Forth 200*x* Standardisation Committee

Forth 2012

10th November, 2014

Notice: *Status of this Document*
This is a draft proposed Standard to replace ANSI X3.215-1994. As such, this is not a completed standard.
The Standardisation Committee may modify this document during the course of its work.

Contents

Foreword

Forth is a language for direct communication between human beings and machines. Forth was invented by Charles Moore to increase programmer productivity without sacrificing machine efficiency. Using natural-language diction and machine-oriented syntax, Forth provides an economical, productive environment for interactive compilation and execution of programs. Forth also provides low-level access to computer-controlled hardware, and the ability to extend the language itself. This extensibility allows the language to be quickly expanded and adapted to special needs and different hardware systems. Forth provides for highly interactive program development and testing.

In the interests of transportability of application software written in Forth, standardization efforts began in the mid-1970s by an international group of users and implementors who adopted the name "Forth Standards Team". This effort resulted in the Forth-77 Standard. As the language continued to evolve, an interim Forth-78 Standard was published by the Forth Standards Team. Following Forth Standards Team meetings in 1979, the Forth-79 Standard was published in 1980. Major changes were made by the Forth Standards Team in the Forth-83 Standard, which was published in 1983.

The ANS Forth committee was formed in 1987 to address the fragmentation within the Forth community caused not only by the difference between Forth 79 and Forth 83 but the exploitation of technical developments. Undertaking a comprehensive review of existing implementations they moved away from prescribing stringent requirements, preferring to describe the operation of the virtual machine, without reference to an implementation. The ANS Forth Standard was published in 1994[1] and was adopted as an international standard in 1997[2].

The Forth 200*x* Standardisation Committee was formed in 2004 to allow the Forth community to contribute to an updated standard. Changes are proposed and discussed in the electronic media: the usenet news group `comp.lang.forth`; the `forth200x@yahoogroups.com` email list; the `www.forth200x.org` web site. Annual public meeting are held to review and vote on the proposed changes.

This document is the result of the public review meetings first held on October 21–22, 2005 (Santander) and subsequently on September 14–15, 2006 (Cambridge), September 13–14, 2007 (Dagstuhl), September 25–26, 2008 (Vienna), March 25–27, 2009 (Neuenkirchen, Rheine), September 2–4, 2009 (Exeter), March 24–26, 2010 (Rostock), September 22–24, 2010 (Hamburg), September 21–23, 2011 (Vienna), September 12–14, 2012 (Oxford), September 25–27, 2013 (Hamburg).

---

[1] ANSI X3.215–1994 Information Systems — Programming Language FORTH
[2] ISO/IEC 15145:1997 Information technology. Programming languages. FORTH

Proposals Process

In developing a standard it is necessary for the standards committee to know what the system implementors and the programmers are already doing in that area, and what they would be willing to do, or wish for.

To that end we have introduced a system of consultation with the Forth community:

a) A proponent of an extension or change to the standard writes a proposal.

b) The proponent publishes the proposal as an *RfD* (Request for Discussion) by sending a copy to the `forth200x@yahoogroups.com` email list *and* to the `comp.lang.forth` usenet news group where it can be discussed. The maintainers of the `www.forth200x.org` web site will then place a copy of the proposal on that web site.

   Be warned, this will generate a lot of heated discussion.

   In order for the results to be available in time for a standards meeting, an RfD should be published at least 12 weeks before the next meeting.

   If a proposal does not propose extensions or changes to the Forth language, but a rewording of the current document, there is nothing for a system implementor to implement, or a programmer to use. In such a case, the proposal should be published as a Request for Comment (RfC). The proposal will be considered, along with any comments, at the next committee meeting.

c) The proponent can modify the proposal, taking any comments into consideration. Where comments have been dismissed, both the comment and the reasons for its dismissal should be given. The revised proposal is published as a revised RfD/RfC.

d) Once a proposal has settled down, it is frozen, and submitted to a vote taker, who then publishes a *CfV* (Call for Votes) on the proposal. The vote taker will normally be a member of the standards committee. In the poll, system implementors can state, whether their systems implement the proposal, or what the chances are that it ever will. Similarly, programmers can state whether they have used something similar to the proposed extension and whether they would use the proposed extension once it is standardized. The results of this poll are used by the standards committee when deciding whether to accept the proposal into the standards document.

   In order for the results to be available in time for a standards meeting, the CfV should be started at least 6 weeks before that meeting.

e) One to two weeks after publishing the CfV, the vote taker will publish a *Current Standings*. Note that the poll will remain open, especially for information on additional systems, and the results will be updated on the Forth200*x* web page. The results considered at a standards meeting are those from four weeks prior to that meeting. If no poll results are available by that deadline, the proposal will be considered at a later meeting.

f) A proposal will only be accepted into the new basis document by consensus of those present at a standards meeting. If you can not attend a meeting, you should ask somebody who is attending to champion the proposal on your behalf.

Should a contributor consider their comments to have been dismissed without due consideration, they are encouraged to submit a counter proposal.

Proposals which have passed the poll will be integrated into the basis document in preparation for the approaching standards committee meeting. Proposals often require some rewording in this process, so the proponent should work with the editor to integrate the proposal into the document.

A proposal should give a rationale for the proposal, so that system implementors and programmers may see the relevance of the proposal and why they should adopt (and vote for) it. The proposal should include the following sections, where appropriate.

Author:
> The name of the author(s) of the proposal.

Change Log:
> A list of changes to the last published edition on the proposal.

Problem:
> This states what problem the proposal addresses.

Solution:
> An informal description of the proposed solution to the problem identified by the proposal.

Typical use:
> Shows a typical use of the word/feature you propose; this should make the formal wording easier to understand.

Remarks:
> This gives the rationale for specific decisions you have taken in the proposal (often in response to comments in the RfD phase), or discusses specific issues that have not been decided yet.

Proposal:
> This is the formal or normative part of the proposal and should be as well specified as possible.
>
> Some issues could be left undecided in the initial RfDs, leaving the issue open for discussion. These issues should be mentioned in the Remarks section as well as in the Proposal section.
>
> If you want to leave something open to the system implementor, make that explicit, e.g., by making it an ambiguous condition.
>
> For the wording of word definitions, it is normally a good idea to take your inspiration from existing word definitions in the basis document. Where possible you should include the rationale for the definition. Should a proposal be accepted where no rationale has been provided, the editor will construct a rationale from other parts of the proposal. The proponent should work with the editor in the development of this rationale.

Reference implementation:
> This makes it easier for system implementors to adopt your proposal. Where possible they should be provided in standard Forth, as defined by this document. Where this is not possible, system specific knowledge is required or non standard words are used, this should be documented.

Testing:
> This should test the feature/words you propose, in particular, it should test boundary conditions. Where possible test cases should be written to conform with John Hayes tester.f test harness.

Experience:

Indicate where the proposal has already been implemented and/or used.

Comments:

Initially this is blank. As comments are made on the proposal, they should be incorporated into the proposal. Comment which can not be incorporated should be included in this section. A response to the comment may be included after the comment itself.

Instructions for responding to the poll:

Once the proposal enters the CfV stage, the vote taker will add these instructions to the proposal.

200*x* Membership

This document is maintained by the Forth 200*x* Standards Committee. The committee meetings are open to the public, anybody is allowed to join the committee in its deliberations.

Membership of the committee is open to anybody who can attend. On attending a meeting of any kind a non-member becomes an observing member (observer). If they attend the next voting meeting, they will become a voting member of the committee, otherwise they revert to non-member status. An observer will not normally be allowed to vote, but may be allowed at the discretion of the committee. A member will be deemed to have resigned from the committee if they fail to attend two consecutive voting meetings.

