 Table of Contents

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.

a) 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.

b) 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.

c) 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

d) 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.

a) 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.

b) 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

c) 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

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

a) Storage and retrieval

Two operators are provided to fetch and store cell pairs:

2@ 2!

b) 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

c) 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 2n+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 36 through 71, respectively.

A.3.2.2 Arithmetic

A.3.2.2.1 Integer 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 VARIABLEs and CONSTANTs.

A.3.2.3.2 Control-flow stack

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

*

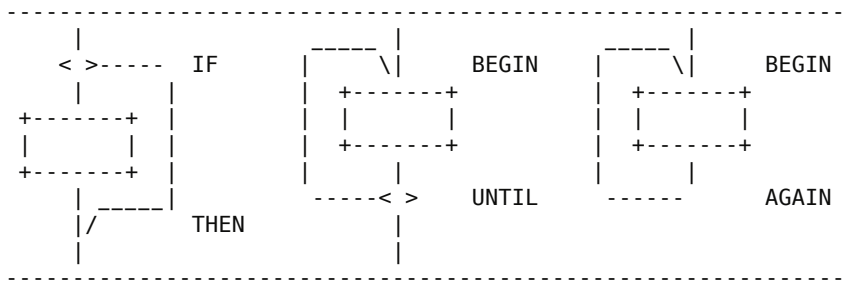


Figure A.1 - The basic control-flow patterns.

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.

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.

*

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
   ---------------------------------------------------------------------
```

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
```

*

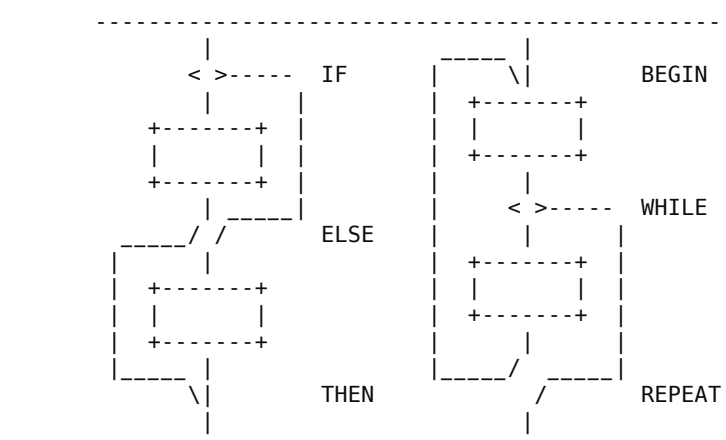


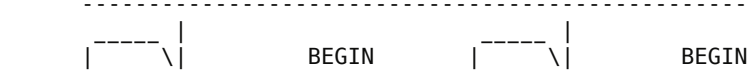
Figure A.2 - Additional basic control-flow patterns.

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 WHILEs 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.

*



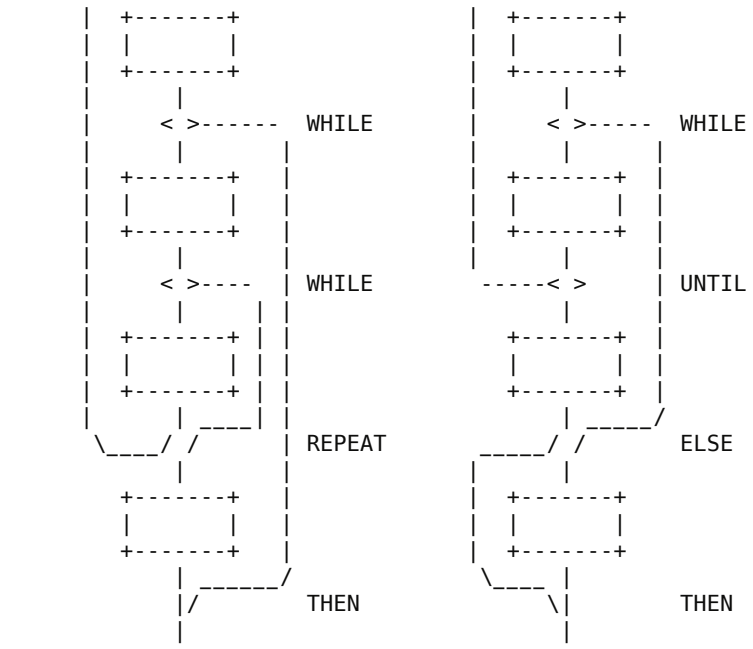


Figure A.3 - Extended control-flow pattern examples.

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 WHILEs 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 ALIGNED`**.

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.

 Table of Contents
 Next Section