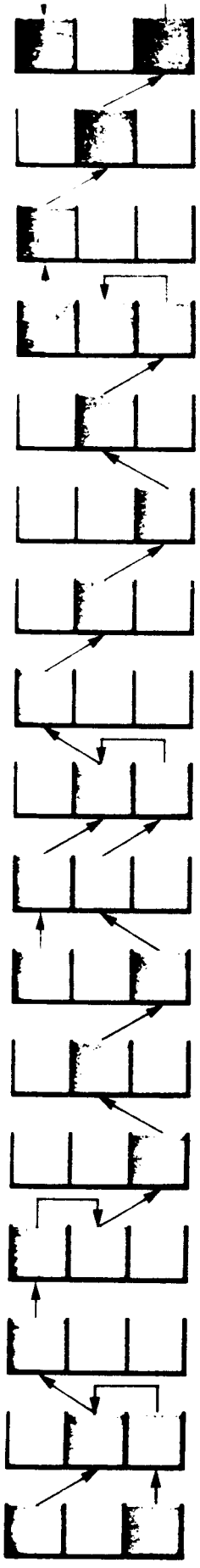


FORTH DIMENSIONS

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

Volume II
Number 3
Price $5.00

INSIDE

FORTH DIMENSIONS

Published by Forth Interest Group

Volume II No. 3 September/October 1980

Publisher Roy C. Martens

Editorial Review Board

Bill Ragsdale
Dave Boulton
Kim Harris
John James
Dave Kilbridge
George Maverick

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 worldwide. 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

Busy, busy, busy. That's what its been for the last couple of months. Here are some of the things that have been happening.

1. We've reorganized the order processing for the mail order items listed on the last page. The volume has increased so much this year that we had gotten several months behind. Now, its all being handled at one location and we even have a phone number for checking on your orders (415) 962-8653. If you have technical questions DO NOT CALL, write to the box number so that your request can be routed to the most helpful person.

2. We can now take VISA and Master Charge orders by mail and by phone, (415) 962-8653. The charge on your monthly statement will be listed as "Mt. View Press". This was done because FIG isn't set up to handle charges. We still aren't ready to handle purchase orders or delayed billings.

3. The August issue of Byte magazine has put FORTH ON THE MAP. We are receiving 50-60 orders and requests for information a day. We have a supply of the issue and can furnish them to you (see the Mail Order form on the last page).

4. This issue of FORTH DIMENSIONS has 60 pages and includes all the CASES that were submitted. Don't get your hopes up for more FD's this long. Next month we go back to our regular size. Congratulations to all entrants!

5. Two events are coming up soon. The FORML Advanced Conference will be held November 26-28, 1980 at Asilomar Conference Grounds, CA. The National FORTH Interest Group Convention will be held on November 29, 1980 at San Mateo, CA. See Page 94 for more information and to register.

My thanks to the Judges and Editorial Review Board for all the help they have given me on this BIG issue. Without their assistance, much of it done late at night, you wouldn't be reading this issue for months to come. Many thanks!

Roy Martens

==CASE CONTEST CLOSES==

This issue of FORTH DIMENSIONS is another special issue, chiefly devoted to FIG's CASE Statement Contest. The contest, announced in FD I/5, Jan./Feb. 1980, brought entries from sixteen individuals and teams, showing a high level of interest and activity among the membership.

All the entries are published here. They show imaginative thinking and hard work, and illustrate the many different ways that FORTH allows the user to implement a single concept. Although no one entry seemed to get it all together, many show some very good work.

Our panel of judges did not settle on a single winner, but instead have decided that the prize will be shared among three entries. These are Dr. Charles E. Eaker, Steve Munson, and the team of Karl Bochert and Dave Lion.

Each of these winners will receive a $50 prize and a one year subscription to Infoworld. The high interest in the contest has justified increasing the overall prize from the $100 announced (including $50 contributed by FORTH, Inc.) to $150. Infoworld kindly donated the subscriptions.

Eaker's entry is particularly well organized, and has a clear, readable writeup. He implemented a keyed CASE statement, and uses non-obvious words. (See below for the difference between positional and keyed CASE statements.)

Munson put so much thought into the contest that he included several versions, differing in the type of data that keys the CASE statement, and in keyed versus positional ordering of cases.

Bochert and Lion submitted a neat positional entry. It includes the ability to alter the binding of cases to case bodies after compile time.

The judging was based on a variety of factors:

1. the approach taken, including degree of generality;

2. the success and efficiency of the implementation, e.g., a minimum of computation and dictionary use should be left to execution time;

3. FORTH-like style, including good documentation on the screens;

4. overall prose description, together with an evaluation of the advantages and limitations of the approach or implementation;

5. adequacy and clarity of examples.

However, the judging did not involve loading and testing the entries on a running FORTH system.

The judges felt that most entrants were not getting close enough to what is possible in FORTH. They seemed to think along narrow lines. A general CASE implementation should be efficient both for the positional case (where the values tested are restricted to the first N integers, for example, similar to FORTRAN's computed GO-TO), and for the general "keyed" case, where a value, not necessarily an integer, is tested against a sequence of explicit values. Very few people tried to solve both.

This collection of contest entries make this issue of FD an excellent source for the comparative study of implementation techniques. Interested FORTH students should read each entry to pick up helpful techniques and evaluate style. (Caution: Any entry may also show poor techniques and weak style.)

Forth Dimensions welcomes more contributors.

JUST IN CASE

Dr. Charles E. Eaker

Even though FORTH provides a variety of program control structures, a CASE structure typically has not been one of them. There is no particular reason for this since, as we shall soon see, it is not difficult to implement one.

There are two different approaches one can take to implementing a CASE structure: vectored jumps and nested IF...ELSE...THEN structures. Vectored jumps provide the greatest speed at run-time but produce enormous compiling complications. So, taking the path of least resistance, here is a proposal for implementing a CASE structure for FORTH which is really just a substitute for nested IF structures. But, even though the proposal is logically redundant, there are a number of practical benefits which make it worthy of consideration.

To help this discussion, consider a word which might appear in an assembler vocabulary with a glossary entry as follows:

GEN operand, opcode, mode selector ---

Used by the ASSEMBLER vocabulary to generate opcodes. 'Mode selector' is the value which indicates which addressing mode has been specified. 'Opcode' is the value placed on the stack by the preceding mnemonic, and 'operand' is the value to be used as the argument of the opcode.

Here is one way of coding GEN.

```
: GEN   0 OVER =
   IF DROP IMMEDIATE
   ELSE 10 OVER =
    IF DROP DIRECT
    ELSE 20 OVER =
     IF DROP INDEXED
     ELSE 30 OVER =
      IF DROP EXTENDED
      ELSE DROP MODE-ERROR
      ENDIF
     ENDIF
    ENDIF
   ENDIF   RESET ;
```

GEN is defined to expect a 16-bit number on top of the stack. For each IF, this number, the "select value," is copied and tested against a constant, the "case value." If the select value equals the case value the appropriate code is executed. If all tests fail, MODE-ERROR is executed. Notice that GEN meticulously keeps the stack clean.

Depending on the select value, some action is performed on the opcode and operand, and GEN removes them from the stack. Consequently, before each test, GEN must copy (OVER) the select value, and if the test is successful, the select value must be dropped from the stack to expose the data values prior to the appropriate routine being called.

But wouldn't you rather code this thing this way?

```
: GEN  CASE
          0 OF    IMMEDIATE      ENDOF
         10 OF    DIRECT         ENDOF
         20 OF    INDEXED        ENDOF
         30 OF    EXTENDED       ENDOF
        MODE-ERROR
     ENDCASE     RESET ;
```

It is certainly easier to see what this routine is doing, so comments are not as necessary, and changes and repairs are far easier to do. Here are the required colon definitions of CASE, OF, ENDOF, and ENDCASE.

```
CASE      ?COMP   CSP @  !CSP   4  ;  IMMEDIATE

OF  4 ?PAIRS  COMPILE OVER  COMPILE =  COMPILE 0BRANCH
        HERE 0 ,    COMPILE DROP   5  ;  IMMEDIATE

ENDOF     5 ?PAIRS    COMPILE BRANCH    HERE 0 ,
        SWAP  2 [COMPILE] ENDIF    4  ;  IMMEDIATE

ENDCASE   4 ?PAIRS    COMPILE DROP
        BEGIN   SP@  CSP @  =  0=
          WHILE  2 [COMPILE] ENDIF   REPEAT
        CSP !  ;  IMMEDIATE
```

It so happens that with these definitions both versions of GEN compile the identical code into the dictionary. Let's look at the compiling details.

CASE makes sure that it is in a colon definition. Then it saves the value of CSP (which contains the position of the stack at the beginning of this case structure) and sets CSP equal to the present position of the stack. The new value of CSP will be used later by ENDCASE to resolve forward references. Finally, it throws a four onto the stack which will be used for checking syntax. CASE compiles no code into the dictionary.

OF first checks that it has been preceded either by CASE or an ENDOF. If the syntax is in order, then code is compiled into the dictionary to duplicate the select value (OVER) and test its equality to the current case value (=). Next, code for a conditional branch is compiled into the dictionary followed by code for DROP. Notice that at run-time the DROP is executed only if the select value equals the constant for this OF...ENDOF pair.

ENDOF first checks that an OF has gone before. If so, then it compiles an absolute branch to whatever code follows ENDCASE. However, the address to branch to is not yet known, so a dummy null is compiled into the address and its location is left on the stack so ENDCASE will know where to stick the address once it is known. But there is already an address on the stack just under the one which ENDOF just pushed. This address was left by OF and it points to an address that should hold a branch address to the code which follows the code generated by ENDOF. So, ENDOF swaps the addresses and calls ENDIF to resolve the address at the address left by OF. Finally, ENDOF leaves a four on the stack for syntax checking.

ENDCASE makes sure it has been preceded by either a CASE or ENDOF. Otherwise an error message is issued and compilation is aborted. Code for a DROP is compiled into the dictionary, then all the unresolved forward branches left by each ENDOF are resolved. Since there may be any number of them, including none, ENDCASE checks the current stack position against what it was when CASE was executed, and performs a fixup by calling ENDIF until the stack no longer contains addresses left by previous ENDOF's. Notice that all of these branches are resolved to point to the code after the DROP generated by ENDCASE. In the case of GEN this is RESET.

It doesn't take long to notice that OF generates an enormous amount of code (10 bytes). This is a classic example of a situation that cries out for a machine language primitive. If a run-time word could be defined, let's call it (OF), then each OF would generate just 4 bytes two to point to (OF) and two for the branch address. What (OF) would have to do is pull the top stack item (the current case value) and test it for equality with the new top stack item (the select value) If the test for equality is true then the next item on the stack the select value is also popped and execution continues after the (OF) If the test is false execution branches using the

branch value following the pointer to (OF), and the select value is left on the stack.

```
CODE  (OF)  A PUL   B PUL   TSX
            1,X B SUB   0,X A SBC  ABA  0=
            IF  INS INS   ' BRANCH CFA @ 6 + ! 11 + JMP
            THEN   ' BRANCH CFA @ JMP
: OF  4 ?PAIRS  COMPILE (OF) HERE 0 , 5 ; IMMEDIATE
```

The M6800 code listed above is straightforward except that is uses code in BRANCH and 0BRANCH. (OF) should work in any FIG 6800 installation provided BRANCH and 0BRANCH have not been altered (it doesn't matter where they are located). Non-6800 users will have to roll their own, but the high-level OF should make it clear what has to be done.

The disadvantages of this CASE proposal are that execution is not as fast as a vectored implementation, and in some versions of FORTH, ENDOF and ENDIF cannot be distinguished. These seem minor compared to the advantages - and there are several.

First, a CASE statement may contain any number of OF...ENDOF pairs, and the constants may be arranged in any order whatever. Actually the constants need not be constants. Between an ENDOF and the next OF the programmer may insert as much code as he or she likes including code which will compute the value of the "constant." CASE statements may be nested; a CASE...ENDCASE pair may appear between an OF...ENDOF pair. Furthermore, there need not be any code between CASE and ENDCASE, nor must there be code between OF and ENDOF. There must be code which pushes a 16-bit number to the stack prior to each OF. Finally, this proposal follows the fig-FORTH style of handling control structures.

fig-FORTH GLOSSARY

CASE --- addr n (compiling)

Used in a colon definition in the form: CASE...OF...ENDOF...ENDCASE. Note that OF...ENDOF pairs may be repeated as necessary.

At compile-time CASE saves the current value of CSP and resets it to the current position of the stack. This information is used by ENDCASE to resolve forward references left on the stack by any ENDOF's which precede it. n is left for subsequent error checking.

CASE has no run-time effects.

OF --- addr n (compiling)
nl n2 --- nl (if no match)
nl n2 --- (if there is a match)

Used in a colon definition in the form: CASE...OF...ENDOF...ENDCASE. Note that OF...ENDOF pairs may be repeated as necessary.

At run-time, OF checks nl and n2 for equality. If equal, nl and n2 are both dropped from the stack, and execution continues to the next ENDOF. If not equal, only n2 is dropped, and execution jumps to whatever follows the next ENDOF.

At compile-time, OF emplaces (OF) and reserves space for an offset at addr. addr is used by ENDOF to resolve the offset. n is used for error checking.

ENDOF addr1 nl --- addr2 n2 (compiling)

Used in a colon definition in the form: CASE...OF...ENDOF...ENDCASE. Note that OF...ENDOF pairs may be repeated as necessary.

At run-time, ENDOF transfers control to the code following the next ENDCASE provided there was a match at the last

OF. If there was not a match at the last OF, ENDOF is the location to which execution will branch.

At compile-time ENDOF emplaces BRANCH reserving a branch offset, leaves the address addr2 and n2 for error checking. ENDOF also resolves the pending forward branch from OF by calculating the offset from addr1 to HERE and storing it at addr1.

```
ENDCASE addr1...addrn n --- (compiling)
        n ---      (if no match)
          ---      (if match was found)
```

Used in a colon definition in the form: CASE...OF...ENDOF...ENDCASE. Note that OF...ENDOF pairs may be repeated as necessary.

At run-time, ENDCASE drops the select value if it does not equal any case values. ENDCASE then serves as the destination of forward branches from all previous ENDOF's.

At compile-time, ENDCASE compiles a DROP then computes forward branch offsets until all addresses left by previous ENDOF's have been resolved. Finally, the value of CSP saved by CASE is restored. n is used for error checking.

```
(OF) n1 n2 --- n1 (if no match)
     n1 n2 ---    (if there is a match)
```

The run-time procedure compiled by OF. See the description of the run-time behavior of OF.

Dr. Charles E. Eaker
Department of Philosophy
State University of New York
Oswego, NY 13126

Judges' Comments -

This is an excellent development and presentation of a key case statement with single integer keys. The following features make it immediately useful:

1. The reader can easily understand what the statement does and how to use it. There are only four words to learn, their functions are immediately clear from the example presented and their names are not confused with each other. (The ENDOF - ENDIF similarity will go away when the FIG model drops ENDIF in favor of the Standards Team decision to use THEN.)

2. One form of the statement can be entered entirely in higher-level fig-FORTH, and run immediately on any FIG system. An optional code word (for 6800) with redefinition of one of the four higher-level words saves run-time memory and time. Either way, the whole statement fits easily on one screen, including compile-time checking.

3. The narrative documentation is excellent. The glossary definitions are detailed (appropriate for this forum). For general distribution they could be condensed to user-only information.

This entry presents one kind of case statement out of several that are desired. We hope that this competent and straightforward work will serve as a model to future development.

COME TO FIG CONVENTION NOVEMBER 29

THE : CASE

Steve Munson

Having grown up on an ancient version of FORTH Inc. micro FORTH, I can appreciate the improvements rendered by fig-FORTH's renames and redefinitions. I was particularly impressed by the source equivalence of HERE NUMBER DROP which functions the same although in one case one is dropping the address of the first non-numeric delimiter, and in the other case one is dropping the most significant half of a double precision number!

My one beef is why was : made IMMEDIATE? Surely nobody wants a header in the middle of a colon definition. By the way, as you probably already know, this tends to mask an error in the definition of ; on the listing I have for the 6502 fig - FORTH. There is no [COMPILE] before the [which means compile mode is never terminated. In fact, I am not sure I see the point of the E property in your glossary. All words ought to be designed, at great pains if necessary, so that they can be compiled. My definition of CASE denies the E property of :, and I would be rash to assume no one would ever want to compile CASE.

Please find enclosed a listing, documentation, glossary entries, and a diskette. The diskette also contains the assembler used to generate the code, as it may be nonstandard If the fig-FORTH does not run on your system as it does on mine, feel free to edit my ideas into polished fig-FORTH (I am a novice figger) and re-list the screens; however I believe they will require no modification.

GLOSSARY ENTRIES

2BYTECASE

Keycase defining word, used in the form:

2BYTECASE cccc key_0 $case_0$ key_1 $case_1$ . . . key_n $case_n$ (default case) END-CASE. Defines cccc as a caseword which expects a 2-byte key on the stack at run-time. If the key equals key_0 (a 2-byte key), $case_0$ (a previously defined word) will execute; if it matches key_1, $case_1$ will execute, and so on. The default case will execute on no match; if no default is specified, NOOP is assumed. Cases may be IMMEDIATE words, but no compile-time execution will occur within the case structure. The structure must be terminated by END-CASE. (See END-CASE, BYTECASE).

BYTECASE

Keycase defining word, used in the form:

BYTECASE cccc key_0 $case_0$ key_1 $case_1$ . . . key_n $case_n$ END-CASE. Defines cccc as a caseword which expects a 1-byte key (most significant byte is ignored) on the stack at run-time. If the key equals key_0 (a 1-byte key), $case_0$ (a previously defined word) will execute; if it equals key_1, $case_1$ will execute, and so on. The default case will execute on no match; if no default is specified, NOOP is assumed. Cases may be IMMEDIATE words, but no compile-time execution will occur within the case structure. The structure must be terminated by END-CASE. (See END-CASE, CASE).

CASE

Case defining word, used in the form:

CASE cccc case$_0$ case$_1$ . . . case$_n$ (default case) END-CASE. Defines cccc as a caseword which expects a 1-byte key (most significant byte is ignored) on the stack at run-time. If the index is 0, case$_0$ (a previously defined word) will execute; if index is 1, case$_1$ executes, and so on. NOOP cases must be inserted for unused values of the index; index limit is 65,535. No protection is made for out-of-range indices or stack underflow. CASE remains in compile mode (by calling :) until terminated by END-CASE. (See END-CASE).

DO-2BYTECASE P, C +

Compiles a reference to the run-time procedure of the same name, a two-byte "exit address", a one-byte case count and case structure identical to that of 2BYTECASE, all inline within a colon definition. Used in the form: : cccc optional words DO-2BYTECASE key$_0$ case$_0$ key$_1$ case$_1$ . . . key$_n$ case$_n$ (default case) END-CASE optional words ; . The keys, cases and run-time activity are exactly as described for 2BYTECASE. (See 2BYTECASE, END-CASE).

DO-BYTECASE P, C +

Copy glossary entry above, substituting BYTECASE for 2BYTECASE everywhere.

DO-CASE P, C +

Compiles a reference to the run-time procedure of the same name, a two-byte "exit address", and a case structure identical to that of CASE, all inline within a colon definition. Used in the form: : cccc optional words ; . The cases and run-time activity are exactly as described for CASE. (See CASE, END-CASE).

DO-STRINGCASE P, C +

Copy glossary entry for DO-2BYTE-CASE, substituting STRINGCASE for 2BYTECASE everywhere.

END-CASE P

Universal caseword delimiter. It has no run-time activity, but at compile-time it may fill in an "exit address" (inline caseword), and/or a case count (keycaseword), or terminate compile mode (CASE, inline CASE for CODE definitions).

STRINGCASE

Keycase defining word, used in the form:

STRINGCASE cccc key$_0$ case$_0$ key$_1$ case$_1$ . . . key$_n$ case$_n$ (default case) END-CASE. Defines cccc as a caseword which expects a byte-string beginning at HERE 1+ with a count of them at HERE (typically fetched by WORD) at run-time. If the string equals key$_0$ (any byte-string of 1 to 255 characters), case$_0$ (a previously defined word) will execute; if it equals key$_1$, case$_1$ executes, and so on. The default case will execute on no match; if no default is specified, NOOP is assumed. Cases may be IMMEDIATE words, but no compile-time execution will occur within the case structure. The structure must be terminated by END-CASE. (See END-CASE).

Explanation of Screens by Number

100-102: To enable the loading of a screen, delete the leftmost parenthesis. For all screens above 108, 109 must be loaded. Some screens load others that they require, hence loading all screens will cause some to be loaded twice. If it is not desired to load all the examples, edit DECIMAL ;S on the same line of any screen in which the word (EXAMPLE) appears on a line by itself.

103: END-CASE is an example of a terminator (or a leader, or any structure) that is common to all members of some group (in this instance, casewords). The structure can be identical for all members of the group only because it behaves slightly differently to each of them. END-CASE accomplishes this by following a binary tree. At each node a flag variable is tested and code common to the branch taken is executed. All members of the group (each caseword) must set or reset the flag variables that must be tested to complete the execution of all their compile-time code.

END-CASE must be expanded for each new class of casewords that use it as a common compile-time terminator. This is done by creating a new flag variable that is 1 only for members of the new class. The affected casewords are then amended to set or reset this variable (at compile-time) depending on their membership in the class, and the new variable is tested in END-CASE. So far, I have included only two classes: the unique, indexed CASE, and the keycasewords. Each is further sub-divided into defining word and inline forms. Note that STATE can serve as a flag for this distinction, providing that the case defining word executes outside of a colon definition, and the inline form does not. Instead of using a binary tree (nested IF tests) with a new flag variable required for each branch, consider using a caseword inside END-CASE, itself, to accomplish an n-way branch based on the value of a single variable!

105: CASE is the simplest form of n-way branch. It compiles a string of consecutive codefield addresses (CFA's) exactly like the parameter field of a colon definition. The : on line 3 creates the header and sets compile mode, END-CASE terminates compile mode. Whereas the CFA's in a colon definition execute sequentially, only one CFA will execute each time a CASE is called. It expects an index on the run-time stack; if it is 0 the first CFA executes, if is 1 the second CFA executes, and so on. No protection is made for out-of-range indices. Credit for the basic form of CASE goes to J. B. Weems, also of Hughes Aircraft, Fullerton.

106-107: Each caseword is presented in three forms: a ;CODE defining word, a <BUILDS DOES> defining word, and an inline version. The inline version is perhaps closest to ordinary usage, the <BUILDS DOES> defining casewords are machine independent and easiest to modify, and the ;CODE defining casewords are, in all cases, the fastest. This is because they take advantage of the available system pointer W (which is set by NEXT) in order to index into the parameter field of the case structure; whereas the inline casewords must move IP beyond the case structure after using it to

select a case. Note that the inline casewords are not defining words, and so do not require an auxiliary name for the case structure.

The method of putting the CFA to be executed into W and jumping to the last half of NEXT (which fetches the code address and puts it into the program counter), is based on the word EXECUTE as a model. The "NEXT 6 + JMP" used here is source truncation for space purposes. It assumes that no insertions are made in the beginning of NEXT (an insertion in NEXT might be forgivable if short and forbidden to execute at run-time, or if turned on momentarily by an EXEVAR). In such a case, the safe thing (and in any case, the fast thing) to do is to copy the code for the last half of NEXT (however it appears on your machine), rather than jumping to it.