Currently the committee has the following voting members:

Dr. M. Anton Ertl (Chair) . . . . Technische Universität Wien
`anton@mips.complang.tuwien.at`   Wien, Austria

Dr. Peter Knaggs (Editor) . . . . Independent Member
`pjk@bcs.org.uk`   Trowbridge, UK

Willem Botha . . . . Construction Computer Software (Pty) Ltd
`willem.botha@ccssa.com`   Cape Town, South Africa

Andrew Haley . . . . Red Hat UK Ltd.
`aph@redhat.com`   Cambridge, UK

Dr. Ulrich Hoffmann . . . . FH Wedel
`uho@forth-ev.de`   Wedel, Germany

Simon Kaphahn . . . . Independent Member
`Simon_K99@gmx.de`   Munich, Germany

Bernd Paysan Net2o
`bernd.paysan@gmx.de`   Munich, Germany

Stephen Pelc . . . . MicroProcessor Engineering Ltd.
`stephen@mpeforth.com`   Southampton, UK

Dr. Willi Stricker . . . . Independent Member
`stricker_w@t-online.de`   Springe, Germany

Leon Wagner . . . . FORTH, Inc.
`leon@forth.com`   Los Angeles, USA

Gerald Wodni . . . . Independent Member
`gerald@wodni.at`   Wien, Austria

The following organizations and individuals have also participated in this project as committee members. The committee recognizes and respects their contributions:

Federico de Ceballos . . . . Universidad de Cantabria
`federico.ceballos@unican.es`   Santander, Spain

Dr. Bill Stoddart. . . . .   Teesside University
w.j.stoddart@gmail.com   Middlesbrough, UK

Carsten Strotmann. . . . .   Independent Member
carsten@strotmann.de   Neuenkirchen, Germany

The committee would like to thank the following contributors:

Forth 2012 Standard

1   Introduction

1.1   Purpose

The purpose of this standard is to promote the portability of Forth programs for use on a wide variety of computing systems, to facilitate the communication of programs, programming techniques, and ideas among Forth programmers, and to serve as a basis for the future evolution of the Forth language.

1.2   Scope

This standard specifies an interface between a Forth System and a Forth Program by defining the words provided by a Standard System.

1.2.1   Inclusions

This standard specifies:

- the forms that a program written in the Forth language may take;

- the rules for interpreting the meaning of a program and its data.

1.2.2   Exclusions

This standard does not specify:

- the mechanism by which programs are transformed for use on computing systems;

- the operations required for setup and control of the use of programs on computing systems;

- the method of transcription of programs or their input or output data to or from a storage medium;

- the program and Forth system behavior when the rules of this standard fail to establish an interpretation;

- the size or complexity of a program and its data that will exceed the capacity of any specific computing system or the capability of a particular Forth system;

- the physical properties of input/output records, files, and units;

- the physical properties and implementation of storage.

1.3   Document organization

1.3.1   Word sets

This standard groups Forth words and capabilities into *word sets* under a name indicating some shared aspect, typically their common functional area. Each word set may have an extension, containing words that offer additional functionality. These words are not required in an implementation of the word set.

The "Core" word set, defined in sections 1 through 6, contains the required words and capabilities of a Standard System. The other word sets, defined in sections 7 through 18, are optional, making it possible to provide Standard Systems with tailored levels of functionality.

1.3.1.1   Text sections

Within each word set, section 1 contains introductory and explanatory material and section 2 introduces terms and notation used throughout the standard. There are no requirements in these sections.

Sections 3 and 4 contain the usage and documentation requirements, respectively, for Standard Systems and Programs, while section 5 specifies their labeling.

Sections x.1–x.6 of each word set have the same section numbering as sections 1–6 of the whole document to make it easy to relate the sections to each other. This may lead to gaps in section numbers if a particular section does not occur in a word set.

1.3.1.2   Glossary sections

Section 6 of each word set specifies the required behavior of the definitions in the word set and the extensions word set.

1.3.2   Annexes

The annexes do not contain any required material.

Annex A provides some of the rationale behind the committee's decisions in creating this standard, as well as implementation examples. It has the same section numbering as the body of the standard to make it easy to relate each requirements section to its rationale section.

Annex B is a short bibliography on Forth.

Annex C discusses the compatibility of this standard with earlier Forths.

Annex D presents some techniques for writing portable programs.

Annex E is an index of all Forth words defined in this standard.

1.4   Future directions

1.4.1   New technology

This standard adopts certain words and practices that are increasingly found in common practice. New words have also been adopted to ease creation of portable programs.

1.4.2   Obsolescent features

This standard adopts certain words and practices that cause some previously used words and practices to become obsolescent. Although retained here because of their widespread use, their use in new implementations or new programs is discouraged, as they may be withdrawn from future revisions of the standard.

This standard designates the following word as obsolescent:

15.6.2.1580   **FORGET**
   6.2.2530   **[COMPILE]**
13.6.2.1795   **LOCALS|**

This standard designates the following practice as obsolescent:

– Using **ENVIRONMENT?** to enquire whether a word set is present.

2   Terms, notation, and references

The phrase "See:" is used throughout this standard to direct the reader to other sections of the standard that have a direct bearing on the current section.

In this standard, "shall" states a requirement on a system or program; conversely, "shall not" is a prohibition; "need not" means "is not required to"; "should" describes a recommendation of the standard; and "may", depending on context, means "is allowed to" or "might happen".

Throughout the standard, typefaces are used in the following manner:

   – This proportional serif typeface is used for text, with *italic* used for symbols and the first appearance of new terms;

   – A bold proportional sans-serif typeface is used for **headings**;

   – A bold monospaced serif typeface is used for Forth-language **`text`**.

2.1   Definitions of terms

Terms defined in this section are used generally throughout this standard. Additional terms specific to individual word sets are defined in those word sets. Other terms are defined at their first appearance, indicated by italic type. Terms not defined in this standard are to be construed according to the *Dictionary for Information Systems,* ANSI X3.172-1990.

address unit: Depending on context, either 1) the units into which a Forth address space is divided for the purposes of locating data objects such as characters and variables; 2) the physical memory storage elements corresponding to those units; 3) the contents of such a memory storage element; or 4) the units in which the length of a region of memory is expressed.

aligned: Divisible by a type-dependent power of 2 (typically used as "⟨*type*⟩-aligned address" or "⟨*type*⟩-aligned value").

aligned address: The address of a memory location at which a character, cell, cell pair, or double-cell integer can be accessed.

ambiguous condition: A circumstance for which this standard does not prescribe a specific behavior. See section **4.1.2 Ambiguous conditions** for a list of such circumstances and **3.4.4 Possible actions on an ambiguous condition**.

cell: The primary unit of information in the architecture of a Forth system.

cell pair: Two cells that are treated as a single unit.

character: Depending on context, either 1) a storage unit capable of holding a character; or 2) a member of a character set.

character-aligned address: The address of a memory location at which a character can be accessed.

character string: Data space that is associated with a sequence of consecutive character-aligned addresses. Character strings usually contain text. Unless otherwise indicated, the term "string" means "character string".

code space: The logical area of the dictionary in which word semantics are implemented.

compile: To transform source code into dictionary definitions.

compilation semantics: The behavior of a Forth definition when its name is encountered by the text interpreter in compilation state.

counted string: A data structure consisting of one character containing a length followed by zero or more contiguous data characters. Normally, counted strings contain text.

cross compiler: A system that compiles a program for later execution in an environment that may be physically and logically different from the compiling environment. In a cross compiler, the term "host" applies to the compiling environment, and the term "target" applies to the run-time environment.

