A.6 Glossary

In this and following sections we present rationales for the handling of specific words: why we included them, why we placed them in certain word sets, or why we specified their names or meaning as we did.

Words in this section are organized by word set, retaining their index numbers for easy cross-referencing to the glossary.

Historically, many Forth systems have been written in Forth. Many of the words in Forth originally had as their primary purpose support of the Forth system itself. For example, WORD and FIND are often used as the principle instruments of the Forth text interpreter, and CREATE in many systems is the primitive for building dictionary entries. In defining words such as these in a standard way, we have endeavored not to do so in such a way as to preclude their use by implementors. One of the features of Forth that has endeared it to its users is that the same tools that are used to implement the system are available to the application programmer - a result of this approach is the compactness and efficiency that characterizes most Forth implementations.

A.6.1 Core words

A.6.1.0070 '

Typical use: `... ' name .`

Many Forth systems use a state-smart tick. Many do not. ANS Forth follows the usage of Forth-83.

See: A.3.4.3.2 Interpretation semantics, A.6.1.1550 FIND

A.6.1.0080 (

Typical use: `... ( ccc) ...`

A.6.1.0140 +LOOP

Typical use: `: X ... limit first DO ... step +LOOP ;`

A.6.1.0150 ,

The use of , (comma) for compiling execution tokens is not portable.

See: 6.2.0945 COMPILE,

A.6.1.0190 ."

Typical use: `: X ... ." ccc" ... ;`

An implementation may define interpretation semantics for ." if desired. In one plausible implementation, interpreting ." would display the delimited message. In another plausible implementation, interpreting ." would compile code to display the message later. In still another plausible implementation, interpreting ." would be treated as an exception. Given this variation a Standard Program may not use ." while interpreting. Similarly, a Standard Program may not compile POSTPONE ." inside a new word, and then use that word while interpreting.

A.6.1.0320 2*

Historically, 2* has been implemented on two's-complement machines as a logical left-shift instruction. Multiplication by two is an efficient side-effect on these machines. However, shifting implies a knowledge of the significance and position of bits in a cell. While the name implies multiplication, most implementors have used a hardware left shift to implement 2*.

A.6.1.0330 2/

This word has the same common usage and misnaming implications as 2*. It is often implemented on two's-complement machines with a hardware right shift that propagates the sign bit.

A.6.1.0350 2@

With 2@ the storage order is specified by the Standard.

A.6.1.0450 :

Typical use: `: name ... ;`

In Forth-83, this word was specified to alter the search order. This specification is explicitly removed in this Standard. We believe that in most cases this has no effect; however, systems that allow many search orders found the Forth-83 behavior of colon very undesirable.

Note that colon does not itself invoke the compiler. Colon sets compilation state so that later words in the parse area are compiled.

A.6.1.0460 ;

Typical use: `: name ... ;`

One function performed by both ; and ;CODE is to allow the current definition to be found in the dictionary. If the current definition was created by :NONAME the current definition has no definition name and thus cannot be found in the dictionary. If :NONAME is implemented the Forth compiler must maintain enough information about the current definition to allow ; and ;CODE to determine whether or not any action must be taken to allow it to be found.

A.6.1.0550 >BODY

a-addr is the address that HERE would have returned had it been executed immediately after the execution of the CREATE that defined xt.

A.6.1.0680 ABORT"

Typical use: `: X ... test ABORT" ccc" ... ;`

A.6.1.0695 ACCEPT

Previous standards specified that collection of the input string terminates when either a **return** is received or when +n1 characters have been received. Terminating when +n1 characters have been received is difficult, expensive, or impossible to implement in some system environments. Consequently, a number of existing implementations do not comply with this requirement. Since line-editing and collection functions are often implemented by system components beyond the control of the Forth implementation, this Standard imposes no such requirement. A Standard Program may only assume that it can receive an input string with ACCEPT or EXPECT. The detailed sequence of user actions necessary to prepare and transmit that line are beyond the scope of this Standard.

Specification of a non-zero, positive integer count (+n1) for ACCEPT allows some implementors to continue their practice of using a zero or negative value as a flag to trigger special behavior. Insofar as such behavior is outside the Standard, Standard Programs cannot depend upon it, but the Technical Committee doesn't wish to preclude it unnecessarily. Since actual values are almost always small integers, no functionality is impaired by this restriction.

ACCEPT and EXPECT perform similar functions. ACCEPT is recommended for new programs, and future use of EXPECT is discouraged.

It is recommended that all non-graphic characters be reserved for editing or control functions and not be stored in the input string.

Commonly, when the user is preparing an input string to be transmitted to a program, the system allows the user to edit that string and correct mistakes before transmitting the final version of the string. The editing function is supplied sometimes by the Forth system itself, and sometimes by external system software or hardware. Thus, control characters and functions may not be available on all systems. In the usual case, the end of the editing process and final transmission of the string is signified by the user pressing a **Return** or **Enter** key.

