A. Rationale (informative annex)

A.1 Introduction

A.1.1 Purpose

A.1.2 Scope

This Standard is more extensive than previous industry standards for the Forth language. Several things made this necessary:

– the desire to resolve conflicts between previous standards;
– the need to eliminate semantic ambiguities and other inadequacies;
– the requirement to standardize common practice, where possible resolving divergences in a way that minimizes the cost of compliance;
– the desire to standardize common system techniques, including those germane to hardware.

The result of the effort to satisfy all of these objectives is a Standard arranged so that the required word set remains small. Thus ANS Forth can be provided for resource-constrained embedded systems. Words beyond those in the required word set are organized into a number of optional word sets and their extensions, enabling implementation of tailored systems that are Standard.

When judging relative merits, the members of the X3J14 Technical Committee were guided by the following goals (listed in alphabetic order):

Consistency	The Standard provides a functionally complete set of words with minimal functional overlap.
Cost of compliance	This goal includes such issues as common practice, how much existing code would be broken by the proposed change, and the amount of effort required to bring existing applications and systems into conformity with the Standard.
Efficiency	Execution speed, memory compactness.
Portability	Words chosen for inclusion should be free of system-dependent features.
Readability	Forth definition names should clearly delineate their behavior. That behavior should have an apparent simplicity which supports rapid understanding. Forth should be easily taught and support readily maintained code.
Utility	Be judged to have sufficiently essential functionality and frequency of use to be deemed suitable for inclusion.

A.1.3 Document organization

A.1.3.1 Word sets

From the beginning, the X3J14 Technical Committee faced not only conflicting ideas as to what “real” Forth is, but also conflicting needs of the various groups within the Forth community. At one extreme were those who pressed for a “bare” Forth. At the other extreme were those who wanted a “fat” Forth. Many were somewhere in between. All were convinced of the rightness of their own position and of the wrongness of at least one of the two extremes. The committee’s composition reflected this full range of interests.

The approach we have taken is to define a Core word set establishing a greatest lower bound for required system functionality and to provide a portfolio of optional word sets for special purposes. This simple approach parallels the fundamental nature of Forth as an extensible language, and thereby achieves a kind of meta-extensibility.

With this key, high-level compromise, regardless of the actual makeup of the individual word sets, a firm and workable framework is established for the long term. One may or may not agree that there should be a Locals word set, or that the word **COMPILE,** belongs in the Core Extensions word set. But at least there is a mechanism whereby such things can be included in a logical and orderly manner.

Several implications of this scheme of optional word sets are significant.

First, ANS Forth systems can continue to be implemented on a greater range of hardware than could be claimed by almost any other single language. Since only the Core word set is required, very limited hardware will be able to accommodate an ANS Forth implementation.

Second, a greater degree of portability of applications, and of programmers, is anticipated. The optional word sets standardize various functions (e.g., floating point) that were widely implemented before, but not with uniform definition names and methodologies, nor the same levels of completeness. With such words now standardized in the optional word sets, communications between programmers -- verbally, via magazine or journal articles, etc. -- will leap to a new level of facility, and the shareability of code and applications should rise dramatically.

Third, ANS Forth systems may be designed to offer the user the power to selectively, even dynamically, include or exclude one or more of the optional word sets or portions thereof. Also, lower-priced products may be offered for the user who needs the Core word set and not much more. Thus, virtually unlimited flexibility will be available to the user.

But these advantages have a price. The burden is on the user to decide what capabilities are desired, and to select product offerings accordingly, especially when portability of applications is important. We do not expect most implementors to attempt to provide all word sets, but rather to select those most valuable to their intended markets.

The basic requirement is that if the implementor claims to have a particular optional word set the entire required portion of that word set must be available. If the implementor wishes to offer only part of an optional word set, it is acceptable to say, for example, “This system offers portions of the [named] word set,” particularly if the selected or excluded words are itemized clearly.

Each optional word set will probably appeal to a particular constituency. For example, scientists performing complex mathematical analysis may place a higher value on the Floating-Point word set than programmers developing simple embedded controllers. As in the case of the core extensions, we expect implementors to offer those word sets they expect will be valued by their users.

Optional word sets may be offered in source form or otherwise factored so that the user may selectively load them.

The extensions to the optional word sets include words which are deemed less essential to performing the primary activity supported by the word set, though clearly relevant to it. As in the case of the Core Extensions, implementors may selectively add itemized subsets of a word set extension providing the labeling doesn’t mislead the user into thinking incorrectly that all words are present.

A.2 Terms and notation

A.2.1 Definitions of terms

ambiguous condition

The response of a Standard System to an ambiguous condition is left to the discretion of the implementor. A Standard System need not explicitly detect or report the occurrence of ambiguous conditions.

cross compiler

Cross-compilers may be used to prepare a program for execution in an embedded system, or may be used to generate Forth kernels either for the same or a different run-time environment.

data field

In earlier standards, data fields were known as “parameter fields”.

On subroutine threaded Forth systems, everything is object code. There are no traditional code or data fields. Only a word defined by **CREATE** or by a word that calls **CREATE** has a data field. Only a data field defined via **CREATE** can be manipulated portably.

word set

This Standard recognizes that some functions, while useful in certain application areas, are not sufficiently general to justify requiring them in all Forth systems. Further, it is helpful to group Forth words according to related functions. These issues are dealt with using the concept of word sets.

The “Core” word set contains the essential body of words in a Forth system. It is the only “required” word set. Other word sets defined in this Standard are optional additions to make it possible to provide Standard Systems with tailored levels of functionality.

A.2.2 Notation

A.2.2.2 Stack notation

The use of *-sys*, *orig*, and *dest* data types in stack effect diagrams conveys two pieces of information. First, it warns the reader that many implementations use the data stack in unspecified ways for those purposes, so that items underneath on either the control-flow or data stacks are unavailable. Second, in cases where *orig* and *dest* are used, explicit pairing rules are documented on the assumption that all systems will implement that model so that its results are equivalent to employment of some stack, and that in fact many implementations do use the data stack for this purpose. However, nothing in this Standard requires that implementations actually employ the data stack (or any other) for this purpose so long as the implied behavior of the model is maintained.

A.3 Usage requirements

Forth systems are unusually simple to develop, in comparison with compilers for more conventional languages such as C. In addition to Forth systems supported by vendors, public-domain implementations and implementation guides have been widely available for nearly twenty years, and a large number of individuals have developed their own Forth systems. As a result, a variety of implementation approaches have developed, each optimized for a particular platform or target market.

The X3J14 Technical Committee has endeavored to accommodate this diversity by constraining implementors as little as possible, consistent with a goal of defining a standard interface between an underlying Forth System and an application program being developed on it.

Similarly, we will not undertake in this section to tell you how to implement a Forth System, but rather will provide some guidance as to what the minimum requirements are for systems that can properly claim compliance with this Standard.

A.3.1 Data-types

Most computers deal with arbitrary bit patterns. There is no way to determine by inspection whether a cell contains an address or an unsigned integer. The only meaning a datum possesses is the meaning assigned by an application.

When data are operated upon, the meaning of the result depends on the meaning assigned to the input values. Some combinations of input values produce meaningless results: for instance, what meaning can be assigned to the arithmetic sum of the ASCII representation of the character “A” and a TRUE flag? The answer may be “no meaning”; or alternatively, that operation might be the first step in producing a checksum. Context is the determiner.

The discipline of circumscribing meaning which a program may assign to various combinations of bit patterns is sometimes called *data typing*. Many computer languages impose explicit data typing and have compilers that prevent ill-defined operations.

Forth rarely explicitly imposes data-type restrictions. Still, data types implicitly do exist, and discipline is required, particularly if portability of programs is a goal. In Forth, it is incumbent upon the programmer (rather than the compiler) to determine that data are accurately typed.

This section attempts to offer guidance regarding *de facto* data typing in Forth.

A.3.1.2 Character types

The correct identification and proper manipulation of the character data type is beyond the purview of Forth's enforcement of data type by means of stack depth. Characters do not necessarily occupy the entire width of their single stack entry with meaningful data. While the distinction between signed and unsigned character is entirely absent from the formal specification of Forth, the tendency in practice is to treat characters as short positive integers when mathematical operations come into play.

Standard Character Set

1) The storage unit for the character data type (**C@**, **C!**, **FILL**, etc.) must be able to contain unsigned numbers from 0 through 255.

2) An implementation is not required to restrict character storage to that range, but a Standard Program without environmental dependencies cannot assume the ability to store numbers outside that range in a "char" location.

3) The allowed number representations are two's-complement, one's-complement, and signed-magnitude. Note that all of these number systems agree on the representation of positive numbers.

4) Since a "*char*" can store small positive numbers and since the character data type is a sub-range of the unsigned integer data type, **C!** must store the n least-significant bits of a cell (8 <= n <= bits/cell). Given the enumeration of allowed number representations and their known encodings, "**TRUE xx C! xx C@**" must leave a stack item with some number of bits set, which will thus will be accepted as non-zero by **IF**.

5) For the purposes of input (**KEY**, **ACCEPT**, etc.) and output (**EMIT**, **TYPE**, etc.), the encoding between numbers and human-readable symbols is ISO646/IRV (ASCII) within the range from 32 to 126 (space to ~). EBCDIC is out (most "EBCDIC" computer systems support ASCII too). Outside that range, it is up to the implementation. The obvious implementation choice is to use ASCII control characters for the range from 0 to 31, at least for the "displayable" characters in that range (TAB, RETURN, LINEFEED, FORMFEED). However, this is not as clear-cut as it may seem, because of the variation between operating systems on the treatment of those characters. For example, some systems TAB to 4 character boundaries, others to 8 character boundaries, and others to preset tab stops. Some systems perform an automatic linefeed after a carriage return, others perform an automatic carriage return after a linefeed, and others do neither.

The codes from 128 to 255 may eventually be standardized, either formally or informally, for use as international characters, such as the letters with diacritical marks found in many European languages. One such encoding is the 8-bit ISO Latin-1 character set. The computer marketplace at large will eventually decide which encoding set of those characters prevails. For Forth implementations running under an operating system (the majority of those running on standard platforms these days), most Forth implementors will probably choose to do whatever the system does, without performing any remapping within the domain of the Forth system itself.

6) A Standard Program can depend on the ability to receive any character in the range 32 ... 126 through **KEY**, and similarly to display the same set of characters with **EMIT**. If a program must be able to receive or display any particular character outside that range, it can declare an environmental dependency on the ability to receive or display that character.

7) A Standard Program cannot use control characters in definition names. However, a Standard System is not required to enforce this prohibition. Thus, existing systems that currently allow control characters in words names from **BLOCK** source may continue to allow them, and programs running on those systems will continue to work. In text file source, the parsing action with space as a delimiter (e.g., **BL WORD**) treats control characters the same as spaces. This effectively implies that you cannot use control characters in definition names from text-file source, since the text interpreter will treat the control characters as delimiters. Note that this "control-character folding" applies only when space is the delimiter, thus the phrase "**CHAR) WORD**" may collect a string containing control characters.

Storage and retrieval

Characters are transferred from the data stack to memory by **C!** and from memory to the data stack by **C@**. A number of lower-significance bits equivalent to the implementation-dependent width of a *character* are transferred from a popped data stack entry to an address by the action of **C!** without affecting any bits which may comprise the higher-significance portion of the cell at the destination address; however, the action of **C@** clears all higher-significance bits of the data stack entry which it pushes that are beyond the implementation-dependent width of a character (which may include implementation-defined display information in the higher-significance bits). The programmer should keep in mind that operating upon arbitrary stack entries with words intended for the character data type may result in truncation of such data.