current definition: The definition whose compilation has been started but not yet ended.

data field: The data space associated with a word defined via **CREATE**.

data space: The logical area of the dictionary that can be accessed.

data-space pointer: The address of the next available data space location, i.e., the value returned by **HERE**.

data stack: A stack that may be used for passing parameters between definitions. When there is no possibility of confusion, the data stack is referred to as "the stack". Contrast with **return stack**.

data type: An identifier for the set of values that a data object may have.

defining word: A Forth word that creates a new definition when executed.

definition: A Forth execution procedure compiled into the dictionary.

dictionary: An extensible structure that contains definitions and associated data space.

display: To send one or more characters to the user output device.

environmental dependencies: A program's implicit assumptions about a Forth system's implementation options or underlying hardware. For example, a program that assumes a cell size greater than 16 bits is said to have an environmental dependency.

execution semantics: The behavior of a Forth definition when it is executed.

execution token: A value that identifies the execution semantics of a definition.

find: To search the dictionary for a definition name matching a given string.

immediate word: A Forth word whose compilation semantics are to perform its execution semantics.

implementation defined: Denotes system behaviors or features that must be provided and documented by a system but whose further details are not prescribed by this standard.

implementation dependent: Denotes system behaviors or features that must be provided by a system but whose further details are not prescribed by this standard.

initiation semantics: Describes the behavior at the start of some word definitions (those of words defined with **:**, **:NONAME**, **CREATE DOES>**). Other parts of the specification of these defining words (and nothing else) refer to initiation semantics.

input buffer: A region of memory containing the sequence of characters from the input source that is currently accessible to a program.

input source: The device, file, block, or other entity that supplies characters to refill the input buffer.

input source specification: A set of information describing a particular state of the input source, input buffer, and parse area. This information is sufficient, when saved and restored properly, to enable the nesting of parsing operations on the same or different input sources.

interpretation semantics: The behavior of a Forth definition when its name is encountered by the text interpreter in interpretation state.

keyboard event: A value received by the system denoting a user action at the user input device. The term "keyboard" in this document does not exclude other types of user input devices.

line: A sequence of characters followed by an actual or implied line terminator.

name space: The logical area of the dictionary in which definition names are stored.

number: In this standard, "number" used without other qualification means "integer". Similarly, "double number" means "double-cell integer".

parse: To select and exclude a character string from the parse area using a specified set of delimiting characters, called delimiters.

parse area: The portion of the input buffer that has not yet been parsed, and is thus available to the system for subsequent processing by the text interpreter and other parsing operations.

pictured-numeric output: A number display format in which the number is converted using Forth words that resemble a symbolic "picture" of the desired output.

program: A complete specification of execution to achieve a specific function (application task) expressed in Forth source code form.

receive: To obtain characters from the user input device.

return stack: A stack that may be used for program execution nesting, do-loop execution, temporary storage, and other purposes.

standard word: A named Forth procedure, formally specified in this standard.

user input device: The input device currently selected as the source of received data, typically a keyboard.

user output device: The output device currently selected as the destination of display data.

variable: A named region of data space located and accessed by its memory address.

word: Depending on context, either 1) the name of a Forth definition; or 2) a parsed sequence of non-space characters, which could be the name of a Forth definition.

word list: A list of associated Forth definition names that may be examined during a dictionary search.

word set: A set of Forth definitions grouped together in this standard under a name indicating some shared aspect, typically their common functional area.

2.2   Notation

2.2.1   Numeric notation

Unless otherwise stated, all references to numbers apply to signed single-cell integers. The inclusive range of values is shown as {*from . . . to*}. The allowable range for the contents of an address is shown in double braces, particularly for the contents of variables, e.g., **BASE** {{2 . . . 36}}.

2.2.2   Stack notation

Stack parameters input to and output from a definition are described using the notation:

> (stack-id: *before* −− *after*)

where *stack-id* specifies which stack is being described, *before* represents the stack-parameter data types before execution of the definition and *after* represents them after execution. The symbols used in *before* and *after* are shown in table 3.1.

The control-flow-stack *stack-id* is "C:", the data-stack *stack-id* is "S:", and the return-stack *stack-id* is "R:". When there is no confusion, the data-stack *stack-id* may be omitted.

When there are alternate *after* representations, they are described by "*after$_1$* | *after$_2$*". The top of the stack is to the right. Only those stack items required for or provided by execution of the definition are shown.

2.2.3   Parsed-text notation

If, in addition to using stack parameters, a definition parses text, that text is specified by an abbreviation from table 2.1, shown surrounded by double-quotes and placed between the *before* parameters and the "--" separator in the first stack described, e.g.,

> (S: *before "parsed-text-abbreviation"* −− *after*)

Table 2.1: Parsed text abbreviations

Abbreviation	Description
⟨*char*⟩	the delimiting character marking the end of the string being parsed
⟨*chars*⟩	zero or more consecutive occurrences of the character ⟨*char*⟩
⟨*space*⟩	a delimiting space character
⟨*spaces*⟩	zero or more consecutive occurrences of the character ⟨*space*⟩
⟨*quote*⟩	a delimiting double quote
⟨*paren*⟩	a delimiting right parenthesis
⟨*eol*⟩	an implied delimiter marking the end of a line
ccc	a parsed sequence of arbitrary characters, excluding the delimiter character
name	a token delimited by space, equivalent to *ccc*⟨*space*⟩ or *ccc*⟨*eol*⟩

2.2.4   Glossary notation

The glossary entries for each word set are listed in the standard ASCII collating sequence. Each glossary entry specifies a Forth word and consists of two parts: an *index line* and the *semantic description* of the definition.

2.2.4.1   Glossary index line

The index line is a single-line entry containing, from left to right:

– Section number, the last four digits of which assign a unique sequential number to all words included in this standard;

– **DEFINITION-NAME** in upper-case, mono-spaced, bold-face letters;

– Natural-language pronunciation in quotes if it differs from English;

– Word-set designator from table 2.2. The designation for extensions word sets includes "EXT".

– Extension designator in sans-serif font under the Word-set designator for words which have been added to the standard via the named extension.

Table 2.2: Word set designators

Word set	Designator
Core word set	CORE
Block word set	BLOCK
Double-Number word set	DOUBLE
Exception word set	EXCEPTION
Facility word set	FACILITY
File-Access word set	FILE
Floating-Point word set	FLOATING
Locals word set	LOCALS
Memory-Allocation word set	MEMORY
Programming-Tools word set	TOOLS
Search-Order word set	SEARCH
String-Handling word set	STRING
Extended-Character word set	XCHAR

2.2.4.2   Glossary semantic description

The first paragraph of the semantic description contains a stack notation for each stack affected by execution of the word. The remaining paragraphs contain a text description of the semantics. See **3.4.3 Semantics**.

2.2.5   BNF notation

The following notation is used to define the syntax of some elements within the document:

– Each component of the element is defined with a rule consisting of the name of the component (italicized in angle-brackets, e.g., ⟨*decdigit*⟩), the characters := and a concatenation of tokens and metacharacters;

– Tokens may be literal characters (in bold face, e.g., **E**) or rule names in angle brackets (e.g., ⟨*decdigit*⟩);

– The metacharacter * is used to specify zero or more occurrences of the preceding token (e.g., ⟨*decdigit*⟩*);

- Tokens enclosed with [and] are optional (e.g., [**-**]);

- Vertical bars separate choices from a list of tokens enclosed with braces (e.g., { **0** | **1** }).

See: **3.4.1.3 Text interpreter input number conversion**, **12.3.7 Text interpreter input number conversion**, **12.6.1.0558** `>FLOAT`, **12.6.2.1613** `FS.`, **13.6.2.2550** `{:`.