108: A curious hybrid of high-level inside a CODE definition. DO-CASE is really a macro that compiles code similar to that executed by the inline version. Note that if the stack is 0, and DO-CASE executes one of the cases, execution will not return to 3TEST, but to the word calling 3TEST (that is, the HPUSH JMP will not execute). There is no danger of name confusion because the two DO-CASE's are in separate vocabularies.

109-110: A keycase is so called because it requires a key associated with each CFA in the case structure. A key of the same type must be supplied at run-time. If a matches a key in the caseword, the associated CFA will be executed. Unless a match is guaranteed, a default CFA is required which is executed on no match. The default may be a NOOP, a pop of the parameter stack, or even a link to another caseword. The default case is optional, if none is specified, ,CFA compiles a reference to NOOP, automatically.

The structure of a keycaseword is as follows:

COUNT KEY_0 CFA_0 KEY_1 CFA_1
default
. . . KEY_n CFA_n CFA

Where count is the number of cases (default excluded), and CFA_0 is the CFA that will be executed if the run-time key matches KEY_0. The count is not supplied by the programmer; it is determined automatically by ,KEYCASE by counting the number of cases till END-CASE at compile-time. The HERE 0 C, on line 13 reserves space for the count, and it is filled in by END-CASE. The 1+ after the BEGIN on line 14 is incrementing the case count on the stack. The compiled count will be picked up at run-time to become a DO LOOP index. When the index runs to 0, it indicates that the list of cases is exhausted, and the defualt address is to be followed.

Just under this count, on the stack, is a flag that indicates whether the programmer has not supplied a default case. It starts at 0 on line 13, may be changed to 1 by line 4, and is tested on line 11.

All of the keycasewords, as written, reserve only one byte for the count of the number of cases. Hence, one is limited to 255 cases per case structure (0 is not allowed, either). However, keys need not be consecutive or ordered in any fashion, as are the indices for CASE. Keys may be 1, 2, or n bytes depending on the kind of caseword; CFA's are 2 bytes.

In addition, inline casewords compile a 2-byte address in front of the count. This "exit address" points to the first byte beyond the end of the case structure. This address is put into IP so that execution may resume after a case has executed. Case defining words do not need this because IP already points correctly; W is used to scan the case structure.

The 1 = IF on lines 1 and 9 is testing for NULL (alias X). NULL cannot perform its usual function of resetting IN, incrementng BLK, and terminating the loading of a screen. The reason is that ,CFA uses ' CFA , which is capable of compiling a reference to even an IMMEDIATE word. This has the advantage that an IMMEDIATE word can be called as a case, but no compile-time execution is permitted in the middle of a case structure. Lines 1 and 9 perform part of the definition of NULL if one is detected. Not that the test assumes 8080 byte order; on some machines, the test for NULL would be 0100 = IF. If, on your system, a block equals a screen, all the testing for NULL may be deleted.

,KEYCASE is designed so that one or more keys, CFA's, default, or END-CASE may be on any given line. A key need not even be on the same line as the associated CFA. Do not skip lines in the middle of a case definition. Keys and CFA's must alternate, the exception is the default CFA which has no key.

,XKEY on line 7, is a dummy which is called by ,KEYCASE where it reserves 2 bytes which will be filled in at compile-time when a particular caseword executes (lines 8 and 9 on screen 111, for example). The CONSTANT ,XKEY-ADDR defined on line 6 is set to point to the two bytes reserved in ,KEYCASE so that a reference to a ,KEY appropriate to a given keycaseword may be stored there (by !KEY , for example). A more elegant solution, beyond the scope of this document, would be to make ,XKEY an EXEVAR , a variable whose value is assumed to be a CFA, and which is executed rather than fetched. !1KEY , !2KEY , etc., would then be used to set the EXEVAR to ' ,1KEY CFA or ' ,2KEY CFA.

NOWSAVE and RECOVER are needed because by the time END-CASE is encountered, one has typically already compiled the default case as a key instead of a CFA. This is because it breaks the pattern of KEY CFA , KEY CFA. And in any case, in order to recognize END-CASE, we must advance the input pointer beyond it, and it is convenient to restore it so that END-CASE can execute and perform its compile-time activity. RECOVER is, then, a way to un-compile and un-interpret what has been done.

The endless loop of line 14 is terminated by the R> DROP on line 10 when END-CASE is encountered. This assumes that ,KEYCASE will always call ,CFA directly.

111: Line 10: BYTECASE is a typical 8080 ;CODE defining word. "HEADER !1KEY ,KEYCASE" is the compile-time activity, and the macro RUN-BYTECASE compiles the run-time code. The run-time code must leave W pointing at a case CFA or the default CFA, and then execute that CFA.

Warning: if your ASSEMBLER does not specifically define BEGIN as HERE (non-IMMEDIATE), then you will fall through the ASSEMBLER into FORTH and find : BEGIN HERE ; IMMEDIATE. This version will not work in macros, because you want to compile a reference

to BEGIN that will execute when the macro executes.

Note how simply each key is paired with the word to execute upon matching the key (32 TWO, for example). The only punctuation needed is spaces (the number of spaces is not important).

112: It is interesting that 5TEST does not behave exactly like 4TEST. They are designed so that pressing the terminal key "0" selects ;S to be executed. Because <BUILDS DOES> is high-level, it has an extra level on the return stack; hence, the endless loop on line 13 does not exit, but screen 111 returns to the terminal with "OK".

Calling the colon definition "R> R> DROP DROP" from 112 would have the same effect as calling the code ;S from 111.

113: Note that the inline code is 6 instructions longer than the run-time code of the defining word. These instructions pick up the "exit address" which was given space at compile time by the "HERE 0 , "on line 7, and filled in by END-CASE.

114: Same idea as 111, but a two-byte key is expected on the stack. The low byte is in L and high byte is in H. Compare the "," on line 12 with the "C," of 111 line 8.

115-117: Self-explanatory.

118: Stringcase uses variable-length keys (up to 255 bytes). At run-time it expects bytes beginning at HERE 1+ with a count of them at HERE. It will match this string against its keys, executing the associated CFA if a match occurs. There is no restriction that the bytes must be printable ASCII, but you may find it hard to edit anything else into a screen. Source numbers may be used as keys, but they will be treated as character strings; the run-time is also a byte string, it is not normally placed on the stack even if it is a number.

The run-time code has two loops. The outer one is counting down the number of cases; the inner one has an index equal to the byte-count of a key plus one (the count, itself, is compared). Saved on the stack is IP and the address of the next key in the case structure (computed from W plus the byte-count plus 3)

119: One application of STRINGCASE is as a compact language translator. The string key is the input word, and the word executed by the associated CFA is the translation. Such an association is faster than a colon definition equating the two, because IP is not saved on the return stack, or restored.

The cases of a stringcase constitute a sort of vocabulary, but the structure is more compact than an ordinary dictionary because it lacks link fields, code fields, and terminators. The arithmetic that advances W from one case to the next is almost as fast as following a dictionary link, and the code for RUN-STRINGCASE compares favorably with (FIND). It is hard to imagine a FORTH - like language translator that would be faster or more compact.

120: High-level version of 118. The two nested loops are still there as DO LOOPS, the address of the next case is saved on the return stack between the loops, and the two pointers to the two byte-strings are on the parameter stack.

121-122: Self-explanatory.

123: The word called by the default case is exactly like INTERPRET except that is does not need to do a BL WORD because the string is already at HERE, and it is not an endless loop (so that it INTERPRET's only one word).

GERMAN is, then, a STRINGCASE that will, first, attempt to translate a word, but if it is not in its vocabulary, it will INTERPRET it normally. The endless loop taken away from INTERPRET is given to TRANSLATE which is then substituted for INTERPRET in the definition of LOAD (it could also be substituted in the definition of QUIT).

If one now loads a screen with TLOAD, it should compile normally, with the addition that EIN, ZWEI, and DREI will be understood as re-names, and executed immediately. In order to be a true translating interpreter, the DOES> part of STRINGCASE must be extended to respect compile mode by testing STATE, and either compile the CFA or execute the CFA, depending.

Note that ;S calls itself as a case. This is not only a way to find ;S, the interpreter would not stop at either ;S or NULL (it would, of course, stop at an undefined). The reason is that there is not an extra level on the return stack (namely, TRANSLATE) between the equivalent of LOAD (TLOAD), and the equivalent of INTERPRET (DEFAULT). Hence, executing ;S from DEFAULT is sufficient to end the execution of GERMAN, but not of TRANSLATE (which will inevitably call GERMAN again). However, calling ;S as a case, since it is a CODE definition, will end the execution of TRANSLATE, and return eventually to the terminal with "OK". But if one had used the <BUILDS DOES> version of STRINGCASE, there would, again, be an extra level on the return stack, and ;S would again fail. In this case, ;S would have to call a word whose definition is R> R> DROP DROP (see explanation of screen 112).

Another possible kind of keycase might be called BITCASE, where the key is a mask, and the associated CFA executes if the mask AND'ed with the value on the stack ≠ 0. The flag variables and compile-time code would be identical with BYTECASE; the run-time code would simply do an AND instead of an XOR, and a 0 = NOT instead of 0 = . The casewords presented here by no means exhaust the possibilities. The structure is deliberately left open-ended to encourage user creativity.

Note that BITCASE, BYTECASE, 2BYTE-CASE, and STRINGCASE all differ in name-length to avoid confusion on WIDTH-3 systems even when prefixed by DO- or RUN- .

Keycases have the property that they can be chained together through their default addresses (the key can be changed at this point, as well). This makes possible complex, high-level sturctures in which casewords feed other kinds of casewords. This is a tree with n brances at each node (a pattern similar to human brain cells).

With two default CFA's one could put keycasewords into 2-link structures such as binary trees. Furthermore, any CFA, including the default case, can be an EXEVAR (see explanation of screen 109), allowing the structure of the tree to change dynamically at run-time.

;S Steve Munson
Hughes Aircraft Company
Fullerton, CA 92634

Judges' Comments - An interesting aproach to error control, by making : IMMEDIATE a part of error control. If a preceding ; is missing, due to misediting, : will be encountered in compile mode. It executes but contains ?EXEC, which produces an error message if compiling. A little confusing but it works.

100

```
( LOAD SCREEN LOADER FOR CASEWORD SELECTION )
FORGET TASK    TASK
   103 LOAD     ( END-CASE)
   109 LOAD     ( WORDS FOR KEYCASEWORD COMPILING)    ( LOADS 110)

( 101 LOAD     ( LOAD SCREEN #1)
   102 LOAD     ( LOAD SCREEN #2)
;S
```

101

```
(     ( LOAD SCREEN #1 FOR CASEWORD SELECTION )
   105 LOAD   ( CASE DEFINING WORD)
   106 LOAD   ( <BUILDS DOES> CASE DEFINING WORD)
   107 LOAD   ( IN-LINE CASE WORD)
   108 LOAD   ( IN-LINE CASE FOR CODE DEFINITIONS)

   111 LOAD   ( BYTECASE DEFINING WORD)
   112 LOAD   ( <BUILDS DOES> BYTECASE DEFINING WORD)
   113 LOAD   ( IN-LINE BYTECASE WORD)                ( LOADS 111)

   114 LOAD   ( 2BYTECASE DEFINING WORD)              ( LOADS 115)
   116 LOAD   ( <BUILDS DOES> 2BYTECASE DEFINING WORD)
   117 LOAD   ( IN-LINE 2BYTECASE WORD)               ( LOADS 114)
;S
```

102

```
(     ( LOAD SCREEN #2 FOR CASEWORD SELECTION )

   118 LOAD   ( STRINGCASE DEFINING WORD)             ( LOADS 119)
   120 LOAD   ( <BUILDS DOES> STRINGCASE DEFINING WORD  ( " 121)
   122 LOAD   ( IN-LINE STRINGCASE WORD)              ( LOADS 119)
;S
```

103

```
(     ( END-CASE)  HEX
0 VARIABLE ?INLINE    0 VARIABLE ?KEYCASE

   +INLINE   1 ?INLINE  !      -INLINE   0 ?INLINE  !
   +KEYCASE  1 ?KEYCASE !      -KEYCASE  0 ?KEYCASE !

   END-CASE   ?KEYCASE @ IF   SWAP C!                 ( UNIVERSAL)
     STATE @ IF   HERE SWAP !   ENDIF   ELSE          ( CASEWORD )
   ?INLINE @ IF   HERE SWAP !   ELSE SMUDGE          ( DELIMITER)
        [COMPILE] [   ENDIF   ENDIF ;    IMMEDIATE
DECIMAL ;S
```

105

```
(     ( CASE DEFINING WORD )    HEX

   CASE   +KEYCASE -INLINE              ( SET FLAG VARIABLES FOR CASE )
      [COMPILE]    :CODE               ( DEFINE CASE WORD   COMP CFAs)
     H POP,  H DAD,  W INX,  W DAD,    ( CODE EXECUTES WORD VIA INDEX)
     M W 1+ MOV,  H INX,  M W MOV,     ( ON STACK     PUT CFA INTO W )
     XCHG,  NEXT 6 + JMP,              ( JUMP TO LAST HALF OF NEXT)

                        ( EXAMPLE )
   ONE  1  ,      TWO  2  ,      THREE  3

CASE PICK  ONE TWO THREE  END-CASE

   0TEST   0     PICK  4  ,                 ( 0 0TEST PRINTS 0 1 4 )
DECIMAL ;S                                  ( 1 0TEST PRINTS 0 2 4 )
                                            ( 2 0TEST PRINTS 0 3 4
```

106

```
(     ( <BUILDS DOES> CASE DEFINING WORD )    HEX

   CASE    <BUILDS  +KEYCASE -INLINE  ]        ( ADD INDEX TO PFA)
           DOES>  OVER + +  @ EXECUTE ;        ( GET CFA, EXECUTE)

                        ( EXAMPLE )
   ONE  1  ,     TWO  2  ,     THREE  3  ,

CASE PICK  ONE TWO THREE  END-CASE

   1TEST   0     PICK  4  ,
DECIMAL ;S
```

107

```
(     ( IN-LINE CASE WORD )    HEX

CODE DO-CASE    IP H MOV,  IP 1+ L MOV,      ( PICK UP EXIT ADDR )
    M IP 1+ MOV,  H INX,  M IP MOV,     ( MOVE IP SECOND END-CASE)
    XCHG,  H POP,  H DAD,  W INX,  W DAD,    ( H = W + 2* OFFSET)
    M W 1+ MOV,  H INX,  M W MOV,     ( CODE FIELD ADDR  INTO W  )
    XCHG,  NEXT 6 + JMP,              ( JUMP TO LAST HALF OF NEXT)

   DO-CASE  -KEYCASE +INLINE                      ( HERE 0   LEAVES)
    COMPILE DO-CASE  HERE 0 ,    IMMEDIATE      ( GAP    END-CASE)
                                                ( FILLS IT IN  )
                        ( EXAMPLE )
   ONE  1  ,     TWO  2  ,     THREE  3  ,

   2TEST   0    DO-CASE  ONE TWO THREE  END-CASE  4
DECIMAL ;S
```

108

```
(     ( IN-LINE CASE FOR CODE DEFINITIONS )    HEX
ASSEMBLER DEFINITIONS
   DO-CASE   HERE 0 - W LXI,        ( LOAD W WITH ADDR BEYOND LAST
    H POP,  H DAD,  W DAD,          ( BYTE OF THIS MACRO , ADD 2* )
    M W 1+ MOV,  H INX,  M W MOV,   ( INDEX ON STACK , FETCH CFA  )
    XCHG,  NEXT 5 + JMP,            ( JUMP TO LAST HALF OF NEXT   )
    -KEYCASE -INLINE  SMUDGE  ]  ;  ( SET FLAGS AND COMPILE MODE  )
FORTH DEFINITIONS

                        ( EXAMPLE )
   ONE  1  ,     TWO  2  ,     THREE  3  ,
                                                ( 1 STEST   PRINTS 1)
CODE STEST   H POP,  L A MOV,  H ORA,           ( 0 0 STEST PRINTS 1)
   0= IF,  DO-CASE  ONE TWO THREE  END-CASE  THEN,  PUSH JMP,
DECIMAL ;S
```

109

```
(     ( WORDS FOR KEYCASEWORD COMPILING )    HEX
0 VARIABLE OLDHERE    0 VARIABLE OLDDL    0 VARIABLE OLDIN

: NOWSAVE   HERE OLDHERE !  DL @ OLDDL !  IN @ OLDIN !
: RECOVER   OLDHERE @ DP !  OLDIN @ IN !  OLDDL @ DL ! ;

   *NUMBER   -1 DPL !  0 0 HERE  DUP 1+ C@ ID =
    DUP 'R +  (NUMBER) DROP DROP  R@ IF  MINUS  ENDIF

   HEADER   CREATE SMUDGE ;

   *'   -FIND  0= 0 ?ERROR  DROP ;                ( ' WITHOUT LITERAL)
DECIMAL --:
```

110

```
(     ( WORDS FOR KEYCASEWORD COMPILING, CONT )  HEX
: ?KEY   -FIND  HERE @ 1 = IF  1 SLIT +!  0 IN !     ( IF NULL  )
    NOWSAVE  DROP DROP DROP -FIND ENDIF  ( GET NEXT BLOCK, SAVE)
       IF  DROP  ( END-CASE = IF          ( NEW ADDR  END-CASE?)
    SWAP 1+ SWAP  RECOVER ENDIF ENDIF ;  ( SET NO-DEFAULT FLAG )

0 CONSTANT ;KEY-ADDR                ( ADDR OF ;KEY IN  KEYCASE)
: ;KEY  HERE 0 ,     ;KEY-ADDR ! ;  IMMEDIATE ( SET ;KEY-ADDR)

: ;CFA   *'  HERE @ 1 = IF  1 SLIT +!  0 IN !  DROP *'  ( PUT A)
  ENDIF  DUP ' END-CASE = IF  RECOVER  R> DROP  DROP  ( CFA IN)
     SWAP IF  ' NOOP ELSE  *'  ENDIF ENDIF  CFA , ;    ( A CASE)

: KEYCASE   +KEYCASE  HERE 0 C,  0
   -1 BEGIN  1+  NOWSAVE  ?KEY  ;KEY  ;CFA  AGAIN
DECIMAL ;S
```

111

```
(     ( BYTECASE DEFINING WORD, ( ONE-BYTE KEYS )    HEX
ASSEMBLER DEFINITIONS
: RUN-BYTECASE   H POP,  W INX,  W LDAX,  A H MOV,  W INX,
   BEGIN, W LDAX,  W INX,  L CMA,  0= NOT IF,      ( MACRO USED)
          W INX,  W INX,  H DCR,  THEN,  0= END,   ( ON LINE 10)
   XCHG,  M W 1+ MOV,  H INX,  M W MOV,            ( 5 SCR 113, )
   XCHG,  NEXT 6 + JMP, ;   FORTH DEFINITIONS      ( LINE 4    )

: ;1KEY   *NUMBER C, ;                      ( COMPILE A 1-BYTE KEY )
: !1KEY   ' ;1KEY CFA , ;KEY-ADDR ! ;    ( PUT ;1KEY IN  KEYCASE)
: BYTECASE   HEADER  !1KEY ;KEYCASE  ;CODE RUN-BYTECASE
                        ( EXAMPLE )
   ONE  1  ,     TWO  2  ,     THREE  3  ,     DEFAULT 4  ,
BYTECASE SHOW
   31 ONE  30 ;S  32 TWO  33 THREE  DEFAULT  END-CASE
   4TEST   BEGIN  KEY SHOW  AGAIN ;    DECIMAL
```