As in previous standards, EXPECT returns the input string immediately after the requested number of characters are entered, as well as when a line terminator is received. The **automatic termination after specified count of**

characters have been entered behavior is widely considered undesirable because the user **loses control** of the input editing process at a potentially unknown time (the user does not necessarily know how many characters were requested from EXPECT). Thus EXPECT and SPAN have been made obsolescent and exist in the Standard only as a concession to existing implementations. If EXPECT exists in a Standard System it must have the **automatic termination** behavior.

ACCEPT does not have the **automatic termination** behavior of EXPECT. However, because external system hardware and software may perform the ACCEPT function, when a line terminator is received the action of the cursor, and therefore the display, is implementation-defined. It is recommended that the cursor remain immediately following the entered text after a line terminator is received.

A.6.1.0705 ALIGN

In this Standard we have attempted to provide transportability across various CPU architectures. One of the frequent causes of transportability problems is the requirement of cell-aligned addresses on some CPUs. On these systems, ALIGN and ALIGNED may be required to build and traverse data structures built with C,. Implementors may define these words as no-ops on systems for which they aren't functional.

A.6.1.0706 ALIGNED

See: `A.6.1.0705 ALIGN`

A.6.1.0760 BEGIN

Typical use:

```
        : X ... BEGIN ... test UNTIL ;
```

or

```
        : X ... BEGIN ... test WHILE ... REPEAT ;
```

A.6.1.0770 BL

Because space is used throughout Forth as the standard delimiter, this word is the only way a program has to find and use the system value of **space**. The value of a space character can not be obtained with CHAR, for instance.

A.6.1.0880 CELL+

As with ALIGN and ALIGNED, the words CELL and CELL+ were added to aid in transportability across systems with different cell sizes. They are intended to be used in manipulating indexes and addresses in integral numbers of cell-widths.

Example:

```
2VARIABLE DATA

0 100 DATA 2!
DATA @ . 100

DATA CELL+ @ .  0
```

A.6.1.0890 CELLS

See: `A.6.1.0880 CELL+`

Example: `CREATE NUMBERS 100 CELLS ALLOT`

(Allots space in the array NUMBERS for 100 cells of data.)

A.6.1.0895 CHAR

Typical use: `... CHAR A CONSTANT "A" ...`

A.6.1.0950 CONSTANT

Typical use: `... DECIMAL 10 CONSTANT TEN ...`

A.6.1.1000 CREATE

The data-field address of a word defined by CREATE is given by the data-space pointer immediately following the execution of CREATE

Reservation of data field space is typically done with ALLOT.

Typical use: `... CREATE SOMETHING ...`

A.6.1.1240 DO

Typical use:

```
        : X ... limit first DO ... LOOP ;
```

or

```
        : X ... limit first DO ... step +LOOP ;
```

A.6.1.1250 DOES>

Typical use: `: X ... DOES> ... ;`

Following DOES>, a Standard Program may not make any assumptions regarding the ability to find either the name of the definition containing the DOES> or any previous definition whose name may be concealed by it. DOES> effectively ends one definition and begins another as far as local variables and control-flow structures are concerned. The compilation behavior makes it clear that the user is not entitled to place DOES> inside any control-flow structures.

A.6.1.1310 ELSE

Typical use: `: X ... test IF ... ELSE ... THEN ;`

A.6.1.1345 ENVIRONMENT?

In a Standard System that contains only the Core word set, effective use of ENVIRONMENT? requires either its use within a definition, or the use of user-supplied auxiliary definitions. The Core word set lacks both a direct method for collecting a string in interpretation state (11.6.1.2165 S" is in an optional word set) and also a means to test the returned flag in interpretation state (e.g. the optional 15.6.2.2532 [IF]).

The combination of 6.1.1345 ENVIRONMENT?, 11.6.1.2165 S", 15.6.2.2532 [IF], 15.6.2.2531 [ELSE], and 15.6.2.2533 [THEN] constitutes an effective suite of words for conditional compilation that works in interpretation state.

A.6.1.1360 EVALUATE

The Technical Committee is aware that this function is commonly spelled EVAL. However, there exist implementations that could suffer by defining the word as is done here. We also find EVALUATE to be more readable and explicit. There was some sentiment for calling this INTERPRET, but that too would have undesirable effects on existing code. The longer spelling was not deemed significant since this is not a word that should be used frequently in source code.

A.6.1.1380 EXIT

Typical use: `: X ... test IF ... EXIT THEN ... ;`

A.6.1.1550 FIND

One of the more difficult issues which the Committee took on was the problem of divorcing the specification of implementation mechanisms from the specification of the Forth language. Three basic implementation approaches can be quickly enumerated:

1) Threaded code mechanisms. These are the traditional approaches to implementing Forth, but other techniques may be used.

2) Subroutine threading with **macro-expansion** (code copying). Short routines, like the code for DUP, are copied into a

definition rather than compiling a JSR reference.