2.3 References

The following national and international standards are referenced in this standard:

- ISO/IEC 15145:1997 *Information technology. Programming languages. FORTH.*

- ANSI X3.215-1994 *Programming Languages – Forth.*

- ANSI X3.172-1990 *Dictionary for Information Systems*, (**2.1 Definitions of terms**);

- ANSI X3.4-1974 *American Standard Code for Information Interchange* (ASCII), (**3.1.2.1 Graphic characters**);

- ISO 646-1983 *ISO 7-bit coded characterset for information interchange, International Reference Version* (IRV) (**3.1.2.1 Graphic characters**);

- ANSI/IEEE 754-1985 *Floating-point Standard*, (**12.2.1 Definition of terms**).

3 Usage requirements

A system shall provide all of the words defined in **6.1 Core words**. It may also provide any words defined in the optional word sets and extensions word sets. No standard word provided by a system shall alter the system state in a way that changes the effect of execution of any other standard word except as provided in this standard. A system may contain non-standard extensions, provided that they are consistent with the requirements of this standard.

The implementation of a system may use words and techniques outside the scope of this standard.

A system need not provide all words in executable form. The implementation may provide definitions, including definitions of words in the Core word set, in source form only. If so, the mechanism for adding the definitions to the dictionary is implementation defined.

A program that requires a system to provide words or techniques not defined in this standard has an environmental dependency.

3.1 Data types

A data type identifies the set of permissible values for a data object. It is not a property of a particular storage location or position on a stack. Moving a data object shall not affect its type.

No data-type checking is required of a system. An ambiguous condition exists if an incorrectly typed data object is encountered.

Table 3.1 summarizes the data types used throughout this standard. Multiple instances of the same type in the description of a definition are suffixed with a sequence digit subscript to distinguish them.

3.1.1 Data-type relationships

Some of the data types are subtypes of other data types. A data type i is a subtype of type j if and only if the members of i are a subset of the members of j. The following list represents the subtype relationships using the phrase "$i \Rightarrow j$" to denote "i is a subtype of j". The subtype relationship is transitive; if $i \Rightarrow j$ and $j \Rightarrow k$ then $i \Rightarrow k$:

$$+n \Rightarrow u \Rightarrow x;$$
$$+n \Rightarrow n \Rightarrow x;$$
$$char \Rightarrow +n;$$
$$a\text{-}addr \Rightarrow c\text{-}addr \Rightarrow addr \Rightarrow u;$$
$$flag \Rightarrow x;$$
$$xt \Rightarrow x;$$
$$ior \Rightarrow n \Rightarrow x;$$
$$+d \Rightarrow d \Rightarrow xd;$$
$$+d \Rightarrow ud \Rightarrow xd.$$

Any Forth definition that accepts an argument of type i shall also accept an argument that is a subtype of i.

Table 3.1: Data types

Symbol	Data type	Size on stack
flag	flag	1 cell
true	true flag	1 cell
false	false flag	1 cell
char	character	1 cell
n	signed number	1 cell
+n	non-negative number	1 cell
u	unsigned number	1 cell
u | n[1]	number	1 cell
x	unspecified cell	1 cell
xt	execution token	1 cell
addr	address	1 cell
a-addr	aligned address	1 cell
c-addr	character-aligned address	1 cell
ior	error result	1 cell
d	double-cell signed number	2 cells
+d	double-cell non-negative number	2 cells
ud	double-cell unsigned number	2 cells
d | ud[2]	double-cell number	2 cells
xd	unspecified cell pair	2 cells
colon-sys	definition compilation	implementation dependent
do-sys	do-loop structures	implementation dependent
case-sys	**CASE** structures	implementation dependent
of-sys	**OF** structures	implementation dependent
orig	control-flow origins	implementation dependent
dest	control-flow destinations	implementation dependent
loop-sys	loop-control parameters	implementation dependent
nest-sys	definition cells	implementation dependent
$i*x, j*x, k*x$[3]	any data type	0 or more cells

[1] May be either a signed number or an unsigned number depending on context.

[2] May be either a double-cell signed number or a double-cell unsigned number depending on context.

[3] May be an undetermined number of stack entries of unspecified type. For examples of use, see **6.1.1370** EXECUTE, **6.1.2050** QUIT.

3.1.2 Character types

Characters shall have the following properties:

  – be at least one address unit wide;
  – contain at least eight bits;
  – be of fixed width;
  – have a size less than or equal to cell size;
  – be unsigned.

The characters provided by a system shall include the graphic characters {32 … 126}, which represent graphic forms as shown in table 3.2.

3.1.2.1 Graphic characters

A graphic character is one that is normally displayed (e.g., A, #, &, 6). These values and graphics, shown in table 3.2, are taken directly from ANS X3.4-1974 (ASCII) and ISO 646-1983, International Reference Version (IRV). The graphic forms of characters outside the hex range {20 … 7E} are implementation defined. Programs that use the graphic hex 24 (the currency sign) have an environmental dependency.

The graphic representation of characters is not restricted to particular type fonts or styles. The graphics here are examples.

Table 3.2: Standard graphic characters

Hex	IRV	ASCII	Hex	IRV	ASCII	Hex	IRV	ASCII	Hex	IRV	ASCII	Hex	IRV	ASCII	Hex	IRV	ASCII
20			30	0	0	40	@	@	50	P	P	60	`	`	70	p	p
21	!	!	31	1	1	41	A	A	51	Q	Q	61	a	a	71	q	q
22	"	"	32	2	2	42	B	B	52	R	R	62	b	b	72	r	r
23	#	#	33	3	3	43	C	C	53	S	S	63	c	c	73	s	s
24	¤	$	34	4	4	44	D	D	54	T	T	63	d	d	74	t	t
25	%	%	35	5	5	45	E	E	55	U	U	64	e	e	75	u	u
26	&	&	36	6	6	46	F	F	56	V	V	65	f	f	76	v	v
27	'	'	37	7	7	47	G	G	57	W	W	66	g	g	77	w	w
28	(	(	38	8	8	48	H	H	58	X	X	67	h	h	78	x	x
29	)	)	39	9	9	49	I	I	59	Y	Y	68	i	i	79	y	y
2A	*	*	3A	:	:	4A	J	J	5A	Z	Z	69	j	j	7A	z	z
2B	+	+	3B	;	;	4B	K	K	5B	[	[	6A	k	k	7B	{	{
2C	,	,	3C	<	<	4C	L	L	5C	\	\	6C	l	l	7C				
2D	-	-	3D	=	=	4D	M	M	5D	]	]	6D	m	m	7D	}	}
2E	.	.	3E	>	>	4E	N	N	5E	^	^	6E	n	n	7E	~	~
2F	/	/	3F	?	?	4F	O	O	5F	_	_	6F	o	o			

3.1.2.2 Control characters

All non-graphic characters included in the implementation-defined character set are defined in this standard as control characters. In particular, the characters {0 … 31}, which could be included in the implementation-defined character set, are control characters.

Programs that require the ability to send or receive control characters have an environmental dependency.

3.1.2.3 Primitive Character

A primitive character (pchar) is a character with no restrictions on its contents. Unless otherwise stated, a "character" refers to a primitive character.

3.1.3 Single-cell types

The implementation-defined fixed size of a cell is specified in address units and the corresponding number of bits. See **D.2 Hardware peculiarities**.

Cells shall be at least one address unit wide and contain at least sixteen bits. The size of a cell shall be an integral multiple of the size of a character. Data-stack elements, return-stack elements, addresses, execution tokens, flags, and integers are one cell wide.