112

```
(     ( <BUILDS DOES> BYTECASE DEFINING WORD )    HEX
: ;1KEY   *NUMBER C, ;                      ( COMPILE A 1-BYTE KEY )
: !1KEY   ' ;1KEY CFA , ;KEY-ADDR ! ;    ( PUT ;1KEY IN  KEYCASE)

: BYTECASE   <BUILDS  !1KEY ;KEYCASE  DOES>
   DUP C@  0 DO  1+ OVER OVER  C@ = IF        ( HIGH LEVEL DOES>
      LEAVE ELSE  2+ ENDIF  LOOP              ( SAME  THING  AS)
   SWAP DROP  1+ @ EXECUTE ;                  ( CODE ON SCR 111
                        ( EXAMPLE )
   ONE  1  ,     TWO  2  ,     THREE  3  ,     DEFAULT 4  ,
BYTECASE +SHOW
   31 ONE  30 ;S  32 TWO  33 THREE  DEFAULT  END-CASE
   5TEST   BEGIN  KEY +SHOW  AGAIN ;
DECIMAL ;S
```

```
                                                                  113
 0    ( IN-LINE BYTECASE WORD )
 1 111 LOAD   HEX
 2
 3 CODE DO-BYTECASE   IP H MOV,  IP 1+ L MOV,  M IP 1+ MOV,
 4    H INX,  M IP MOV,  XCHG,  RUN-BYTECASE  ( SEE MACRO SCR 111)
 5
 6   DO-BYTECASE
 7    COMPILE DO-BYTECASE  HERE 0  'S'KEY ,'KEYCASE ,   IMMEDIATE
 8
 9                         ( EXAMPLE )
10   ONE  1  ,      TWO  2  ,      THREE  3  ,      DEFAULT 4  ,
11
12  :TEST   BEGIN  KEY  DO-BYTECASE  21 ONE  32 TWO
13    33 THREE  30 ,S  DEFAULT  END-CASE  AGAIN ;
14 DECIMAL ;S
15
```

```
                                                                  114
 0    ( 2BYTECASE DEFINING WORD, ( TWO-BYTE KEYS )     HEX
 1 ASSEMBLER DEFINITIONS                              ( H = KEY ,)
 2   RUN-2BYTECASE   H POP,  IP PUSH,                 ( SAVE IP ,)
 3    W INX,  W LDAX,  A IP MOV,  W INX,              ( IP = # OF CASES)
 4    BEGIN, W LDAX,  W INX,  L XRA,  0= IF,          ( 1ST BYTE = ?)
 5           W LDAX,  W INX   H XRA,  0= NOT IF,      ( 2ND BYTE = ?)
 6           W INX,  W INX,  IP DCR,  THEN,  ELSE,    ( IF NO MATCH)
 7           W INX,  W INX,  W INX,  IP DCR, THEN,    ( GO NEXT CASE)
 8    0= END,  IP POP,                                ( ELSE GET IP)
 9    XCHG,  M W 1+ MOV,  H INX,  M W MOV,            ( PICK UP CFA)
10    XCHG,  NEXT 0 + JMP ,    FORTH DEFINITIONS      ( EXECUTE CFA )
11
12   :2'KEY   *NUMBER , ;                            ( COMPILE A 2-BYTE KEY )
13   '2'KEY     ,2'KEY CFA  ,)'KEY-ADDR ' ,    ( PUT ,2'KEY IN ,'KEYCASE)
14   2BYTECASE   HEADER  '2'KEY ,'KEYCASE  ,CODE RUN-2BYTECASE
15 DECIMAL --:
```

```
                                                                  115
 0    ( 2BYTECASE DEFINING WORD, CONT )    HEX
 1 ( TEST FOR SCREEN 114)
 2                         ( EXAMPLE )
 3   ONE  1  ,      TWO  2  ,      THREE  3  ,      DEFAULT 4  ,
 4
 5 2BYTECASE 7PICK
 6    0111 ONE  0222 TWO  0333 THREE  DEFAULT  END-CASE
 7   7TEST   0    7PICK  4  ,
 8 DECIMAL ;S
 9
10
11
12
13
14
15
```

```
                                                                  116
 0    ( :BUILDS DOES> 2BYTECASE DEFINING WORD )    HEX
 1   :2'KEY   *NUMBER , ;                            ( COMPILE A 2-BYTE KEY )
 2   '2'KEY     ,2'KEY CFA  ,)'KEY-ADDR ' ,    ( PUT ,2'KEY IN ,'KEYCASE)
 3
 4   2BYTECASE   :BUILDS  '2'KEY ,'KEYCASE  DOES>  ( DOES SAME THING
 5    DUP 1+ SWAP C@  0 DO  OVER OVER  @ = IF     ( AS CODE IN 114 )
 6       2+ LEAVE ELSE  4 + ENDIF  LOOP           ( COMPARES  BOTH
 7    SWAP DROP  @ EXECUTE ,                      (  BYTES AT ONCE )
 8
 9                         ( EXAMPLE )
10   ONE  1  ,      TWO  2  ,      THREE  3  ,      DEFAULT 4  ,
11
12 2BYTECASE 3PICK
13    0111 ONE  0222 TWO  0333 THREE  DEFAULT  END-CASE
14  :3TEST   0    3PICK  4  ,
15 DECIMAL ;S
```

```
                                                                  117
 0    ( IN-LINE 2BYTECASE WORD )
 1 114 LOAD   HEX
 2
 3 CODE DO-2BYTECASE   IP H MOV,  IP 1+ L MOV,  M IP 1+ MOV,
 4    H INX,  M IP MOV,  XCHG,  RUN-2BYTECASE ( SEE MACRO SCR 114)
 5
 6   DO-2BYTECASE
 7    COMPILE DO-2BYTECASE  HERE 0 ,  '2'KEY ,'KEYCASE ,   IMMEDIATE
 8
 9                         ( EXAMPLE )
10   ONE  1  ,      TWO  2  ,      THREE  3  ,      DEFAULT 4  ,
11
12  :9TEST   0    DO-2BYTECASE  0111 ONE  0222 TWO
13    0333 THREE  0 ;S  DEFAULT  END-CASE  4  ,
14 DECIMAL ;S
15
```

```
                                                                  118
 0    ( STRINGCASE DEFINING WORD, ( STRING KEYS      HEX
 1 ASSEMBLER DEFINITIONS
 2   RUN-STRINGCASE   IP PUSH,  W INX,          ( SAVE IP ON STACK)
 3           W LDAX,  A IP MOV,  W INX,          ( IP = # OF CASES)
 4   BEGIN, W LDAX,  A INR,  A IP 1+ MOV,  W 1+ ADC,   ( IP 1+ =)
 5           A L MOV,  W A MOV,  0 # ADC,  A H MOV,   ( BYTE COUNT)
 6           H INX,  H INX,  H PUSH,          ( STACK = NEXT CASE)
 7           DP ( RUNTIME HERE) LHLD,         ( H = RUN-TIME HERE)
 8   BEGIN, W LDAX,  W INX,  M XRA,  H INX,  0= IF,  ( BYTES = ?)
 9           IP 1+ DCR,  SWAP ( AT COMPILE TIME) 0= END,  ( AGAIN)
10           H POP,  HERE 3 + JMP,  THEN,        ( THROW AWAY ADDR )
11           W POP,  IP DCR, 0= END,  IP POP,  ( ELSE GO NEXT CASE)
12    XCHG,  M W 1+ MOV,  H INX,  M W MOV,          ( PICK UP CFA )
13    XCHG,  NEXT 5 + JMP ,    FORTH DEFINITIONS    ( JUMP TO NEXT)
14 DECIMAL --:
15
```

```
                                                                  119
 0    ( STRINGCASE DEFINING WORD, CONT )    HEX
 1
 2   :S'KEY   HERE C@ 1+  ALLOT ;                ( COMPILE A STRING KEY )
 3   'S'KEY     ,SKEY CFA  ,)'KEY-ADDR '  ,    ( PUT ,S'KEY IN ,'KEYCASE)
 4
 5   STRINGCASE   HEADER  'S'KEY ,'KEYCASE  ,CODE RUN-STRINGCASE
 6
 7                         ( EXAMPLE )
 8   ONE  1         TWO  2  ,      THREE  3         DEFAULT 4
 9                    ( DECIMAL 123 LOAD ;S
10 STRINGCASE GERMAN
11    EIN ONE  ZWEI TWO  DREI THREE  DEFAULT  END-CASE
12
13   TRANSLATE   BL WORD  GERMAN      ( 'TRANSLATE EIN'  PRINTS )
14 DECIMAL ;S
15
```

```
                                                                  120
 0    ( :BUILDS DOES> STRINGCASE DEFINING WORD )    HEX
 1   :S'KEY   HERE C@ 1+  ALLOT ;                ( COMPILE A STRING KEY )
 2   'S'KEY     ,SKEY CFA  ,)'KEY-ADDR '  ,    ( PUT ,S'KEY IN ,'KEYCASE)
 3
 4   STRINGCASE   :BUILDS  'S'KEY ,'KEYCASE  DOES>
 5   HERE OVER 1+  ROT C@ 0 DO                        ( DO # OF CASES )
 6       DUP DUP C@ + 3 + SWAP  DUP C@ 1+ 0 DO        ( SAVE NEXT CASE)
 7       OVER C@  OVER C@ = IF  1+ SWAP  1+ SWAP  (  COMPARE BYTE
 8       ELSE  DROP DROP HERE 0  LEAVE ENDIF LOOP    IF MATCH  DUMP
 9    DUP IF  R> DROP  LEAVE ELSE  DROP R>  ENDIF    ADDR  ELSE GO
10    LOOP  SWAP DROP  @ EXECUTE ;                   ( TO NEXT CASE )
11 DECIMAL --:
12
13
14
15
```

```
                                                                  121
 0    ( :BUILDS DOES> STRINGCASE DEFINING WORD, CONT )    HEX
 1 ( TEST FOR SCREEN 120)
 2                         ( EXAMPLE )
 3   ONE  1  ,      TWO  2  ,      THREE  3  ,      DEFAULT 4  ,
 4
 5 STRINGCASE GERMAN
 6    EIN ONE  ZWEI TWO  DREI THREE  DEFAULT  END-CASE
 7
 8   TRANSLATE   BL WORD  GERMAN ;
 9 DECIMAL ;S
10
11
12
13
14
15
```

```
                                                                  122
 0    ( IN-LINE STRINGCASE WORD )
 1 118 LOAD   HEX
 2
 3 CODE DO-STRINGCASE   IP H MOV,  IP 1+ L MOV,  M IP 1+ MOV,
 4    H INX,  M IP MOV,  XCHG,  RUN-STRINGCASE ( SEE MACRO SCR 118)
 5
 6   DO-STRINGCASE
 7    COMPILE DO-STRINGCASE  HERE 0 ,  'S'KEY ,'KEYCASE ,   IMMEDIATE
 8
 9                         ( EXAMPLE )
10   ONE  1  ,      TWO  2  ,      THREE  3  ,      DEFAULT 4  ,
11
12   TRANSLATE   BL WORD  DO-STRINGCASE
13    EIN ONE  ZWEI TWO  DREI THREE  DEFAULT  END-CASE ;
14 DECIMAL ;S
15
```

```
                                                                  123
 0    ( EXAMPLE OF A TRANSLATING INTERPRETER     HEX
 1   DEFAULT  HERE CONTEXT @ @ (FIND)  DUP 0= IF    ( = INTERPRET
 2       DROP  HERE LATEST (FIND)  ENDIF    ( WITHOUT BEGIN-AGAIN)
 3    IF  STATE @ ; IF  CFA ,  ELSE  CFA EXECUTE  ENDIF ?STACK
 4    ELSE  HERE NUMBER  DPL @ 1+ IF  [COMPILE] DLITERAL    ALSO )
 5          ELSE  DROP  [COMPILE] LITERAL  ENDIF ?STACK    ( LAUNCH)
 6    ENDIF ;                                              ( BL WORD)
 7
 8 STRINGCASE GERMAN
 9    EIN ONE  ZWEI TWO  DREI THREE  ;S  0  DEFAULT  END-CASE
10
11   TRANSLATE   BEGIN  BL WORD  GERMAN  AGAIN
12
13  TLOAD   BLK @ >R  IN @ >R                    ( SAME AS LOAD)
14    0 IN !  B/SCR *  BLK !                     ( BUT TRANSLATE)
15       TRANSLATE  R> IN !  R> BLK !     DECIMAL    ( FOR INTERPRET)
```

COME TO FIG CONVENTION
NOVEMBER 29

A PROPOSED CASE STATEMENT FOR FORTH

Karl Bochert/Dave Lion

General Description

The CASE statement suggested here is done in high level code for the 6800 version of fig-FORTH. It may have to have some minor changes in order to conform to the FORTH-79 standard. The names of the words were chosen for descriptive value.

The word that initiates the set of cases is:

CASE

Following that are as many sets of:

<forth code> ENDCASE

as needed to represent all the desired cases which are to be executed. The first set is for case 0, and each successive set is for the next higher case number. After the last set comes the terminating set:

<forth code> ENDCASES

which indicates the default code to be executed if the case number is outside the legal limits. It also marks the end of all of the cases, and causes the look-up table to be compiled. Word (CASE) , which is the run-time word, is surrounded by parentheses according to fig-FORTH convention, indicating that it is normally never typed in by the user.

At run-time word (CASE) uses one integer parameter from the data stack and leaves none. The given parameter specifies which one of many cases will be executed. A single case is defined as a set of FORTH words which is preceded by the word CASE or END-CASE, and followed by the word ENDCASE or ENDCASES. Within a single case, the usual rules of pairing still apply to the words: DO, LOOP, IF, ELSE, THEN, BEGIN, AGAIN, WHILE, REPEAT. That is, they must be properly matched with each other.

Case 0 will be executed if the parameter is 0, case 1 if it is 1, etc. The parameter will normally be in the range: 0 thru (# of cases)-1. Thus, the case function works like the computed GOTO found in some versions of BASIC, with the exception that this code is in-line.

Advantages

CASE is very compactly compiled, so the number of 16-bit words of overhead is 2 * (# of cases +1) + 3. This excludes the code within each of the cases, but includes the ;S which follows each case. The following use of the CASE function, having 3 empty cases and an empty default case will compile as 22 bytes of code:

```
CASE
   ENDCASE
   ENDCASE
   ENDCASE
ENDCASES
```

Here, it should be pointed out that the CASE function is only used within a definition, and the above sample is part of a definition.

More Advantages

CASE statements have little overhead run-time code. In the FIG model this version of (CASE) executes 41 FORTH words, 37 of which are code words. This may be shortened by leaving out the two protective features, thus executing 25 words, 22 of which are code words. The fastest method takes about 0.002 seconds to execute.

There is practically no limit upon the number of cases that may be compiled. The table of pointers will contain an address for each case plus an address for the default case.

Two protective functions in word (CASE) will handle negative numbers and numbers that are too high. For negative numbers, the equivalent positive case is executed. For numbers too high, a default case is executed. It should also be noted that any intermediate case that will never be executed still needs an ENDCASE, but the compiled code will contain only a ;S . The default case may be left out, and will then compile like an empty case.

One additional feature to point out is that CASE statements may be nested much the same way as 'DO' loops can.

Disadvantages

There is one machine dependent factor that must be considered before installing these words. Since we fool around with return addresses in the return stack, we must know whether the return stack of the machine stacks 'return to' addresses or 'came from' addresses. The former is the situation where the address is not incremented before doing the first fetch after a ;S . The latter type of machine (my 6800 version) does do a pre-increment after a ;S . Appropriate comments for patching are included in the definition of (CASE) .

The way to find out which type of FORTH machine you are using is:

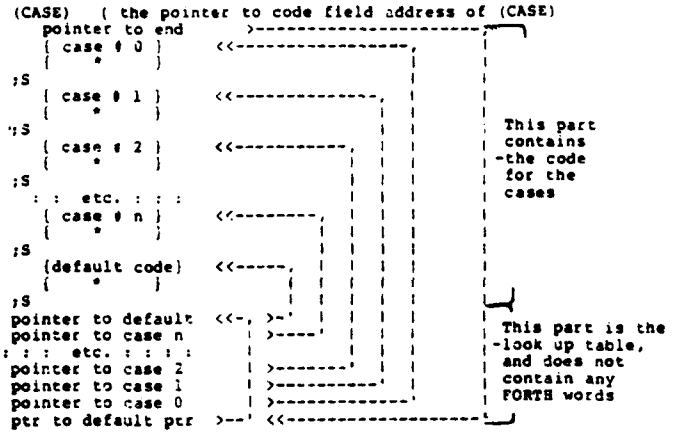
```
: P1  R   ;
: P2  P1  ;
  P2  P2  -  .
  FORGET P1
```

The printout will be 0 for the 'came from' type of FORTH machine, and 2 for the 'return to' type.

Another thing to watch out for is that while inside a CASE statement you no longer have access to any loop counter (I) which was created outside the CASE statement. During execution of the chosen case there is one extra address on the return stack, covering up what was there.

Compiled Structure

Note: Each line shows the contents of one 16-bit word of memory except for the lines within braces: , which signify any amount of memory, including none, which may contain FORTH instructions.



* Any case code may be left out. The resultant case segment will have only a ;S in it.

Definitions for 6800 Fig-FORTH

```
(CASE)              (    the run-time function                    )
   ABS              ( #  make sure parameter is +                 )
   R>               (    get address of pointer to table          )
   2+            ( delete this line for 'return to' machines <======)
   @ DUP            (    get pointer to table                     )
   2+             add this line for 'return to' machines <=========)
   >R               (    save final return address                )
   SWAP 1+ DUP +    (    find addresses into table of the ....    )
   OVER  SWAP       ( #    highest legal case, ....               )
   -                (      and the desired case ....              )
   SWAP @           ( #    then choose the ....                   )
   MAX              ( #    best one                               )
   @                (    read table entry for chosen case         )
   2  -          ( delete this line for 'return to' machines <======)
   >R  ;            (    stack it & 'return' to it                )

NOTE: the lines marked '#' may be deleted to speed up execution    )
      while sacrificing protection.                                )

:CASE
   COMPILE  (CASE)  (    compile the run-time executor            )
   HERE  0  ,       (    init table pointer & get its addr        )
   0                (    stack a marker on data stack             )
   ;   IMMEDIATE

ENDCASE
   COMPILE  ;S      ( end of a case                               )
   HERE             ( stack a ptr. to next case                   )
   ;   IMMEDIATE

ENDCASES            ( this word writes the look-up table          )
   COMPILE  ;S      ( end of default case                         )
   ,                ( put pointer for default case into table     )
   HERE  >R         ( temporarily save addr of pntr to case[n]    )
   DUP              ( look for case[0]                            )
   IF               ( didn't find marker, so:                     )
      BEGIN
      ,  DUP  0=    ( store pntrs to case[n] thru case[1]         )
      END           ( until reaching the marker                   )
   THEN
   DROP             ( drop the marker                             )
   DUP              ( dup the pfa                                 )
   2+  ,            ( store pntr to case[0]                       )
                    ( data stack is down to 1 item: the pfa       )
   HERE SWAP  !     ( store this addr into pfa                    )
   R>               ( fetch addr of ptr to highest normal case    )
   2  -  ,
   ;   IMMEDIATE
```

A Test of the 'CASE' Function:

```
TEST-WORD
 CR
 5  -4    ( try a range of parameters, some of which are illegal )
 DO
   I  DUP .   ( precede each line with the case #  being tried )
   CASE
   ." This is the case # 0 code"     ENDCASE
   ( ---case #1 does nothing--- )    ENDCASE
   ." This is the case # 2 code"     ENDCASE
   ." default case"
   ENDCASES
   CR           ( do the next case on a new line )
 LOOP ;
```

The Result is:

```
TEST-WORD    ( typed by human)
-4 default case
-3 default case
-2 This is the case # 2 code
-1
0 This is the case # 0 code
1
2 This is the case # 2 code
3 default case
4 default case
 OK
```

Time Trials:

Here we find out how long it takes to get to the proper case. The CPU clock is set at 1.000 MHz. The word (CASE) was defined leaving out the protection features. Then the following definitions for timing loops are tried, executing null cases which do nothing. 100,000 loops are timed:

```
DECIMAL
: INNER 1000 0 DO 1 CASE ENDCASE ENDCASE ENDCASE ENDCASES LOOP ;
: SPEED ." X" 100 0 DO INNER LOOP ." X" ;
SPEED XX OK     ( this was 210 seconds on the 6800 FORTH )
```

1000,000 loops are timed, leaving out the CASE portion:

```
: INNER2 1000 0 DO        LOOP ;
: SPEED2 2 ." X" 100 0 DO  INNER2  LOOP ." X" ;
SPEED2 XX OK     ( this was 13 seconds on the 6800 FORTH )
```

Thus, it can be seen that it takes about 2 milliseconds to vector to the desired case if the two protection features are left out. Putting in the protection would increase the time to about 3.5 mSec.

Karl Bochert
Dave Lion
Los Altos, CA 94022

Judges' Comments -

Karl and Dave were the only entry to make provision for pre-incrementing and post-incrementing versions of NEXT. This refers to when the interpretive pointer IP is advanced within NEXT. They give a test to check your system. This version uses a compiled table of indexes to give minimum execution time. The style and documentation is to be complimented.

CASE AND PROD CONSTRUCTS

Steve Brecher

```
( syntax:   www CASEOF
            www CASE www ESAC
                  .          ( 0 or more CASE/ESAC pairs
                  .            allowed, at least 2 pairs
                  .            for semantic sense.)
            www CASE www ESAC
            OTHERWISE www      ( OTHERWISE optional)
            ENDCASEOF
```

www stands for 0+ Forth words, possibly including complete case expression[s], these possibly still further nested. But code represented by www can make no net change to the return stack, as the case selector value is stored there. Runtime: CASEOF pops, saves top of compute stack as selector. CASE pops, tests top of stack vs. selector; if =, executes words up to next ESAC followed by words after ENDCASEOF. If <>, executes words after next ESAC. OTHERWISE is optional for readability. SELECTOR used anywhere between CASEOF and ENDCASEOF leaves the selector value, provided no net change has been made to the return stack since CASEOF; SELECTOR is an alias for 'R'.)

```
31 CONSTANT      CASSYNTX (Error number, case construct syntax)

: CASEOF

( -> 0 4  .  Pronounced "case of", after Pascal.)
COMPILE >R       ( to save selector for testing by CASEs)
0                ( end-of-data signal to ENDCASEOF)
4                ( For CASE syntax check) ;

IMMEDIATE

CODE CASEBRANCH  ( n -> . Forth branch to the offset following inline if n <> @RP, else bump IP over offset. Compiled by CASE.)
S )+   RP () CMP,
NE IF,                  ( If n <> @RP, )
  IP ()   IP ADD,       ( add inline offset to IP)
  NEXT, ENDIF,          ( and "branch")
IP )+ TST,              ( else bump IP over inline offset)
NEXT, C;                ( and continue there.)

: ?CASE          ( n1 n2 -> . Compile-time check for n1=n2. If fail issue syntax error)
<> IF CASSYNTX ERROR ENDIF ;

: CASE           ( 4 -> addr 5 . Executes ?CASE syntax check; compiles CASEBRANCH with a zero offset; pushes address of offset so ESAC can fix it later; pushes 5 syntax check signal.)
4 ?CASE ( Syntax check)
COMPILE CASEBRANCH
HERE     ( Push address of offset so ESAC can patch it)
0 ,      ( ESAC will change the 0 to +offset for CASEBRANCH)
5        ( For ESAC syntax check) ;

IMMEDIATE

: ESAC           ( addr1 5 -> addr2 4 . Pronounced "eesack"; "case" spelled backward. Executes ?CASE syntax check; fixes the offset at addr1 so the CASEBRANCH there will branch to the code after ESAC; compiles BRANCH with a 0 offset, pushes the address of the 0 offset so ENDCASEOF can fix it later; leaves 4 for syntax check by later word.)
5 ?CASE                 ( Syntax check)
2                       ( ELSE will be checking for this)
[COMPILE] ELSE          ( ELSE fixes CASE offset, pushes addr of 0 offset it compiles with BRANCH)
2+ ;    ( ELSE leaves 2, CASE/OTHERWISE/ENDCASEOF want 4)

IMMEDIATE
```

```
: OTHERWISE      ( 4 -> 4 . For
                   readability,
                   optionally written
                   after last ESAC to
                   identify code which
                   is executed if no
                   cases match.
                   Performs compile-
                   time checks.)
                 ?COMP
                 4 ?CASE
                 4 ;
IMMEDIATE

: ENDCASEOF      ( 0 addr1 addr2 ...
                   addrn 4 -> .
                   addrx is the addr of
                   an inline offset
                   following a BRANCH
                   compiled by an ESAC.
                   Executes ?CASE
                   syntax check; 0 on
                   the stack is an
                   end-of-data signal
                   which was pushed by
                   CASEOF; For each
                   CASE...ESAC, patches
                   the offset at addrx
                   so that the BRANCH
                   compiled by ESAC
                   will branch to the
                   R>DROP which END-
                   CASEOF compiles.)
                 4 ?CASE ( Syntax check)
BEGIN -DUP WHILE        ( there's a nonzero offset on stack)
  2                     ( ENDIF will be checking for this)
  [COMPILE] ENDIF       ( ENDIF will compute, emplace offset)
                  REPEAT
COMPILE R>DROP ;        ( code drops case value from R stack)
IMMEDIATE

ALIAS SELECTOR   R

( PRODS/PROD/DCRP/CATCHAL/ENDPRODS are analogous to
  CASEOF/CASE/ESAC/OTHERWISE/ENDCASEOF except there
  is no selector value:  each PROD tests for tf on stack.)

33 CONSTANT      PROSYNTX ( Error number, production set syntax)

: PRODS          ( -> 0 6 .   Compile-time setup for PROD set.)
                 0       ( end-of-data signal to ENDPRODS)
                 6 ;     ( For PROD syntax check)

IMMEDIATE

: ?PROD          ( n1 n2 -> . )
                 <> IF PROSYNTX ERROR ENDIF ;

: PROD           ( 6 -> addr 7 )
                 6 ?PROD

                 6 ;
IMMEDIATE

: ENDPRODS       ( 0 addr1 addr2 ... addrn 6 -> )
                 6 ?PROD                ( Syntax check.)
                 BEGIN -DUP WHILE       ( there's a nonzero offset of stack)
                 2                      ( ENDIF will be checking for this)
                 [COMPILE] ENDIF        ( ENDIF will compute, emplace offset)
                 REPEAT ;
IMMEDIATE
```

Steve Brecher
Software Supply
Long Beach, CA

Judges' Comments -

This entry supports essentially the same syntax and semantics as the FORTH-85 CASE statement (see FD I/5), but offers the following advantages:

1. Compile-time syntax checking.
2. Explicit OTHERWISE clause.
3. Case selector is kept on return stack instead of in a special variable. This allows nesting of CASE constructs.
4. 16-bit branch offsets are used, rather than a mixture of 16-bit addresses and 8-bit offsets. This eliminates the need for a special run-time END-CASE word and simplifies compilation.