3) Native coding with optimization. This may include stack optimization (replacing such phrases as `SWAP ROT +` with one or two machine instructions, for example), parallelization (the trend in the newer RISC chips is to have several functional subunits which can execute in parallel), and so on.

The initial requirement (inherited from Forth-83) that compilation addresses be compiled into the dictionary disallowed type 2 and type 3 implementations.

Type 3 mechanisms and optimizations of type 2 implementations were hampered by the explicit specification of immediacy or non-immediacy of all standard words. POSTPONE allowed de-specification of immediacy or non-immediacy for all but a few Forth words whose behavior must be STATE-independent.

One type 3 implementation, Charles Moore's cmForth, has both compiling and interpreting versions of many Forth words. At the present, this appears to be a common approach for type 3 implementations. The Committee felt that this implementation approach must be allowed. Consequently, it is possible that words without interpretation semantics can be found only during compilation, and other words may exist in two versions: a compiling version and an interpreting version. Hence the values returned by FIND may depend on STATE, and ' and ['] may be unable to find words without interpretation semantics.

A.6.1.1561 FM/MOD

By introducing the requirement for **floored** division, Forth-83 produced much controversy and concern on the part of those who preferred the more common practice followed in other languages of implementing division according to the behavior of the host CPU, which is most often symmetric (rounded toward zero). In attempting to find a compromise position, this Standard provides primitives for both common varieties, floored and symmetric (see SM/REM). FM/MOD is the floored version.

The Technical Committee considered providing two complete sets of explicitly named division operators, and declined to do so on the grounds that this would unduly enlarge and complicate the Standard. Instead, implementors may define the normal division words in terms of either FM/MOD or SM/REM providing they document their choice. People wishing to have explicitly named sets of operators are encouraged to do so. FM/MOD may be used, for example, to define:

```
: /_MOD ( n1 n2 -- n3 n4) >R S>D R> FM/MOD ;

: /_  ( n1 n2 -- n3)  /_MOD SWAP DROP ;

: _MOD ( n1 n2 -- n3)   /_MOD DROP ;

: */_MOD ( n1 n2 n3 -- n4 n5)  >R M* R> FM/MOD ;

: */_  ( n1 n2 n3 -- n4 )   */_MOD SWAP DROP ;
```

A.6.1.1700 IF

Typical use:

```
        : X ... test IF ... THEN ... ;
```

or

```
        : X ... test IF ... ELSE ... THEN ... ;
```

A.6.1.1710 IMMEDIATE

Typical use: `: X ... ; IMMEDIATE`

A.6.1.1720 INVERT

The word NOT was originally provided in Forth as a flag operator to make control structures readable. Under its intended usage the following two definitions would produce identical results:

```
: ONE  ( flag -- )
    IF ." true" ELSE ." false" THEN ;

: TWO ( flag -- )
```

```
    NOT IF ." false" ELSE ." true" THEN ;
```

This was common usage prior to the Forth-83 Standard which redefined NOT as a cell-wide one's-complement operation, functionally equivalent to the phrase `-1 XOR`. At the same time, the data type manipulated by this word was changed from a flag to a cell-wide collection of bits and the standard value for true was changed from **1** (rightmost bit only set) to **-1** (all bits set). As these definitions of TRUE and NOT were incompatible with their previous definitions, many Forth users continue to rely on the old definitions. Hence both versions are in common use.

Therefore, usage of NOT cannot be standardized at this time. The two traditional meanings of NOT - that of negating the sense of a flag and that of doing a one's complement operation - are made available by 0= and INVERT, respectively.

A.6.1.1730 J

J may only be used with a nested DO...LOOP, DO...+LOOP, ?DO...LOOP, or ?DO...+LOOP, for example, in the form:

```
        : X ... DO ... DO ... J ... LOOP ... +LOOP ... ;
```

A.6.1.1760 LEAVE

Note that LEAVE immediately exits the loop. No words following LEAVE within the loop will be executed. Typical use:

```
        : X ... DO ... IF ... LEAVE THEN ... LOOP ... ;
```

A.6.1.1780 LITERAL

Typical use: `: X ... [ x ] LITERAL ... ;`

A.6.1.1800 LOOP

Typical use:

```
        : X ... limit first DO ... LOOP ... ;
```

or

```
        : X ... limit first ?DO ... LOOP ... ;
```

A.6.1.1810 M*

This word is a useful early step in calculation, going to extra precision conveniently. It has been in use since the Forth systems of the early 1970's.

A.6.1.1900 MOVE

CMOVE and CMOVE> are the primary move operators in Forth-83. They specify a behavior for moving that implies propagation if the move is suitably invoked. In some hardware, this specific behavior cannot be achieved using the best move instruction. Further, CMOVE and CMOVE> move characters; ANS Forth needs a move instruction capable of dealing with address units. Thus MOVE has been defined and added to the Core word set, and CMOVE and CMOVE> have been moved to the String word set.