Manipulation on the stack

In addition to **C@** and **C!**, characters are moved to, from and upon the data stack by the following words:

>R ?DUP DROP DUP OVER PICK R> R@ ROLL ROT SWAP

Additional operations

The following mathematical operators are valid for character data:

+ - * / /MOD MOD

The following comparison and bitwise operators may be valid for characters, keeping in mind that display information cached in the most significant bits of characters in an implementation-defined fashion may have to be masked or otherwise dealt with:

AND OR > < U> U< = <> 0= 0<> MAX MIN
LSHIFT RSHIFT

A.3.1.3 Single-cell types

A single-cell stack entry viewed without regard to typing is the fundamental data type of Forth. All other data types are actually represented by one or more single-cell stack entries.

Storage and retrieval

Single-cell data are transferred from the stack to memory by **!**; from memory to the stack by **@**. All bits are transferred in both directions and no type checking of any sort is performed, nor does the Standard System check that a memory address used by **!** or **@** is properly aligned or properly sized to hold the datum thus transferred.

Manipulation on the stack

Here is a selection of the most important words which move single-cell data to, from and upon the data stack:

! @ >R ?DUP DROP DUP OVER PICK R> R@ ROLL ROT SWAP

Comparison operators

The following comparison operators are universally valid for one or more single cells:

= <> 0= 0<>

A.3.1.3.1 Flags

A **FALSE** flag is a single-cell datum with all bits unset, and a **TRUE** flag is a single-cell datum with all bits set. While Forth words which test flags accept any non-null bit pattern as true, there exists the concept of the *well-formed flag*. If an operation whose result is to be used as a flag may produce any bit-mask other than **TRUE** or **FALSE**, the recommended discipline is to convert the result to a well-formed flag by means of the Forth word **0<>** so that the result of any subsequent logical operations on the flag will be predictable.

In addition to the words which move, fetch and store single-cell items, the following words are valid for operations on one or more flag data residing on the data stack:

AND OR XOR INVERT

A.3.1.3.2 Integers

A single-cell datum may be treated by a Standard Program as a signed integer. Moving and storing such data is performed as for any single-cell data. In addition to the universally-applicable operators for single-cell data specified above, the following mathematical and comparison operators are valid for single-cell signed integers:

*** */ */MOD /MOD MOD + +! - / 1+ 1- ABS MAX MIN NEGATE**
0< 0> < >

Given the same number of bits, unsigned integers usually represent twice the number of absolute values representable by signed integers.

A single-cell datum may be treated by a Standard Program as an unsigned integer. Moving and storing such data is performed as for any single-cell data. In addition, the following mathematical and comparison operators are valid for single-cell unsigned integers:

UM* UM/MOD + +! - 1+ 1- * U< U>

A.3.1.3.3 Addresses

An address is uniquely represented as a single cell unsigned number and can be treated as such when being moved to, from, or upon the stack. Conversely, each unsigned number represents a unique address (which is not necessarily an address of accessible memory). This one-to-one relationship between addresses and unsigned numbers forces an equivalence between address arithmetic and the corresponding operations on unsigned numbers.

Several operators are provided specifically for address arithmetic:

CHAR+ CHARS CELL+ CELLS

and, if the floating-point word set is present:

FLOAT+ FLOATS SFLOAT+ SFLOATS DFLOAT+ DFLOATS

A Standard Program may never assume a particular correspondence between a Forth address and the physical address to which it is mapped.

A.3.1.3.4 Counted strings

The trend in ANS Forth is to move toward the consistent use of the "*c-addr u*" representation of strings on the stack. The use of the alternate "address of counted string" stack representation is discouraged. The traditional Forth words **WORD** and **FIND** continue to use the "address of counted string" representation for

historical reasons. The new word **C"** , added as a porting aid for existing programs, also uses the counted string representation.

Counted strings remain useful as a way to store strings in memory. This use is not discouraged, but when references to such strings appear on the stack, it is preferable to use the "*c-addr u*" representation.

A.3.1.3.5 Execution tokens

The association between an execution token and a definition is static. Once made, it does not change with changes in the search order or anything else. However it may not be unique, e.g., the phrases

' 1+ and

' CHAR+

might return the same value.

A.3.1.4 Cell-pair types

Storage and retrieval

Two operators are provided to fetch and store cell pairs:

2@ 2!

Manipulation on the stack

Additionally, these operators may be used to move cell pairs from, to and upon the stack:

2>R 2DROP 2DUP 2OVER 2R> 2SWAP 2ROT

Comparison

The following comparison operations are universally valid for cell pairs:

D= D0=

A.3.1.4.1 Double-Cell Integers

If a double-cell integer is to be treated as signed, the following comparison and mathematical operations are valid:

D+ D- D< D0< DABS DMAX DMIN DNEGATE M*/ M+

If a double-cell integer is to be treated as unsigned, the following comparison and mathematical operations are valid:

D+ D- UM/MOD DU<

A.3.1.4.2 Character strings

See: A.3.1.3.4 Counted Strings.

A.3.2 The Implementation environment

A.3.2.1 Numbers

Traditionally, Forth has been implemented on two's-complement machines where there is a one-to-one mapping of signed numbers to unsigned numbers–any single cell item can be viewed either as a signed or unsigned number. Indeed, the signed representation of any positive number is identical to the equivalent unsigned representation. Further, addresses are treated as unsigned numbers: there is no distinct pointer type. Arithmetic ordering on two's complement machines allows **+** and **–** to work on both signed and unsigned numbers. This arithmetic behavior is deeply embedded in common Forth practice. As a consequence of these behaviors, the likely ranges of signed and unsigned numbers for implementations hosted on each of the permissible arithmetic architectures is:

Arithmetic architecture	signed numbers	unsigned numbers
Two's complement	-*n*-1 to *n*	0 to 2*n*+1
One's complement	-*n* to *n*	0 to *n*
Signed magnitude	-*n* to *n*	0 to *n*

where *n* is the largest positive signed number. For all three architectures, signed numbers in the 0 to N range are bitwise identical to the corresponding unsigned number. Note that unsigned numbers on a signed magnitude machine are equivalent to signed non-negative numbers as a consequence of the forced correspondence between addresses and unsigned numbers and of the required behavior of **+** and **−**.

For reference, these number representations may be defined by the way that **NEGATE** is implemented:

two's complement: `: NEGATE INVERT 1+ ;`
one's complement: `: NEGATE INVERT ;`
signed-magnitude: `: NEGATE HIGH-BIT XOR ;`

where **HIGH-BIT** is a bit mask with only the most-significant bit set. Note that all of these number systems agree on the representation of non-negative numbers.

Per **3.2.1.1 Internal number representation** and **6.1.0270 0=**, the implementor must ensure that no standard or supported word return negative zero for any numeric (non-Boolean or flag) result. Many existing programmer assumptions will be violated otherwise.

There is no requirement to implement circular unsigned arithmetic, nor to set the range of unsigned numbers to the full size of a cell. There is historical precedent for limiting the range of *u* to that of +*n*, which is permissible when the cell size is greater than 16 bits.

A.3.2.1.2 Digit conversion

For example, an implementation might convert the characters "a" through "z" identically to the characters "A" through "Z," or it might treat the characters "[" through "~" as additional digits with decimal values 35 through 71, respectively.

A.3.2.2 Arithmetic

A.3.2.2.1 division

The Forth-79 Standard specifies that the signed division operators (**/**, **/MOD**, **MOD**, ***/MOD**, and ***/**) round non-integer quotients towards zero (symmetric division). Forth 83 changed the semantics of these operators to round towards negative infinity (floored division). Some in the Forth community have declined to convert systems and applications from the Forth-79 to the Forth-83 divide. To resolve this issue, an ANS Forth system is permitted to supply either floored or symmetric operators. In addition, ANS Forth systems must provide a floored division primitive (**FM/MOD**), a symmetric division primitive (**SM/REM**), and a mixed precision multiplication operator (**M***).

This compromise protects the investment made in current Forth applications; Forth-79 and Forth-83 programs are automatically compliant with ANS Forth with respect to division. In practice, the rounding direction rarely matters to applications. However, if a program requires a specific rounding direction, it can use the floored division primitive **FM/MOD** or the symmetric division primitive **SM/REM** to construct a division operator of the desired flavor. This simple technique can be used to convert Forth-79 and Forth-83 programs to ANS Forth without any analysis of the original programs.

A.3.2.2.2 Other integer operations

Whether underflow occurs depends on the data-type of the result. For example, the phrase **1 2 −** underflows if the result is unsigned and produces the valid signed result -1.

A.3.2.3 Stacks

The only data type in Forth which has concrete rather than abstract existence is the stack entry. Even this primitive typing Forth only enforces by the hard reality of stack underflow or overflow. The programmer must have a clear idea of the number of stack entries to be consumed by the execution of a word and the number of entries that will be pushed back to a stack by the execution of a word. The observation of anomalous occurrences on the data stack is the first line of defense whereby the programmer may recognize errors in an application program. It is also worth remembering that multiple stack errors caused by erroneous application code are frequently of equal and opposite magnitude, causing complementary (and deceptive) results.

For these reasons and a host of other reasons, the one unambiguous, uncontroversial, and indispensable programming discipline observed since the earliest days of Forth is that of providing a stack diagram for all additions to the application dictionary with the exception of static constructs such as **VARIABLE**s and **CONSTANT**s.

A.3.2.2.2 Control-flow stack

The simplest use of control-flow words is to implement the basic control structures shown in **figure A.1**.

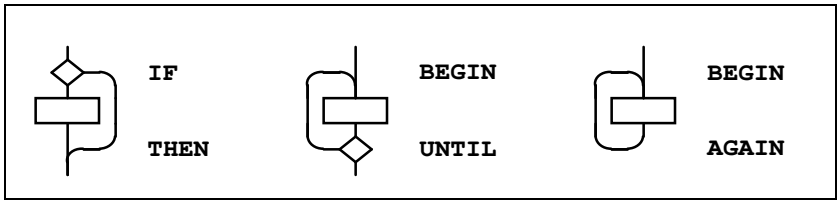

Figure A.1 – The basic control-flow patterns.

In control flow every branch, or transfer of control, must terminate at some destination. A natural implementation uses a stack to remember the origin of forward branches and the destination of backward branches. At a minimum, only the location of each origin or destination must be indicated, although other implementation-dependent information also may be maintained.

An origin is the location of the branch itself. A destination is where control would continue if the branch were taken. A destination is needed to resolve the branch address for each origin, and conversely, if every control-flow path is completed no unused destinations can remain.

With the addition of just three words (**AHEAD**, **CS-ROLL** and **CS-PICK**), the basic control-flow words supply the primitives necessary to compile a variety of transportable control structures. The abilities required are compilation of forward and backward conditional and unconditional branches and compile-time management of branch origins and destinations. **Table A.1** shows the desired behavior.

Table A.1 – Compilation behavior of control-flow words

at compile time, word:	supplies:	resolves:	is used to:
IF	*orig*		mark origin of forward conditional branch
THEN		*orig*	resolve **IF** or **AHEAD**
BEGIN	*dest*		mark backward destination
AGAIN		*dest*	resolve with backward unconditional branch
UNTIL		*dest*	resolve with backward conditional branch
AHEAD	*orig*		mark origin of forward unconditional branch
CS-PICK			copy item on control-flow stack
CS-ROLL			reorder items on control-flow stack

The requirement that control-flow words are properly balanced by other control-flow words makes reasonable the description of a compile-time implementation-defined *control-flow stack*. There is no prescription as to how the control-flow stack is implemented, e.g., data stack, linked list, special array. Each element of the control-flow stack mentioned above is the same size.