NEW PRODUCT

Z-80

We have a Z-80 implementation of FIG-FORTH that was derived directly from 8080 FIG-FORTH 1.1 and will run under either CP/M or Cromemco CDOS. The code is optomized to exploit the additional Z-80 registers and instructions. Although this was developed for our own internal use we are willing to make it available at cost to interested FIG members. For $25.00 to cover media, copying, and shipping, we will send two soft- sectored single density eight inch diskettes containing executable Z-80 FORTH interpreter, all source files, and sample FORTH programs. Payment may be sent by check or money order to the address below. Please allow us 30 days for shipment. LABORATORY MICROSYSTEMS, 4147 Beethoven Street, Los Angeles, CA 90066, (213) 390-9292.

A CASE STATEMENT

Mike Brothers

Approximately a year ago I was writing a program and needed a more powerful branching construction than the standard IF..ELSE..ENDIF construction. Somehow I decided on implementing Pascal's CASE statement in FORTH, and this is the one which is described here. This CASE statement is also included in the standard SL5 package, available from the Stackworks.

Some of the advantages of SL5's CASE statement are:

1) Infinite nesting is possible.

2) The CODE is machine independent.

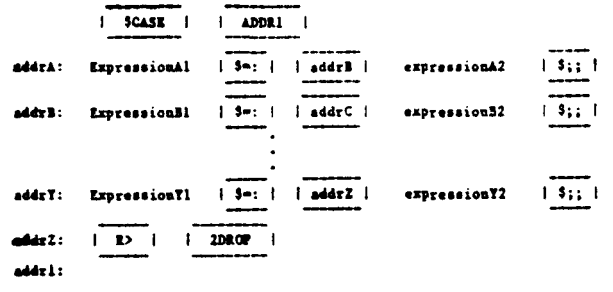
3) Programs are easier to read because of its simplicity.

CASE statement definitions

```
: $CASE R> DUP 2 + SWAP @ >R >R ;
: $=: OVER = IF
    DROP R> 2 + >R
    ELSE R> @ >R
 ENDIF ;
: $;; R> DROP ;          : NOCASE DUP ;
: CASE  \ $CASE    HERE 0 , ;                    IMMEDIATE
: =:  \ $=:    HERE 0 , ;                        IMMEDIATE
: ;;  \ $;;    HERE SWAP ! ;                     IMMEDIATE
: CASEND  \ R>   \ 2DROP   HERE SWAP ! ;         IMMEDIATE
```

Compilation

During compilation, "CASE" compiles the address of "$CASE" and a 0 for the address field. Every subsequent "=:" causes "$=:" to be compiled along with a dummy address field (to be set by the next ";;"). The word ";;" then compiles "$;;" and replaces the address field of the previous "=:" with addrx. When "CASEND" is finally processed, the something resembling figure 2 should be present.



Execution

Upon entry, a number corresponding to the case is assumed to be on the stack. "$CASE" then places ADDR1 on the return stack. ExpressionA1 is then executed, and "$=:" compares the value on the tos (top of stack) with the nos (next on stack), which should be the entry value. If these are equal, the entry value is dropped and ExpressionA2 is executed before "$;;" sets the interpreter pointer to ADDR1 (which is on the return stack). If the two values are not equal, "$=:" sets the IP to addrB and execution continues until a valid case is found or the cases are exhausted, which causes ADDR1 to be removed from the return stack and the entry value dropped.

The word "NOCASE" always causes the $=: to execute the following expression, simply by setting the tos equal to the entry value.

Examples of CASE statement usage

```
: EXAMPLE1 BEGIN
    ." CHOICE? "  GCH CASE
        41 =: ." APPLE "       1 ;;    ( A is for APPLE )
        42 =: ." BLUEBERRY "   1 ;;    ( B is for BLUEBERRY )
        43 =: ." CHERRY "      1 ;;    ( C is for CHERRY )
        44 =: ." DATE "        1 ;;    ( D is for DATE )
        45 =: ." ELDERBERRY "  1 ;;    ( E is for ELDERBERRY )
        NOCASE =: ." WRONG "   0 ;;    ( repeat till valid )
    CASEND
END ;
```

Figure 3. Example of CASE statement usage

CASE statement example

The example shown above illustrates the CASE statement's simplicity and power. When EXAMPLE1 is executed, a character is read from the keyboard. If the character is an "A", the string "APPLE" is displayed. If the character is a "B", the string "BLUEBERRY" is shown. If none of the five are selected, the string "WRONG" is

displayed and the loop is executed again until a valid (A-E) choice is entered.

The COND Statement

One particular advantage of the case statement is that an additional branching structure which executed an expression based on a boolean expression can be defined with a few more words. I call this structure the COND statement, and the extra words needed are shown in figure 3. The structure is much like that of the CASE statement, as shown in the example in figure 4.

```
: COND  COMPILE CASE ;                                  IMMEDIATE
: CONDEND  \ R>     \ 2DROP     HERE SWAP - ;           IMMEDIATE
: $:: IF
     R> 2 + >R
     ELSE R> @ >R
 ENDIF ;
: ::  \ $::      HERE 0 , ;                             IMMEDIATE
```

Figure 4. COND statement definitions

```
: EXAMPLE2 COND
       DUP 0>   ::  T" POSITIVE"  ;;
       DUP 0=   ::  T" ZERO" ;;
       1        ::  T" NEGATIVE"  ;;
 CONDEND ;
```

Figure 5. Examples of COND statement usage

The COND Example

The Example shown above illustrates the similarity between the COND construction and the CASE statement. Upon entry to EXAMPLE2, an integer is assumed to be on the stack. One of the strings "POSITIVE", "ZERO", or "NEGATIVE" is displayed depending on the integer.

Mike Brothers
The Stackworks
Bloomington, IN 47401

Judges' Comments - This is a practical method but not as portable as it might appear. The 2+ in $CASE and $=: will have to be relocated for preincrementing 6800 systems. The COND statement is a nice variation on CASE.

NEW PRODUCT

FORTH FOR CP/M

Mitchell E. Timin Engineering Co. has an enhanced version of FIG FORTH ready for immediate delivery. It is supplied on an 8 in. single density diskette, ready to run on any system with CP/M and at least 24K of memory. A FORTH style editor with 20 commands is included, as well as a virtual memory sub-system for software which is permanently stored on diskettes, then loaded when needed. The user may also make permanent additions to the resident FORTH vocabulary. A Z-80/8080 assembler is also included, allowing the user to create new FORTH definitions which compile directly into machine code. All Z-80 or 8080 instructions may be used. The IF...ELSE..., BEGIN...UNTILL, and BEGIN...WHILE...control structures may be included in assembler definitions; these will automatically compile into appropriate machine code.

Other enhancements include an interleaved disk format that minimizes the time required for disk access. A 1024 byte disk block may be read or written in as little as 1/6 second. Eight of these blocks are maintained in RAM for immediate access and automatically swapped with others on the disk as they are needed.

The price is $75 for the 8 in. single density version, $90 for other diskette formats. Adequate documentation is included, suitable for the beginner as well as the experienced computer user.

FIG FORTH was originally defined by the FORTH INTEREST GROUP and is very close to the FORTH-79 international standard.

Mitchel E. Timin Engineering Co., 9575 Genesse Avenue, Suite E-2, San Diego, CA 92121.

DO-CASE STATEMENT

Dwight K. Elvey

OVERVIEW OF STATEMENT:

This is a DO-CASE written in FIG FORTH. It allows the operations of statements on the condition of a match of a case value and a case key. This DO-CASE also has a range case that allows the use of the condition to be done on a range of case key values. The NOT CASE and the NULL CASE concept are also allowed in this DO-CASE.

```
SCR # 19
  0 ( DO-CASE ALSO COMPILE { ...} LIKE COMPILE )
  1 : } ;
  2 : COMPILE{ ?COMP BEGIN R> DUP 2+ >R @ DUP ' }CFA = IF DROP
  3 1 ELSE , 0 ENDIF UNTIL ;
  4
  5 : DO-CASE COMPILE >R 0 5 ; IMMEDIATE
  6 : CASE 5 ?PAIRS COMPILE{ R = OBRANCH } HERE 0 , 7 ; IMMEDIATE
  7 : RANGE-CASE 5 ?PAIRS COMPILE{ R SWAP - 0< 0= OBRANCH } HERE 0 ,
  8  COMPILE{ R - 0< OBRANCH} HERE 0 , HERE COMPILE BRANCH HERE 0 ,
  9  HERE SWAP >R ROT >R R - R> 1 OVER - SWAP 1 R> 7 ; IMMEDIATE
 10 : END-CASE 7 ?PAIRS COMPILE BRANCH HERE 0 , SWAP HERE OVER -
 11  SWAP ! SWAP 1+ 5 ; IMMEDIATE
 12 : END-DO-CASE 5 ?PAIRS -DUP IF 0 DO HERE OVER - SWAP ! LOOP
 13  ENDIF COMPILE{ R> DROP} ; IMMEDIATE ;S
 14
 15
```

```
SCR # 29
  0   ( EXAMPLE OF DO CASE )
  1 : EXAMPLE DO-CASE
  2 4 CASE ." THE NUMBER WAS 4 " CR END-CASE
  3 5 3 RANGE-CASE ." THE NUMBER IS 3 OR 5 " CR END-CASE
  4 6 CASE ." THE NUMBER IS 6 " CR END-CASE
  5 ( NULL OR NOT CASE ) ." THE NUMBER ISN'T 3,4,5 OR 6 " CR
  6 END-CASES ; ;S
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

WHAT EACH DEFINITION FOR DO-CASE DOES:

DO-CASE consumes the case key value to be used later by the individual cases. This is the initialization statement for a DO-CASE field.

CASE does a comparison of the case key value and a case value. If a match is found the statements between CASE and the next END-CASE are done, then operation is picked up after the END-DO-CASE statement; else operation continues after the END-CASE statement and continues until END-DO-CASE or the next successful case.

RANGE-CASE does a comparison of the case key value and an inclusive range of values set by the two case values. The first case value on the stack must be greater in value then the next case value on the stack. The operation of RANGE-CASE is otherwise the same as CASE.

END-CASE indicates that the conditional CASE or RANGE-CASE is ended. It must be paired with any use of CASE or RANGE-CASE.

END-DO-CASE is used to close a DO-CASE field. Its main purpose is to do the cleaning of the stack and provide an exit point for the CASE statements. DO-CASE must be paired with a closing END-DO-CASE.

GLOSSARY ENTRIES

CASE

n --- (run-time)
n --- addr n (compile)
Used in a colon-definition in the form:

```
n(1) DO-CASE ... n(2) CASE (fp) ... END-CASE

(fp) ... END-DO-CASE
```

At run-time a comparison of n(1) and n(2) is done. If there is a match the true part is executed, then execution resumes after END-DO-CASE. If there is no match execution continues at the false part (fp). It must be followed by an END-CASE and an END-DO-CASE. It must be preceded by a DO-CASE.

At compile-time CASE compiles a branch and reserves space for an offset at addr. addr and n are used by END-CASE to resolve the offset and for error testing.

DO-CASE

```
n ---       (run-time)
--- n1 n2   (compile)
```

Used in a colon-definition in the form:

```
n(1) DO-CASE ... n(2) CASE (tp) ... END-CASE
(fp) ... END-DO-CASE
```

At run-time it consumes the value on the stack to be used later by case statements. This is used to initialize a do case field. See CASE for its use.

At compile-time DO-CASE leaves a case count (n1) and a value for error testing (n2).

END-CASE

```
---                 (run-time)
n1 addr1 n2 --- addr2 n3 n4 (compile)
```

At run-time it is used to terminate a CASE or RANGE-CASE statement. See CASE or RANGE-CASE for its use.

At compile-time it takes a value for an error check (n1), an address (addr1) to resolve an offset and a value that is the number of cases. It leaves a value for error checking (n4) , a value with a new case count (n3 = n1 + 1) and an offset at address (addr2) to be used later.

END-DO-CASE

```
---                 (run-time)
addr(1) addr(2) ... addr(n1) n1 n2
---                 (compile)
```

At run-time this terminates a DO-CASE field. See DO-CASE or CASE for its use.

At compile time it takes a case count (n1) and the count number of addresses to be used to resolve offsets and a value to use for error checking (n2).

RANGE-CASE

```
n1  n2  ---             (run-time)
n --- addr n   (compile)
```

Used in a colon-definition in the form:

```
n(1) DO-CASE ... n(2) n(3) RANGE-CASE (tp)
... END-CASE (fp) ... END-DO-CASE
```

At run-time a comparison of n(1) and the inclusive range of n(2) and n(3) is done. If there is a match the true part (tp) is executed, then execution resumes after END-DO-CASE. It must be preceded by a DO-CASE. n(2) must be greater than or equal to n(3) to do a successful case.

At compile-time RANGE-CASE compiles a branch and reserves space for an offset at addr. addr and n are used by END-CASE to resolve the offset and for error testing.

EXAMPLE OF USE:

SCR # 29 is an example of the use of DO-CASE. It shows the use of CASE, RANGE-CASE and null or not-case. In order to use it type in SCR # 19 first then SCR # 29. It is used by

typing a number, then EXAMPLE. The result will be a comparison of the number you typed and the comparisons done in the DO-CASE.

ADVANTAGES AND DISADVANTAGES:

The main disadvantage is that DO-CASE uses the return stack like DO ... LOOP does. This means that a value can not be passed on the R-stack from the outside of the DO-CASE field to the inside or vice-versa. Also this means that if the loop value I is to be used it must be on the operation stack before entering the DO-CASE.

The advantages of this DO-CASE are that it has a RANGE-CASE and the ability to allow the concept of not or null-case. This allows it to be used for something like an input entering routine for something like an editor. The CASEs can be used to prescan for special keys, the RANGE-CASEs can be used as a capitals only routine and the null-case used to do the normal entry.

Dwight K. Elvey
Santa Cruz, CA 94065

Judge's Comments -

This entry performs the functions of the FORTH-85 CASE statement. It also provides compile-time syntax checking, allows a range of indices to be treated as a single case, and offers a "none-of-the-above" case.

Compiling the same list of run-time words for each case results in excessive space overhead (about 28 bytes for each RANGE-CASE). Defining some new run-time words would save space without adding much execution time.

Also, using " - 0<" to check the index against a range gives the wrong result if the subtraction overflows.

FIG NORTHERN CALIFORNIA MONTHLY MEETING REPORT

28 June 80

FORML Session -

Tom Zimmer described the product of his last two weeks effort - tinyPASCAL (written in FORTH, of course). Two of his remarkable routines include the use of Ragsdale's table structure (C.F., Morse code tutorial, 24 May 80 FIG Meeting) in a Tokenizer and his technique of recursion uning dummy pointer-variables. The PASCAL design came from a "Byte" (Sep-Nov 78) series of articles which instructed the reader to do it in BASIC. Tom's first version of PASCAL-under-FORTH occupies some forty blocks.

FIG Meeting -

Three technical talks were delivered. Michael Perry described a CP/M File System written in FORTH which gives a 8080/Z80 version of FORTH compatibility with and use of extant CP/M data files.

Kim Harris spoke about arrays, i.e. how tables are created by alloting space to named variables and accessing array components by manipulating an index.

Bill Ragsdale discussed database concepts after FORTH, Inc.'s poly-FORTH and the organization of fields within files. Their FORTH definitions and demonstrations of file manipulation "How to talk to mass storage".

Announced was the availability of source code for FORTH on the 6809 running under SWTPC's FLEX 1.9. This is copyrighted by Talbot and is available from FIG for $10. See order blank.

Regarding FIG organizational business, two volunteers were asked to step forward - one to organize meetings, sequence schedules and distribute tasks (Ragsdale estimates 3 hrs/mo effort needed) and the other to take up the meeting announcement effort.

;s Jay Melvin

=A CASE IMPLEMENTATION=

William S. Emery

Yet another CASE implementation, this for either the TI990 or the Motorola 6800.

The objectives of this implementation were:

1. To provide a clear source program structure when using CASE, i.e. no compiler directives.

2. To provide a direct exit from any executed CASE to the next program statement.

3. To provide an ELSE (or Trap) statement within the CASE structure.

Please note: in both my 990 and 6800 implementations of FORTH all compiled addresses are 16 bits. No relative addressing is used.

The compiling word 'CASE creates a dictionary entry as follows:

1. The code address of (CASE).

2. The source argument to be compared. This eliminates the compilation of LITERAL and the necessity of moving the argument to the stack.

3. The branch address for I when not true. This is the address of the next CASE statement in the list.

A complete CASE statement requires three unique words:

<CASE , pronounced "open case,"
CASE , pronounced "case," and
CASE> , pronounced "case closed."

A sample of use is:

```
: TEST
     <CASE 1  ." FIRST"
      CASE 5  ." FIFTH"
      CASE 7  ." SEVENTH"
      ELSE .  ." NOT VALID"
      CASE> ;
```

At compile time <CASE places a zero delimeter on the stack, compiles to the dictionary (CASE), the source argument, and a nul, which will become the not true branch address. CASE then compiles a standard ELSE, which resolves the preceding not true, and deposits a nul address, to be resolved by CASE>. The address of this nul cell is left on the stack. Finally, CASE> resolves all addresses on the stack to itself until the opening nul is encountered.

```
CODE (CASE)     ( TI990 ASSEMBLER )
     I )+ S ) C     ( COMPARE ARGUMENT TO STACK )
     0= IF   S INCT   I INCT  ( POP STACK ENTER PROC )
        ELSE   I ) I MOV      ( SET UP BRANCH ADDR )
        THEN   NEXT

: 'ELSE   \ (ELSE)   HERE 0 ,   HERE ROT !  ;
: 0,        32 WORD  NUMBER , ;

: 'CASE    \ (CASE)   0,   HERE 0 ,  ;

: <CASE      0   'CASE  ;                        IMMEDIATE
: CASE     'ELSE 'CASE  ;                        IMMEDIATE
: CASE>    BEGIN   HERE SWAP !   ?DUP 0= END  ;  IMMEDIATE
```

A dictionary map of the compiled source would be as follows:

```
(headers omitted - addresses in hex )
XX00   (case) 0001 XX12 (.") 5F IR ST (else) XX44
XX12   (case) 0005 XX24 (.") 5F IF TH (else) XX44
XX24   (case) 0007 XX34 (.") 7S EV EN TH
XX34   (else) XX44 (.") 9N OT bV AL ID
XX44   (;)
```

While using byte offset addressing for the branches would have saved one or two bytes per CASE statement, to do so would violate the definition of word aligned dictionary established at the recent Standards Team meeting.

The word incorporating the CASE paragraph is entered with any 16 bit value on the stack. Any CASE statement finding the stack equal to its argument pulls the entry from the stack. If no CASE statement matches the stack parameter the value remains for the ELSE statement, if used, or beyond the "case closed" point.

This procedure executes (and compiles) nicely on the byte oriented Motorola 6800 by using the following definition for (CASE).

```
CODE (CASE)       ( M6800 ASSEMBLER )
     I LDX   0 ) LDX  N STX   ( SAVE ARGUMENT )
       TSX   0 ) LDX  N CPX   ( COMPARE TO STACK )
     0= IF     A PUL  B PUL   ( POP STACK )
               I LDX  INX INX INX INX  ( ENTER PROC )
       ELSE    I LDX  2 ) LDX  I  STX  ( SET BRANCH )
       THEN    NEXT
```

Thank you for the opportunity to submit this. I think the contest idea is a great one. How about some future contests on +LOOP, the Bartholdi "TO" concept and/or Data Structures. If publication space permits I'd also be interested in a competition on SORT and/or an approach to precompiled, relocatable FORTH for virtual memory processing.

William S. Emery
Costa Mesa, CA 92626

Judges' Comments - This entry achieves its objectives with only 7 short and well-factored new word definitions. The CODE word could have been written in high-level. While having to specify the case keys as numbers at compile time is a restriction, it is adequate for many applications. And it does simplify the source code.

FIG NORTHERN CALIFORNIA MONTHLY MEETING REPORT

26 July 80

FORML Session -

Henry Laxon presented his string package which has been his first FORTH programming effort. He pointed out that this package was designed for a computerized type-setting task and not text editing. The word "string" takes a length parameter and name and is manipulated so to find, concatenate, parse, move and so forth.

John Cassady then outlined his string package which he fashioned after Northstar's BASIC. He pointed out it's file handling utility and a discussion arose regarding screen windows, input windows and video segmentation. Amazing how FORTH gets strung along.

FIG Meeting -

Announcements included the report of over 25 attendees at Kim Harris' Humbolt State FORTH class.

Allyn Saroyan described the problems he's had trying to convert code from other machines and asserted that we ought to submit code along with its algorithm and perhaps even assembler particulars.

Don Colburn, from Creative Solutions, mentioned a FORTHcoming tutorial under CP/M with stackgraphics.

Bob Smith reviewed progress and problems of the floating point standards team effort.

John James described Cap'n Software's Apple editor.

Bill Ragsdale spoke briefly about the Installation Manual version editor and code was shown on how to extend FORTH, Inc.'s editor.

A preview copy of the August 1980 Byte magazine was passed around. See the order form to get your copy.

;s Jay Melvin

APPLE - 4th CASE

F.W. Fittery

Here is a select case for Apple-4th. The Apple works so-far and allows any level of nesting of any of the allowable structures plus more BEGIN-CASES, END-CASES. You will get a lot of failures if you do not balance your (BEGIN-CASES==END-CASES) and your (CASE==END-CASE). Also be aware the the top of stack is still available if none of the case statements are executed. Otherwise the top of stack is eaten up by the case statement. When BEGIN-CASES is encountered 0 is placed on the stack for END-CASES. When CASE is encountered at compile time OVER = ZERO-BRANCH 0 , DROP is compiled inline. When END-CASE is encountered the ZERO-BRANCH for the matching case is patched to the proper jump point. When END-CASE is found all forward jumps set-up by END-CASE are resolved. This is done with a BEGIN END looking for the 0 put on the compile time stack by BEGIN-CASES. Good luck.

Note: The general approach of the CASE statement is:

```
:TEST 5 OVER = IF DROP ." FIVE "  ELSE
      6 OVER = IF DROP ." SIX  "  ELSE
      7 OVER = IF DROP ." SEVEN"  ELSE
      DROP ." BAD INPUT"
                THEN  THEN  THEN  ;
```

Generates the same code as:

```
: TEST
   BEGIN-CASES
     34 CASE 34 . END-CASE
     35 CASE 35 . END-CASE
     36 CASE 36 . END-CASE
     DROP ." BAD INPUT"
   END-CASES ;
```

Note: You must use up so if no case is executed as if is left on the stack.

Case Documentation

The CASE statement format is as follows:

The result of the BEGIN-CASES, END-CASES is:

1 0 if a CASE option is executed
1 1 if no CASE option is executed

If no CASE option is executed the flow of execution starts after the last END-CASE. Because of this and the fact that the top of stack passed to the BEGIN-CASE is still on top of the stack you may drop the parameter or you may use it to do a calculation which is done only when none of the case options are selected:

Note: Though the code executes exactly the same code the format in Figure 1 is much easier to understand than that in Figure 2. It is also much preferable.

Case Statement

Figure 1.