A.6.1.2033 POSTPONE

Typical use:

```
: ENDIF  POSTPONE THEN ;  IMMEDIATE

: X  ... IF ... ENDIF ... ;
```

POSTPONE replaces most of the functionality of COMPILE and [COMPILE]. COMPILE and [COMPILE] are used for the same purpose: postpone the compilation behavior of the next word in the parse area. COMPILE was designed to be applied to non-immediate words and [COMPILE] to immediate words. This burdens the programmer with needing to know which words in a system are immediate. Consequently, Forth standards have had to specify the immediacy or non-immediacy of all words covered by the Standard. This unnecessarily constrains implementors.

A second problem with COMPILE is that some programmers have come to expect and exploit a particular implementation, namely:

```
        :  COMPILE  R>  DUP  @  ,  CELL+  >R  ;
```

This implementation will not work on native code Forth systems. In a native code Forth using inline code expansion and peephole optimization, the size of the object code produced varies; this information is difficult to communicate to a **dumb** COMPILE. A **smart** (i.e., immediate) COMPILE would not have this problem, but this was forbidden in previous standards.

For these reasons, COMPILE has not been included in the Standard and [COMPILE] has been moved in favor of POSTPONE. Additional discussion can be found in Hayes, J.R., **Postpone**, Proceedings of the 1989 Rochester Forth Conference.

A.6.1.2120 RECURSE

Typical use: `: X ... RECURSE ... ;`

This is Forth's recursion operator; in some implementations it is called MYSELF. The usual example is the coding of the factorial function.

```
: FACTORIAL ( +n1 -- +n2)
    DUP 2 < IF  DROP 1 EXIT  THEN
    DUP 1-  RECURSE *
;
```

n2 = n1(n1-1)(n1-2)...(2)(1), the product of n1 with all positive integers less than itself (as a special case, zero factorial equals one). While beloved of computer scientists, recursion makes unusually heavy use of both stacks and should therefore be used with caution. See alternate definition in A.6.1.2140 REPEAT.

A.6.1.2140 REPEAT

Typical use:

```
: FACTORIAL ( +n1 -- +n2)
    DUP 2 < IF  DROP 1 EXIT  THEN
    DUP
    BEGIN DUP 2 > WHILE
        1-  SWAP OVER *  SWAP
    REPEAT  DROP
;
```

A.6.1.2165 S"

Typical use: `: X ... S" ccc" ... ;`

This word is found in many systems under the name " (quote). However, current practice is almost evenly divided on the use of ", with many systems using the execution semantics given here, while others return the address of a counted string. We attempt here to satisfy both camps by providing two words, S" and the Core Extension word C" so that users may have whichever behavior they expect with a simple renaming operation.

A.6.1.2214 SM/REM

See the previous discussion of division under FM/MOD. SM/REM is the symmetric-division primitive, which allows programs to define the following symmetric-division operators:

```
: /-REM  ( n1 n2 -- n3 n4 )  >R  S>D  R> SM/REM ;

: /-  (  n1 n2 -- n3 )  /-REM SWAP DROP ;

: -REM  ( n1 n2 -- n3 )  /-REM DROP ;

: */-REM  (  n1 n2 n3 -- n4 n5 )  >R  M*  R> SM/REM ;

: */-  ( n1 n2 n3 -- n4 )  */-REM SWAP DROP ;
```

A.6.1.2216 SOURCE

SOURCE simplifies the process of directly accessing the input buffer by hiding the differences between its location for different input sources. This also gives implementors more flexibility in their implementation of buffering mechanisms for different input sources. The committee moved away from an input buffer specification consisting of a collection of individual variables, declaring TIB and #TIB obsolescent.

SOURCE in this form exists in F83, POLYFORTH, LMI's Forths and others. In conventional systems it is equivalent to the phrase

```
        BLK @  IF BLK @ BLOCK 1024  ELSE TIB #TIB @ THEN
```

A.6.1.2250 STATE

Although EVALUATE, LOAD, INCLUDE-FILE, and INCLUDED are not listed as words which alter STATE, the text interpreted by any one of these words could include one or more words which explicitly alter STATE. EVALUATE, LOAD, INCLUDE-FILE, and INCLUDED do not in themselves alter STATE.

STATE does not nest with text interpreter nesting. For example, the code sequence:

```
        : FOO  S" ]" EVALUATE ;       FOO
```

will leave the system in compilation state. Similarly, after LOADing a block containing], the system will be in compilation state.

Note that] does not affect the parse area and that the only effect that : has on the parse area is to parse a word. This entitles a program to use these words to set the state with known side-effects on the parse area. For example:

```
: NOP  : POSTPONE ; IMMEDIATE ;

NOP ALIGN    NOP ALIGNED
```

Some non-ANS Forth compliant systems have] invoke a compiler loop in addition to setting STATE. Such a system would inappropriately attempt to compile the second use of NOP.