With these tools, the remaining basic control-structure elements, shown in **figure A.2**, can be defined. The stack notation used here for immediate words is (*compilation* / *execution*).

```
: WHILE  ( dest -- orig dest / flag -- )
   \ conditional exit from loops
   POSTPONE IF      \ conditional forward branch
   1 CS-ROLL ;  \ keep dest on top
 IMMEDIATE
: REPEAT  ( orig dest -- / -- )
   \ resolve a single WHILE and return to BEGIN
   POSTPONE AGAIN   \ uncond. backward branch to dest
   POSTPONE THEN ;  \ resolve forward branch from orig
 IMMEDIATE
: ELSE  ( orig1 -- orig2 / -- )
   \ resolve IF supplying alternate execution
   POSTPONE AHEAD   \ unconditional forward branch orig2
   1 CS-ROLL    \ put orig1 back on top
   POSTPONE THEN ;  \ resolve forward branch from orig1
 IMMEDIATE
```

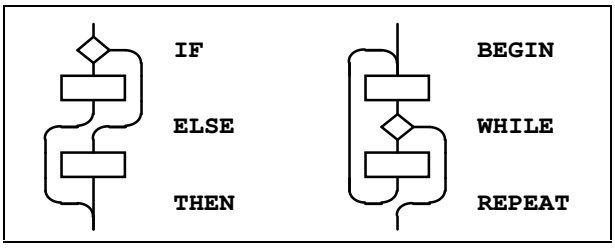

Figure A.2 – Additional basic control-flow patterns.

Forth control flow provides a solution for well-known problems with strictly structured programming.

The basic control structures can be supplemented, as shown in the examples in **figure A.3**, with additional **WHILE**s in **BEGIN ... UNTIL** and **BEGIN ... WHILE ... REPEAT** structures. However, for each additional **WHILE** there must be a **THEN** at the end of the structure. **THEN** completes the syntax with **WHILE** and indicates where to continue execution when the **WHILE** transfers control. The use of more than one additional **WHILE** is possible but not common. Note that if the user finds this use of **THEN** undesirable, an alias with a more likable name could be defined.

Additional actions may be performed between the control flow word (the **REPEAT** or **UNTIL**) and the **THEN** that matches the additional **WHILE**. Further, if additional actions are desired for normal termination and early termination, the alternative actions may be separated by the ordinary Forth **ELSE**. The termination actions are all specified after the body of the loop.

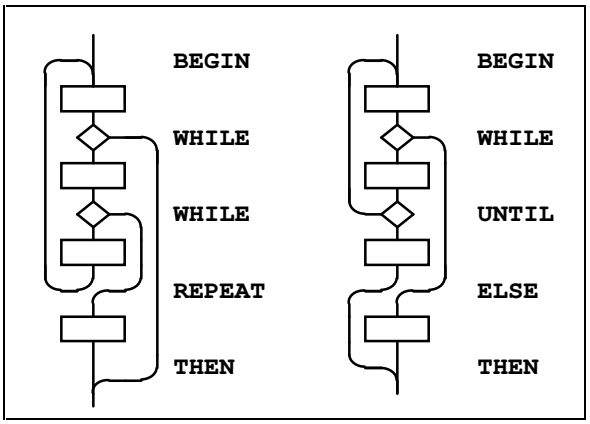

Figure A.3 – Extended control-flow pattern examples.

Note that **REPEAT** creates an anomaly when matching the **WHILE** with **ELSE** or **THEN**, most notable when compared with the **BEGIN...UNTIL** case. That is, there will be one less **ELSE** or **THEN** than there are **WHILE**s because **REPEAT** resolves one **THEN**. As above, if the user finds this count mismatch undesirable, **REPEAT** could be replaced in-line by its own definition.

Other loop-exit control-flow words, and even other loops, can be defined. The only requirements are that the control-flow stack is properly maintained and manipulated.

The simple implementation of the ANS Forth **CASE** structure below is an example of control structure extension. Note the maintenance of the data stack to prevent interference with the possible control-flow stack usage.

```
0 CONSTANT CASE IMMEDIATE  ( init count of OFs )


: OF  ( #of -- orig #of+1 / x -- )
   1+    ( count OFs )
   >R    ( move off the stack in case the control-flow )
         ( stack is the data stack. )
   POSTPONE OVER  POSTPONE = ( copy and test case value)
   POSTPONE IF    ( add orig to control flow stack )
   POSTPONE DROP  ( discards case value if = )
   R> ;           ( we can bring count back now )
 IMMEDIATE


: ENDOF ( orig1 #of -- orig2 #of )
   >R   ( move off the stack in case the control-flow )
        ( stack is the data stack. )
   POSTPONE ELSE
   R> ; ( we can bring count back now )
 IMMEDIATE


: ENDCASE  ( orig1..orign #of -- )
   POSTPONE DROP  ( discard case value )
   0 ?DO
     POSTPONE THEN
   LOOP ;
 IMMEDIATE
```

A.3.2.3.3 Return stack

The restrictions in section **3.2.3.3 Return stack** are necessary if implementations are to be allowed to place loop parameters on the return stack.

A.3.2.6 Environmental queries

The size in address units of various data types may be determined by phrases such as **`1 CHARS`**. Similarly, alignment may be determined by phrases such as **`1 ALIGN`**.

The environmental queries are divided into two groups: those that always produce the same value and those that might not. The former groups include entries such as **`MAX-N`**. This information is fixed by the hardware or by the design of the Forth system; a user is guaranteed that asking the question once is sufficient.

The other group of queries are for things that may legitimately change over time. For example an application might test for the presence of the Double Number word set using an environment query. If it is missing, the system could invoke a system-dependent process to load the word set. The system is permitted to change **`ENVIRONMENT?`**'s database so that subsequent queries about it indicate that it is present.

Note that a query that returns an "unknown" response could produce a "known" result on a subsequent query.

A.3.3 The Forth dictionary

A Standard Program may redefine a standard word with a non-standard definition. The program is still Standard (since it can be built on any Standard System), but the effect is to make the combined entity (Standard System plus Standard Program) a non-standard system.

A.3.3.1 Name space

A.3.3.1.2 Definition names

The language in this section is there to ensure the portability of Standard Programs. If a program uses something outside the Standard that it does not provide itself, there is no guarantee that another implementation will have what the program needs to run. There is no intent whatsoever to imply that all Forth programs will be somehow lacking or inferior because they are not standard; some of the finest jewels of the programmer's art will be non-standard. At the same time, the committee is trying to ensure that a program labeled "Standard" will meet certain expectations, particularly with regard to portability.

In many system environments the input source is unable to supply certain non-graphic characters due to external factors, such as the use of those characters for flow control or editing. In addition, when interpreting from a text file, the parsing function specifically treats non-graphic characters like spaces; thus words received by the text interpreter will not contain embedded non-graphic characters. To allow implementations in such environments to call themselves Standard, this minor restriction on Standard Programs is necessary.

A Standard System is allowed to permit the creation of definition names containing non-graphic characters. Historically, such names were used for keyboard editing functions and "invisible" words.

A.3.3.2 Code space

A.3.3.3 Data space

The words **`#TIB`**, **`>IN`**, **`BASE`**, **`BLK`**, **`SCR`**, **`SOURCE`**, **`SOURCE-ID`**, **`STATE`**, and **`TIB`** contain information used by the Forth system in its operation and may be of use to the application. Any assumption made by the application about data available in the Forth system it did not store other than the data just listed is an environmental dependency.

There is no point in specifying (in the Standard) both what is and what is not addressable.

A Standard Program may NOT address:

– Directly into the data or return stacks;
– Into a definition's data field if not stored by the application.

The read-only restrictions arise because some Forth systems run from ROM and some share I/O buffers with other users or systems. Portable programs cannot know which areas are affected, hence the general restrictions.

A.3.3.3.1 Address alignment

Many processors have restrictions on the addresses that can be used by memory access instructions. For example, on a Motorola 68000, 16-bit or 32-bit data can be accessed only at even addresses. Other examples include RISC architectures where 16-bit data can be loaded or stored only at even addresses and 32-bit data only at addresses that are multiples of four.

An implementor of ANS Forth can handle these alignment restrictions in one of two ways. Forth's memory access words (**@**, **!**, **+!**, etc.) could be implemented in terms of smaller-width access instructions which have no alignment restrictions. For example, on a 68000 Forth with 16-bit cells, **@** could be implemented with two 68000 byte-fetch instructions and a reassembly of the bytes into a 16-bit cell. Although this conceals hardware restrictions from the programmer, it is inefficient, and may have unintended side effects in some hardware environments. An alternate implementation of ANS Forth could define each memory-access word using the native instructions that most closely match the word's function. On a 68000 Forth with 16-bit cells, **@** would use the 68000's 16-bit move instruction. In this case, responsibility for giving **@** a correctly-aligned address falls on the programmer. A portable ANS Forth program must assume that alignment may be required and follow the requirements of this section.

A.3.3.3.2 Contiguous regions

The data space of a Forth system comes in discontinuous regions! The location of some regions is provided by the system, some by the program. Data space is contiguous within regions, allowing address arithmetic to generate valid addresses only within a single region. A Standard Program cannot make any assumptions about the relative placement of multiple regions in memory.

Section **3.3.3.2** does prescribe conditions under which contiguous regions of data space may be obtained. For example:

```
CREATE TABLE   1 C, 2 C, ALIGN 1000 , 2000 ,
```

makes a table whose address is returned by **TABLE**. In accessing this table,

```
TABLE C@                        will return 1
TABLE CHAR+ C@                  will return 2
TABLE 2 CHARS + ALIGNED @           will return 1000
TABLE 2 CHARS + ALIGNED CELL+ @ will return 2000.
```

Similarly,

```
CREATE DATA   1000 ALLOT
```

makes an array 1000 address units in size. A more portable strategy would define the array in application units, such as:

```
500 CONSTANT NCELLS
CREATE CELL-DATA  NCELLS CELLS ALLOT
```

This array can be indexed like this:

```
: LOOK   NCELLS 0 DO  CELL-DATA I CELLS + ? LOOP ;
```

A.3.3.3.6 Other transient regions

In many existing Forth systems, these areas are at **HERE** or just beyond it, hence the many restrictions.

(2*n)+2 is the size of a character string containing the unpunctuated binary representation of the maximum double number with a leading minus sign and a trailing space.

Implementation note: Since the minimum value of n is 16, the absolute minimum size of the pictured numeric output string is 34 characters. But if your implementation has a larger n, you must also increase the size of the pictured numeric output string.

A.3.4 The Forth text interpreter

A.3.4.3 Semantics

The "initiation semantics" correspond to the code that is executed upon entering a definition, analogous to the code executed by **EXIT** upon leaving a definition. The "run-time semantics" correspond to code fragments, such as literals or branches, that are compiled inside colon definitions by words with explicit compilation semantics.

In a Forth cross-compiler, the execution semantics may be specified to occur in the host system only, the target system only, or in both systems. For example, it may be appropriate for words such as **CELLS** to execute on the host system returning a value describing the target, for colon definitions to execute only on the target, and for **CONSTANT** and **VARIABLE** to have execution behaviors on both systems. Details of cross-compiler behavior are beyond the scope of this Standard.

A.3.4.3.2 Interpretation semantics

For a variety of reasons, this Standard does not define interpretation semantics for every word. Examples of these words are **>R**, **."**, **DO**, and **IF**. Nothing in this Standard precludes an implementation from providing interpretation semantics for these words, such as interactive control-flow words. However, a Standard Program may not use them in interpretation state.

A.3.4.5 Compilation

Compiler recursion at the definition level consumes excessive resources, especially to support locals. The Technical Committee does not believe that the benefits justify the costs. Nesting definitions is also not common practice and won't work on many systems.