3.1.3.1 Flags

Flags may have one of two logical states, *true* or *false*. Programs that use flags as arithmetic operands have an environmental dependency. A true flag returned by a standard word shall be a single-cell value with all bits set. A false flag returned by a standard word shall be a single-cell value with all bits clear.

3.1.3.2 Integers

The implementation-defined range of signed integers shall include {-32767 ... +32767}. The implementation-defined range of non-negative integers shall include {0 ... 32767}. The implementation-defined range of unsigned integers shall include {0 ... 65535}.

3.1.3.3 Addresses

An address identifies a location in data space with a size of one address unit, which a program may fetch from or store into except for the restrictions established in this standard. The size of an address unit is specified in bits. Each distinct address value identifies exactly one such storage element. See **3.3.3 Data space**.

The set of character-aligned addresses, addresses at which a character can be accessed, is an implementation-defined subset of all addresses. Adding the size of a character to a character-aligned address shall produce another character-aligned address.

The set of aligned addresses is an implementation-defined subset of character-aligned addresses. Adding the size of a cell to an aligned address shall produce another aligned address.

3.1.3.4 Counted strings

A counted string in memory is identified by the address (*c-addr*) of its length character.

The length character of a counted string shall contain a binary representation of the number of data characters, between zero and the implementation-defined maximum length for a counted string. The maximum length of a counted string shall be at least 255.

3.1.3.5 Execution tokens

Different definitions may have the same execution token if the definitions are equivalent.

3.1.3.6 Error results

A value of zero indicates that the operation completed successfully; other values are in the range {-4095 ... -1} and represent a valid **THROW** code.

The meanings of values in the range {-255 ... -1} are defined by table **9.1 THROW code assignments**. Values in the range {-4095 ... -256} and their meanings are implementation defined.

A word that returns an *ior* will not **THROW** that *ior* as an exception, but indicates the exception through the *ior*. This allows a program to take appropriate actions, which may include throwing the exception.

3.1.4   Cell-pair types

A cell pair in memory consists of a sequence of two contiguous cells. The cell at the lower address is the first cell, and its address is used to identify the cell pair. Unless otherwise specified, a cell pair on a stack consists of the first cell immediately above the second cell.

3.1.4.1   Double-cell integers

On the stack, the cell containing the most significant part of a double-cell integer shall be above the cell containing the least significant part.

The implementation-defined range of double-cell signed integers shall include {-2147483647 ... +2147483647}.

The implementation-defined range of double-cell non-negative integers shall include {0 ... 2147483647}.

The implementation-defined range of double-cell unsigned integers shall include {0 ... 4294967295}. Placing the single-cell integer zero on the stack above a single-cell unsigned integer produces a double-cell unsigned integer with the same value. See **3.2.1.1 Internal number representation**.

3.1.4.2   Character strings

A string is specified by a cell pair (*c-addr u*) representing its starting address and length in characters.

3.1.5   System types

The system data types specify permitted word combinations during compilation and execution.

3.1.5.1   System-compilation types

These data types denote zero or more items on the control-flow stack (see 3.2.3.2). The possible presence of such items on the data stack means that any items already there shall be unavailable to a program until the control-flow-stack items are consumed.

The implementation-dependent data generated upon beginning to compile a definition and consumed at its close is represented by the symbol *colon-sys* throughout this standard.

The implementation-dependent data generated upon beginning to compile a do-loop structure such as **DO** ... **LOOP** and consumed at its close is represented by the symbol *do-sys* throughout this standard.

The implementation-dependent data generated upon beginning to compile a **CASE** ... **ENDCASE** structure and consumed at its close is represented by the symbol *case-sys* throughout this standard.

The implementation-dependent data generated upon beginning to compile an **OF** ... **ENDOF** structure and consumed at its close is represented by the symbol *of-sys* throughout this standard.

The implementation-dependent data generated and consumed by executing the other standard control-flow words is represented by the symbols *orig* and *dest* throughout this standard.

3.1.5.2   System-execution types

These data types denote zero or more items on the return stack. Their possible presence means that any items already on the return stack shall be unavailable to a program until the system-execution items are consumed.

The implementation-dependent data generated upon beginning to execute a definition and consumed upon exiting it is represented by the symbol *nest-sys* throughout this standard.

The implementation-dependent loop-control parameters used to control the execution of do-loops are represented by the symbol *loop-sys* throughout this standard. Loop-control parameters shall be available inside the do-loop for words that use or change these parameters, words such as **I**, **J**, **LEAVE** and **UNLOOP**.

3.2   The implementation environment

3.2.1   Numbers

3.2.1.1   Internal number representation

This standard allows one's complement, two's complement, or sign-magnitude number representations and arithmetic. Arithmetic zero is represented as the value of a single cell with all bits clear.

The representation of a number as a compiled literal or in memory is implementation dependent.

3.2.1.2   Digit conversion

Numbers shall be represented externally by using characters from the standard character set. Conversion between the internal and external forms of a digit shall behave as follows:

The value in **BASE** is the radix for number conversion. A digit has a value ranging from zero to one less than the contents of **BASE**. The digit with the value zero corresponds to the character "0". This representation of digits proceeds through the character set to the decimal value nine corresponding to the character "9". For digits beginning with the decimal value ten the graphic characters beginning with the character "A" are used. This correspondence continues up to and including the digit with the decimal value thirty-five which is represented by the character "Z". The characters "a" though to "z" should be treated the same as "A" though "Z", with "a" having the value ten and "z" the value thirty-five. The conversion of digits outside this range is implementation defined.

3.2.1.3   Free-field number display

Free-field number display uses the characters described in digit conversion, without leading zeros, in a field the exact size of the converted string plus a trailing space. If a number is zero, the least significant digit is not considered a leading zero. If the number is negative, a leading minus sign is displayed.

Number display may use the pictured numeric output string buffer to hold partially converted strings (see **3.3.3.6 Other transient regions**).

3.2.2   Arithmetic

3.2.2.1   Integer division

Division produces a quotient *q* and a remainder *r* by dividing operand *a* by operand *b*. Division operations return *q, r*, or both. The identity $b \times q + r = a$ shall hold for all *a* and *b*.

When unsigned integers are divided and the remainder is not zero, *q* is the largest integer less than the true quotient.

When signed integers are divided, the remainder is not zero, and *a* and *b* have the same sign, *q* is the largest integer less than the true quotient. If only one operand is negative, whether *q* is rounded toward negative infinity (floored division) or rounded towards zero (symmetric division) is implementation defined.

Floored division is integer division in which the remainder carries the sign of the divisor or is zero, and the quotient is rounded to its arithmetic floor. Symmetric division is integer division in which the remainder carries the sign of the dividend or is zero and the quotient is the mathematical quotient "rounded towards zero" or "truncated". Examples of each are shown in tables 3.3 and 3.4.

In cases where the operands differ in sign and the rounding direction matters, a program shall either include code generating the desired form of division, not relying on the implementation-defined default result, or have an environmental dependency on the desired rounding direction.

Table 3.3: Floored Division Example				Table 3.4: Symmetric Division Example			
Dividend	Divisor	Remainder	Quotient	Dividend	Divisor	Remainder	Quotient
10	7	3	1	10	7	3	1
-10	7	4	-2	-10	7	-3	-1
10	-7	-4	-2	10	-7	3	-1
-10	-7	-3	1	-10	-7	-3	1

3.2.2.2 Other integer operations

In all integer arithmetic operations, both overflow and underflow shall be ignored. The value returned when either overflow or underflow occurs is implementation defined.

3.2.3 Stacks

3.2.3.1 Data stack

Objects on the data stack shall be one cell wide.