Also note that nothing in the Standard prevents a program from finding the execution tokens of] or [and using these to affect STATE. These facts suggest that implementations of] will do nothing but set STATE and a single interpreter/compiler loop will monitor STATE.

A.6.1.2270 THEN

Typical use:

```
        : X ... test IF ... THEN ... ;
```

or

```
        : X ... test IF ... ELSE ... THEN ... ;
```

A.6.1.2380 UNLOOP

Typical use:

```
: X  ...

   limit first DO

       ... test IF ... UNLOOP EXIT THEN ...

   LOOP
   ...
;
```

UNLOOP allows the use of EXIT within the context of DO ... LOOP and related do-loop constructs. UNLOOP as a function has been called UNDO. UNLOOP is more indicative of the action: nothing gets undone -- we simply stop doing it.

A.6.1.2390 UNTIL

Typical use: `: X ... BEGIN ... test UNTIL ... ;`

A.6.1.2410 VARIABLE

Typical use: `... VARIABLE XYZ ...`

A.6.1.2430 WHILE

Typical use: `: X ... BEGIN ... test WHILE ... REPEAT ... ;`

A.6.1.2450 WORD

Typical use: `char WORD ccc<char>`

A.6.1.2500 [

Typical use: `: X ... [ 4321 ] LITERAL ... ;`

A.6.1.2510 [']

Typical use: `: X ... ['] name ... ;`

See: A.6.1.1550 FIND

A.6.1.2520 [CHAR]

Typical use: `: X ... [CHAR] ccc ... ;`

A.6.1.2540]

Typical use: `: X ... [ 1234 ] LITERAL ... ;`

A.6.2 Core extension words

The words in this collection fall into several categories:

- Words that are in common use but are deemed less essential than Core words (e.g., 0<>);
- Words that are in common use but can be trivially defined from Core words (e.g., FALSE);
- Words that are primarily useful in narrowly defined types of applications or are in less frequent use (e.g., PARSE);
- Words that are being deprecated in favor of new words introduced to solve specific problems (e.g., CONVERT).

Because of the varied justifications for inclusion of these words, the Technical Committee does not encourage implementors to offer the complete collection, but to select those words deemed most valuable to their clientele.

A.6.2.0060 #TIB

The function of #TIB has been superseded by SOURCE.

A.6.2.0200 .(

Typical use: `.( ccc)`

A.6.2.0210 .R

In .R, **R** is short for RIGHT.

A.6.2.0340 2>R

Historically, 2>R has been used to implement DO. Hence the order of parameters on the return stack.

The primary advantage of 2>R is that it puts the top stack entry on the top of the return stack. For instance, a double-cell number may be transferred to the return stack and still have the most significant cell accessible on the top of the return stack.

A.6.2.0410 2R>

Note that 2R> is not equivalent to R> R>. Instead, it mirrors the action of 2>R (see A.6.2.0340).

A.6.2.0455 :NONAME

:NONAME allows a user to create an execution token with the semantics of a colon definition without an associated name. Previously, only : (colon) could create an execution token with these semantics. Thus, Forth code could only be compiled using the syntax of :, that is:

```
        : NAME  ...  ;
```

:NONAME removes this constraint and places the Forth compiler in the hands of the programmer.

:NONAME can be used to create application-specific programming languages. One technique is to mix Forth code fragments with application-specific constructs. The application-specific constructs use :NONAME to compile the Forth code and store the corresponding execution tokens in data structures.

The functionality of :NONAME can be built on any Forth system. For years, expert Forth programmers have exploited intimate knowledge of their systems to generate unnamed code fragments. Now, this function has been named and can be used in a portable program.

For example, :NONAME can be used to build a table of code fragments where indexing into the table allows executing a particular fragment. The declaration syntax of the table is:

```
:NONAME .. code for command 0 .. ;  0 CMD !

:NONAME .. code for command 1 .. ;  1 CMD !
   ...

:NONAME .. code for command 99 .. ; 99 CMD !

   ... 5 CMD @ EXECUTE ...
```

The definitions of the table building words are:

```
CREATE CMD-TABLE  \ table for command execution tokens
100 CELLS ALLOT

: CMD ( n -- a-addr ) \ nth element address in table
    CELLS CMD-TABLE + ;
```

As a further example, a defining word can be created to allow performance monitoring. In the example below, the number of times a word is executed is counted. : must first be renamed to allow the definition of the new ;.

```
: DOCOLON ( -- )     \ Modify CREATEd word to execute like a colon def
     DOES> ( i*x a-addr -- j*x )
     1 OVER +!         \ count executions
     CELL+ @ EXECUTE   \ execute :NONAME definition
;

: OLD: : ;           \ just an alias

OLD: : ( "name" -- a-addr xt colon-sys )
                     \ begins an execution-counting colon definition
     CREATE  HERE 0 ,  \ storage for execution counter
     0 ,               \ storage for execution token
     DOCOLON           \ set run time for CREATEd word
    :NONAME           \ begin unnamed colon definition
;
```