A.4 Documentation requirements

A.4.1 System documentation

A.4.2 Program documentation

A.5 Compliance and labeling

A.5.1 ANS Forth system

Section **5.1** defines the criteria that a system must meet in order to justify the label "ANS Forth System." Briefly, the minimum requirement is that the system must "implement" the Core word set. There are several ways in which this requirement may be met. The most obvious is that all Core words may be in a pre-compiled kernel. This is not the only way of satisfying the requirement, however. For example, some words may be provided in source blocks or files with instructions explaining how to add them to the system if they are needed. So long as the words are provided in such a way that the user can obtain access to them with a clear and straightforward procedure, they may be considered to be present.

A Forth cross-compiler has many characteristics in common with an ANS Forth System, in that both use similar compiling tools to process a program. However, in order to fully specify an ANS Forth cross compiler it would be necessary to address complex issues dealing with compilation and execution semantics in both host and target environments as well as ROMability issues. The level of effort to do this properly has proved to be impractical at this time. As a result, although it may be possible for a Forth cross-compiler to correctly prepare an ANS Forth program for execution in a target environment, it is inappropriate for a cross-compiler to be labeled an ANS Forth System.

A.5.2 ANS Forth programs

A.5.2.2 Program labeling

Declaring an environmental dependency should not be considered undesirable, merely an acknowledgment that the author has taken advantage of some assumed architecture. For example, most computers in common use are based on two's complement binary arithmetic. By acknowledging an environmental dependency on this architecture, a programmer becomes entitled to use the number -1 to represent all bits set without significantly restricting the portability of the program.

Because all programs require space for data and instructions, and time to execute those instructions, they depend on the presence of an environment providing those resources. It is impossible to predict how little of some of these resources (e.g. stack space) might be necessary to perform some task, so this Standard does not do so.

On the other hand, as a program requires increasing levels of resources, there will probably be sucessively fewer systems on which it will execute sucessfully. An algorithm requiring an array of 10^9 octets might run on fewer computers than one requiring only 10^3.

Since there is also no way of knowing what minimum level of resources will be implemented in a system useful for at least some tasks, any program performing real work labeled simply an "ANS Forth Program" is unlikely to be labeled correctly.

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.3.4.2.2 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 $+n_1$ characters have been received. Terminating when $+n_1$ 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 ($+n_1$) 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  DUP 1-
   RECURSE  * THEN ;
```

n2 = n1(n1-1)(n1-2)...(2)(1), the product of n1 with all positive integers less than itself. 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
   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 **LOAD**ing 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 **CREATE**d 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: **: X ... limit first ?DO ... 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**:

1. **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.

2. **WORD** returns its result as a counted string. This has four bad effects:

   a) 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.

   b) **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.)

   c) 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**.

   d) 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 on the parse area will be predictable when it is used at the end of a line.

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 n_1 in **DATA** and returns the prior value n_2.

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 $n_2|u_2..n_3|u_3$ range straddles the overflow/underflow points or it does not. Lets examine the non-straddle case first:

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

The [denotes $n_2|u_2$, the) denotes $n_3|u_3$, and the dots and [are numbers **WITHIN** the range. $n_3|u_3$ is greater than $n_2|u_2$, so the following tests will determine if $n_1|u_1$ is **WITHIN** $n_2|u_2$ and $n_3|u_3$:

$n_2|u_2 \le n_1|u_1$ and $n_1|u_1 < n_3|u_3$.

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

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

$n_3|u_3$ is less than $n_2|u_2$ and the following tests will determine if $n_1|u_1$ is **WITHIN** $n_2|u_2$ and $n_3|u_3$:

$n_2|u_2 \le n_1|u_1$ or $n_1|u_1 < n_3|u_3$.

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

```
: WITHIN  ( test low high -- flag )
   >R OVER < 0=    ( test flag )
   SWAP R> <  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 -- )   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**

A.7 The Block word set

Early Forth systems ran stand-alone, with no host OS. Blocks of 1024 bytes were designed as a convenient unit of disk, and most native Forth systems still use them. It is relatively easy to write a native disk driver that maps head/track/sector addresses to block numbers. Such disk drivers are extremely fast in comparison with conventional file-oriented operating systems, and security is high because there is no reliance on a disk map.

Today many Forth implementations run under host operating systems, because the compatibility they offer the user outweighs the performance overhead. Many people who use such systems prefer using host OS files only; however, people who use both native and non-native Forths need a compatible way of accessing disk. The Block Word set includes the most common words for accessing program source and data on disk.

In order to guarantee that Standard Programs that need access to mass storage have a mechanism appropriate for both native and non-native implementations, ANS Forth requires that the Block word set be available if any mass storage facilities are provided. On non-native implementations, blocks normally reside in host OS files.

A.7.2 Additional terms

block

Many Forth systems use blocks to contain program source. Conventionally such blocks are formatted for editing as 16 lines of 64 characters. Source blocks are often referred to as “screens”.

A.7.6 Glossary

A.7.6.2.2190 SCR

SCR is short for screen.

A.8 The Double-Number word set

Forth systems on 8-bit and 16-bit processors often find it necessary to deal with double-length numbers. But many Forths on small embedded systems do not, and many users of Forth on systems with a cell size of 32-bits or more find that the necessity for double-length numbers is much diminished. Therefore, we have factored the words that manipulate double-length entities into this optional word set.

Please note that the naming convention used in this word set conveys some important information:

1. Words whose names are of the form 2xxx deal with cell pairs, where the relationship between the cells is unspecified. They may be two-vectors, double-length numbers, or any pair of cells that it is convenient to manipulate together.
2. Words with names of the form Dxxx deal specifically with double-length integers.
3. Words with names of the form Mxxx deal with some combination of single and double integers. The order in which these appear on the stack is determined by long-standing common practice.

Refer to **A.3.1** for a discussion of data types in Forth.

A.8.6 Glossary

A.8.6.1.0360 2CONSTANT

Typical use: **x1 x2 2CONSTANT name**

A.8.6.1.0390 2LITERAL

Typical use: **: X ... [x1 x2] 2LITERAL ... ;**

A.8.6.1.0440 2VARIABLE

Typical use: **2VARIABLE** *name*

A.8.6.1.1070 D.R

In **D.R**, the "R" is short for RIGHT.

A.8.6.1.1090 D2*

See **A.6.1.0320 2*** for applicable discussion.

A.8.6.1.1100 D2/

See **A.6.1.0330 2/** for applicable discussion.

A.8.6.1.1140 D>S

There exist number representations, e.g., the sign-magnitude representation, where reduction from double- to single-precision cannot simply be done with **DROP**. This word, equivalent to **DROP** on two's complement systems, desensitizes application code to number representation and facilitates portability.

A.8.6.1.1820 M*/

M*/ was once described by Chuck Moore as the most useful arithmetic operator in Forth. It is the main workhorse in most computations involving double-cell numbers. Note that some systems allow signed divisors. This can cost a lot in performance on some CPUs. The requirement for a positive divisor has not proven to be a problem.

A.8.6.1.1830 M+

M+ is the classical method for integrating.

A.9 The Exception word set

CATCH and **THROW** provide a reliable mechanism for handling exceptions, without having to propagate exception flags through multiple levels of word nesting. It is similar in spirit to the "non-local return" mechanisms of many other languages, such as C's **setjmp()** and **longjmp()**, and LISP's **CATCH** and **THROW**. In the Forth context, **THROW** may be described as a "multi-level **EXIT**", with **CATCH** marking a location to which a **THROW** may return.

Several similar Forth "multi-level **EXIT**" exception-handling schemes have been described and used in past years. It is not possible to implement such a scheme using only standard words (other than **CATCH** and **THROW**), because there is no portable way to "unwind" the return stack to a predetermined place.

THROW also provides a convenient implementation technique for the standard words **ABORT** and **ABORT"**, allowing an application to define, through the use of **CATCH**, the behavior in the event of a system **ABORT**.

This sample implementation of **CATCH** and **THROW** uses the non-standard words described below. They or their equivalents are available in many systems. Other implementation strategies, including directly saving the value of **DEPTH**, are possible if such words are not available.

SP@ (-- *addr*) returns the address corresponding to the top of data stack.

SP! (*addr* --) sets the stack pointer to *addr*, thus restoring the stack depth to the same depth that existed just before *addr* was acquired by executing **SP@**.

RP@ (-- *addr*) returns the address corresponding to the top of return stack.

RP! (*addr* --) sets the return stack pointer to *addr*, thus restoring the return stack depth to the same depth that existed just before *addr* was acquired by executing **RP@**.

```
VARIABLE HANDLER   0 HANDLER !  \ last exception handler

: CATCH  ( xt -- exception# | 0 ) \ return addr on stack
   SP@ >R          ( xt ) \ save data stack pointer
   HANDLER @ >R    ( xt ) \ and previous handler
   RP@ HANDLER !   ( xt ) \ set current handler
   EXECUTE         ( )    \ execute returns if no THROW
   R> HANDLER !    ( )    \ restore previous handler
   R> DROP         ( )    \ discard saved stack ptr
   0 ;             ( 0 )  \ normal completion

: THROW  ( ??? exception# -- ??? exception# )
   ?DUP IF           ( exc# ) \ 0 THROW is no-op
     HANDLER @ RP ! ( exc# ) \ restore prev return stack
     R> HANDLER !   ( exc# ) \ restore prev handler
     R> SWAP >R     ( saved-sp ) \ exc# on return stack
     SP! DROP R>    ( exc# ) \ restore stack
     \  Return to the caller of CATCH because return
     \  stack is restored to the state that existed
     \  when CATCH began execution
   THEN ;
```

In a multi-tasking system, the **HANDLER** variable should be in the per-task variable area (i.e., a user variable).

This sample implementation does not explicitly handle the case in which **CATCH** has never been called (i.e., the **ABORT** behavior). One solution is to add the following code after the **IF** in **THROW**:

```
HANDLER @ 0= IF ( empty the stack ) QUIT THEN
```

Another solution is to execute **CATCH** within **QUIT**, so that there is always an "exception handler of last resort" present. For example:

```
: QUIT
   ( empty the return stack and )
   ( set the input source to the user input device )
   POSTPONE [
   BEGIN
     REFILL
   WHILE
     ['] INTERPRET  CATCH
     CASE
        0 OF STATE @ 0= IF ." OK" THEN CR  ENDOF
       -1 OF ( Aborted) ENDOF
       -2 OF ( display  message from ABORT" ) ENDOF
       ( default ) DUP ." Exception # "  .
     ENDCASE
   REPEAT BYE ;
```

This example assumes the existance of a system-implementation word **INTERPRET** that embodies the text interpreter semantics described in **3.4 The Forth text interpreter**. Note that this implementation of **QUIT** automatically handles the emptying of the stack and return stack, due to **THROW**'s inherent restoration of the data and return stacks. Given this definition of **QUIT**, it's easy to define:

```
: ABORT  -1 THROW ;
```

In systems with other stacks in addition to the data and return stacks, the implementation of **CATCH** and **THROW** must save and restore those stack pointers as well. Such an "extended version" can be built on top

of this basic implementation. For example, with another stack pointer accessed with **FP@** and **FP!** only **CATCH** needs to be redefined:

```
: CATCH   ( xt -- exception# | 0 )
   FP@ >R  CATCH  R> OVER IF FP! ELSE DROP THEN ;
```

No change to **THROW** is necessary in this case. Note that, as with all redefinitions, the redefined version of **CATCH** will only be available to definitions compiled after the redefinition of **CATCH**.