```
: ESC-ESC ." ESC-ESC" ;
: NEC ." ESC-CTL-N" ;
: LEC ." ESC-CTL-L" ;
: SEC ." ESC-CTL-S" ;
: ESC KEY
     BEGIN-CASES
       27 CASE ESC-ESC END-CASE
       14 CASE NEC     END-CASE
       12 CASE LEC     END-CASE
       19 CASE SEC     END-CASE
        .
     END-CASES ;
: OUTPUT
   BEGIN-CASES
     27 CASE ESC     END-CASE
     14 CASE 91 DOT  END-CASE
     12 CASE 92 DOT  END-CASE
     19 CASE 95 DOT  END-CASE
     DOT
     END-CASES ;
```

```
 : MONITER BEGIN KEY DUP OUTPUT
          32 = END ;
;S
```

Figure 2.

```
: ESC-ESC ." ESC-ESC" ;
: SDC ." ESC-CTL-S" ;
: LEC ." ESC-CTL-L" ;
: NEC ." ESC-CTL-N" ;
: OUTPUT
    BEGIN-CASES
      27 CASE KEY
           BEGIN-CASES
              27 CASE ESC-ESC  END-CASE
              14 CASE NEC     END-CASE
              12 CASE LEC     END-CASE
              19 CASE SEC     END-CASE
              .
           END-CASES
                       END-CASE
      14 CASE 91 DOT   END-CASE
      12 CASE 92 DOT   END-CASE
      19 CASE 95 DOT   END-CASE
      DOT
      END-CASES ;
 : MONITER BEGIN KEY DUP OUTPUT
          32 = END ;
;S
```

Support Words

```
: DOT 0  'S 1 + 1 TYPE DROP ;
: '?  WORD HERE CONTEXT @ @ FIND ;
: LIT R> 2 + DUP >R @ 2 + ;
: .PFETCH R> R> 2 + DUP >R @ 2 + SWAP
         >R ;
;S
```

(CASE)

```
: BACKSLASH .PFETCH 2 - , ;
: BRANCH R> 2 + @ >R ;
: ZERO-BRANCH 0= IF R> 2 + @ >R
                 ELSE R> 2 + >R THEN ;
: BEGIN-CASES 0 ; IMMEDIATE
: OVER= OVER = ;
: CASE BACKSLASH OVER=
  BACKSLASH ZERO-BRANCH HERE 0 ,
  BACKSLASH DROP ; IMMEDIATE
: END-CASE BACKSLASH BRANCH HERE 0 ,
  SWAP HERE 2 - SWAP ! ; IMMEDIATE
: END-CASES BEGIN -DUP IF HERE 2 - SWAP
  ! 0 ELSE 1 THEN END ; IMMEDIATE
PRINT-OFF
85 85 PLIST
( GENERAL 8-BIT SELECT CASE CODE        )
( NEEDS BACKSLASH ( \ )  TO WORK        )
( FOR 16 BIT VERSION SEE # S 1,2,3      )
: XXX IF ELSE THEN ;
' XXX DUP @ SWAP 2 + @
( #1:  2 BECOMES 1 IN THE ABOVE LINE)
  FORGET XXX
  CONSTANT BRANCH
  CONSTANT ZERO-BRANCH
: BEGIN-CASES 0 ; IMMEDIATE
: OVER= OVER = ;
: CASE BACKSLASH OVER=
  ZERO-BRANCH , HERE 0 ,
  BACKSLASH DROP ; IMMEDIATE
: END-CASE BRANCH , HERE 0 ,
  SWAP HERE 2 - SWAP ! ; IMMEDIATE
( #2:    2 BECOMES 1 IN THE ABOVE LINE)
: END-CASES BEGIN -DUP IF HERE 2 - SWAP
( #3:    2 BECOMES 1 IN THE ABOVE LINE)
  ! 0 ELSE 1 THEN END ; IMMEDIATE
```

E. W. Fittery
International Computers
Mount Arlington, NJ 07856

Judges' Comments - Interesting but rather limited.

DO-CASE EXTENSIONS

Bob Giles

Upon using the DO-CASE structure offered by Rick Main in the Vol. 1, No. 5 issue of Forth Dimensions, I came across several instances where the power of this tremendously useful construct can be improved. The first is where several options are defined using the CASE and END-CASE structure, but all remaining cases have a common option. The other feature is where the DO-CASE variable is to be tested within a certain range of values instead of strict equality to one value per CASE. In order to maintain symmetry, some renaming of the keywords was necessary. The old structure looks like this:

```
DO-CASE
  w CASE.............END-CASE
  x CASE.............END-CASE
              ...
              ...
  z CASE.............END-CASE
END-CASES
```

My structure looks like this:

```
DO-CASE
  a CASE.............END-CASE
b c CASES............END-CASES
              ...
              ...
  J CASE.............END-CASE
k l CASES............END-CASES
  m CASE.............END-CASE
    OTHERWISE........
END-DO-CASE
```

The lower case letters indicate operations that leave a 16 bit value on the stack. DO-CASE is symmetrical with END-DO-CASE, CASE is symmetrical with END-CASE, CASES is symmetrical with END-CASES and OTHERWISE, well....

OTHERWISE is useful when there are several courses of action for certain values of the DO-CASE variable, and a common routine for all the other cases. This closes any "loopholes" for erroneous values that can occur. This is easily implemented by putting the common routine after the last END-CASE and before the END-CASES in Rick's DO-CASE structure. However, for readability and documentation, I defined a dummy word, OTHERWISE , (i.e. : OTHERWISE ; IMMEDIATE), to mark the action. Making this work an IMMEDIATE word assures that run time is not affected. OTHERWISE must be used at this particular point in the DO-CASE structure, and has no meaning or usage anywhere else.

The need to test for equality to a value within a range leads to the CASES structure. Whereas x CASE tests the DO-CASE variable (VCASE) for x = VCASE, lo hi CASES tests VCASE to see if it satisfies lo $\leq$ VCASE $\leq$ hi. If VCASE is within the range of the lower boundary, lo, and the higher boundary, hi, then the appropriate statements are executed within the CASES...END-CASES statement (this is the newer word - don't confuse it with Rick's END-CASES). If VCASE is out of range, these statements are skipped and execution resumes after the END-CASES (new word) statement.

The listing of the structure is in the figure (see enclosure). The minor changes include - changing the name of END-CASES, making a dummy word called OTHERWISE, and defining the new word CASES.

The simplicity of CASES does not reflect the time it took to get it working. (A fairly lengthy interactive Forth debugger was written to help with the development). The basic idea is to subtract the upper limit from VCASE minus one and see if the result is zero or positive (i.e., the carry flag IS set). If the carry flag IS set, then the result is out of range and the Forth instruction pointer (kept in

the BC pair) has to be incremented so that the next "instruction" executed will be the one after END-CASES. The action is the same as when VCASE does not match in the CASE statement. If the carry flag is NOT set, then VCASE is less than or equal to the upper bound and possibly in range. If VCASE is less than the upper bound, the lower bound is subtracted from VCASE. If the result is negative (i.e., carry is NOT set), then VCASE is out of range and IP is incremented to resume after END-CASES. If the result is positive or zero (i.e., the carry flag IS set), then VCASE is between or equal to the upper and lower boundaries. In this case, the statements between CASES and END-CASES are executed. At END-CASES, execution jumps to after the END-DO-CASE statement and continues.

Two interesting concepts were included in this implementation. The first was the use of the assembly language CALL. The ' (tick) causes the code field pointer of the next word to be placed on the stack. The Forth CALL takes this address from the stack and assembles the CALL opcode and the address into the dictionary. At run time, the call to the -TOP subroutine is executed, and the 8080 program counter is pushed on the top of the stack. Within -TOP, the H POP takes the return address to HL, and then exchanges it with the top item (the boundary) so that the return address will be on top of the stack when RETurn is executed.

' AFTER-END-CASES leaves an address on the top of the compile time stack which is assembled into the dictionary by the JMP in code CASES. At run time, this AFTER-END-CASES segment serves as an extension to machine code in code CASES. Although this type of programming is a GOTO type of construct, it is used here to keep the definition of code CASES short. It also adds insight as to the intent of extended segment by the use of a name. My advice to other programmers is to use this jump around feature very sparingly, so as to remain in keeping with the concepts of structured programming.

The TEST for the new DO-CASE is listed on screen 153. It differs from the program that Rick submitted in that the various variables are to be entered on the stack before executing TEST. This way, all 65,536 possibilities can be tried instead of only the 128 available from an ASCII keyboard.

All of the following was done using Zendex SBC-FORTH V 1.0 for an 8080 processor.

A final note is in order. The earlier DO-CASE had a bug in it pertaining to the address used to store VCASE. Notice that my routines deleted the ' (tick) which preceded VCASE in lines 3 and 4 of the first screen that Rick sent (see Vol. 1, #5 of Forth Dimensions, pg. 51). This is because ' VCASE causes the address of the parameter field to be put on the stack, rather than the location of VCASE in the RAM area. Although the earlier DO-CASE works, fetching VCASE always yields a zero.

Bob Giles
Magnetic Media, Inc.
Tulsa, OK

Judges' Comments -

More of an extension of previous work than a new CASE.

```
SCREEN  150
  0 ( DO-CASE STATEMENTS                              4-4-80 BG )
  1 BASE C@
  2   VOCABULARY FORTH+    FORTH+  DEFINITIONS
  3
  4   ( DO-CASE CASE END-CASE CODE DEFINITIONS )
  5 0 VARIABLE VCASE
  6 CODE DO-CASE H POP VCASE SHLD I INX I INX NEXT JMP
  7 CODE CASE W POP VCASE LHLD L A MOV W 1+ CMP
  8      0= NOT IF I LDAX I 1+ ADD A I 1+ MOV NEXT JNC
  9               I INR NEXT JMP THEN H A MOV W CMP
 10      0= NOT IF I LDAX I 1+ ADD A I 1+ MOV NEXT JNC
 11               I INR NEXT JMP THEN I INX NEXT JMP
 12 CODE END-CASE  I LDAX A L MOV I INX I LDAX A H MOV
 13      H PUSH I POP NEXT JMP
 14 BASE C! ;S    ( END CODE DEFINITIONS )
 15 ( COPIES FROM FORTH DIMENSIONS V1-5 pg  50/51    BG 4-4-80 )


SCREEN  151
  0 ( DO-CASE EXTENTIONS                          6-2-80 BG  )
  1
  2 CODE -TOP     H POP  XTHL  XCHG  E A MOV  CMA  A E MOV  D A MOV
  3               CMA  A D MOV  D INX  D DAD RET
  4 (NOT TO BE CALLED FROM HIGH-LEVEL)
  5 CODE AFTER-END-CASES  B LDAX  C ADD  A C MOV  NEXT JNC
  6                       B INR   NEXT JMP
  7
  8 CODE CASES  VCASE LHLD H DCX XCHG ' -TOP CALL
  9     CS      IF D POP ' AFTER-END-CASES JMP  THEN
 10             VCASE LHLD XCHG ' -TOP CALL
 11     CS NOT  IF ' AFTER-END-CASES JMP  THEN  B INX NEXT JMP
 12
 13 CODE END-CASES  I LDAX  A L MOV  I INX  I LDAX  A H MOV  H PUSH
 14                 I POP   NEXT JMP
 15     ;S


SCREEN 152
  0 ( CASES&OTHERWISE EXTENSIONS            5-22-80 BG )
  1 ( FORTH+  DEFINITIONS - COMPILER DO-CASE STATEMENTS )
  2
  3 : DO-CASE  COMPILE  DO-CASE HERE 0 0 , ; IMMEDIATE
  4 : CASE     COMPILE  CASE SWAP HERE 0 C, ; IMMEDIATE
  5 : END CASE COMPILE  END-CASE HERE 0 , SWAP HERE
  6            OVER - SWAP C! ; IMMEDIATE
  7 ( COPIED FROM FORTH DIMENSIONS V1-5 pg  50/51    BG 4-4-80 )
  8
  9 : CASES  COMPILE CASES SWAP HERE 0 C, ; IMMEDIATE
 10 : END-CASES  COMPILE  END-CASES HERE 0 , SWAP HERE OVER - SWAP
 11              C! ;   IMMEDIATE
 12 : OTHERWISE ; IMMEDIATE   ( NULL DEFINITION )
 13 : END-DO-CASE BEGIN HERE SWAP ! -DUP 0 = END ; IMMEDIATE
 14 FORTH+     ;S
 15


SCREEN  153
  0 ( TEST FOR EXTENDED DO-CASE                    5-22-80 BG )
  1  BASE C@ HEX
  2 : MONITOR     DO-CASE
  3           40  CASE  QUIT          END-CASE
  4           41  CASE  ." AAAA "    END-CASE
  5           42  CASE  ." BBBB "    END-CASE
  6           43  CASE  ." CAT  "    END-CASE
  7       30  39  CASES ." NUMBERS " END-CASES
  8       OFE 102 CASES ." CROSS "   END-CASES
  9               OTHERWISE ." NOT TESTED "
 10               END-DO-CASE ;
 11
 12 : TEST  BEGIN DUP MONITOR 0 = END ;
 13
 14 BASE C! ;S
 15
```

ENTRY FOR THE FIG CASE CONTEST

Arie Kattenberg

An Overview of the CASE Statement

Externally the CASE statement looks like:

```
m  n  CASE ..... ESAC
   k  CASE ..... ESAC
   .  . .          . .
   .  . .          . .
   .  . .          . .
   .  . .          . .
   .  . .          . .
   .  . .          . .
   l  CASE ..... ESAC
      ENDCASE
```

- If a comparison is not 'true' (m≠n) the m stays on the stack and is tested against the next CASE.
- If a CASE is met the m is dropped and after the case body is executed, the ESAC transfers control to words following ENDCASE.
- If none of the CASES is met, ENDCASE has compiled a DROP that now drops the m instead of one of the CASES doing that.

If we want explicitly some (stack) operations to be done when none of the cases is met, the m that remains on the stack there would be bothering. We then use:

```
m  n  CASE ..... ESAC
   k  CASE ..... ESAC
   .  . .          . .
   .  . .          . .
   .  . .          . .
   .  . .          . .
   l  CASE ..... ESAC
      OTHER ..... ENDCASE
```

Now the 'OTHER' has compiled a drop for the m and ENDCASE does not compile a drop.

In both the above examples we can nest other case structures in any of the case bodies. This is another reason for using 'OTHER' sometimes.

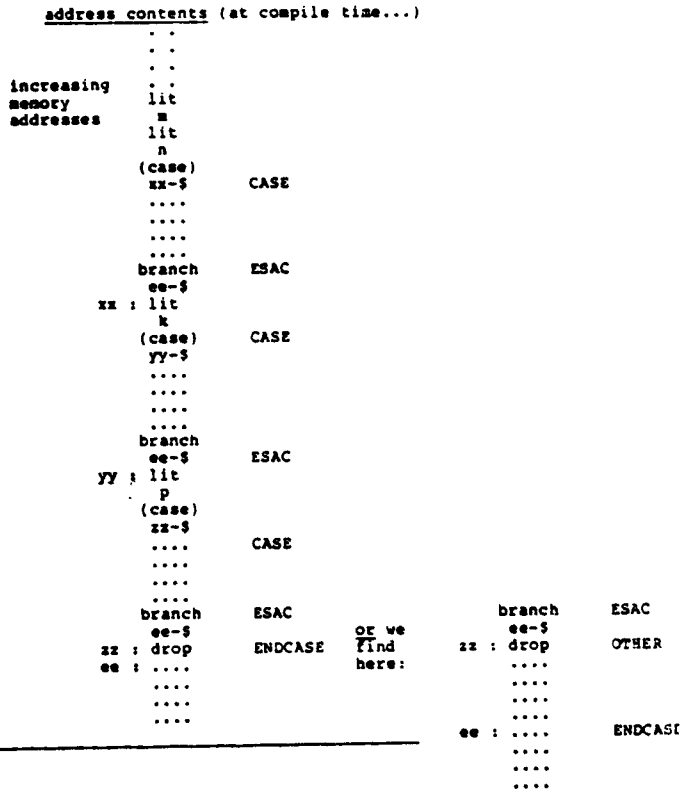
Though this is in no way essential to the above structures I have chosen a high level branch in the conditional branch that is compiled by CASE (i.e. (CASE) manipulates the return stack contents to effectuate a branch). Now it is simple, machine independent and self explaining to make words like:

>CASE <CASE CASES ODDCASE etc.

that can take the place of CASE in the above examples. (Of course this can be done using machine language conditional branches for these elements just as well.)

By the way: The m, n, k and l in the examples may be any amount of FORTH that puts a number on the stack.

Internally making a picture of a compiled CASE structure: (e.g.)



Instead of the (CASE) cfa's we may find examples of:

(>CASE), (<CASE), (CASES) etc.

there in a more advanced example.

The m, n, k, p here are compiled literals, but there may be all sorts of FORTH compiled there.

Source definitions in fig-FORTH words

```
SCR #100

:  CASE control structure                                  AK-80Feb29 )
:  ZBRAN           (Bi-level branch if BOT is zero, used by CASE *)
       RP@    2+  SWAP
       IF SWAP DROP 2 ELSE DUP @ @ THEN SWAP +! ;
:  COBR                        ( Complete a pending forward branch *)
       HERE OVER - SWAP ! ;
:   (CASE)    (Compiled by CASE, do a test and conditionally branch *)
       OVER - ZBRAN ;
:  CASE        (Execute until ESAC if Key-2 equals Case-1 *)
       ?COMP COMPILE (CASE) HERE 0 , 5 ; IMMEDIATE
:  ESAC          (Close a CASE; Key is left if case not done*)
       5 ?PAIRS
       COMPILE BRANCH HERE 0 , SWAP COBR 4 ; IMMEDIATE
:  OTHER          (After last ESAC, if stack or nested CASEs used there *
    4  ? PAIRS  COMPILE  DROP   6  ; IMMEDIATE
;S
```

```
SCR #101

:  CASE control structure                          AK-80Feb 29 )
:  ENDCASE              (Close a CASE control structure  *)
       DUP 4 - IF COMPILE DROP ELSE 6 ?PAIRS 4 THEN
       BEGIN  DUP  4  -  SP@  CSP  @  <  AND
       WHILE  DROP  COBR
       REPEAT  ;  IMMEDIATE
;S
```


An 'English' explanation of how the words work:

ZBRAN

Finds 'true' or 'false' on the stack. It fetches the address of the second return stack number which is the pointer (stored IP) in the list where a branch can occur.

- If 'true' was on stack, the pointer is incremented by 2 (making next skip to the CFA following the branch) and the second stack number is dropped. (It was the 'key' to the 'case'.)

- If 'false' was on stack, the pointer is incremented by the value that is found in the location where it is pointing to (making NEXT to resume interpretation of the list where the branch was compiled at a new location).

COBR

Finds an address on stack; a distance from the actual DP value to that address is compiled in the address.

(CASE)

Finds two numbers on stack, compares these and leaves the 2nd on stack. Then control is transferred to ZBRAN. The CFA of (CASE) appears in compiled lists as a relative branch (the relative jump following it in the list).

CASE

Has precedence and checks whether we are compiling when we use it. It compiles (CASE) and puts the address following (CASE)'s CFA in the list, on stack. It stores a temporary Ø in that location and puts a 5 for pair checking on the stack.

ESAC

Does a pair check on the 5 it expects from CASE. It compiles a BRANCH, puts a temporary Ø in the location following that BRANCH and puts the address of that location on stack.

The branch that is half made by the previous CASE is completed. For pair check by the following ENDCASE a 4 is put on stack; the change of check digit (from 5 to 4) makes the nesting of other case structures in CASE .... ESAC possible. (ESAC has precedence.)

OTHER

Has precedence, it does pair check on the 4 it expects from ESAC. It compiles a DROP for the key (for the CASES that are all not fulfilled when this point is reached). The check digit 6 is put on stack. The change from 4 to 6 as a check digit signals to ENDCASE that the 'OTHER' is used and it makes nesting of other case structures in OTHER .... ENDCASE possible.

ENDCASE

Checks for a 4 on stack; in case there is a 4 the "OTHER" is not used and we must compile a DROP here. If there is not a 4 there must be a 6 (which is checked); it is replaced by a 4.

The rest of ENDCASE looks for 4's on stack that are placed there by the previous ESAC's (since there may be 4's on stack already before the definition that contains the case structure. ENDCASE also checks SP@ against CSP contents).

The incomplete branches from the ESAC's are completed until none is left.

Glossary entries for each word in sheet B

ZBRAN mf --- (if f is true)
mf --- m (if f is false)

Procedure to perform the branch for a high level run time conditional branch in a CASE control structure.

If f is false (zero), the in-line parameter following the compiled reference to the run time conditional branch is added to the stored interpretive pointer (second word on the return stack) to effectuate a branch.

If f is true, 2 is added to skip the in-line parameter and m is dropped. Used by (CASE), (>CASE), (CASES) etc.

COBR addr1 ---

Calculate the branch offset from addr1 to HERE and store in addr1, thus resolving a pending forward branch.

(CASE) m n --- (if m equal n) C2
m n --- m (if m unequal n)

The run time procedure to conditionally branch in a CASE control structure.

If m equals n, no branching occurs and NEXT interprets the words following the branch offset in the dictionary after the CFA of (CASE).

If m is unequal to n, m remains on stack and NEXT resumes interpretation with a new interpretive pointer value according to the branch offset.

Compiled by CASE. For branching, 2BRANCH is used.

```
CASE m n ---   (run-time, if
               m equal to n)          P,C2
     m n --- m (run-time, if
               m unequal to n)
         --- addr c (compile)
```

Occurs in a colon-definition in the form:

```
CASE ..... ESAC
CASE ..... ESAC
  ....      ....
CASE ..... ESAC
ENDCASE
```

or:

```
CASE ..... ESAC
CASE ..... ESAC
  ....      ....
CASE ..... ESAC
OTHER ..... ENDCASE
```

At run-time CASE selects execution based on an equality test of the two numbers on stack. If m equals n the part until the next ESAC is executed and then control is passed to after ENDCASE. If m is not equal to n, m remains on the stack and control passes to after the following ESAC. The use of OTHER and its 'other' part are optional. ENDCASE, or (if present) OTHER, drops the remaining m.

At compile time CASE compiles (CASE) and reserves space for an offset at addr. addr and c are used later for resolution of the offset and error testing.

```
ESAC addr1 c1 --- addr2 c2
     (compiling)                      P,C2
```

Occurs within a colon-definition in the form:

```
CASE ..... ESAC
CASE ..... ESAC
  ...       ...
CASE ..... ESAC
ENDCASE
```

or:

```
CASE ..... ESAC
CASE ..... ESAC
  ...       ...
CASE ..... ESAC
OTHER ..... ENDCASE
```

At run-time ESAC executes after the part selected by the CASE it pairs to. ESAC branches over the following cases and resumes execution after ENDCASE.

At compile time ESAC compiles a BRANCH, reserving room for a branch offset at addr2, leaving addr2 and c2 for later resolving of the offset and error checking. ESAC also resolves the pending branch from the previous CASE at addr1, storing the ofset from addr1 to HERE.