(Note the placement of DOES>: DOES> must modify the CREATEd word and not the :NONAME definition, so DOES> must execute before :NONAME.)

```
OLD: ; ( a-addr xt colon-sys -- )
                      \ ends an execution-counting colon definition )
    POSTPONE ;        \ complete compilation of colon def
    SWAP CELL+ !      \ save execution token
;  IMMEDIATE
```

The new : and ; are used just like the standard ones to define words:

```
        ... : xxx  ... ;  ...  xxx  ...
```

Now however, these words may be **ticked** to retrieve the count (and execution token):

```
        ... ' xxx >BODY ? ...
```

A.6.2.0620 ?DO

Typical use: `: FACTORIAL ( +n1 -- +n2 ) 1 SWAP 1+ ?DO I * LOOP ;`

This word was added in response to many requests for a resolution of the difficulty introduced by Forth-83's DO, which on a 16-bit system will loop 65,535 times if given equal arguments. As this Standard also encourages 32-bit systems, this behavior can be intolerable. The Technical Committee considered applying these semantics to DO, but declined on the grounds that it might break existing code.

A.6.2.0700 AGAIN

Typical use: `: X ... BEGIN ... AGAIN ... ;`

Unless word-sequence has a way to terminate, this is an endless loop.

A.6.2.0855 C"

Typical use: `: X ... C" ccc" ... ;`

It is easy to convert counted strings to pointer/length but hard to do the opposite. C" is the only new word that uses the **address of counted string** stack representation. It is provided as an aid to porting existing programs to ANS Forth systems. It is relatively difficult to implement C" in terms of other standard words, considering its **compile string into the current definition** semantics.

Users of C" are encouraged to migrate their application code toward the consistent use of the preferred **c-addr u** stack representation with the alternate word S". This may be accomplished by converting application words with counted string input arguments to use the preferred **c-addr u** representation, thus eliminating the need for C" .

See: `A.3.1.3.4 Counted strings`

A.6.2.0873 CASE

Typical use:

```
   : X ...
      CASE
      test1 OF ... ENDOF
      testn OF ... ENDOF
      ... ( default )
      ENDCASE ...
   ;
```

A.6.2.0945 COMPILE,

COMPILE, is the compilation equivalent of EXECUTE. In many cases, it is possible to compile a word by using POSTPONE without resorting to the use of COMPILE,. However, the use of POSTPONE requires that the name of the word must be known at compile time, whereas COMPILE, allows the word to be located at any time. It is sometime possible to use EVALUATE to compile a word whose name is not known until run time. This has two possible problems:

- EVALUATE is slower than COMPILE, because a dictionary search is required.
- The current search order affects the outcome of EVALUATE.

In traditional threaded-code implementations, compilation is performed by , (comma). This usage is not portable; it doesn't work for subroutine-threaded, native code, or relocatable implementations. Use of COMPILE, is portable.

In most systems it is possible to implement COMPILE, so it will generate code that is optimized to the same extent as code that is generated by the normal compilation process. However, in some implementations there are two different **tokens** corresponding to a particular definition name: the normal **execution token** that is used while interpreting or with EXECUTE, and another **compilation token** that is used while compiling. It is not always possible to obtain the compilation token from the execution token. In these implementations, COMPILE, might not generate code that is as efficient as normally compiled code.

A.6.2.0970 CONVERT

CONVERT may be defined as follows:

```
        : CONVERT   CHAR+ 65535 >NUMBER DROP ;
```

A.6.2.1342 ENDCASE

Typical use:

```
   : X ...
       CASE
       test1 OF ... ENDOF
       testn OF ... ENDOF
       ... ( default )
       ENDCASE ...
   ;
```

A.6.2.1343 ENDOF

Typical use:

```
: X ...
   CASE
   test1 OF ... ENDOF
   testn OF ... ENDOF
   ... ( default )
   ENDCASE ...
;
```

A.6.2.1390 EXPECT

Specification of positive integer counts (+n) for EXPECT allows some implementors to continue their practice of using a zero or negative value as a flag to trigger special behavior. Insofar as such behavior is outside the Standard, Standard Programs cannot depend upon it, but the Technical Committee doesn't wish to preclude it unnecessarily. Since actual values are almost always small integers, no functionality is impaired by this restriction.

A.6.2.1850 MARKER

As dictionary implementations have gotten more elaborate and in some cases have used multiple address spaces, FORGET has become prohibitively difficult or impossible to implement on many Forth systems. MARKER greatly eases the problem by making it possible for the system to remember **landmark information** in advance that specifically marks the spots where the dictionary may at some future time have to be rearranged.