CATCH and **THROW** provide a convenient way for an implementation to "clean up" the state of open files if an exception occurs during the text interpretation of a file with **INCLUDE-FILE**. The implementation of **INCLUDE-FILE** may guard (with **CATCH**) the word that performs the text interpretation, and if **CATCH** returns an exception code, the file may be closed and the exception re**THROW**n so that the files being included at an outer nesting level may be closed also. Note that the Standard allows, but does not require, **INCLUDE-FILE** to close its open files if an exception occurs. However, it does require **INCLUDE-FILE** to unnest the input source specification if an exception is **THROW**n.

A.9.3 Additional usage requirements

One important use of an exception handler is to maintain program control under many conditions which **ABORT**. This is practicable only if a range of codes is reserved. Note that an application may overload many standard words in such a way as to **THROW** ambiguous conditions not normally **THROW**n by a particular system.

A.9.3.6 Exception handling

The method of accomplishing this coupling is implementation dependent. For example, **LOAD** could "know" about **CATCH** and **THROW** (by using **CATCH** itself, for example), or **CATCH** and **THROW** could "know" about **LOAD** (by maintaining input source nesting information in a data structure known to **THROW**, for example). Under these circumstances it is not possible for a Standard Program to define words such as **LOAD** in a completely portable way.

A.9.6 Glossary

A.9.6.1.2275 THROW

If **THROW** is executed with a non zero argument, the effect is as if the corresponding **CATCH** had returned it. In that case, the stack depth is the same as it was just before **CATCH** began execution. The values of the *i*x* stack arguments could have been modified arbitrarily during the execution of *xt*. In general, nothing useful may be done with those stack items, but since their number is known (because the stack depth is deterministic), the application may **DROP** them to return to a predictable stack state.

Typical use:

```
: could-fail ( -- char )
   KEY DUP [CHAR] Q =  IF  1 THROW THEN ;
: do-it ( a b -- c)   2DROP could-fail ;
: try-it ( --)
   1 2 ['] do-it  CATCH  IF
     ( x1 x2 ) 2DROP ." There was an exception" CR
   ELSE ." The character was " EMIT CR
   THEN ;
: retry-it ( -- )
   BEGIN  1 2 ['] do-it CATCH  WHILE
     ( x1 x2) 2DROP  ." Exception, keep trying" CR
   REPEAT ( char )
   ." The character was " EMIT CR ;
```

A.10 The Facility word set

A.10.6 Glossary

A.10.6.1.0742 AT-XY

Most implementors supply a method of positioning a cursor on a CRT screen, but there is great variance in names and stack arguments. This version is supported by at least one major vendor.

A.10.6.1.1755 KEY?

The Technical Committee has gone around several times on the stack effects. Whatever is decided will violate somebody's practice and penalize some machine. This way doesn't interfere with type-ahead on some systems, while requiring the implementation of a single-character buffer on machines where polling the keyboard inevitably results in the destruction of the character.

Use of **KEY** or **KEY?** indicates that the application does not wish to bother with non-character events, so they are discarded, in anticipation of eventually receiving a valid character. Applications wishing to handle non-character events must use **EKEY** and **EKEY?**. It is possible to mix uses of **KEY?** / **KEY** and **EKEY?** / **EKEY** within a single application, but the application must use **KEY?** and **KEY** only when it wishes to discard non-character events until a valid character is received.

A.10.6.2.1305 EKEY

EKEY provides a standard word to access a system-dependent set of "raw" keyboard events, including events corresponding to members of the standard character set, events corresponding to other members of the implementation-defined character set, and keystrokes that do not correspond to members of the character set.

EKEY assumes no particular numerical correspondence between particular event code values and the values representing standard characters. On some systems, this may allow two separate keys that correspond to the same standard character to be distinguished from one another.

In systems that combine both keyboard and mouse events into a single "event stream", the single number returned by **EKEY** may be inadequate to represent the full range of input possibilities. In such systems, a single "event record" may include a time stamp, the x,y coordinates of the mouse position, the keyboard state, and the state of the mouse buttons. In such systems, it might be appropriate for **EKEY** to return the address of an "event record" from which the other information could be extracted.

Also, consider a hypothetical Forth system running under MS-DOS on a PC-compatible computer. Assume that the implementation-defined character set is the "normal" 8-bit PC character set. In that character set, the codes from 0 to 127 correspond to ASCII characters. The codes from 128 to 255 represent characters from various non-English languages, mathematical symbols, and some graphical symbols used for line drawing. In addition to those characters, the keyboard can generate various other "scan codes", representing such non-character events as arrow keys and function keys.

There may be multiple keys, with different scan codes, corresponding to the same standard character. For example, the character representing the number "1" often appears both in the row of number keys above the alphabetic keys, and also in the separate numeric keypad.

When a program asks the MS-DOS operating system for a keyboard event, it receives either a single non-zero byte, representing a character, or a zero byte followed by a "scan code" byte, representing a non-character keyboard event (e.g., a function key).

EKEY represents each keyboard event as a single number, rather than as a sequence of numbers. For the system described above, the following would be a reasonable implementation of **EKEY** and related words:

The **MAX-CHAR** environmental query would return 255.

Assume the existence of a word **DOS-KEY** (-- *char*) which executes the MS-DOS "Direct STDIN Input" system call (Interrupt 21h, Function 07h) and a word **DOS-KEY?** (-- *flag*) which executes the MS-DOS "Check STDIN Status" system call (Interrupt 21h, Function 0Bh).

```
: EKEY?  ( -- flag )  DOS-KEY?  0<>  ;
: EKEY  ( -- u )
   DOS-KEY ?DUP 0= IF
     DOS-KEY  256 +
   THEN ;
: EKEY>CHAR  ( u -- u false | char true )
   DUP 256 >= IF           ( u )
     DUP 259 = IF          \ 259 is Ctrl-@ (ASCII NUL)
       DROP 0 TRUE EXIT    \ so replace with character
     THEN FALSE EXIT       \ otherwise extended character
   THEN  TRUE ;            \ normal extended ASCII char.
VARIABLE PENDING-CHAR   -1 PENDING-CHAR !
: KEY?  ( -- flag )
   PENDING-CHAR @ 0>=  IF  TRUE EXIT  THEN
   BEGIN  EKEY? WHILE
     EKEY EKEY>CHAR IF
       PENDING-CHAR !  TRUE EXIT
     THEN DROP
   REPEAT  FALSE ;
: KEY  ( -- char )
   PENDING-CHAR @ 0>= IF
     PENDING-CHAR @  -1 PENDING-CHAR !  EXIT
   THEN
   BEGIN  EKEY  EKEY>CHAR 0= WHILE
     DROP
   REPEAT ;
```

This is a full-featured implementation, providing the application program with an easy way to either handle non-character events (with **EKEY**), or to ignore them and to only consider "real" characters (with **KEY**).

Note that **EKEY** maps scan codes from 0 to 255 into numbers from 256 to 511. **EKEY** maps the number 259, representing the keyboard combination Ctrl-Shift-@, to the character whose numerical value is 0 (ASCII NUL). Many ASCII keyboards generate ASCII NUL for Ctrl-Shift-@, so we use that key combination for ASCII NUL (which is otherwise unavailable from MS-DOS, because the zero byte signifies that another scan-code byte follows).

One consequence of using the "Direct STDIN Input" system call (function 7) instead of the "STDIN Input" system call (function 8) is that the normal DOS "Ctrl-C interrupt" behavior is disabled when the system is waiting for input (Ctrl-C would still cause an interrupt while characters are being output). On the other hand, if the "STDIN Input" system call (function 8) were used to implement **EKEY**, Ctrl-C interrupts would be enabled, but Ctrl-Shift-@ would also cause an interrupt, because the operating system would treat the second byte of the 0,3 sequence as a Ctrl-C, even though the 3 is really a scan code and not a character. One "best of both worlds" solution is to use function 8 for the first byte received by **EKEY**, and function 7 for the scan code byte. For example:

```
: EKEY  ( -- u )
   DOS-KEY-FUNCTION-8  ?DUP  0=  IF
     DOS-KEY-FUNCTION-7  DUP 3  =  IF
       DROP 0  ELSE  256 +
```

```
    THEN
  THEN ;
```

Of course, if the Forth implementor chooses to pass Ctrl-C through to the program, without using it for its usual interrupt function, then DOS function 7 is appropriate in both cases (and some additional care must be taken to prevent a typed-ahead Ctrl-C from interrupting the Forth system during output operations).

A Forth system might also choose a simpler implementation of **KEY**, without implementing **EKEY**, as follows:

```
: KEY   ( -- char )  DOS-KEY  ;
: KEY?  ( -- flag )  DOS-KEY? 0<>  ;
```

The disadvantages of the simpler version are:

a) An application program that uses **KEY**, expecting to receive only valid characters, might receive a sequence of bytes (e.g., a zero byte followed by a byte with the same numerical value as the letter "A") that appears to contain a valid character, even though the user pressed a key (e.g., function key 4) that does not correspond to any valid character.

b) An application program that wishes to handle non-character events will have to execute **KEY** twice if it returns zero the first time. This might appear to be a reasonable and easy thing to do. However, such code is not portable to other systems that do not use a zero byte as an "escape" code. Using the **EKEY** approach, the algorithm for handling keyboard events can be the same for all systems; the system dependencies can be reduced to a table or set of constants listing the system-dependent key codes used to access particular application functions. Without **EKEY**, the algorithm, not just the table, is likely to be system dependent.

Another approach to **EKEY** on MS-DOS is to use the BIOS "Read Keyboard Status" function (Interrupt 16h, Function 01h) or the related "Check Keyboard" function (Interrupt 16h, Function 11h). The advantage of this function is that it allows the program to distinguish between different keys that correspond to the same character (e.g. the two "1" keys). The disadvantage is that the BIOS keyboard functions read only the keyboard. They cannot be "redirected" to another "standard input" source, as can the DOS STDIN Input functions.

A.10.6.2.1306 EKEY>CHAR

EKEY>CHAR translates a keyboard event into the corresponding member of the character set, if such a correspondence exists for that event.

It is possible that several different keyboard events may correspond to the same character, and other keyboard events may correspond to no character.

A.10.6.2.1325 EMIT?

An indefinite delay is a device related condition, such as printer off-line, that requires operator intervention before the device will accept new data.

A.10.6.2.1905 MS

Although their frequencies vary, every system has a clock. Since many programs need to time intervals, this word is offered. Use of milliseconds as an internal unit of time is a practical "least common denominator" external unit. It is assumed implementors will use "clock ticks" (whatever size they are) as an internal unit and convert as appropriate.

A.10.6.2.2292 TIME&DATE

Most systems have a real-time clock/calendar. This word gives portable access to it.

A.11 The File-Access word set

Many Forth systems support access to a host file system, and many of these support interpretation of Forth from source text files. The Forth-83 Standard did not address host OS files. Nevertheless, a degree of similarity exists among modern implementations.

For example, files must be opened and closed, created and deleted. Forth file-system implementations differ mostly in the treatment and disposition of the exception codes, and in the format of the file-identification strings. The underlying mechanism for creating file-control blocks might or might not be visible. We have chosen to keep it invisible.

Files must also be read and written. Text files, if supported, must be read and written one line at a time. Interpretation of text files implies that they are somehow integrated into the text interpreter input mechanism. These and other requirements have shaped the file-access extensions word set.

Most of the existing implementations studied use simple English words for common host file functions: **OPEN**, **CLOSE**, **READ**, etc. Although we would have preferred to do likewise, there were so many minor variations in implementation of these words that adopting any particular meaning would have broken much existing code. We have used names with a suffix **-FILE** for most of these words. We encourage implementors to conform their single-word primitives to the ANS behaviors, and hope that if this is done on a widespread basis we can adopt better definition names in a future standard.

Specific rationales for members of this word set follow.