3.2.3.2 Control-flow stack

The control-flow stack is a last-in, first out list whose elements define the permissible matchings of control-flow words and the restrictions imposed on data-stack usage during the compilation of control structures.

The elements of the control-flow stack are system-compilation data types.

The control-flow stack may, but need not, physically exist in an implementation. If it does exist, it may be, but need not be, implemented using the data stack. The format of the control-flow stack is implementation defined.

3.2.3.3 Return stack

Items on the return stack shall consist of one or more cells. A system may use the return stack in an implementation-dependent manner during the compilation of definitions, during the execution of do-loops, and for storing run-time nesting information.

A program may use the return stack for temporary storage during the execution of a definition subject to the following restrictions:

- A program shall not access values on the return stack (using `R@`, `R>`, `2R@`, `2R>` or `NR>`) that it did not place there using `>R`, `2>R` or `N>R`;

- A program shall not access from within a do-loop values placed on the return stack before the loop was entered;

- All values placed on the return stack within a do-loop shall be removed before `I`, `J`, `LOOP`, `+LOOP`, `UNLOOP`, or `LEAVE` is executed;

- All values placed on the return stack within a definition shall be removed before the definition is terminated or before `EXIT` is executed.

3.2.4   Operator terminal

See **1.2.2 Exclusions**.

3.2.4.1   User input device

The method of selecting the user input device is implementation defined.

The method of indicating the end of an input line of text is implementation defined.

3.2.4.2   User output device

The method of selecting the user output device is implementation defined.

3.2.5   Mass storage

A system need not provide any standard words for accessing mass storage.

3.2.6   Environmental queries

The name spaces for `ENVIRONMENT?` and definitions are disjoint. Names of definitions that are the same as `ENVIRONMENT?` strings shall not impair the operation of `ENVIRONMENT?`. Table 3.5 contains the valid input strings and corresponding returned value for inquiring about the programming environment with `ENVIRONMENT?`.

If an environmental query (using `ENVIRONMENT?`) returns *false* (i.e., unknown) in response to a string, subsequent queries using the same string may return *true*. If a query returns *true* (i.e., known) in response to a string, subsequent queries with the same string shall also return *true*. If a query designated as constant in the above table returns *true* and a value in response to a string, subsequent queries with the same string shall return *true* and the same value.

3.2.7   Obsolescent Environmental Queries

This standard designates the practice of using `ENVIRONMENT?` to inquire whether a given word set is present as obsolescent. If such a query, as listed in table 3.6, returns *true*, the word set is present in the form defined by Forth 94. As these queries will be withdrawn from future revisions of the standard their use in new programs is discouraged.

See **A.3.2.7 Obsolescent Environmental Queries**.

Table 3.5: Environmental Query Strings

String	Value data type	Constant?	Meaning
/COUNTED-STRING	*n*	yes	maximum size of a counted string, in characters
/HOLD	*n*	yes	size of the pictured numeric output string buffer, in characters
/PAD	*n*	yes	size of the scratch area pointed to by **PAD**, in characters
ADDRESS-UNIT-BITS	*n*	yes	size of one address unit, in bits
FLOORED	*flag*	yes	true if floored division is the default
MAX-CHAR	*u*	yes	maximum value of any character in the implementation-defined character set
MAX-D	*d*	yes	largest usable signed double number
MAX-N	*n*	yes	largest usable signed integer
MAX-U	*u*	yes	largest usable unsigned integer
MAX-UD	*ud*	yes	largest usable unsigned double number
RETURN-STACK-CELLS	*n*	yes	maximum size of the return stack, in cells
STACK-CELLS	*n*	yes	maximum size of the data stack, in cells

3.3 The Forth dictionary

Forth words are organized into a structure called the dictionary. While the form of this structure is not specified by the standard, it can be described as consisting of three logical parts: a name space, a code space, and a data space. The logical separation of these parts does not require their physical separation.

A program shall not fetch from or store into locations outside data space. An ambiguous condition exists if a program addresses name space or code space.

3.3.1 Name space

The relationship between name space and data space is implementation dependent.

3.3.1.1 Word lists

The structure of a word list is implementation dependent. When duplicate names exist in a word list, the latest-defined duplicate shall be the one found during a search for the name.

3.3.1.2 Definition names

Definition names shall contain {1 … 31} characters. A system may allow or prohibit the creation of definition names containing non-standard characters. A system may allow the creation of definition names longer than 31 characters. Programs with definition names longer than 31 characters have an environmental dependency.

Programs that use lower case for standard definition names or depend on the case-sensitivity properties of a system have an environmental dependency.

A program shall not create definition names containing non-graphic characters.

Table 3.6: Obsolescent Environmental Query Strings

String	Value data type	Constant?	Meaning
CORE	*flag*	no	true if complete core word set of Forth 94 is present (i.e., not a subset as defined in 5.1.1)
CORE-EXT	*flag*	no	true if the core extensions word set of Forth 94 is present
BLOCK	*flag*	no	Forth 94 block word set present.
BLOCK-EXT	*flag*	no	Forth 94 block extensions word set present.
DOUBLE	*flag*	no	Forth 94 double number word set present.
DOUBLE-EXT	*flag*	no	Forth 94 double number extensions word set present.
EXCEPTION	*flag*	no	Forth 94 exception word set present.
EXCEPTION-EXT	*flag*	no	Forth 94 exception extensions word set present.
FACILITY	*flag*	no	Forth 94 facility word set present.
FACILITY-EXT	*flag*	no	Forth 94 facility extensions word set present.
FILE	*flag*	no	Forth 94 file word set present.
FILE-EXT	*flag*	no	Forth 94 file extensions word set present.
FLOATING	*flag*	no	Forth 94 floating-point word set present.
FLOATING-EXT	*flag*	no	Forth 94 floating-point extensions word set present.
LOCALS	*flag*	no	Forth 94 locals word set present.
LOCALS-EXT	*flag*	no	Forth 94 locals extensions word set present.
MEMORY-ALLOC	*flag*	no	Forth 94 memory-allocation word set present.
MEMORY-ALLOC-EXT	*flag*	no	Forth 94 memory-allocation extensions word set present.
TOOLS	*flag*	no	Forth 94 programming-tools word set present.
TOOLS-EXT	*flag*	no	Forth 94 programming-tools extensions word set present.
SEARCH-ORDER	*flag*	no	Forth 94 search-order word set present.
SEARCH-ORDER-EXT	*flag*	no	Forth 94 search-order extensions word set present.
STRING	*flag*	no	Forth 94 string word set present.
STRING-EXT	*flag*	no	Forth 94 string extensions word set present.

3.3.2 Code space

The relationship between code space and data space is implementation dependent.

3.3.3 Data space

Data space is the only logical area of the dictionary for which standard words are provided to allocate and access regions of memory. These regions are: contiguous regions, variables, text-literal regions, input buffers, and other transient regions, each of which is described in the following sections. A program may read from or write into these regions unless otherwise specified.

3.3.3.1 Address alignment

Most addresses are cell aligned (indicated by *a-addr*) or character aligned (*c-addr*). **ALIGNED**, **CHAR+**, and arithmetic operations can alter the alignment state of an address on the stack. **CHAR+** applied to an aligned address returns a character-aligned address that can only be used to access characters. Applying **CHAR+** to a character-aligned address produces the succeeding character-aligned address. Adding or subtracting an arbitrary number to an address can produce an unaligned address that shall not be used to fetch

or store anything. The only way to find the next aligned address is with **ALIGNED**. An ambiguous condition exists when memory is accessed using an address that is not aligned according to the requirements for the accessed type.