```
OTHER m ---      (run-time)             C,P
      c1 --- c2  (compiling)
```

Occurs within a colon definition in the form:

```
CASE ..... ESAC
CASE ..... ESAC
  ...       ...
CASE ..... ESAC
OTHER ..... ENDCASE
```

At run-time OTHER executes when none of the cases is met. OTHER drops the m against which the cases were tested.

At compile time OTHER compiles a DROP. OTHER also checks the c1 from the last ESAC for error testing and puts c2 on stack to signal ENDCASE that OTHER has been used and to make nesting of new case structures possible between OTHER and ENDCASE.

```
ENDCASE addr1 c1 addr2 c1 .....
        addrn-1 c1 addr-n c2 ---
        (compiling)                     P,C
        m --- (run-time, no OTHER used)
```

Occurs in a colon-definition in the form:

```
   CASE ..... ESAC
   CASE ..... ESAC
     ...       ...
   CASE        ESAC
   ENDCASE
or:
   CASE ..... ESAC
   CASE ..... ESAC
     ...       ...
   CASE        ESAC
   OTHER ..... ENDCASE
```

At run-time ENDCASE serves as the destination of all forward branches from the ESAC's in the case structure. If OTHER does not occur, ENDCASE drops the m that remained on stack when no case is met.

At compile time ENDCASE compiles a DROP if OTHER was not used in the case structure, which can be known from the value of C2. ENDCASE resolves all the pending forward branches from the ESAC's by storing the offset from addri to HERE in addri for addrl thru addrn. The C1's indicate the presence of such an unresolved branch as long as the control stack pointer is not passed.

Examples of the use of this statement

```
SCR #102

 0 ( Examples of CASE control structure              AK-80Feb 29 )
 1 : SEL             ( Write number -1 between 0 and 9 as text  * )
 2     0 CASE  ." Zero"   ESAC
 3     1 CASE  ." One"    ESAC
 4     2 CASE  ." Two"    ESAC
 5     3 CASE  ." Three"  ESAC
 6     4 CASE  ." Four"   ESAC
 7     5 CASE  ." Five"   ESAC
 8     6 CASE  ." Six"    ESAC
 9     7 CASE  ." Seven"  ESAC
10     8 CASE  ." Eight"  ESAC
11     9 CASE  ." NINE"   ESAC
12     ." Outside range 0-9" ENDCASE ;
13
14
15 --
```

```
SCR #103

 0 ( Examples of CASE control structure                    AK-80Feb29 )
 1 : EXAMPL ( Some text is printed, selected by number -1 & number -2 *)
 2   5 CASE 12 CASE   ." This"          ESAC
 3             3 CASE   ." is"          ESAC
 4               OTHER  ." only"        ENDCASE ESAC
 5   18 CASE 2 CASE   ." a very"        ESAC
 6             7 CASE   ." silly"       ESAC
 7               OTHER  ." example"     ENDCASE ESAC
 8     OTHER 2 CASE   ." of the use"    ESAC
 9             9 CASE   ." of nested"   ESAC
10               OTHER  ." cases"       ENDCASE ENDCASE ;
11
12 ; S
13
14
15
```

Short discussion on the case statement presented here

The history of this case statement is outlined below:

A first try for CASE was a replacement of IF so we could produce case structures like:

```
Ia.  ... CASE .... THEN ...
     ... CASE .... THEN ...
     ... CASE .... THEN DROP ...
or
ib.  ... CASE .... ELSE
     ... CASE .... ELSE
     ... CASE .... ELSE DROP
         THEN THEN THEN
```

At run-time, Ib is the faster one since no other cases are tested once a case is done. A disadvantage however is the necessity to write the THEN ... THEN ... THEN series.

An improvement on Ib was to organize the DROP THEN THEN ... THEN by a new compiler word:

```
II.  ... CASE .... ELSE
     ... CASE .... ELSE
     ... CASE .... ELSE
         ENDCASE
```

But since ENDCASE can only see by the 2's for ?PAIR on stack how many branches have to be completed this structure II cannot be nested inside ... IF ... ELSE ... THEN or inside an other CASE ... ELSE.

This could be avoided by making a new "ELSE" and then using other numbers for ?PAIR checking in the structure.

By changing the number for pair checking in the new "ELSE" (ESAC), also nesting in other case structures is possible:

```
III.   ... CASE   ? PAIRS "a"   ESAC   "?PAIRS "b"
       ... CASE                 ESAC
       ... CASE                 ESAC

   ?PAIRS "b"   ENDCASE
```

Now, a remaining problem is the part between ESAC (the last) and ENDCASE. There the number against which the cases are checked is still on stack, so we cannot easily manipulate the stack there; also, at compile time we have the "b"'s for ?PAIR checking on stack so we cannot nest a new case structure there.

To solve these two problems we made the optional "OTHER" that performs the DROP at run time and that at compile time changes again the "check number" to inform ENDCASE that the DROP already has been compiled and to make nesting of other case structures possible.

Of course the nesting problem could have been solved by using an opening word like is done in the example on page 50 and 51 of Forth Dimensions, Vol. 1, No. 5. But this forces the use of an extra word at compile time. This opening word could e.g. store the top stack word on the return stack (not in a variable as is done in the example!, since this prohibits nesting of case structures). But I doubt whether it is an advantage to remove the number against which the cases are checked from the stack: This costs (run) time, makes it difficult to change that number between an ESAC and the next CASE (after all, why should one not be allowed to do this) and the number is not in the way as far as I can see.

The use of a high level (CASE) and the use of the separate ZBRAN there are not mandatory.

To have a fast executing case structure one may rewrite (CASE) in low level without affecting the essence of this case structure.

However, as presented here the structure is machine independent for standard fig-FORTH's.

Also this high level (CASE) makes it easy to extend the possibilities, e.g.:

```
: (>CASE) OVER < ?BRAN ; (BRANCH IF SMALLER THAN)
: >CASE COMPILE (>CASE) HERE 0 , 5 ; IMMEDIATE
```

and we have a new type of case.

or:

```
: (CASES) ROT >R R < SWAP R > AND R>
          SWAP ?BRAN ; (BRANCH IF NOT IN RANGE)
: CASES COMPILE (CASES) HERE 0 , 5 ; IMMEDIATE
```

etc. Any odd case you expect to use more than once can be incorporated in the set and used just like CASE.

;S

P.S. In re-reading all this I notice that "?COMP" is not needed in the definition of CASE; please omit.

P.P.S. My native language is Dutch; please forgive me any errors in the language.

Arie Kattenberg
Utrecht, Netherlands

Judges' Comments - This entry has a number of interesting ideas in it and could be useful to developers. The presentation is a bit hard to follow in places. A plus is the short history of the development of this CASE structure through several earlier forms.

=CASE CONTEST STATEMENT=

George Lyons

This entry submitted to the FIG Case Statement Contest is limited to providing a compiler syntax for writing equivalents of ALGOL "case" and "switch" statements in FORTH and some additional words to use in conjunction with ALGOL style case expressions. As such it does not solve all the problems posed in the contest announcement.

In formulating a case expression syntax the first decision was to treat case lists as in-line or literal expressions within : definitions rather than provide a special defining word creating words of a case list type. This increases flexibility of use at the expense of storage saving otherwise obtainable by exploiting the code address field of a case-type word. A second decision was to allow use of a list either to execute a case selected at run time or to compile the execution address of the case--for use in more complex compiler features. Storage for a list which was to be only executed turned out less than when compiling so different commands are provided for these two circumstances. A third decision was to include in the compiled code for a case list no number-of-cases parameter; hence no checking of the run time inputsubscript's validity is done in executing the cases. Instead separate words ?INDEX and EXCEPT are provided to do this checking, taking more storage when used than if their functions were built into the case list code, but saving time and space when they are not needed, as when the validity of the input is established elsewhere in a program.

The in-line case lists are handled as one instance of a general approach to in-line list functions in which a list is represented in the form ccc(...list data...). cc(is a Word which begins compiling the list and) is introduced as a word, in addition to its role in comments, to terminate the list. Different words ccc(perform different functions involving data or code stored in the list. The parenthesis was defined as a word because of the similarity of the run time process of skipping around data embedded within a definition and the compile time skipping past a comment in source code. The general approach compiles all lists with an execution address at the front which processes the data and returns controls to the point following the list; the address of this return point is stored immediately following the execution address at the beginning, and it has more uses than just returning control. When a list contains variable length elements a vector of addresses of the elements is appended to the end of the list in reverse order. The case lists are an example of this structure in which the data is a list of variable length code segments, written for instance using EXECUTE(CASE case 0 code... CASE casel code... CASE ...). The case compiling words such as EXECUTE(are written using utility Words available for building additional functions along the same lines.

Examples of the latter are some Words that might be used in conjunction with EXECUTE(...). These are mentioned briefly here, and some are implemented in the glossary and code section.

```
[ n ] EXCEPT EXECUTE( ... )
(compile time source)
```

At run time EXCEPT will check the subscript on the stack intended to select a case in EXECUTE(...) and replace it by zero if negative or greater than n. Case zero can then be especially written to handle these exceptions.

```
W+  addr n --- addr+2*n
```

Address arithmetic operation for byte addressable computers. Increments an address adr by n words. Can be implemented in machine code using shift operations instead of multiply.

```
: W+ 2 * + ;
```

W- addr n --- addr-2*n

Address arithmetic operation for byte addressable computers. Decrements an address adr by n words. See W+.

COMPILES --- addr n P,C

Used in conjunction with EXECUTES in a : definition to combine a procedure performing compiler operations with run time code for the procedure compiled, in a single definition. In the form : ccc ... COMPILES ... EXECUTES ... ; IMMEDIATE, ccc performs the operations up to EXECUTES at compile time; these compile time operations include COMPILES which compiles the address of the code following EXECUTES. EXECUTES places at that address a pointer to the code for : definitions, so that the code following EXECUTES is in effect a : definition without a name field.

```
:COMPILES ?COMP COMPILE COMPILE HERE n 2 ALLOT ; IMMEDIATE
```

EXECUTES addr n --- P,C

See COMPILES. Compiles ;S to terminate the compile time part of a dual definition, and stores the address of the next dictionary location in the space reserved by COMPILES. Compiles the address of the code for : definitions to begin the run time part of the Word.

```
: EXECUTES ?COMP [ HERE 2 - ] COMPILE ;S n ?PAIRS HERE
  SWAP ! COMPILE [ @ , ] ; IMMEDIATE
```

[n] ?INDEX EXECUTE(...)
(compile time source)

At run time ?INDEX will issue a system error message if the subscript on the stack intended to control EXECUTE(is invalid, instead of writing a special case zero in the case list.

```
FIND( n1 , n2 , n3 , ...) EXECUTE( ... )
```

FIND(is another example of the in-line expression approach which performs the inverse of a simple vector. It searches at run time for a match to the stack item in the list, returning either its subscript or zero if not found. Again, case zero can be written to handle the exceptions.

INTERVAL(n1 , n2 , n3 , ... nk)
EXECUTE(...)

Another in-line expression type, INTERVAL(contains a vector of values in ascending order dividing the number domain into the intervals between them. At run time a subscript is returned identifying the interval in which the stack item falls, and the item itself is preserved for processing by the selected case.

RANGE(n1 , m1 , n2 , m2 , ...)
EXECUTE(...)

Similar to INTERVAL(except that each n,m pair defines a separate range, and a subscript is generated identifying the first range found which embraces the stack item, or a zero if outside all of the ranges.

n MENU ccc EXECUTE(...) ;

MENU is a defining Word to create a menu-driven application named ccc which at run time will present screen n to the user, who will select options by entering a number, which is finally processed by the case list compiled within ccc.

Glossary and Code

The implementation below is written entirely in high level code assuming a byte addressing machine. Literal "n" used with ?PAIRS is left unspecified for consistent specification of all ?PAIRS values.

BEGIN(--- addr n ff n

Used in certain compiling words to begin compilation of an in-line, or literal data structure within a : definition. The next word in the dictionary is reserved for the address of the location following the entire structure, to be filled in by) at address addr. n is for compiler error checking. ff marks the stack so that other compiling words may push pointers to internal parts of the data block, to be appended to the end of the block by). See).

```
: BEGIN ?COMP HERE n 0 2 ALLOT . ;
```

) addr n ff addr0... n ---
(when used as a Word)

Has two entirely different uses. One terminates a comment begun by (, in which case it is not processed by the compiler. When used outside of a comment it completes compilation of an in-line data structure begun by BEGIN(. addr0... is a possibly empty list of addresses of points internal to the data block left by other compiling Words; if present it is appended to the data in reverse order. The address of the location following the data is then stored back at its beginning point addr. Also resumes compilation mode.

```
: ) n ?PAIRS BEGIN - DUP WHILE , REPEAT n ?PAIRS HERE SWAP
! ] ; IMMEDIATE
```

LIT(---

Used in Words processing in-line data structures to set up the return and computation stacks for accessing the data and branching around it. The Word in whose definition LIT(appears must be used immediately in front of an in-line data block, so that the address of the location at which to resume control is found in the following location; see BEGIN(. Consequently on entry to LIT(the return stack contains the address of the code following LIT(itself on top and the address of the data block just below. LIT(replaces the second return stack item by the address of the code following the data, and pushes the address of the first data item onto the computation stack. Also see)LIT.

```
: LIT( R> R> DUP @ >R 2+ SWAP >R ;
```

)LIT ---

Similar to LIT(except returns on the computation stack the address of the last word in the data structure instead of the first word, for accessing any address vector stored there in reverse order by).

```
: )LIT R> R> @ DUP >R 2 - SWAP >R ;
```

EXECUTE(--- addr n ff n (compile) P,C
n --- (run time)

Used within a : definition to define a list of routines, or cases in high level code in the form:

```
EXECUTE( CASE case0... ;S CASE case1... ;S ... ;S )
```

At run time case n is executed and control returns beyond the list. Unpredictable results occur if n is not a valid subscript at run time. Executes BEGIN(at compile time.

```
EXECUTE( ?COMP COMPILES BEGIN( EXECUTES )LIT W- @ >R ; IMMEDIATE
```

```
CALL(  --- addr n ff n  (compile)   P,C
       n ---            (run time)
```

Similar to EXECUTE(except the case routines are in assembly language in the form CALL(CASE case0... CASE case1... ...). Invokes the assembler vocabulary and suspends compilation.

```
CALL( ?COMP COMPILES BEGIN( [COMPILE] [ [COMPILE] ASSEMBLER
EXECUTES )LIT W- @ SP@ 2 - SWAP DROP EXECUTE; IMMEDIATE
```

```
CASE  addr n ff addr0... n ---
      addr n ff addr0...addr1 n       P
```

Used to begin each cae in a case list defined by EXECUTE(or CALL(. Adds the address of the next case addr1 to the list of case addresses addr0... on the stack, using n for error checking.

```
: CASE n ?PAIRS HERE n ; IMMEDIATE
```

```
COMPILE( --- addr n ff n (compile)  P,C
         n --- addr      (run time)
```

Used within a : definition to define a list of routines, or cases in either machine code or high level code in the form COMPILE(:CASE ... ;S ... CODECASE ...) which returns at run time the execution address of the case whose subscript is on the stack. The input subscript must be valid or unpredictable results will occur. To actually compile the execution address returned use ,. See also EXECUTE(and LITERAL(. Compiles using BEGIN(.

```
COMPILE( ?COMP COMPILES BEGIN( EXECUTES )LIT W- @ ; IMMEDIATE
```

```
:CASE  addr n ff addr0... n ---
       addr n ff addr0... addr1 n    P
```

Used to begin each high level code case in a case list defined by COMPILE(. Executes CASE and compiles the address of the code for executing : definitions. The routine begun by :CASE should be terminated by ;S as in EXECUTE(expressions.

```
: :CASE [ HERE 2 - ] [COMPILE] CASE  ] COMPILE [ @ , ] ; IMMEDIATE
```

```
CODECASE  addr n ff addr0... n ---
          addr n ff addr0...addr1 n    P
```

Used to begin each assembly code routine in a case list defined by COMPILE(. Executes CASE and compiles the address of the next dictionary location (as in the code field for a CODE definition). Compilation is suspended and the assembler vocabulary invoked as in CALL(. A jump to NEXT within a machine code case will resume high level execution following the case list.

```
: CODECASE [COMPILE] CASE 2 ALLOT HERE DUP 2 - !
  [COMPILE] [ [COMPILE] ASSEMBLER ; IMMEDIATE
```

```
LITERAL( --- addr n ff n (compile)
         n1 --- n2       (run time)  P,C
```

Used within a : definition to define a vector of 16-bit values. These values may be made Word execution addresses using the form

```
LITERAL( Word0 Word1 Word2 ... )
```

or may be made literal numbers using the form

```
LITERAL( [ n0 , n1 , n2 , ...] )
```

At run time the element whose subscription is on the stack is returned (without checking the validity of the stack value). When used with EXECUTE in the form LITERAL(...) EXECUTE the same result is achieved as using EXECUTE(...) except that storage requirements are less because no extra addresses are needed at the end of the vector. Uses BEGIN(to compile the list.

```
:LITERAL( ?COMP COMPILES BEGIN( EXECUTES LIT( W+ @ ; IMMEDIATE
```

```
EXCEPT  n ---        (compile)
        n --- n      (run time)          P,C
```

Used before an in-line case list or literal vector defined by EXECUTE(...) or similar Words, in the form [n] EXCEPT. Compiles an execution address and the value n, presumed to be the number of caes or vector elements in the subsequent in-line expression. At run time replaces any input value that is negative or greater than n by zero, allowing case or element zero to represent the "exceptions." This may be an error message or other explicit operation, or may simply bypass the entire case list by leaving case zero empty, i.e. compiling high level cases as ;S and machine code cases as a jump to NEXT. The EXCEPT function is not built into the case list expression code itself to allowing saving the storage when it is not needed.

```
: EXCEPT >R COMPILES R> , EXECUTES R> DUP 2+ >R OVER 0< IF
  DROP DROP 0 ELSE @ OVER < IF DROP 0 ENDIF ENDIF ; IMMEDIATE
```

```
FIND(  --- addr n ff n (compile)
       n1 --- n2       (run time)    P,C
```

Used in a : definition to define an array similar to LITERAL(but to perform the reverse operation at run time, i.e. the value is on the stack and the subscript is returned, or zero if not found.

```
: FIND( COMPILES BEGIN( 2 ALLOT ( element zero reserved )
( for copy of input) [COMPILE] ! [EXECUTED LIT(
OVER OVER I SWAP OVER R 2 - DO DUP I @ = IF DROP I LEAVE
ENDIF -2 +LOOP SWAP - 2 / ; IMMEDIATE
```

```
INTERVAL( --- addr n ff n (compile) P,C
          n1 --- n1 n2     (run time)
```

Used in a : definition to define a literal vector of interval boundary points in increasing order; at run time the subscript of the smallest boundary above n1 is added to n1 already on the stack, to control a subsequent case list processing n1. Compiles using BEGIN).

```
: INTERVAL COMPILES BEGIN( [COMPILE] [ EXECUTES ] LIT(
OVER OVER R 2 - DO I OVER I @ < IF LEAVE ENDIF
2 +LOOP SWAP DROP SWAP - 2 / ; IMMEDIATE
```

APPENDIX

The words COMPILES and :CASE above share a common function which might preferably be in a separate Word by itself. That function is compiling into the next dictionary location the code address used in : definitions. Rather than define a new Word, however, this function may be added to the existing definition of the : operator, as the function to be performed when STATE is the compiling mode, in contrast to the regular function performed when STATE is the execution mode, as it is when a definition is begun using :. Similarly, the Word CODE can be expanded to include a function to be performed in the compile state which consists of compiling the code address of a CODE definition (the address of the following location...), setting STATE to execute and invoking the ASSEMBLER vocabulary for beginning assembly language programming immediately following. Revised definitions from the case statement glossary above would then be:

```
: EXECUTES ? COMP COMPILE ;S n ?PAIRS HERE SWAP !
  [COMPILE ] : : IMMEDIATE
```

The Words :CASE and CODECASE are eliminated and the syntax for COMPILE(is:

```
COMPILE( CASE : ...high level case...;S ... )
COMPILE( CASE CODE ...machine code case...  .. )
```

George Lyons
Jersey City, NJ 07302

Judges' Comments - George got off to a great start but went on to solve many more problems than CASE, i.e. compiling in-line machine code by CODECASE. There are numerous ideas here, deserving of further analysis and examples of CASE.

=A FORTH CASE STATEMENT=

R. D. Perry

The Case Statements presented here are an extension of the FORTH IF Statement. The structure of the CASE Statement is such that it allows an N-way branch as contrasted to the IF statement two way branch. This version allows a CASE to be tested against a single value or a range of values. It does not require contiguous values for the tests. The value or range of values to be tested against are determined at run-time, this allows variables to determine CASE selection. No preprocessing is required as with the vector selection approach. It will execute faster than an IF statement preceeded by preprocessing (Example: = IF) assuming code implementation of N=Branch and NRANGE=BRANCH.

I became interested in the CASE Statement while implementing a CRT Screen Editor for FORTH Editing and Word Processor use.