A.11.3 Additional usage requirements

A.11.3.2 Blocks in files

Many systems reuse file identifiers; when a file is closed, a subsequently opened file may be given the same identifier. If the original file has blocks still in block buffers, these will be incorrectly associated with the newly opened file with disastrous results. The block buffer system must be flushed to avoid this.

A.11.6 Glossary

A.11.6.1.0765 BIN

Some operating systems require that files be opened in a different mode to access their contents as an unstructured stream of binary data rather than as a sequence of lines.

The arguments to **READ-FILE** and **WRITE-FILE** are arrays of character storage elements, each element consisting of at least 8 bits. The Technical Committee intends that, in **BIN** mode, the contents of these storage elements can be written to a file and later read back without alteration. The Technical Committee has declined to address issues regarding the impact of “wide” characters on the File and Block word sets.

A.11.6.1.1010 CREATE-FILE

Typical use:

```
: X ... S" TEST.FTH" R/W CREATE-FILE
   ABORT" CREATE-FILE FAILED" ... ;
```

A.11.6.1.1717 INCLUDE-FILE

Here are two implementation alternatives for saving the input source specification in the presence of text file input:

1) Save the file position (as returned by **FILE-POSITION**) of the beginning of the line being interpreted. To restore the input source specification, seek to that position and re-read the line into the input buffer.

2) Allocate a separate line buffer for each active text input file, using that buffer as the input buffer. This method avoids the "seek and reread" step, and allows the use of "pseudo-files" such as pipes and other sequential-access-only communication channels.

A.11.6.1.1718 INCLUDED

Typical use: **... S" filename" INCLUDED ...**

A.11.6.1.1970 OPEN-FILE

Typical use:

```
: X ... S" TEST.FTH" R/W OPEN-FILE
   ABORT" OPEN-FILE FAILED" ... ;
```

A.11.6.1.2080 READ-FILE

A typical sequential file-processing algorithm might look like:

```
BEGIN              (  )
   . . . READ-FILE THROW   ( length )
?DUP WHILE            ( length )
   . . .           (  )
REPEAT                (  )
```

In this example, **THROW** is used to handle (unexpected) exception conditions, which are reported as non-zero values of the *ior* return value from **READ-FILE**. End-of-file is reported as a zero value of the "length" return value.

A.11.6.1.2090 READ-LINE

Implementations are allowed to store the line terminator in the memory buffer in order to allow the use of line reading functions provided by host operating systems, some of which store the terminator. Without this provision, a temporary buffer might be needed. The two-character limitation is sufficient for the vast majority of existing operating systems. Implementations on host operating systems whose line terminator sequence is longer than two characters may have to take special action to prevent the storage of more than two terminator characters.

Standard Programs may not depend on the presence of any such terminator sequence in the buffer.

A typical line-oriented sequential file-processing algorithm might look like:

```
BEGIN              (  )
   . . . READ-LINE  THROW  ( length not-eof-flag )
WHILE              ( length )
   . . .           (  )
REPEAT DROP           (  )
```

In this example, **THROW** is used to handle (unexpected) I/O exception condition, which are reported as non-zero values of the "*ior*" return value from **READ-LINE**.

READ-LINE needs a separate end-of-file flag because empty (zero-length) lines are a routine occurrence, so a zero-length line cannot be used to signify end-of-file.

A.11.6.1.2165 S"

Typical use: **... S" ccc" ...**

The interpretation semantics for **S"** are intended to provide a simple mechanism for entering a string in the interpretation state. Since an implementation may choose to provide only one buffer for interpreted strings, an interpreted string is subject to being overwritten by the next execution of **S"** in interpretation

state. It is intended that no standard words other than **S"** should in themselves cause the interpreted string to be overwritten. However, since words such as **EVALUATE**, **LOAD**, **INCLUDE-FILE** and **INCLUDED** can result in the interpretation of arbitrary text, possibly including instances of **S"**, the interpreted string may be invalidated by some uses of these words.

When the possibility of overwriting a string can arise, it is prudent to copy the string to a "safe" buffer allocated by the application.

Programs wishing to parse in the fashion of **S"** are advised to use **PARSE** or **WORD COUNT** instead of **S"**, preventing the overwriting of the interpreted string buffer.

A.12 The Floating-Point word set

The Technical Committee has considered many proposals dealing with the inclusion and makeup of the Floating-Point Word Sets in ANS Forth. Although it has been argued that ANS Forth should not address floating-point arithmetic and numerous Forth applications do not need floating-point, there are a growing number of important Forth applications from spread sheets to scientific computations that require the use of floating-point arithmetic. Initially the Technical Committee adopted proposals that made the Forth Vendors Group Floating-Point Standard, first published in 1984, the framework for inclusion of Floating-Point in ANS Forth. There is substantial common practice and experience with the Forth Vendors Group Floating-Point Standard. Subsequently the Technical Committee adopted proposals that placed the basic floating-point arithmetic, stack and support words in the Floating-Point word set and the floating-point transcendental functions in the Floating-Point Extensions word set. The Technical Committee also adopted proposals that:

- Changed names for clarity and consistency; e.g., **REALS** to **FLOATS**, and **REAL+** to **FLOAT+** .
- Removed words; e.g., **FPICK** .
- Added words for completeness and increased functionality; e.g., **FSINCOS**, **F~**, **DF@**, **DF!**, **SF@** and **SF!**

Several issues concerning the Floating-Point word set were resolved by consensus in the Technical Committee:

Floating-point stack: By default the floating-point stack is separate from the data and return stacks; however, an implementation may keep floating-point numbers on the data stack. A program can determine whether floating-point numbers are kept on the data stack by passing the string **FLOATING-STACK** to **ENVIRONMENT?** It is the experience of several members of the Technical Committee that with proper coding practices it is possible to write floating-point code that will run identically on systems with a separate floating-point stack and with floating-point numbers kept on the data stack.

Floating-point input: The current base must be **DECIMAL**. Floating-point input is not allowed in an arbitrary base. All floating-point numbers to be interpreted by an ANS Forth system must contain the exponent indicator "**E**" (see **12.3.7 Text interpreter input number conversion**). Consensus in the Technical Committee deemed this form of floating-point input to be in more common use than the alternative that would have a floating-point input mode that would allow numbers with embedded decimal points to be treated as floating-point numbers.

Floating-point representation: Although the format and precision of the significand and the format and range of the exponent of a floating-point number are implementation defined in ANS Forth, the Floating-Point Extensions word set contains the words **DF@**, **SF@**, **DF!**, and **SF!** for fetching and storing double- and single-precision IEEE floating-point-format numbers to memory. The IEEE floating-point format is commonly used by numeric math co-processors and for exchange of floating-point data between programs and systems.

A.12.3 Additional usage requirements

A.12.3.5 Address alignment

In defining custom floating-point data structures, be aware that **CREATE** doesn't necessarily leave the data space pointer aligned for various floating-point data types. Programs may comply with the requirement for the various kinds of floating-point alignment by specifying the appropriate alignment both at compile-time and execution time. For example:

```
: FCONSTANT ( F:  r -- )
   CREATE FALIGN  HERE  1 FLOATS ALLOT  F!
   DOES> ( F:  -- r )  FALIGNED F@ ;
```

A.12.3.7 Text interpreter input number conversion

The Technical Committee has more than once received the suggestion that the text interpreter in Standard Forth systems should treat numbers that have an embedded decimal point, but no exponent, as floating-point numbers rather than double cell numbers. This suggestion, although it has merit, has always been voted down because it would break too much existing code; many existing implementations put the full digit string on the stack as a double number and use other means to inform the application of the location of the decimal point.

A.12.6 Glossary

A.12.6.1.0558 **>FLOAT**

>FLOAT enables programs to read floating-point data in legible ASCII format. It accepts a much broader syntax than does the text interpreter since the latter defines rules for composing source programs whereas **>FLOAT** defines rules for accepting data. **>FLOAT** is defined as broadly as is feasible to permit input of data from ANS Forth systems as well as other widely used standard programming environments.

This is a synthesis of common FORTRAN practice. Embedded spaces are explicitly forbidden in much scientific usage, as are other field separators such as comma or slash.

While **>FLOAT** is not required to treat a string of blanks as zero, this behavior is strongly encouraged, since a future version of ANS Forth may include such a requirement.

A.12.6.1.1427 **F.**

For example, **1E3 F.** displays **1000.** .

A.12.6.1.1492 **FCONSTANT**

Typical use: **r FCONSTANT name**

A.12.6.1.1552 **FLITERAL**

Typical use: **: X ... [... (r)] FLITERAL ... ;**

A.12.6.1.1630 **FVARIABLE**

Typical use: **FVARIABLE name**

A.12.6.1.2143 **REPRESENT**

This word provides a primitive for floating-point display. Some floating-point formats, including those specified by IEEE-754, allow representations of numbers outside of an implementation-defined range. These include plus and minus infinities, denormalized numbers, and others. In these cases we expect that

REPRESENT will usually be implemented to return appropriate character strings, such as "+infinity" or "nan", possibly truncated.

A.12.6.2.1489 FATAN2

FSINCOS and **FATAN2** are a complementary pair of operators which convert angles to 2-vectors and vice-versa. They are essential to most geometric and physical applications since they correctly and unambiguously handle this conversion in all cases except null vectors, even when the tangent of the angle would be infinite.

FSINCOS returns a Cartesian unit vector in the direction of the given angle, measured counter-clockwise from the positive X-axis. The order of results on the stack, namely y underneath x, permits the 2-vector data type to be additionally viewed and used as a ratio approximating the tangent of the angle. Thus the phrase **FSINCOS F/** is functionally equivalent to **FTAN**, but is useful over only a limited and discontinuous range of angles, whereas **FSINCOS** and **FATAN2** are useful for all angles. This ordering has been found convenient for nearly two decades, and has the added benefit of being easy to remember. A corollary to this observation is that vectors in general should appear on the stack in this order.

The argument order for **FATAN2** is the same, converting a vector in the conventional representation to a scalar angle. Thus, for all angles, **FSINCOS FATAN2** is an identity within the accuracy of the arithmetic and the argument range of **FSINCOS**. Note that while **FSINCOS** always returns a valid unit vector, **FATAN2** will accept any non-null vector. An ambiguous condition exists if the vector argument to **FATAN2** has zero magnitude.

A.12.6.2.1516 FEXPM1

This function allows accurate computation when its arguments are close to zero, and provides a useful base for the standard exponential functions. Hyperbolic functions such as cosh(*x*) can be efficiently and accurately implemented by using **FEXPM1**; accuracy is lost in this function for small values of *x* if the word **FEXP** is used.

An important application of this word is in finance; say a loan is repaid at 15% per year; what is the daily rate? On a computer with single precision (six decimal digit) accuracy:

1. Using **FLN** and **FEXP**:

   FLN of 1.15 = 0.139762,
   divide by 365 = 3.82910E-4,
   form the exponent using **FEXP** = 1.00038, and
   subtract one (1) and convert to percentage = 0.038%.

Thus we only have two digit accuracy.

2. Using **FLNP1** and **FEXPM1**:

   FLNP1 of 0.15 = 0.139762, (this is the same value as in the first example, although with the argument closer to zero it may not be so)
   divide by 365 = 3.82910E-4,
   form the exponent and subtract one (1) using **FEXPM1** = 3.82983E-4, and
   convert to percentage = 0.0382983%.

This is full six digit accuracy.

The presence of this word allows the hyperbolic functions to be computed with usable accuracy. For example, the hyperbolic sine can be defined as:

```
: FSINH  ( r1 -- r2 )
   FEXPM1  FDUP  FDUP 1.0E0 F+  F/  F+  2.0E0 F/ ;
```

A.12.6.2.1554 FLNP1

This function allows accurate compilation when its arguments are close to zero, and provides a useful base for the standard logarithmic functions. For example, **FLN** can be implemented as:

```
: FLN   1.0E0 F-  FLNP1 ;
```

See: **A.12.2.1516 FEXPM1**.