A.6.2.1950 OF

Typical use:

```
   : X ...
       CASE
       test1 OF ... ENDOF
       testn OF ... ENDOF
       ... ( default )
       ENDCASE ...
   ;
```

A.6.2.2000 PAD

PAD has been available as scratch storage for strings since the earliest Forth implementations. It was brought to our attention that many programmers are reluctant to use PAD, fearing incompatibilities with system uses. PAD is specifically intended as a programmer convenience, however, which is why we documented the fact that no standard words use it.

A.6.2.2008 PARSE

Typical use: `char PARSE ccc<char>`

The traditional Forth word for parsing is WORD. PARSE solves the following problems with WORD:

a) WORD always skips leading delimiters. This behavior is appropriate for use by the text interpreter, which looks for sequences of non-blank characters, but is inappropriate for use by words like (, .(, and ." . Consider the following (flawed) definition of .(:

```
        : .(   [CHAR] )  WORD COUNT TYPE ;  IMMEDIATE
```

This works fine when used in a line like:

```
        .( HELLO)   5 .
```

but consider what happens if the user enters an empty string:

```
        .( )   5 .
```

The definition of .(shown above would treat the) as a leading delimiter, skip it, and continue consuming characters until it located another) that followed a non-) character, or until the parse area was empty. In the example shown, the 5 . would be treated as part of the string to be printed.

With PARSE, we could write a correct definition of .(:

```
        : .(   [CHAR] ) PARSE TYPE ; IMMEDIATE
```

This definition avoids the **empty string** anomaly.

b) WORD returns its result as a counted string. This has four bad effects:

1) The characters accepted by WORD must be copied from the input buffer into a temporary buffer, in order to make room for the count character that must be at the beginning of the counted string. The copy step is inefficient, compared to PARSE, which leaves the string in the input buffer and doesn't need to copy it anywhere.

2) WORD must be careful not to store too many characters into the temporary buffer, thus overwriting something beyond the end of the buffer. This adds to the overhead of the copy step. (WORD may have to scan a lot of characters before finding the trailing delimiter.)

3) The count character limits the length of the string returned by WORD to 255 characters (longer strings can easily be stored in blocks!). This limitation does not exist for PARSE.

4) The temporary buffer is typically overwritten by the next use of WORD. This introduces a temporal dependency; the value returned by WORD is only valid for a limited duration. PARSE has a temporal dependency, too, related to the lifetime of the input buffer, but that is less severe in most cases than WORD's temporal dependency.

The behavior of WORD with respect to skipping leading delimiters is useful for parsing blank-delimited names. Many system implementations include an additional word for this purpose, similar to PARSE with respect to the **c-addr u** return value, but without an explicit delimiter argument (the delimiter set is implicitly **white space**), and which does skip leading delimiters. A common description for this word is:

```
        PARSE-WORD  ( <spaces>name -- c-addr u )
```

Skip leading spaces and parse name delimited by a space. c-addr is the address within the input buffer and u is the length of the selected string. If the parse area is empty, the resulting string has a zero length.

If both PARSE and PARSE-WORD are present, the need for WORD is largely eliminated.

A.6.2.2030 PICK

0 PICK is equivalent to DUP and 1 PICK is equivalent to OVER.

A.6.2.2040 QUERY

The function of QUERY may be performed with ACCEPT and EVALUATE.

A.6.2.2125 REFILL

This word is a useful generalization of QUERY. Re-defining QUERY to meet this specification would have broken existing code. REFILL is designed to behave reasonably for all possible input sources. If the input source is coming from the user, as with QUERY, REFILL could still return a false value if, for instance, a communication channel closes so that the system knows that no more input will be available.

A.6.2.2150 ROLL

2 ROLL is equivalent to ROT, 1 ROLL is equivalent to SWAP and 0 ROLL is a null operation.

A.6.2.2182 SAVE-INPUT

SAVE-INPUT and RESTORE-INPUT allow the same degree of input source repositioning within a text file as is available with BLOCK input. SAVE-INPUT and RESTORE-INPUT **hide the details** of the operations necessary to accomplish this repositioning, and are used the same way with all input sources. This makes it easier for programs to reposition the input source, because they do not have to inspect several variables and take different action depending on the values of those variables.

SAVE-INPUT and RESTORE-INPUT are intended for repositioning within a single input source; for example, the following scenario is NOT allowed for a Standard Program:

```
   : XX
      SAVE-INPUT  CREATE
      S" RESTORE-INPUT" EVALUATE
      ABORT" couldn't restore input"
   ;
```

This is incorrect because, at the time RESTORE-INPUT is executed, the input source is the string via EVALUATE, which is not the same input source that was in effect when SAVE-INPUT was executed.

The following code is allowed:

```
: XX
    SAVE-INPUT  CREATE
    S" .( Hello)" EVALUATE
    RESTORE-INPUT ABORT" couldn't restore input"
;
```

After EVALUATE returns, the input source specification is restored to its previous state, thus SAVE-INPUT and RESTORE-INPUT are called with the same input source in effect.