```
SCR # 81
  0 ( CASE STATEMENTS        RDP 800322  )
  1 HEX
  2 CODE N=BRANCH ( IF BOT NOT EQU SEC BRANCH FROM INLINE LITERAL )
  3    INX, INX, FE ,X LDA, BOT CMP, 0=
  4     IF, FF ,X LDA, BOT 1+ CMP, 0=
  5        IF, INX, INX,  ' 0BRANCH 8 +  ( BUMP ) JMP,
  6        ENDIF,
  7     ENDIF,  ' BRANCH JMP, C;
  8
  9 CODE NRANGE=BRANCH ( IF THIRD<SEC OF THIRD>BOT BRANCH FROM LIT )
 10    INX, INX, INX, INX, SEC, FC ,X LDA, BOT SBC,
 11    FD ,X LDA, BOT 1+ SBC, 0< NOT
 12       IF, SEC, BOT LDA, FE ,X SBC, BOT 1+ LDA, FF ,X SBC, 0< NOT
 13          IF, INX, INX,  ' 0BRANCH 8 +  ( BUMP ) JMP, ENDIF,
 14       ENDIF,  ' BRANCH JMP,  C:
 15 DECIMAL -->
```

```
SCR # 82

  0 ( CASE STATEMENTS        RDP 800322
  1  -->   ( REMOVE THIS LINE IF CODE VERSIONS NOT USED )
  2 DECIMAL   ( R IS POINTING TO NEXT LOCATION )
  3 : N=BRANCH
  4    OVER =
  5    IF R> 2+ >R DROP
  6    ELSE R> DUP   @ + >R
  7    THEN  ;
  8
  9 : NRANGE=BRANCH
 10    ROT DUP ROT ( L,V,V,H ) >
 11    IF SWAP DROP R> DUP @ + >R ( OVER RANGE )
 12    ELSE DUP ROT ( V,V,L ) <
 13       IF R> DUP @ + >R ( UNDER RANGE )
 14       ELSE R> 2+ >R DROP ( IN RANGE )
 15  THEN THEN ; -->
```

```
SCR # 83

  0 ( MORE CASE        RDP 800322  )
  1 : BEGIN-CASES   ?COMP   0  4 ; IMMEDIATE
  2
  3 : CASE    ?COMP ( EL,4 ) 4 ?PAIRS ( EL )
  4    COMPILE N=BRANCH  HERE  0 ,  ( EL,NBL  )
  5    5 ; IMMEDIATE  ( EL, NBL,5 )
  6
  7 : RANGE-CASE  ?COMP ( EL,4 )   4 ?PAIRS   ( EL )
  8    COMPILE NRANGE=BRANCH  HERE   0 ,      ( EL,NBL )
  9    5 ; IMMEDIATE  ( EL,NBL,5)
 10
 11 : ELSE-CASE  ?COMP 4 ?PAIRS ( EL )
 12    COMPILE DROP  0  5 ; IMMEDIATE   ( EL,0,5 )
 13 -->
 14
 15
```

```
SCR # 84

  0 ( MORE CASE        RDP 800322  )
  1 : END-CASE   ?COMP  5 ?PAIRS  COMPILE BRANCH ( EL,BL )
  2     DUP ( EL,BL,BL, )
  3    IF HERE  2+  OVER - SWAP ! ( EL )
  4    ELSE DROP
  5    THEN  HERE  SWAP  ,   4 ( NEL,4 ) ;   IMMEDIATE
  6
  7 : END-CASES ?COMP 4 ?PAIRS  ( EL )
  8    DUP 0= 0= 1 ?PAIRS ( ERROR IF NO CASES )
  9    COMPILE DROP
 10    BEGIN  DUP
 11    WHILE  DUP @ SWAP HERE OVER -  SWAP !
 12    REPEAT DROP ; IMMEDIATE     ;S
 13 ( NAMES OF STACK ITEMS )
 14 EL -> END LINK            NEL -> NEW END LINK
 15 BL -> BEGIN LINK          NBL -> NEW BEGIN LINK
```

```
SCR # 85
  0 ( CASE STATEMENT TEST         RDP 800320 )
  1 : TEST BEGIN-CASES
  2      1  CASE  " ONE" END-CASE
  3      2  CASE  " TWO" END-CASE
  4      -9 9  RANGE-CASE  " > NEG.  TEN AND < TEN" END-CASE
  5      ELSE-CASE  " OTHER" END-CASE
  6    END-CASES  CR ;
  7 ;S
  8
  9 TYPE A NUMBER FOLLOWED BY "TEST", OUTPUT WILL BE
 10 ACCORDING TO CASE ABOVE
 11
 12
 13
 14
 15
```

```
N=BRANCH n1 n2 --- (run-time, n1=n2)
        n1 n2 --- (run-time, n1<>n2)
```

The Run-Time procedure to conditionally branch. If n1 does not equal n2 the following In-Line parameter is added to the interpretive pointer to branch ahead (or back) and n2 is dropped. If n1 equals n2 the interpretive pointer is advanced passed the in-line parameter and both n1 and n2 are dropped. Compiled by CASE.

```
NRANGE=BRANCH n1 n2 n3 ---
              (run-time, n1>=n2 & n1<=n3)
              n1 n2 n3 --- n1
              (run-time, n1<n2 or n1>n3)
```

The Run-Time procedure to conditionally branch. If n1 is less than n2 or n1 is greater than n3 the following in-line parameter is added to the interpretive pointer to branch ahead (or back) and both n2 and n3 are dropped. If n1 is greater than or equal to n2 and n1 is less than or equal to n3 and n1, n2, and n3 are dropped and the interpretive pointer is advanced passed the In-Line parameter. Compiled by RANGE-CASE.

```
BEGIN-CASES --- n1 n2
            (compile time)
```

Occurs in a colon-definition in the form:

```
BEGIN-CASES
... CASE ... END-CASE
... RANGE-CASE ... END-CASE
ELSE-CASE --- END-CASE
END-CASES
```

At compile-time BEGIN-CASES places n1 and n2 on the stack. n1 will later be used by END-CASES to signal that there is no prior END-CASE to link to. n2 is used for error testing.

```
CASE n1 n2 --- n1 (routine, N1<>n2)
     n1 n2 ---      (routine, n1=n1)
addr1   N1 --- Addr1 Addr2 N2
               (compile time)
```

At Run-Time CASE selects execution based on equality of the bottom two values on the stack. If they are equal signalling that the CASE is to be executed, both n1 and n2 are dropped and execution proceeds through CASE. CASE. If they are not equal only N2 is dropped and execution skips to just after END-CASE. (See BEGIN-CASES)

At Compile-Time CASE compiles N=BRANCH and reserves space for an offset value at addr2. addr1 is the address for the offset value of the last END-CASE. n1 and n2 are used for error testing.

```
RANGE-CASE
   N1 N2 --- N1 (Run-Time, N1<>N2 ) P,C2
   N1 N2 ---    (Run-Time, N1=N2)
addr1 N2 --- addr1 addr2 N2 (Compile-Time)
```

At Run-Time selects execution bases on whether n1 is in the range n2 to n3 (n2<n3). If in range execution proceeds through RANGE-CASE. If not in RANGE execution skips to just after END-CASE. (See BEGIN-CASES)

At Compile-Time RANGE-CASE compiles NRANGE=BRANCH and reserves space for an offset at addr2. addr1 is the location for the offset value of the last END-CASE. n1 and n2 are used for error testing.

```
ELSE-CASE
     n1 --- (run-time)
addr1 n1 --- addr n2 n3
             (compile-time)
```

At Run-Time n1 is dropped and execution continues through ELSE-CASE. (See BEGIN-CASES)

At Compile-Time ELSE compiles DROP. ADDR is the location for the offset of the last END-CASE. n2 is used by END-CASES to signal that the last case was an ELSE-CASE. n1 and n3 are used for error testing.

```
ELSE-CASE --- (run-time)
  addr1 addr2 n1 ---
        addr3 n2 (compile time)
```

At Run-Time causes execution to skip to after END-CASES. (See BEGIN-CASES)

At Compile-Time uses ADDR2 to set the offset of the last CASE or RANGE-CASE to point to after this END-CASE. Compiles BRANCH with an offset to be calculated later by END-CASES. The location for the offset of the last END-CASE is temporarily stored in this offset location and the new offset location is put on the stack. N1 and N2 are used for error testing.

END-CASES
--- (run-time with ELSE-CASE)
n --- (other run-time)
addr1 n1 --- (compile-time)

At Run-Time drops a stack value if no ELSE-CASE exists. Any END-CASE will continue execution just after the DROP. (See BEGIN-CASES)

At Compile-Time DROP is compiled and all of the offsets from an END-CASE are calculated and stored in their proper locations. ADDR is the location for the last offset for an END-CASE. That location holds the address for the prior offset and so on. The first offset location holds a value (0) which tells END-CASE that there are no more offsets to calculate.

R.D. Perry
San Diego, CA 92106

Judges' Comments - This is quite a complete and well documented entry. The range-of-cases feature is well done. Note that high level alternatives are given for the 6502 machine CODE words.

NEW PRODUCT

pico FORTH

HERMOSA BEACH, CA, JUNE 24, 1980 picoFORTH™, a new subset of polyFORTH™, is available for 1802 (disk or PROM) and 8080 micro- processors. Designed for interactive evaluation, picoFORTH includes all the essentials for programming, debugging, and testing a single-task application. This complete operating system features the polyFORTH assembler, compiler, text interpreter, editor, disk utilities, and basic documentation. picoFORTH can be upgraded at any time, either for a single purpose (with one or more of three packages: Source, Target Compiler™, or Mulititasker) or to full polyFORTH. A File Management Option package is also available. In addition to the current versions, picoFORTH will soon be implemented on the 8086, 6800, and LSI-11 processors. Price for picoFORTH is $495. Write or call Tom at FORTH, Inc., 2309 Pacific Coast Highway, Hermosa Beach, CA 90254 (213) 372-8493.

NEW PRODUCT

ALPHA MICRO FORTH

This system implements the Forth Interest Group language model, with full-length names to 31 characters, and extensive compile-time checks.

In addition, the diskette includes an editor, a FORTH assembler, and a string package, in FORTH source. The PDP-11 FORTH User's Guide, which includes extensive annotated examples of FORTH programming.

This FORTH system runs under AMOS. The distribution disk is single density. The complete system price is $190: Professional Management Services, 724 Arastradero Road, Suite 109, Palo Alto, California 94306, (408) 252-2218.

CASE STATEMENT

William H. Powell

The case structure by R.B. Main looks very powerful and flexible, but it seems to me to be unnecessarily complicated. My suggestion is for a word that does OVER = IF for his word CASE. This fits the existing FORTH compiler very well. The example by Main would read

```
: MONITOR
    41 CASE ." ASSIGN " THEN
    44 CASE ." DISPLAY " THEN
    46 CASE ." FILL " THEN
    47 CASE ." GO " THEN
    53 CASE ." SUBSTITUTE " THEN
       ELSE ." INSERT " THEN
       DROP ;
```

You will note that I have made the 'insert' message unconditional. This illustrates just how little need be added to the present FORTH structure and also how use of the present FORTH conditionals can be harnessed to the simple case structure as above. The normal FORTH syntax holds, and can be relied upon if case structures are nested into other structures, or into another set of case conditions.

This structure is neither the optimum for speed nor bytes. On the other hand we should avoid adding to FORTH in such a way that the nucleus and compiler grow any more than necessary. I favor a CASE structure that makes the program clearer, encourages sound software design and adds power to the language without adding significantly to the system software overhead.

Using the fig-FORTH model I need ideally one more nucleus word, and one for the compiler....

```
CODE /=BRANCH       ( Branch if SEC - BOT non-zero)
      INX, INX,     ( Drop BOT  only)
      SEC, FE ,X LDA,  0 ,X SBC, ' BRANCH 0= END,
           FF ,X LDA,  1 ,X SBC, ' BRANCH 0= END,
           BUMP: JMP,

: CASE ( n1 n2 --- n1 Case is executed if n1 = n2)
    COMPILE /=BRANCH HERE 0 , 2 ; IMMEDIATE
```

You will see that /=BRANCH does the same as OVER = IF and the case structure could be implemented without introducing /=BRANCH but I think speed and clarity better if one adds a code word as I have.

W.H. Powell
Sawbridgeworth
Herts. CM21 9NB
ENGLAND

Judges' Comments - Bill Powell didn't submit this as a contest entry, but it appeared in our mail just as the contest started. We took the liberty of including it as a mini-Case appropriate for the 6502.

A CASE STATEMENT

Major Robert A Selzer

OVERVIEW OF THE STATEMENT

CASE - The "CASE" statement is a special form of the IF-ELSE-THEN that permits the selection of one of many cases depending upon the top word on the stack being equal to a specified word (the value that precedes "CASE").

u (stack value) ul (case value) CASE (true action) ELSE (false action) THEN

u (stack value) ul (case value) CASE (true action) THEN

If u = ul, drop u,ul and execute true action following CASE until ELSE or THEN. Otherwise, drop ul but leave u on stack and execute ELSE (false action) or THEN if no ELSE. Implementation is the same as IF-ELSE-THEN, however each subsequent use of "CASE" will save 2 words (4 bytes) over the explicit use of OVER = IF DROP. CASE use also improves the readability of the source and if used often, will save code as well as being more convenient to the user.

SOURCE DEFINITIONS

See attached source listing. Note that fig-FORTH word COMPILE should replace FORTH, INC. word (backslash) or X (in later FORTH, INC versions) and a 16 bit emplace word , (comma) replaces the 8 bit emplace C, (C-comma). So, for SCR # 198, line 6. The fig-FORTH definition for CASE would be:

```
: CASE COMPILE (CASE) COMPILE 0BRANCH HERE 00 , ; IMMEDIATE
```

ENGLISH EXPLANATION

Only two new words need to be defined to use the CASE statement. (CASE) is the execution version that duplicates (OVER) the top of stack value then compares (=) it to the case value. If they are equal, the true action through the IF statement is taken and the stack value u is dropped (DROP). As part of the true action a flag (1) is pushed on the stack for 0BRANCH to test when CASE is executed. If the stack and case values are not equal the false action (ELSE) is taken and a false flag (0) is pushed on the stack over the original stack value tested (u). Both actions exit with THEN. CASE compiles the address of (CASE) and the address of the run-time IF called 0BRANCH . A 16 bit zero is compiled (,) at HERE in the dictionary, by HERE 00 , to reserve space for the branch to ELSE or THEN. The precedence bit of CASE is set so that CASE compiles 6 bytes whenever it is executed. Like IF, CASE must be used inside a colon definition and each use of CASE requires a corresponding THEN (or ELSE) to complete the structure.

GLOSSARY ENTRIES

(CASE)

The run-time procedure that is used by CASE, Equivalent to OVER = IF DROP. (CASE) is compiled by CASE.

CASE u ul --- u P,C2+

u ul CASE true action for u=ul
ELSE u false action THEN

If u=ul, drop u and ul and execute true action following CASE until next ELSE or THEN. If u is not equal to ul, drop ul but leave u and execute false action following ELSE or drop ul but leave u if no ELSE and exit to THEN.

```
u  u1  CASE  true action for u=u1  ELSE
u  u2  CASE  true action for u=u2  ELSE
u  un  CASE  true action for u=un  ELSE
          u  false action  THEN  THEN...THEN
```

EXAMPLES

See screens #199 and #200.

SCR #199 is used to demonstrate simple CASE use in the same application of the example published in FORTH-DIMENSIONS v 1/5, p. 51 to show conformity to an existing structure.

SCR #200 is a simple, but elegant example of CASE use in a video editor which occupies about 355 bytes of dictionary space for the COMPLETE editor. This is a good example of the CASE structure in fig-FORTH used to save code space and provide clarity of structure. While the editor is written for the ADM-3A terminal, line 1 defines a word which controls the cursor position sequence, so that any terminal can be used by making appropriate changes to the word YXCUR . The integer values in line 2 (2 and 4), determine the initial Y,X offset of the cursor in the HOME position (upper left corner + Y,X offset). This allows for adjustment of different LIST formats and edit screen positions. The vertical line at the right margin of the screens is generated by a 7C EMIT compiled in LIST. This vertical line gives the video editor user a positive indication of the editor limits of the right margin by setting up a window in which to edit. The ESC ($1B) key is used to exit the video editor VEDIT when finished. In fig-FORTH, use EMIT in place of ECHO in line 1. Don't forget to FLUSH .

DISCUSSION

This implementation of CASE in this form is fig-FORTH transportable to different machines (ie., 6502, 8080, 6800 etc.), however there is a 6 byte requirement for each use of CASE versus only 4 bytes for each use of IF. In applications like the example shown in SCR #200, the 2 byte overhead in CASE (6 bytes vs. 4 bytes for IF) saves 4 bytes for each use in lieu of OVER = IF DROP (10 bytes). More importantly, its use significantly enhances the readability and structure of the source code at the minimum cost of only 2 new FORTH words.

```
      SCR# 198
 0 (  CASE DEFINITION       RAS-09FEB80 )
 1 FORTH DEFINITIONS  BASE  @  HEX  FORGET  TASK  :  TASK  ;
 2
 3
 4 :  (CASE )  OVER = IF DROP 1 ELSE 0 THEN  ; ( EXECUTION CODE )
 5
 6 : CASE          (CASE)     0BRANCH  HERE  0  C,  ;  IMMEDIATE
 7
 8 BASE !
 9
10
11
12
13
14
15
```

```
      SCR# 199
 0 (  TEST  " CASE "  STRUCTURE )  BASE  @  HEX
 1
 2 :  MONITOR
 3     41 CASE  ." ASSIGN "      ELSE
 4     44 CASE  ." DISPLAY "     ELSE
 5     46 CASE  ." FILL  "       ELSE
 6     47 CASE  ." GO "          ELSE
 7     49 CASE  ." INSERT "      ELSE
 8     53 CASE  ." SUBSTITUTE "  ELSE
 9      THEN THEN THEN THEN THEN THEN  ;
10
11 : KEYBOARD  BEGIN  KEY 7F AND DUP MONITOR 20 = END  ;
12
13 BASE !   ;S
14
15
```

```
      SCR# 200
 0 ( VIDEO EDITOR, COPYRIGHT RCS 1978     ) HEX   00 VARIABLE  CUR
 1 : YXCUR  1B ECHO  3D ECHO  20 + ECHO 20 + ECHO ; ( ADM-3A)
 2 : .CUR CUR @ 40  /MOD 2 + SWAP 4 + SWAP YXCUR  ; : !CUR  0 MAX
 3   3FF MIN CUR ! ;  : +CUR  CUR @ + !CUR ;  : +.CUR  +CUR .CUR ;
 4 : +LIN CUR @ 40 / ( LINE #) + 40 * !CUR ;  : HOM    00 CUR !  ;
 5 : !BLK SCR @ 8 * CUR @ 80 /MOD ROT + BLOCK + C! UPDATE 1 +.CUR ;
 6   : VEDIT         LIST CR CR CR CR CR HOM .CUR BEGIN
 7           KEY 1B  CASE  0 12 YXCUR QUIT  ELSE  ( ESCAPE)
 8               0B  CASE    -1 +.CUR       ELSE  ( LEFT  CURSOR)
 9               0A  CASE    40 +.CUR       ELSE  ( DOWN  CURSOR)
10               08  CASE   -40 +.CUR       ELSE  ( UP    CURSOR)
11               0C  CASE     1 +.CUR       ELSE  ( RIGHT CURSOR)
12               0D  CASE     1  +LIN .CUR  ELSE  ( NEW   LINE  )
13               1E  CASE         HOM .CUR  ELSE  ( HOME  CURSOR)
14               DUP ECHO !BLK
15        THEN THEN THEN THEN THEN THEN THEN AGAIN ; DECIMAL ;S
```

```
Copyright 1977, RCS Associates                          FORTH-65 Ver
OK
```

Major Robert A. Selzer
APO San Francisco, 96301

Judges' Comments - This entry has the unfortunate need for closing the CASE by a correct number of THENs. It is written for microFORTH. The example of a screen text editor is outstanding and should be carefully read by all.

A CASE STATEMENT

Kenneth A. Wilson

CASE STATEMENT CONTEST

1.0 Description of the entry (coded in microFORTH)

1.1 Screen 338 defines the 4 words needed to generate a complete CASE statement.

1.2 Screen 339 contains a CASE test example.

1.3 The next 2 pages contain the printout obtained by executing the word TRIAL.

2.0 Definition of CASE words

	word	vocabulary	block	stack in	stack out
2.1	<CASE	FORTH	338	1	0

A defining word which creates a named array of n + 1 cells. Example: n <CASE name.

2.2	->	FORTH	338	0	1

A redefinition of for visual clarity. Pushes onto the stack the address of the parameter field of the word that follows in the current input stream.

2.3	=CASE	FORTH	338	3	0

Puts the address of a word (S1) into an array (S0) at cell n (S2).

Example: n word array =CASE
Read as: "n" becomes "word" in "array" case.

2.4	CASE	FORTH	338	2	0

Executes the word whose address is contained in the array (S0) at cell location n (S1).

Example: n name CASE

3.0 Explanation of the Example in Screen 339.

3.1 Line 1 defines 3 Cases:

3.1.1 FIRST is a Case of 4 cells
3.1.2 SEC is a Case of 4 cells
3.1.3 THIRD is a Case of 4 cells

3.2 Lines 2 thru 5 define "printing" words as follows:

3.2.1 Pronouns: I, YOU, WE, THEY
3.2.2 Verbs: RUN, WAL, SIT, JOG
3.2.3 Adverbs: HOME, BACK, DOWN UP

3.3 Line 6 thru 9 define the contents of the three Cases as follows:

3.3.1 FIRST Case contains 4 Pronouns

3.3.2 SEC Case contains 4 Verbs

3.3.3 THIRD Case contains 4 Adverbs

3.4 Lines 10 thru 14 define the word TRIAL which when executed, will cause the three Cases to be executed in sequence for each different possible combination of the index. i.e.:

```
111 FIRST CASE SEC CASE THIRD CASE
112 FIRST CASE SEC CASE THIRD CASE

554 FIRST CASE SEC CASE THIRD CASE
555 FIRST CASE SEC CASE THIRD CASE
```

An Overview

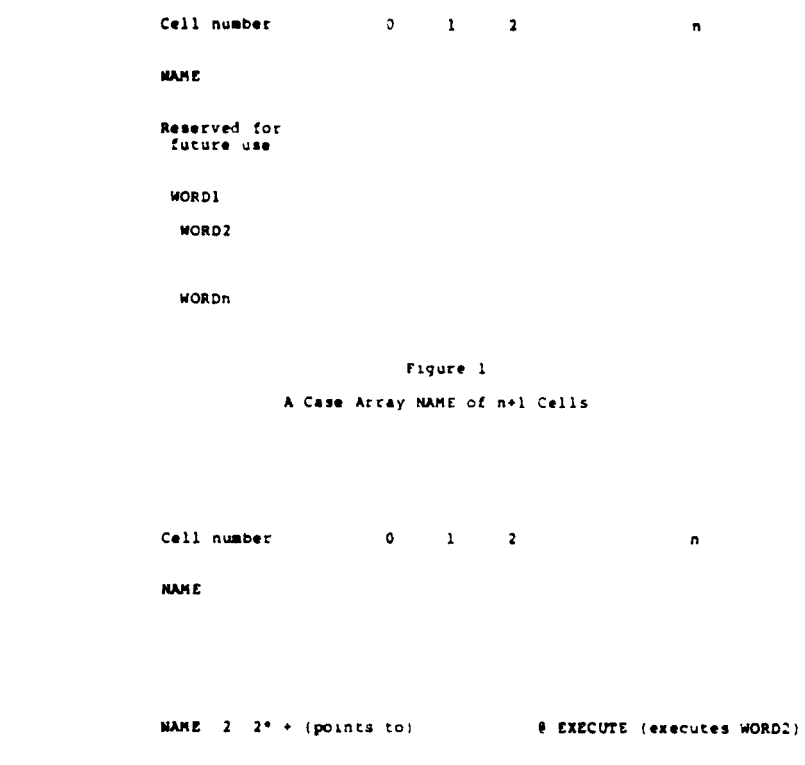


Figure 1

A Case Array NAME of n+1 Cells

Figure 2

Storing and Executing Cell 2

```
338 LIST

  0  ( CASE TEST WORDS)
  1  DISPLAY DEFINITIONS
  2  : <CASE  0 VARIABLE   2* 8 +! ;
  3  : ->    ;
  4  : =CASE  ROT 2*  +  !  ;
  5  : CASE  SWAP 2* +  @  EXECUTE ;
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15  DECIMAL      ;S    KAW        2-18-80
OK