A.12.6.2.1616 FSINCOS

See: **A.12.2.1489 FATAN2**.

A.12.6.2.1640 F~

This provides the three types of “floating point equality” in common use -- “close” in absolute terms, exact equality as represented, and “relatively close”.

A.13 The Locals word set

The Technical Committee has had a problem with locals. It has been argued forcefully that ANS Forth should say nothing about locals since:

– there is no clear accepted practice in this area;
– not all Forth programmers use them or even know what they are; and
– few implementations use the same syntax, let alone the same broad usage rules and general approaches.

It has also been argued, it would seem equally forcefully, that the lack of any standard approach to locals is precisely the reason for this lack of accepted practice since locals are at best non-trivial to implement in a portable and useful way. It has been further argued that users who have elected to become dependent on locals tend to be locked into a single vendor and have little motivation to join the group that it is hoped will “broadly accept” ANS Forth unless the Standard addresses their problems.

Since the Technical Committee has been unable to reach a strong consensus on either leaving locals out or on adopting any particular vendor’s syntax, it has sought some way to deal with an issue that it has been unable to simply dismiss. Realizing that no single mechanism or syntax can simultaneously meet the desires expressed in all the locals proposals that have been received, it has simplified the problem statement to be to define a locals mechanism that:

– is independent of any particular syntax;
– is user extensible;
– enables use of arbitrary identifiers, local in scope to a single definition;
– supports the fundamental cell size data types of Forth; and
– works consistently, especially with respect to re-entrancy and recursion.

This appears to the Technical Committee to be what most of those who actively use locals are trying to achieve with them, and it is at present the consensus of the Technical Committee that if ANS Forth has anything to say on the subject this is an acceptable thing for it to say.

This approach is proposed as one that can be used with a small amount of user coding to implement some, but not all, of the locals schemes in use. The following coding examples illustrate how it can be used to implement two syntaxes.

1. A proposed syntax: (**LOCAL** *name*) with additional usage rules:

```
: LOCAL  ( "name")  BL WORD COUNT (LOCAL) ;  IMMEDIATE
: END-LOCALS   0 0 (LOCAL) ;  IMMEDIATE
```

```
: EXAMPLE  ( n -- n n2 n3)  LOCAL N  END-LOCALS
   N DUP N * DUP N * ;
```

2. The syntax used by systems of Creative Solutions, Inc.:

```
: LOCALS|  ( "name...name |" -- )
   BEGIN BL WORD   COUNT OVER C@
      [CHAR] | - OVER 1 - OR
   WHILE (LOCAL)
   REPEAT 2DROP   0 0 (LOCAL);  IMMEDIATE

: EXAMPLE  ( n -- n2 n3)
   LOCALS| N | N  DUP N *  DUP N * ;
```

Other syntaxes can be implemented, although some will admittedly require considerably greater effort or in some cases program conversion. Yet other approaches to locals are completely incompatible due to gross differences in usage rules and in some cases even scope identifiers. For example, the complete local scheme in use at Johns Hopkins had elaborate semantics that cannot be duplicated in terms of this model.

To reinforce the intent of section **13**, here are two examples of actual use of locals. The first illustrates correct usage:

(1.1)
```
: {  BEGIN  BL WORD COUNT OVER C@
        [CHAR] } - OVER 1 - OR
     WHILE (LOCAL)
     REPEAT 2DROP 0 0 (LOCAL) ;  IMMEDIATE
```

(1.2)
```
: JOE  ( a b c -- n ) >R 2* R> 2DUP + 0
   { ANS 2B+C C 2B A }
   2 0 DO  1 ANS + I + TO ANS  ANS .  LOOP
   ANS . 2B+C . C . 2B . A . ANS ;
```

(1.3)
```
100 300 10 JOE .
```

(1.1) defines a local declaration syntax that surrounds the list of locals with braces. It doesn't do anything fancy, such as reordering locals or providing initial values for some of them, so locals are initialized from the stack in the default order. (1.2) illustrates a use of this syntax in the definition of **JOE**. Note that work is performed at execution time in that definition before locals are declared. It's OK to use the return stack as long as whatever is placed there is removed before the declarations begin.

Note that before declaring locals, **B** is doubled, a subexpression (**2B+C**) is computed, and an initial value (zero) for **ANS** is provided. After locals have been declared, **JOE** proceeds to use them. Note that locals may be accessed and updated within do-loops. The effect of interpreting (1.3) is to display the following values:

1 (**ANS** the first time through the loop),
3 (**ANS** the second time),
3 (**ANS**) 610 (**2B+C**). 10 (**C**), 600 (**2B**) 100 (**A**), and
3 (**ANS** left on the stack by **JOE**).

The *names* of the locals vanish after **JOE** has been compiled. The *storage and meaning* of locals appear when **JOE**'s locals are declared and vanish as **JOE** returns to its caller at **;** (semicolon).

A second set of examples illustrates various things that break the rules. We assume that the definitions of **LOCAL** and **END-LOCALS** above are present, along with **{** from the preceding example.

(2.1)
```
: ZERO   0 POSTPONE LITERAL POSTPONE LOCAL ; IMMEDIATE
```

(2.2)
```
: JOE  ( a b)  ZERO TEMP  LOCAL B  1+ LOCAL A+  ZERO ANSWER ;
```

(2.3)
```
: BOB  ( a b c d)  { D C }  { B A } ;
```

Here are two definitions with various violations of rule **13.3.3** (1). In (2.2) the declaration of **TEMP** is legal and creates a local whose initial value is zero. It's OK because the executable code that **ZERO**

generates precedes the first use of **(LOCAL)** in the definition. However, the **1+** preceding the declaration of **A+** is illegal. Likewise the use of **ZERO** to define **ANSWER** is illegal because it generates executable code between uses of **(LOCAL)**. Finally, **JOE** terminates illegally (no **END-LOCALS**). **BOB** in (2.3) violates the rule against declaring two sets of locals.

(2.4) **: ANN (a b -- b) DUP >R DUP IF { B A } THEN R> ;**

(2.5) **: JANE (a b -- n) { B A } A B + >R A B - R> / ;**

ANN in (2.4) violates two rules. The **IF** ... **THEN** around the declaration of its locals violates **13.3.3** (2), and the copy of **B** left on the return stack before declaring locals violates **13.3.3** (3). **JANE** in (2.5) violates **13.3.3** (4) by accessing locals after placing the sum of **A** and **B** on the return stack without first removing that sum.

(2.6)
```
: CHRIS ( a b) { B A } ['] A EXECUTE
  5 ['] B >BODY ! [ ' A ] LITERAL LEE ;
```

CHRIS in (2.6) illustrates three violations of **13.3.3** (5). The attempt to **EXECUTE** the local called **A** is inconsistent with some implementations. The store into **B** via **>BODY** is likely to cause tragic results with many implementations; moreover, if locals are in registers they can't be addressed as memory no matter what is written.

The third violation, in which an execution token for a definition's local is passed as an argument to the word **LEE** would, if allowed, have the unpleasant implication that **LEE** could **EXECUTE** the token and obtain a value for **A** from the particular execution of **CHRIS** that called **LEE** this time.

A.13.3 Additional usage requirements

Rule **13.3.3** (4) could be relaxed without affecting the integrity of the rest of this structure. **13.3.3** (3) could not be.

13.3.3 (2) forbids the use of the data stack for local storage because no usage rules have been articulated for programmer users in such a case. Of course, if the data stack is somehow employed in such a way that there are no usage rules, then the locals are invisible to the programmer, are logically not on the stack, and the implementation conforms.

The minimum required number of locals can (and should) be adjusted to minimize the cost of compliance for existing users of locals.

Access to previously declared local variables is prohibited by Section **13.3.3**(4) until any data placed onto the return stack by the application has been removed, due to the possible use of the return stack for storage of locals.

Authorization for a Standard Program to manipulate the return stack (e.g., via **>R R>**) while local variables are active overly constrains implementation possibilities. The consensus of users of locals was that Local facilities represent an effective functional replacement for return stack manipulation, and restriction of standard usage to only one method was reasonable.

Access to Locals within **DO..LOOP**s is expressly permitted as an additional requirement of conforming systems by Section **13.3.3**(7). Although words, such as **(LOCALS)**, written by a System Implementor, may require inside knowledge of the internal structure of the return stack, such knowledge is not required of a user of compliant Forth systems.

A.13.6 Glossary

A.13.6.1.2295 TO

Typical use: **x TO name**

See: **A.6.2.2295 TO**.

A.13.6.2.1795 LOCALS|

A possible implementation of this word and an example of usage is given in **A.13**. It is intended as an example only; any implementation yielding the described semantics is acceptable.

A.14 The Memory-Allocation word set

The Memory-Allocation word set provides a means for acquiring memory other than the contiguous data space that is allocated by **ALLOT**. In many operating system environments it is inappropriate for a process to pre-allocate large amounts of contiguous memory (as would be necessary for the use of **ALLOT**). The Memory-Allocation word set can acquire memory from the system at any time, without knowing in advance the address of the memory that will be acquired.

A.15 The Programming-Tools word set

These words have been in widespread common use since the earliest Forth systems.

Although there are environmental dependencies intrinsic to programs using an assembler, virtually all Forth systems provide such a capability. Insofar as many Forth programs are intended for real-time applications and are intrinsically non-portable for this reason, the Technical Committee believes that providing a standard window into assemblers is a useful contribution to Forth programmers.

Similarly, the programming aids **DUMP**, etc., are valuable tools even though their specific formats will differ between CPUs and Forth implementations. These words are primarily intended for use by the programmer, and are rarely invoked in programs.

One of the original aims of Forth was to erase the boundary between “user” and “programmer” -- to give all possible power to anyone who had occasion to use a computer. Nothing in the above labeling or remarks should be construed to mean that this goal has been abandoned.

A.15.6 Glossary

A.15.6.1.0220 .S

.S is a debugging convenience found on almost all Forth systems. It is universally mentioned in Forth texts.

A.15.6.1.2194 SEE

SEE acts as an on-line form of documentation of words, allowing modification of words by decompiling and regenerating with appropriate changes.

A.15.6.1.2465 WORDS

WORDS is a debugging convenience found on almost all Forth systems. It is universally referred to in Forth texts.

A.15.6.2.0470 ;CODE

Typical use: **: namex ... <create> ... ;CODE ...**

where **namex** is a defining word, and **<create>** is **CREATE** or any user defined word that calls **CREATE**.

A.15.6.2.0930 CODE

Some Forth systems implement the assembly function by adding an **ASSEMBLER** word list to the search order, using the text interpreter to parse a postfix assembly language with lexical characteristics similar to Forth source code. Typically, in such systems, assembly ends when a word **END-CODE** is interpreted.

A.15.6.2.1015 CS-PICK

The intent is to reiterate a *dest* on the control-flow stack so that it can be resolved more than once. For example:

```
\ Conditionally transfer control to beginning of loop
\ This is similar in spirit to C's "continue" statement.
: ?REPEAT  ( dest -- dest ) \ Compilation
           ( flag -- )      \ Execution
   0 CS-PICK   POSTPONE UNTIL ;  IMMEDIATE
: XX  ( -- ) \ Example use of ?REPEAT
   BEGIN
     ...
   flag ?REPEAT  ( Go back to BEGIN if flag is false )
     ...
   flag ?REPEAT  ( Go back to BEGIN if flag is false )
     ...
   flag UNTIL    ( Go back to BEGIN if flag is false )
   ...  ;
```

A.15.6.2.1020 CS-ROLL

The intent is to modify the order in which the *origs* and *dests* on the control-flow stack are to be resolved by subsequent control-flow words. For example, **WHILE** could be implemented in terms of **IF** and **CS-ROLL**, as follows:

```
: WHILE  ( dest -- orig dest )
   POSTPONE IF  1 CS-ROLL  ; IMMEDIATE
```

A.15.6.2.1580 FORGET

Typical use: `... FORGET name ...`

FORGET assumes that all the information needed to restore the dictionary to its previous state is inferable somehow from the forgotten word. While this may be true in simple linear dictionary models, it is difficult to implement in other Forth systems; e.g., those with multiple address spaces. For example, if Forth is embedded in ROM, how does **FORGET** know how much RAM to recover when an array is forgotten? A general and preferred solution is provided by **MARKER**.