The definitions of **6.1.1000** CREATE and **6.1.2410** VARIABLE require that the definitions created by them return aligned addresses.

After definitions are compiled or the word **ALIGN** is executed the data-space pointer is guaranteed to be aligned.

3.3.3.2   Contiguous regions

A system guarantees that a region of data space allocated using **ALLOT**, **,** (comma), **C,** (c-comma), and **ALIGN** shall be contiguous with the last region allocated with one of the above words, unless the restrictions in the following paragraphs apply. The data-space pointer **HERE** always identifies the beginning of the next data-space region to be allocated. As successive allocations are made, the data-space pointer increases. A program may perform address arithmetic within contiguously allocated regions. The last region of data space allocated using the above operators may be released by allocating a corresponding negatively-sized region using **ALLOT**, subject to the restrictions of the following paragraphs.

CREATE establishes the beginning of a contiguous region of data space, whose starting address is returned by the **CREATE**d definition. This region is terminated by compiling the next definition.

Since an implementation is free to allocate data space for use by code, the above operators need not produce contiguous regions of data space if definitions are added to or removed from the dictionary between allocations. An ambiguous condition exists if deallocated memory contains definitions.

3.3.3.3   Variables

The region allocated for a variable may be non-contiguous with regions subsequently allocated with **,** (comma) or **ALLOT**. For example, in:

>   **VARIABLE** X   1 **CELLS ALLOT**

the region X and the region **ALLOT**ted could be non-contiguous.

Some system-provided variables, such as **STATE**, are restricted to read-only access.

3.3.3.4   Text-literal regions

The text-literal regions, specified by strings compiled with **S"**, **S\"** and **C"**, may be read-only.

A program shall not store into the text-literal regions created by **S"**, **S\"** and **C"** nor into any read-only system variable or read-only transient regions.

3.3.3.5   Input buffers

The address, length, and content of the input buffer may be transient. A program shall not write into the input buffer. In the absence of any optional word sets providing alternative input sources, the input buffer is either the terminal-input buffer, used by **QUIT** to hold one line from the user input device, or a buffer specified by **EVALUATE**. In all cases, **SOURCE** returns the beginning address and length in characters of the current input buffer.

The minimum size of the terminal-input buffer shall be 80 characters.

The address and length returned by **SOURCE**, the string returned by **PARSE**, and directly computed input-buffer addresses are valid only until the text interpreter does I/O to refill the input buffer or the input source is changed.

A program may modify the size of the parse area by changing the contents of **>IN** within the limits imposed by this standard. For example, if the contents of **>IN** are saved before a parsing operation and restored afterwards, the text that was parsed will be available again for subsequent parsing operations. The extent of permissible repositioning using this method depends on the input source (see **7.3.2 Block buffer regions** and **11.3.3 Input source**).

A program may directly examine the input buffer using its address and length as returned by **SOURCE**; the beginning of the parse area within the input buffer is indexed by the number in **>IN**. The values are valid for a limited time. An ambiguous condition exists if a program modifies the contents of the input buffer.

3.3.3.6   Other transient regions

The data space regions identified by **PAD**, **WORD**, and **#>** (the pictured numeric output string buffer) may be transient. Their addresses and contents may become invalid after:

 – a definition is created via a defining word;

 – definitions are compiled with **:** or **:NONAME**;

 – data space is allocated using **ALLOT**, **,** (comma), **C,** (c-comma), or **ALIGN**.

The previous contents of the regions identified by **WORD** and **#>** may be invalid after each use of these words. Further, the regions returned by **WORD** and **#>** may overlap in memory. Consequently, use of one of these words can corrupt a region returned earlier by a different word. The other words that construct pictured numeric output strings (**<#**, **#**, **#S**, **HOLD**, **HOLDS**, **XHOLD**) may also modify the contents of these regions. Words that display numbers may be implemented using pictured numeric output words. Consequently, **.** (dot), **.R**, **.S**, **?**, **D.**, **D.R**, **U.**, **U.R** could also corrupt the regions.

The size of the scratch area whose address is returned by **PAD** shall be at least 84 characters. The contents of the region addressed by **PAD** are intended to be under the complete control of the user: no words defined in this standard place anything in the region, although changing data-space allocations as described in **3.3.3.2 Contiguous regions** may change the address returned by **PAD**. Non-standard words provided by an implementation may use **PAD**, but such use shall be documented.

The size of the region identified by **WORD** shall be at least 33 characters.

The size of the pictured numeric output string buffer shall be at least $(2 \times n) + 2$ characters, where n is the number of bits in a cell. Programs that consider it a fixed area with unchanging access parameters have an environmental dependency.

3.4   The Forth text interpreter

Upon start-up, a system shall be able to interpret, as described by **6.1.2050** QUIT, Forth source code received interactively from a user input device.

Such interactive systems usually furnish a "prompt" indicating that they have accepted a user request and acted on it. The implementation-defined Forth prompt should contain the word "OK" in some combination of upper or lower case.

Text interpretation (see **6.1.1360** EVALUATE and **6.1.2050** QUIT) shall repeat the following steps until either the parse area is empty or an ambiguous condition exists:

   a) Skip leading spaces and parse a *name* (see 3.4.1);

   b) Search the dictionary name space (see 3.4.2). If a definition name matching the string is found:

      1) if interpreting, perform the interpretation semantics of the definition (see 3.4.3.2), and continue at a).

      2) if compiling, perform the compilation semantics of the definition (see 3.4.3.3), and continue at a).

   c) If a definition name matching the string is not found, attempt to convert the string to a number (see 3.4.1.3). If successful:

      1) if interpreting, place the number on the data stack, and continue at a);

      2) if compiling, compile code that when executed will place the number on the stack (see **6.1.1780** LITERAL), and continue at a);

   d) If unsuccessful, an ambiguous condition exists (see 3.4.4).

3.4.1 Parsing

Unless otherwise noted, the number of characters parsed may be from zero to the implementation-defined maximum length of a counted string.

If the parse area is empty, i.e., when the number in `>IN` is equal to the length of the input buffer, or contains no characters other than delimiters, the selected string is empty. Otherwise, the selected string begins with the next character in the parse area, which is the character indexed by the contents of `>IN`. An ambiguous condition exists if the number in `>IN` is greater than the size of the input buffer.

If delimiter characters are present in the parse area after the beginning of the selected string, the string continues up to and including the character just before the first such delimiter, and the number in `>IN` is changed to index immediately past that delimiter, thus removing the parsed characters and the delimiter from the parse area. Otherwise, the string continues up to and including the last character in the parse area, and the number in `>IN` is changed to the length of the input buffer, thus emptying the parse area.

Parsing may change the contents of `>IN`, but shall not affect the contents of the input buffer. Specifically, if the value in `>IN` is saved before starting the parse, resetting `>IN` to that value immediately after the parse shall restore the parse area without loss of data.

3.4.1.1 Delimiters

If the delimiter is the space character, hex 20 (`BL`), control characters may be treated as delimiters. The set of conditions, if any, under which a "space" delimiter matches control characters is implementation defined.

To skip leading delimiters is to pass by zero or more contiguous delimiters in the parse area before parsing.

3.4.1.2 Syntax

Forth has a simple, operator-ordered syntax. The phrase `A B C` returns values as if `A` were executed first, then `B` and finally `C`. Words that cause deviations from this linear flow of control are called control-flow words. Combinations of control-flow words whose stack effects are compatible form control-flow structures. Examples of typical use are given for each control-flow word in Annex A.

Forth syntax is extensible; for example, new control-flow words can be defined in terms of existing ones. This standard does not require a syntax or program-construct checker.