339 LIST
  0  ( CASE TEST EXAMPLE )  DISPLAY DEFINITIONS    DECIMAL
  1  4 <CASE FIRST   4 <CASE SEC     4 <CASE THIRD
  2  : II [ I ] ;  : YOU [ YOU ]  ;  : WE [ WE ] ;  :  THEY [ THEY ] ;
  3  : RUN [ RUN ] ; : WALK [ WALK ]  ; : SIT [ SIT ]  ;
  4  : JOG [ JOG ] ; : HOME [ HOME ]  ; : BACK [ BACK ] ;
  5  : DOWN [ DOWN ] ; : UP [ UP ] ;
  6  1 -> II FIRST =CASE  1 -> RUN SEC =CASE  1 -> HOME THIRD =CASE
  7  2 -> YOU FIRST =CASE 2 -> WALK SEC =CASE  2 -> BACK THIRD =CASE
  8  3 -> WE FIRST = CASE 3 -> SIT SEC =CASE  3 -> DOWN THIRD =CASE
  9  4 -> THEY FIRST =CASE 4 -> JOG SEC =CASE  4 -> UP THIRD =CASE
 10  : TRIAL  CR  5 1 DO I
 11           5 1 DO I
 12           5 1 DO OVER OVER I ROT ROT
 13           FIRST CASE  SEC CASE  THIRD CASE  CR
 14           LOOP DROP CR   LOOP DROP CR   LOOP ;
 15  DECIMAL      ;S      KAW      2-28-80
OK
```

```
TRIAL

I RUN HOME
I RUN BACK
I RUN DOWN
I RUN UP

YOU RUN HOME
YOU RUN BACK
YOU RUN DOWN
YOU RUN UP

WE RUN HOME
WE RUN BACK
WE RUN DOWN
WE RUN UP

THEY RUN HOME
THEY RUN BACK
THEY RUN DOWN
THEY RUN UP

I WALK HOME
I WALK BACK
I WALK DOWN
I WALK UP

YOU WALK HOME
YOU WALK BACK
YOU WALK DOWN
YOU WALK UP

WE WALK HOME
WE WALK BACK
WE WALK DOWN
WE WALK UP

THEY WALK HOME
THEY WALK BACK
THEY WALK DOWN
THEY WALK UP

I SIT HOME
I SIT BACK
I SIT DOWN
I SIT UP

YOU SIT HOME
YOU SIT BACK
YOU SIT DOWN
YOU SIT UP
```

```
WE SIT HOME
WE SIT BACK
WE SIT DOWN
WE SIT UP

THEY SIT HOME
THEY SIT BACK
THEY SIT DOWN
THEY SIT UP

I JOG HOME
I JOG BACK
I JOG DOWN
I JOG UP

YOU JOG HOME
YOU JOG BACK
YOU JOG DOWN
YOU JOG UP

WE JOG HOME
WE JOG BACK
WE JOG DOWN
WE JOG UP

THEY JOG HOME
THEY JOG BACK
THEY JOG DOWN
THEY JOG UP

OK
```

Kenneth Wilson
Waltham, MA 02154

Judges' Comments - This is a very simple positional (jump table) type of CASE. The whole thing can be defined in three short lines of code. At first glance, however, the presentation looks more difficult than it is. Part of the problem is that the notation - the word names - does not suggest, very well, what is going on. This entry looks like a good complement to Eaker's. Both are simple mechanisms for doing a single job and the jobs that they each do are very different. Work is needed on integration and further development of these models.

NEW PRODUCT

68000

CREATIVE SOLUTIONS, INC. announces the availability of the FORTH programming approach for the Motorola 68000 16-bit Microprocessor.

Featuring: FORTH Interest Group Model and FORTH-79 Standard Compatibility, Virtual Disk Operating System, Text Editor, Inline Macro Assembler, Computer Aided Instruction Course on the FORTH Proramming Approach.

Also Available: Customized I/O Drivers for Non-Standard configurations, Suitable Hardware Configurations, Complete Source (written in FORTH), Meta Compiler, Multi-tasker, Extended Data Base Management and File System.

The standard software product, available for configurations utilizing the Motorola MEX68KDM (D2) 68000 evaluation model with Persci 1070 controller and compatible floppy disk drives retails for between $1500 - $5000 (depending upon options) for single user systems.

For further information please contact Creative Solutions, Inc., 14625 Tynewick Terrace, Silver Spring, Maryland 20906, Phone: (301) 598-5805.

NEW PRODUCT

AVAILABLE FROM ANCON

The following manuals and other information is available from ANCON, 17370 Hawkins Lane, Morgan Hill, CA 95037. Write for detailed list.

- FORTH Systems Reference Manual
- The FORTH Language
- FORTH-11 Reference Manual
- Indirect Threaded Code Reprints
- FORTH, a Programmers Guide
- PDP-11 FORTH Users Guide
- PH21-MX FORTH Manual
- CYBOS Programmers Manual
- Program FORTH, A Primer
- The JKL FORTH Manual

CASE STATEMENT

WAYNE WITT/BILL BUSLER

Overview

The CASE word provides the capability to vector to a particular word based on an input parameter, similar to the FORTRAN computed go-to. The CASE word also provides automatic limit checking on the input parameter with an optional out-of-range capability (OTHERCASE).

```
    49

 0 ( NEW CASE  -  CODE CASE            WW & WB  2/15/80  ) HEX
 1
 2 CODE I!   Y' PULU   NEXT         ( IP = TOP OF STACK )
 3
 4 : (CASE)                    ( CODE CASE -- CASE PARAMETER N -1 )
 5     I @ 7FFF AND OVER SWAP <<       ( TRUE IF N IN LIST RANGE )
 6       IF 2* I + 2+ @ 2+ EXECUTE          ( EXEC. LIST MODULE N )
 7       ELSE DROP I @ 0<           ( TRUE IF OTHERCASE SPECIFIED )
 8        IF I @ 7FFF AND 2* I + 2+ @ 2+ EXECUTE     ( OTHERCASE )
 9       THEN THEN                  ( NOW TO CONTINUE EXEC. AT DONE )
10     I DUP @ DUP 0<          ( GET ADDR. AND VALUE OF CASE-INDEX )
11       IF 7FFF AND 1+ THEN ( INCR INDEX IF OTHERCASE SPECIFIED )
12     2* 2+ + R> DROP I! ;       ( CONTINUE EXECUTION AFTER DONE )
13
14 ( NOTE: INTERPRETER POINTER MOVED TO END OF LIST OR)
15 (       AFTER THE DONE )   DECIMAL   ;S
```

```
    50

 0 ( NEW   CASE - OTHERCASE - DONE      WW & WB   2/15/80   ) HEX
 1                        ( PUT CODE CASE ADDRESS IN DICTIONARY , )
 2                                              ( PUT 0 ON STACK , )
 3 : CASE    (CASE) HERE 0 , ;  IMMEDIATE     ( CREATE CASE-INDEX )
 4                                ( IN DICTIONARY AND ZERO IT )
 5
 6 : OTHERCASE  DUP 8000 SWAP ! ; IMMEDIATE  ( SET OTHERCASE BIT )
 7                                              ( IN CASE-INDEX )
 8
 9 : DONE  DUP HERE SWAP - 2 / 1 -     ( CALC. CNT FOR CASE-INDEX )
10     SWAP DUP @      ( GET THE CASE-INDEX TO TEST FOR OTHERCASE )
11     ROT DUP 0=                      ( TRUE IF NO ITEMS IN LIST )
12       IF DROP DROP 0                  ( SET CASE-INDEX TO ZERO )
13       ELSE SWAP                   ( TRUE IF OTHERCASE SPECIFIED )
14         IF 1 - 8000 OR THEN       ( = -1 AND OTHERCASE BIT SET )
15       THEN SWAP ! ; IMMEDIATE  DECIMAL ;S  ( STORE CASE-INDEX )
```

This listing is from a 6809 version of FORTH.

CASE

n CASE mO ml ... mi DONE

CASE is used as a structured construction where n = 0 to i and mO ml ... mi represent a list of word names with the list being terminated by the word DONE.

When the definition containing the case construction is executed, module mn will execute, then execution will continue after the DONE. If n is not in the range 0 to i, execution continues after the DONE.

Alternative CASE usage with OTHERCASE

n CASE mO ml ... mi OTHERCASE mx DONE

When the definition containing the case construction is executed, module mn will execute if n is in the range 0 to i; then execution will continue after the DONE. If n is not in the range 0 to i, module mx will execute and then execution will continue after the DONE.

Only executable modules should be used in the case list; literals and compiler words, especially:

CASE OF ELSE THEN BEGIN END BUILDS DOES e

Should NOT be used.

OTHERCASE

Used in conjunction with CASE word for out of range conditions. See CASE usage.

DONE

CASE word terminator. See CASE usage.

I!

n ---

Replaces the interpreter pointer with the top stack item (n).

(CASE)

n ---

The execution time portion of the CASE word.

<<

n1 n2 --- f

Unsigned 16 bit less than.

Example of CASE usage

```
: TXX  CASE  TX1  TX2  TX3  DONE ;
```

If TXX is executed, then execution will continue as follows based on the value on the stack.

STACK VALUE	EXECUTE
0	TX1
1	TX2
2	TX3

Execution then continues after the DONE. If the stack value was not 0, 1 or 2 then execution continues after the DONE.

Examples of CASE usage with OTHERCASE.

```
: MH2  TX1  CASE  ENQ  VOICE  SYNC  NULL  OTHERCASE  TM3  DONE ;
: MH1  CASE  NULL  FIFO  TIME  XMIT-MSG  OTHERCASE  MH2  DONE CLEANUP ;
```

If MH1 is executed, then execution will continue as follows based on the value on the stack.

STACK VALUE	EXECUTE
0	NULL
1	FIFO
2	TIME
3	XMIT-MSG
Any Other Value	MH2

Execution then continues after the DONE, in this instance CLEANUP.

MH2 illustrates the nesting capability of the CASE word.

This form of CASE conforms with the unwritten rule of FORTH to keep it simple and basic. The user needs to remember only three words, CASE, OTHERCASE and DONE to construct simple to complex forms of the structured CASE. The CASE in providing automatic limit checking and out of range recovery elliminates the need for user limit testing of the parameters. This out of range checking capability does slow the execution speed slightly, but it was felt that the added capability was worth the slight loss of speed.

Bill Busler
Odessa, Florida 33556

Wayne Witt
Tampa, Florida 33615

Judge's Comments - The run-time word (CASE) seems much too long for the job it does. This is partly because the out-of-range case is handled by a special construction. Nevertheless, the code could be reorganized or factored. Also, pushing the DONE address back on the return stack at the end of (CASE) would eliminate the need for I! and make the package more portable.

The GODO ... THEN construction in Kitt Peak FORTH accomplishes all the same functions much more efficiently.

THE KITT PEAK GODO CONSTRUCT

By David Kilbridge

The GODO construct, as specified in the glossary of the Kitt Peak FORTH Primer, is a type of CASE statement. An index on the stack is truncated to fall within a contiguous range and used to select a word from an in-line execution vector. I present here a very simple implementation in fig-FORTH.

As an example of usage, here is a word which accepts a 0 or 1 from the terminal and selects the corresponding disk drive, and rings the bell if any other key is pressed.

```
: GET-DRIVE   ." DISK DRIVE? "
    KEY  2F -
    GODO  BELL DR0 DR1 BELL  THEN  ;
```

The necessary source definitions are

```
: (GODO)   2*
    0  MAX   R  @  4  -  MIN
    R>  DUP  DUP  @  +  >R
    +  2+  @  EXECUTE  ;

: GODO
    COMPILE  (GODO)
    HERE  0  ,  2  ;
    IMMEDIATE
```

How it works: GODO compiles (GODO) and leaves space for a branch offset to be calculated by THEN. The address of the cell and an error-checking flag are left on the stack. At run time (GODO) doubles the index on the stack and truncates it both above and below so that the reference executed will always be chosen from the list provided. Then (GODO) uses the branch offset to step its return address over the reference list and finally executes the selected reference.

Glossary:

```
GODO     --- addr n (compile-time)    P,C
(GODO) n ---        (run-time)
```

Used in the sequence

```
... GODO  R0  R1 ... Rn  THEN ...
```

At run-time, GODO selects execution based on a signed integer index. If the index is <=0 then R0 is executed; if =1 then R1 is executed; ... if >=n then Rn is executed. After executing the selected reference, execution resumes after THEN.

Discussion: The GODO construct provides a basic contiguous-range type of CASE statement requiring very little supporting code. The compile-time word is simple because most of the work is done by THEN. The run-time word is simple because truncating the index allows out-of-range cases to be handled just like in-range cases.

If other means are used to insure that the index is always within range, the "catch-all" references R0 and/or Rn can be omitted. However, there is still the time overhead needed to truncate the index (unless (GODO) is recompiled without the second line of its definition).

The principal limitation of this construct is that only single words can be referenced. This prevents direct nesting of GODO's. However, one can nest by defining the inner GODO as a separate word and referencing it in the outer GODO. By letting Ro and/or Rn be such references, several noncontiguous ranges can be covered.

Kitt Peak PRIMER available from FIG for $20.00 in US and $25.00 Overseas.

COME TO FIG CONVENTION NOVEMBER 29

FIG NORTHERN CALIFORNIA MONTHLY MEETING REPORT

26 April 80

The FORML session consisted of three presentations covering FORTH File proposals. John James and John Cassady discussed Directories consisting of bit maps named FileControlBlock (FCB) wherein allocation of strings of blocks (Files) were managed. Particulars of bitmap manipulation at the Buffer, Block and Disk (file and volume) levels were explicated. Some other concepts included user transparency, hierarchy of directories, commands, security and integrity. Kim Harris described Record types and management within a File and gave examples of FORTH, Inc. styled I/O at the Field level. The pros and cons of the various approaches will be debated at the next meeting where also String manipulation will be discussed. Attendees were requested to prepare written proposals of anticipated requirements and arguments for and against the different approaches. Though not a tutorial, the FORML session was very instructive.

The April Northern California FIG meeting consisted of a presentation by Jim Brick (of M&B Design) of a polyFORTH bootup under CP/M. Jim described the application requirements that produced the need and the technique he used to develop this bootup package sold by FORTH, Inc. He demonstrated the hybrid package on a TRS-80 with I/O accessories which allowed 8" disks and remapping of the TRS-80 memory for polyFORTH-CP/M compatability.

Bill Ragsdale initiated a tutorial on overflow correction which spontaneously escalated into a discussion on error signals, repair and recovery. Kim Harris, Lafarr Stuart and Dave Boulton described their respective approaches to dealing with errors. Bill elaborated the "Utrecht approach" to error signaling and recovery and noted two lessons learned: high level words can define error recovery and the return stack can be usefully unthreaded. He congratulated our Dutch colleagues for their imaginative applications of "tricks" garnished from other computer languages.

Henry Laxen was congratulated for his excellent article on FORTH in the 80 April 28 issue of INFOWORLD.

Kim Harris announced his FORTHcoming course on FORTH programming at Humbolt State University (80 July 21-25) and also reported on a talk he delivered earlier this month at the Asilomar I.E.E.E. conference on megatransistor chips.

...HANDOUTS provided at the meeting included:

-polyFORTH-CP/M (Brick)

-INFOWORLD reprint (Laxen)

-TIC-TAC-TOE (in FORTH, of course) (George Flammer)

-overflow correction (Ragsdale)

-Match CPM for 8080 figFORTH (anon)

-Double number support (Ragsdale)

-String match for Editor (Peter Midnight)

;s Jay Melvin

Publisher's Note:

Come on, you other FIGGERS, send in reports on your meetings. We'll publish them.

FIG NORTHERN CALIFORNIA MONTHLY MEETING REPORT

24 May 80

FORML Session -

Kim Harris directed a review of last month's session to compare and contrast file systems presented by:

1. John James
2. John Cassady
3. Kim Harris (FORTH, Inc. system)

The most striking difference between the three file systems was that FORTH, Inc.'s did not utilize a bit map in the directory which would allow for a distinction between physical and logical files. The bit map implemented in James' and Cassady's systems provide for easier file manipulation.

FIG Meeting -

Bill Ragsdale opened the meeting by introducing guests Ed Murray from the University of South Africa and Don Colburn who is marketing a FORTH Teaching Tutorial to be configured for various machines.

The meeting was devoted to a two fold tutorial where Kim Harris explained FORTH tools ranging from NUMERIC output and base conversion to test interpretation. I/O formating examples included the definition of HOLD, ASCII and PAD. These "tools" were applied in a temperature conversion program. Bill Ragsdale followed with a presentation on problem solving techniques using the task of printing Morse (dits/dahs) characters to the screen in response to text input. Top down techniques were delineated by listing the subtasks and writing code then testing each module.

John Draper described CAP'N Software's Version 1.7 FORTH for the Apple; the system was up and running for demonstration. Ragsdale notified us that Computer magazine wants articles for a FORTH issue next year and that Byte's August issue will have a Robert Tinney cover displaying three blocks in a field of stars, each block containing a word (2*, DUP, +) and threaded together by a ribbon terminating in a space needle.

Handouts included: Kim's tool kit, Bill's Morse Code worksheet (a blank page!), John's Version 1.7 brochure, and Benchmark by DRC for measuring FORTH execution speeds on CRAY-1 through micros. Also, a floating point package by NHC, a paper on file word concepts by Jim Berkey and the HomeBrew Computer Club's newsletter by (ed.) Bill Reiling were available.

;s Jay Melvin

FORTH Interest Group Meetings

Northern California 4th Saturday	FIG Monthly Meeting, 1:00 p.m., at Liberty House Department Store, Hayward, CA. FORML Workshop at 10:00 a.m.
Massachusetts 3rd Wednesday	MMSFORTH Users Group, 7:00 p.m., Cochituate, MA. Call Dick Miller at (617) 653-6136 for site.
San Diego Thursdays	FIG Meeting, 12:00 noon. Call Guy Kelly at (714) 268-3100 x 4784 for site.
Seattle Various times	Contact Chuck Pliske or Dwight Vandenburg at (206) 542-8370.
Potomac Various times	Contact Paul van der Eijk at (703) 354-7443 or Joel Shprentz at (703) 437-9218.
Texas Various times	Contact Jeff Lewis at (713) 729-3320 or John Earls at (214) 661-2928 or Dwayne Gustaus at (817) 387-6976. John Hastings (512) 835-1918.
Arizona Various times	Contact Dick Wilson at (602) 277-6611 x 3257.
Oregon Various times	Contact Ed Krammerer at (503) 644-2688.
New York Various times	Contact Tom Jung at (212) 746-4062.
Detroit Various times	Contact Dean Vieau at (313) 493-5105.
Japan Various times	Contact Mr. Okada, President, ASR Corp. Int'l, 3-15-8, Nishi-Shimbashi Minato-ku, Tokyo, Japan.

Publisher's Note:

Please send notes (and reports) about your meetings.

----HELP WANTED----

BUSINESS SYSTEMS IN FORTH

We need two good FORTH programmers.

You should have solid FORTH experience, a year or two, and be generally competent in Computer Science.

We are building an exciting range of business application systems using FORTH - the advantages are obvious! - and our approach is unique. We'll have a range of configurations - single and multi-processor, both Winchester and large fixed disks and color graphics screens.

Ideally you'll live in Orange County - be attracted by a small, quality team - and like to grab your own projects with a strong sense of self management - we haven't got the time or the inclination to be overbearing.

Please send brief description of your background to:

The Software Development Director
4861 McKay Circle
Anaheim, CA 92807

and let us know why you think you'd like to work with us.

CALL FOR PAPERS

FORML CONFERENCE

(FORTH Modification Laboratory)

Papers are requested for a three day technical workshop to be held November 26-28, 1980 at the Asilomar Conference Grounds in Pacific Grove, California (on the Monterey Peninsula). The purpose of the workshop is to discuss advanced technical topics related to FORTH implementation, language and application. Papers on any of the following or related topics are requested for presentation and discussion:

1. Programming methodology
   - problem analysis and design
   - implementation style
   - development team management
   - documentation
   - debugging
2. Virtual machine implementation
   - arithmetic
   - address enlargement
   - position independent object code
   - metaFORTH
3. Concurrency
   - resource management
   - scheduling
   - intertask communication and control
   - integrity, privacy and protection
4. Language and compiler
   - typing and generic operations
   - data and control structures
   - optimization
5. Applications
   - file systems
   - string handling
   - text editing
   - graphics
6. Standardization
   - Review and discussion of 79-STANDARD
   - Input for the Standards Team

FORML is an organization (sponsored by the FORTH Interest Group) which promotes the exchange of ideas on the use, modification and extension of the FORTH approach to systems development. This will be an advanced technical workshop; no introductory tutorials will be held.

Abstracts of papers must be received by October 1, 1980 for inclusion in the conference program. Complete papers must be received by November 1, 1980 to be included in the conference proceedings. Send both abstracts and completed papers to:

FORML Conference
P. O. Box 51351
Palo Alto, CA 94303

----HELP WANTED----

TITLE: Product Support Programmer

DUTIES: Responsible for maintaining existing list of software products, including the polyFORTH Operating System and Programming Language, file management options, math options and utilities and their documentation, and providing technical support to customers of these products.

Requirements for candidates:

1. Good familiarity with FORTH—preferably through one complete target-compiled application.

2. Good assembler level programming skills.

3. Assembler level familiarity with the 8080 and PDP/LSI-11 processors and preferably some of these: 8086, M6800, CDP1802, NOVA, IBM Series I, TI990.

4. Excellent communications skills--both oral and written; ability to work well with customers.

5. Excellent organizational ability.

Contact: Elizabeth Rather
FORTH, Inc.
2309 Pacific Coast Hwy.
Hermosa Beach, CA 90254
(213) 372-8493

FORML CONFERENCE

(FORTH Modification Laboratory)

November 26-28. 1980 at the Asilomar Conference Grounds, Pacific Beach, California. A three day advanced technical workshop for the discussion of topics related to FORTH implementation, language and application. No introductory tutorials will be held.

FORML is an organization (sponsored by the FORTH Interest Group) which promotes the exchange of ideas on the use, modification and extension of the FORTH approach to systems development.

Asilomar is a comfortable, rustic resort located on the Pacific Ocean near Monterey in Northern California. Attendees are urged to bring family members to Asilomar as they will enjoy the area and Thanksgiving dinner. Costs are very reasonable, especially for families, and include room (double occupancy) and meals.

Attendees and/or participants	$100.00 (includes conference registration and materials)
Non-conference guest (wife and/or husband, friend, and children 12 or over)	$ 75.00
Children 11 or younger	$ 50.00

Send request for registration and list of guests by October 15th with a check to:

FORML Conference
P.O. Box 51351
Palo Alto, CA 94303

NATIONAL CONVENTION

FORTH Interest Group

November 29, 1980 at the Villa Hotel, San Mateo, California, 8:30 a.m. - 4:30 p.m. for exhibits and papers; 6:00 p.m. cocktails; 7:30 p.m. for dinner (with speaker). This one day convention will include presentations, workshops, hands-on equipment and a number of vendor exhibits. An evening dinner will include a talk by one of the foremost authorities on FORTH (more about the speaker in a later release).

Pre-registration for the convention is available for $4.00.

Pre-registration for the dinner and speech is required by October 15th at $15.00.

Vendors may contact FIG about the cost and availability of booth and table space.

To pre-register or for more information write:

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

Vendors may contact Roy Martens at (415) 962-8653 for details about exhibiting.

Room arrangements can also be made through FIG.

**********FLASH LATE NEWS**********

FIG NATIONAL CONVENTION BANQUET SPEAKER

ALAN TAYLOR

Author of The Taylor Report for Computer World. 30 years in computer field.

*********MAKE YOUR RESERVATION*********