My method of blanks compaction is to use the MSB of each text character to flag a compaction count byte. When listing a program in the editor, the compacted blanks can be re-expanded while they can be interpreted as blanks (due to changes in the WORD routine in URTH) when loading the text. Text will be stored on disk blocks as an integral number of lines of text per block with each line being defined as 0 or more characters followed by a carriage return character.

Text will be able to span multiple random blocks to avoid any "artificial" program length constraints due to fixed block size. Blocks are associated together via a doubly linked (forward and backward) pointer scheme while block usage is kept track of via a bit map (more on this later) corresponding to the disk's block utilization. So far the text editor has been written, but not fully debugged. However, the bit map and filing name system has been written and used for several months. I'd like to discuss them here as the type of entity which should be standardized for FIG FORTH usage. Let me try to motivate this building of file structures by analogy to building data structures in FORTH.

IN FORTH (or at least URTH) one can use some system features to define any arbitrary data structure. One which I've used recently is:

```
: IPARAM <BUILDS  2 ALLO7 DOES >
```

which might be used:

```
IPARAM MY-VIRTUAL-INTEGER
```

The important things to notice in this example are that the IPARAM data type first uses standard dictionary features to add new specific variables - in this case MY-VIRTUAL-INTEGER - to the dictionary. IPARAM also sets aside some dictionary space - in this case just one word - to store data for MY-VIRTUAL-INTEGER. Thus there are two important actions here - that of linking a variable's name into the dictionary, and that of reserving dictionary space for a variable's storage requirements.

The file system that I have been evolving also achieves two analogous actions to those above. First, it has a way of linking a file's name into a diskettes name directory, and second, it has a way of reserving disk block space for a file's sole use. Note, that it does not concern itself in any manner with how the file is logically formatted. As such, it is not a complete file management system, but only a common protocol for various logical file structures!

Let me explore two uses of file types built on this foundation. The previously mentioned text file system logically builds a file structure by the use of doubly linked random blocks. But in another case, the file is logically built up as an array of consecutive integers in consecutive disk blocks - thus linked only implicitly. Other logical structures are as diverse as are FORTH data types.

In summary, what I am proposing to be discussed and hopefully standardized is a common structure which can be used to name files and reserve disk space for files. I am not suggesting any specific file structures or formats for standardization. I am enclosing a copy of the source listings and some (hastily written) documentation for this file system so that it might stimulate comments and improvements from the public domain.

Thanks very much, and keep up the good work....

Peter H. Helmers
University of Rochester
Rochester, N.Y.

In December I got tired of waiting and implemented FORTH-65 from the fig-FORTH model. By the end of December I had it up and running. This version follows the model exactly except for printer control, the disk kinkage, and the inner interpreter.

The jump indirect in the inner interpreter doesn't always work, JMP ($XXFF) doesn't work correctly on a 6502. If a CFA ends in $FF it's goodbye FORTH.

This bus bit after my third re-assembly of FORTH-65. The inner interpreter I'm now using is considerably slower (60 cycles) but it is reliable.

I assembled FORTH-65 through the disk I/O (SCR #69), Screens 72 through 92 reside on disk and are compiled as needed. What I need now is the ASSEMBLER vocabulary. Has anyone done any work on a FORTH assembler for the 6502?

```
SCR #44
( RANDOM NUMBER GENERATOR E )

DECIMAL

0 VARIABLE SEED

: (RAND)   SEED @ 259 * 3 + 32767 AND DUP SEED ! ;

: RANDOM   (RAND) 32767 */ ;             (RANGE -1  )

;S





```

J.E. Rickenbacker
Houston, TX