In the above examples, the EVALUATE phrase could have been replaced by a phrase involving INCLUDE-FILE and the same rules would apply.

The Standard does not specify what happens if a program violates the above rules. A Standard System might check for the violation and return an exception indication from RESTORE-INPUT, or it might fail in an unpredictable way.

The return value from RESTORE-INPUT is primarily intended to report the case where the program attempts to restore the position of an input source whose position cannot be restored. The keyboard might be such an input source.

Nesting of SAVE-INPUT and RESTORE-INPUT is allowed. For example, the following situation works as expected:

```
: XX
    SAVE-INPUT
    S" f1" INCLUDED      \ The file "f1" includes:
    \   ... SAVE-INPUT ... RESTORE-INPUT ...
    \ End of file "f1"
    RESTORE-INPUT  ABORT" couldn't restore input"
;
```

In principle, RESTORE-INPUT could be implemented to **always fail**, e.g.:

```
: RESTORE-INPUT  ( x1 ... xn n -- flag )
    0 ?DO DROP LOOP TRUE
```

```
;
```

Such an implementation would not be useful in most cases. It would be preferable for a system to leave SAVE-INPUT and RESTORE-INPUT undefined, rather than to create a useless implementation. In the absence of the words, the application programmer could choose whether or not to create **dummy** implementations or to work-around the problem in some other way.

Examples of how an implementation might use the return values from SAVE-INPUT to accomplish the save/restore function:

```
Input Source    possible stack values
------------    ---------------------
block           >IN @  BLK @  2
EVALUATE        >IN @  1
keyboard        >IN @  1
text file       >IN @  lo-pos  hi-pos  3
```

These are examples only; a Standard Program may not assume any particular meaning for the individual stack items returned by SAVE-INPUT.

A.6.2.2290 TIB

The function of TIB has been superseded by SOURCE.

A.6.2.2295 TO

Historically, some implementations of TO have not explicitly parsed. Instead, they set a mode flag that is tested by the subsequent execution of name. ANS Forth explicitly requires that TO must parse, so that TO's effect will be predictable when it is used at the end of the parse area.

Typical use: `x TO name`

A.6.2.2298 TRUE

TRUE is equivalent to the phrase `0 0=`.

A.6.2.2405 VALUE

Typical use:

```
0 VALUE DATA

: EXCHANGE ( n1 -- n2 ) DATA SWAP TO DATA ;
```

EXCHANGE leaves n1 in DATA and returns the prior value n2.

A.6.2.2440 WITHIN

We describe WITHIN without mentioning circular number spaces (an undefined term) or providing the code. Here is a number line with the overflow point (o) at the far right and the underflow point (u) at the far left:

```
u--------------------------------------------------------------o
```

There are two cases to consider: either the n2|u2..n3|u3 range straddles the overflow/underflow points or it does not. Lets examine the non-straddle case first:

```
u-------------------[.....................)------------------------o
```

The [denotes n2|u2, the) denotes n3|u3, and the dots and [are numbers WITHIN the range. n3|u3 is greater than n2|u2, so the following tests will determine if n1|u1 is WITHIN n2|u2 and n3|u3:

```
           n2|u2 < n1|u1  and  n1|u1 < n3|u3.
```

In the case where the comparison range straddles the overflow/underflow points:

u..............)-----------------------------[.......................o n3|u3 is less than n2|u2 and the following tests will determine if n1|u1 is WITHIN n2|u2 and n3|u3:

```
          n2|u2 > n1|u1  and  n1|u1 > n3|u3.
```

WITHIN must work for both signed and unsigned arguments. One obvious implementation does not work:

```
: WITHIN  ( test low high -- flag )
    >R  OVER < 0= ( test flag1 )
    SWAP R> <     ( flag1 flag2 )
    AND
;
```

Assume two's-complement arithmetic on a 16-bit machine, and consider the following test:

```
        33000  32000 34000  WITHIN
```

The above implementation returns false for that test, even though the unsigned number 33000 is clearly within the range {{32000 .. 34000}}.

The problem is that, in the incorrect implementation, the signed comparison < gives the wrong answer when 32000 is compared to 33000, because when those numbers are treated as signed numbers, 33000 is treated as negative 32536, while 32000 remains positive.

Replacing < with U< in the above implementation makes it work with unsigned numbers, but causes problems with certain signed number ranges; in particular, the test:

```
        1  -5  5  WITHIN
```

would give an incorrect answer.

For two's-complement machines that ignore arithmetic overflow (most machines), the following implementation works in all cases:

```
:  WITHIN  ( test low high -- flag )   OVER - >R - R>  U<  ;
```

A.6.2.2530 [COMPILE]

Typical use: `: name2 ... [COMPILE] name1 ... ; IMMEDIATE`

A.6.2.2535 \

Typical use: `5 CONSTANT THAT \ THIS IS A COMMENT ABOUT THAT`

Table of Contents

Next Section