A.15.6.2.2531 [ELSE]

Typical use: `... flag [IF] ... [ELSE] ... [THEN] ...`

A.15.6.2.2532 [IF]

Typical use: `... flag [IF] ... [ELSE] ... [THEN] ...`

A.15.6.2.2533 [THEN]

Typical use: `... flag [IF] ... [ELSE] ... [THEN] ...`

Software that runs in several system environments often contains some source code that is environmentally dependent. Conditional compilation -- the selective inclusion or exclusion of portions of the source code at compile time -- is one technique that is often used to assist in the maintenance of such source code.

Conditional compilation is sometimes done with “smart comments” -- definitions that either skip or do not skip the remainder of the line based on some test. For example:

```
\ If 16-Bit? contains TRUE, lines preceded by 16BIT\
\ will be skipped.  Otherwise, they will not be skipped.
VARIABLE 16-BIT?
: 16BIT\   16-BIT? @  IF  POSTPONE \  THEN ;  IMMEDIATE
```

This technique works on a line by line basis, and is good for short, isolated variant code sequences.

More complicated conditional compilation problems suggest a nestable method that can encompass more than one source line at a time. The words included in the ANS Forth Programmers-Tool-kit extensions word set are useful for this purpose. The implementation given below works with any input source (keyboard, **EVALUATE**, **BLOCK**, or text file).

```
: [ELSE]  ( -- )
   1 BEGIN                                \ level
     BEGIN  BL WORD COUNT  DUP  WHILE     \ level adr len
       2DUP  S" [IF]"  COMPARE 0= IF      \ level adr len
         2DROP 1+                         \ level'
       ELSE                               \ level adr len
         2DUP  S" [ELSE]"  COMPARE 0= IF  \ level adr len
            2DROP 1- DUP IF 1+ THEN       \ level'
         ELSE                             \ level adr len
           S" [THEN]"  COMPARE 0= IF      \ level
             1-                           \ level'
           THEN
         THEN
       THEN ?DUP 0=  IF EXIT THEN         \ level'
     REPEAT  2DROP                        \ level
   REFILL 0= UNTIL                        \ level
   DROP ;  IMMEDIATE
: [IF]  ( flag -- )
   0= IF POSTPONE [ELSE] THEN ;  IMMEDIATE
: [THEN]  ( -- )  ;  IMMEDIATE
```

A.16 The Search-Order word set

Search-order specification and control mechanisms vary widely. The FIG-Forth, Forth-79, polyFORTH, and Forth-83 vocabulary and search order mechanisms are all mutually incompatible. The complete list of incompatible mechanisms, in use or proposed, is much longer. The **ALSO/ONLY** scheme described in a Forth-83 Experimental Proposal has substantial community support. However, many consider it to be fundamentally flawed, and oppose it vigorously.

Recognizing this variation, this Standard specifies a new "primitive" set of tools from which various schemes may be constructed. This primitive search-order word set is intended to be a portable "construction set" from which search-order words may be built, rather than a user interface. **ALSO/ONLY** or the various "vocabulary" schemes supported by the major Forth vendors can be defined in terms of the primitive search-order word set.

The encoding for word list identifiers *wid* might be a small-integer index into an array of word-list definition records, the data-space address of such a record, a user-area offset, the execution token of a Forth-83 style sealed vocabulary, the link-field address of the first definition in a word list, or anything else. It is entirely up to the system implementor.

In some systems the interpretation of numeric literals is controlled by including "pseudo word lists" that recognize numbers at the end of the search order. This technique is accommodated by the "default search order" behavior of **SET-ORDER** when given an argument of -1. In a system using the traditional implementation of **ALSO/ONLY** , the minimum search order would be equivalent to the word **ONLY**.

There has never been a portable way to restore a saved search order. F83 (not Forth 83) introduced the word **PREVIOUS** , which almost made it possible to "unload" the search order by repeatedly executing the phrase **CONTEXT @ PREVIOUS**. The search order could be "reloaded" by repeating **ALSO CONTEXT !**. Unfortunately there was no portable way to determine how many word lists were in the search order.

ANS Forth has removed the word **CONTEXT** because in many systems its contents refer to more than one word list, compounding portability problems.

Note that **:** (colon) no longer affects the search order. The previous behavior, where the compilation word list replaces the first word list of the search order, can be emulated with the following redefinition of **:** (colon).

```
: : GET-ORDER SWAP DROP GET-CURRENT SWAP SET-ORDER : ;
```

A.16.2 Additional terms

search order

Note that the use of the term "list" does not necessarily imply implementation as a linked list.

A.16.3.3 Finding definition names

In other words, the following is not guaranteed to work:

```
: FOO  ... [ ... SET-CURRENT ] ... RECURSE ... ;
IMMEDIATE
```

RECURSE, **;** (semicolon), and **IMMEDIATE** may or may not need information stored in the compilation word list.

A.16.6 Glossary

A.16.6.1.2192 SEARCH-WORDLIST

The string argument to **SEARCH-WORDLIST** is represented by *c-addr u*, rather than by just *c-addr* as with **FIND**. The committee wishes to establish *c-addr u* as the preferred representation of a string on the stack, and has adopted that representation for all new functions that accept string arguments. While this decision may cause the implementation of **SEARCH-WORDLIST** to be somewhat more difficult in existing systems, the committee feels that the additional difficulty is minor.

When **SEARCH-WORDLIST** fails to find the word, it does not return the string, as does **FIND**. This is in accordance with the general principle that Forth words consume their arguments.

A.16.6.2.0715 ALSO

Here is an implementation of **ALSO/ONLY** in terms of the primitive search-order word set.

```
WORDLIST CONSTANT ROOT   ROOT SET-CURRENT
: DO-VOCABULARY  ( -- ) \ Implementation factor
  DOES>  @ >R           (  ) ( R: widnew )
   GET-ORDER  SWAP DROP ( wid1 ... widn-1 n )
   R> SWAP SET-ORDER ;
: DISCARD  ( x1 .. xu u - ) \ Implementation factor
   0 ?DO DROP LOOP ;        \ DROP u+1 stack items
CREATE FORTH  FORTH-WORDLIST , DO-VOCABULARY
: VOCABULARY  ( name -- )
   WORDLIST CREATE ,  DO-VOCABULARY ;
: ALSO  ( -- )  GET-ORDER  OVER SWAP 1+ SET-ORDER ;
```

```
: PREVIOUS  ( --  )  GET-ORDER
   SWAP DROP 1- SET-ORDER ;
: DEFINITIONS  ( -- )  GET-ORDER
   OVER SET-CURRENT DISCARD ;
: ONLY ( -- )  ROOT ROOT  2 SET-ORDER ;
\ Forth-83 version; just removes ONLY
: SEAL  ( -- )  GET-ORDER 1- SET-ORDER DROP ;
\ F83 and F-PC version; leaves only CONTEXT
: SEAL  ( -- )  GET-ORDER OVER 1 SET-ORDER DISCARD ;
```

The preceding definition of **ONLY** in terms of a "**ROOT**" word list follows F83 usage, and assumes that the default search order just includes **ROOT** and **FORTH**. A more portable definition of **FORTH** and **ONLY**, without the assumptions, is:

```
<omit the  ... WORDLIST CONSTANT ROOT ... line>
CREATE FORTH GET-ORDER OVER , DISCARD DO-VOCABULARY
: ONLY  ( -- )  -1 SET-ORDER ;
```

Here is a simple implementation of **GET-ORDER** and **SET-ORDER**, including a corresponding definition of **FIND**. The implementations of **WORDLIST**, **SEARCH-WORDLIST**, **GET-CURRENT** and **SET-CURRENT** depend on system details and are not given here.

```
16 CONSTANT #VOCS
VARIABLE #ORDER
CREATE CONTEXT  #VOCS CELLS ALLOT
: GET-ORDER  ( -- wid1 .. widn n )
   #ORDER @ 0 ?DO
     #ORDER @  I - 1- CELLS CONTEXT + @
   LOOP
   #ORDER @ ;
: SET-ORDER  ( wid1 .. widn n -- )
   DUP -1 = IF
     DROP  <push system default word lists and n>
   THEN
   DUP #ORDER !
   0 ?DO  I CELLS CONTEXT + ! LOOP ;
: FIND  ( c-addr -- c-addr 0 | w 1 | w -1 )
   0                      ( c-addr 0 )
   #ORDER @ 0 ?DO
     OVER COUNT           ( c-addr 0 c-addr' u )
     I CELLS CONTEXT + @  ( c-addr 0 c-addr' u wid)
     SEARCH-WORDLIST      ( c-addr 0; 0 | w 1 | w -1 )
     ?DUP IF              ( c-addr 0; w 1 | w -1 )
       2SWAP 2DROP LEAVE  ( w 1 | w -1 )
     THEN                 ( c-addr 0 )
   LOOP ;                 ( c-addr 0 | w 1 | w -1 )
```

In an implementation where the dictionary search mechanism uses a hash table or lookup cache to reduce the search time, **SET-ORDER** might need to reconstruct the hash table or flush the cache.

A.17 The String word set

A.17.6 Glossary

A.17.6.1.0245 /STRING

/STRING is used to remove or add characters relative to the "left" end of the character string. Positive values of *n* will exclude characters from the string while negative values of *n* will include characters to the left of the string. **/STRING** is a natural factor of **WORD** and commonly available.

A.17.6.1.0910 CMOVE

If *c-addr*$_2$ lies within the source region, memory propagation occurs. (*c-addr*$_2$ lies within the source region if *c-addr*$_2$ is not less than *c-addr*$_1$ and *c-addr*$_2$ is less than the quantity *c-addr*$_1$ *u* **CHARS +**).

Typical use: Assume a character string at address 100: "ABCD". Then after
100 DUP CHAR+ 3 CMOVE the string at address 100 is "AAAA".

Rationale for **CMOVE** and **CMOVE>** follows **MOVE**.

A.17.6.1.0920 CMOVE>

If *c-addr*$_1$ lies within the destination region, memory propagation occurs. (*c-addr*$_1$ lies within the destination region if *c-addr*$_1$ is greater than or equal to *c-addr*$_2$ and if *c-addr*$_2$ is less than the quantity *c-addr*$_1$ *u* **CHARS +**.)

Typical use: Assume a character string at address 100: "ABCD". Then after
100 DUP CHAR+ SWAP 3 CMOVE> the string at address 100 is "DDDD".

A.17.6.1.0935 COMPARE

Existing Forth systems perform string comparison operations using words that differ in spelling, input and output arguments, and case sensitivity. One in widespread use was chosen.

A.17.6.1.2191 SEARCH

Existing Forth systems perform string searching operations using words that differ in spelling, input and output arguments, and case sensitivity. One in widespread use was chosen.

A.17.6.1.2212 SLITERAL

The current functionality of **6.1.2165 S"** may be provided by the following definition:

```
: S" ( "ccc<quote>" -- )
   [CHAR] " PARSE   POSTPONE SLITERAL ; IMMEDIATE
```