3.4.1.3 Text interpreter input number conversion

When converting input numbers, the text interpreter shall recognize integer numbers in the form ⟨*anynum*⟩.

$$\begin{aligned}
\langle anynum\rangle &:= \{\ \langle BASEnum\rangle \mid \langle decnum\rangle \mid \langle hexnum\rangle \mid \langle binnum\rangle \mid \langle cnum\rangle\ \}\\
\langle BASEnum\rangle &:= \text{[-]}\langle bdigit\rangle\langle bdigit\rangle\text{*}\\
\langle decnum\rangle &:= \text{\#[-]}\langle decdigit\rangle\langle decdigit\rangle\text{*}\\
\langle hexnum\rangle &:= \text{\$[-]}\langle hexdigit\rangle\langle hexdigit\rangle\text{*}\\
\langle binnum\rangle &:= \text{\%[-]}\langle bindigit\rangle\langle bindigit\rangle\text{*}\\
\langle cnum\rangle &:= \text{'}\langle char\rangle\text{'}\\
\langle bindigit\rangle &:= \{\ \mathbf{0}\mid\mathbf{1}\ \}\\
\langle decdigit\rangle &:= \{\ \mathbf{0}\mid\mathbf{1}\mid\mathbf{2}\mid\mathbf{3}\mid\mathbf{4}\mid\mathbf{5}\mid\mathbf{6}\mid\mathbf{7}\mid\mathbf{8}\mid\mathbf{9}\ \}\\
\langle hexdigit\rangle &:= \{\ \langle decdigit\rangle\mid\mathbf{a}\mid\mathbf{b}\mid\mathbf{c}\mid\mathbf{d}\mid\mathbf{e}\mid\mathbf{f}\mid\mathbf{A}\mid\mathbf{B}\mid\mathbf{C}\mid\mathbf{D}\mid\mathbf{E}\mid\mathbf{F}\ \}
\end{aligned}$$

⟨*bdigit*⟩ represents a digit according to the value of **BASE** (see **3.2.1.2 Digit conversion**). For ⟨*hexdigit*⟩, the digits **a**…**f** have the values 10…15. ⟨*char*⟩ represents any printable character.

The radix used for number conversion is:

⟨*BASEnum*⟩	the value in **BASE**
⟨*decnum*⟩	10
⟨*hexnum*⟩	16
⟨*binnum*⟩	2
⟨*cnum*⟩	the number is the value of ⟨*char*⟩

See **2.2.5 BNF notation**.

3.4.2 Finding definition names

A string matches a definition name if each character in the string matches the corresponding character in the string used as the definition name when the definition was created. The case sensitivity (whether or not the upper-case letters match the lower-case letters) is implementation defined. A system may be either case sensitive, treating upper- and lower-case letters as different and not matching, or case insensitive, ignoring differences in case while searching.

The matching of upper- and lower-case letters with alphabetic characters in character set extensions such as accented international characters is implementation defined.

A system shall be capable of finding the definition names defined by this standard when they are spelled with upper-case letters.

3.4.3 Semantics

The semantics of a Forth definition are implemented by machine code or a sequence of execution tokens or other representations. They are largely specified by the stack notation in the glossary entries, which shows what values shall be consumed and produced. The prose in each glossary entry further specifies the definition's behavior.

Each Forth definition may have several behaviors, described in the following sections. The terms "initiation semantics" and "run-time semantics" refer to definition fragments, and have meaning only within the individual glossary entries where they appear.

3.4.3.1 Execution semantics

The execution semantics of each Forth definition are specified in an "Execution:" section of its glossary entry. When a definition has only one specified behavior, the label is omitted.

Execution may occur implicitly, when the definition into which it has been compiled is executed, or explicitly, when its execution token is passed to **EXECUTE**. The execution semantics of a syntactically correct definition under conditions other than those specified in this standard are implementation dependent.

Glossary entries for defining words include the execution semantics for the new definition in a "*name* Execution:" section.

3.4.3.2 Interpretation semantics

Unless otherwise specified in an "Interpretation:" section of the glossary entry, the interpretation semantics of a Forth definition are its execution semantics.

A system shall be capable of executing, in interpretation state, all of the definitions from the Core word set and any definitions included from the optional word sets or word set extensions whose interpretation semantics are defined by this standard.

A system shall be capable of executing, in interpretation state, any new definitions created in accordance with **3 Usage requirements**.

3.4.3.3 Compilation semantics

Unless otherwise specified in a "Compilation:" section of the glossary entry, the compilation semantics of a Forth definition shall be to append its execution semantics to the execution semantics of the current definition.

3.4.4 Possible actions on an ambiguous condition

When an ambiguous condition exists, a system may take one or more of the following actions:

- ignore and continue;
- display a message;
- execute a particular word;
- set interpretation state and begin text interpretation;
- take other implementation-defined actions;

– take implementation-dependent actions.

The response to a particular ambiguous condition need not be the same under all circumstances.

3.4.5 Compilation

A program shall not attempt to nest compilation of definitions.

During the compilation of the current definition, a program shall not execute any defining word, **:NONAME**, or any definition that allocates dictionary data space. The compilation of the current definition may be suspended using **[** (left-bracket) and resumed using **]** (right-bracket). While the compilation of the current definition is suspended, a program shall not execute any defining word, **:NONAME**, or any definition that allocates dictionary data space.

4 Documentation requirements

When it is impossible or infeasible for a system or program to define a particular behavior itself, it is permissible to state that the behavior is unspecifiable and to explain the circumstances and reasons why this is so.

4.1 System documentation

4.1.1 Implementation-defined options

The implementation-defined items in the following list represent characteristics and choices left to the discretion of the implementor, provided that the requirements of this standard are met. A system shall document the values for, or behaviors of, each item.

  – aligned address requirements **3.1.3.3 Addresses**;

  – behavior of **6.1.1320** EMIT for non-graphic characters;

  – character editing of **6.1.0695** ACCEPT;

  – character set (**3.1.2 Character types**, **6.1.1320** EMIT, **6.1.1750** KEY);

  – character-aligned address requirements (**3.1.3.3 Addresses**);

  – character-set-extensions matching characteristics (**3.4.2 Finding definition names**);

  – conditions under which control characters match a space delimiter (**3.4.1.1 Delimiters**);

  – format of the control-flow stack (**3.2.3.2 Control-flow stack**);

  – conversion of digits larger than thirty-five (**3.2.1.2 Digit conversion**);

  – display after input terminates in **6.1.0695** ACCEPT;

  – exception abort sequence (as in **6.1.0680** ABORT");

  – input line terminator (**3.2.4.1 User input device**);

  – maximum size of a counted string, in characters (**3.1.3.4 Counted strings**, **6.1.2450** WORD);

  – maximum size of a parsed string (**3.4.1 Parsing**);

  – maximum size of a definition name, in characters (**3.3.1.2 Definition names**);

  – maximum string length for **6.1.1345** ENVIRONMENT?, in characters;

  – method of selecting **3.2.4.1 User input device**;

  – method of selecting **3.2.4.2 User output device**;

  – methods of dictionary compilation (**3.3 The Forth dictionary**);

  – number of bits in one address unit (**3.1.3.3 Addresses**);

  – number representation and arithmetic (**3.2.1.1 Internal number representation**);

- ranges for *n*, *+n*, *u*, *d*, *+d*, and *ud* (**3.1.3 Single-cell types**, **3.1.4 Cell-pair types**);

- read-only data-space regions (**3.3.3 Data space**);

- size of buffer at **6.1.2450** `WORD` (**3.3.3.6 Other transient regions**);

- size of one cell in address units (**3.1.3 Single-cell types**);

- size of one character in address units (**3.1.2 Character types**);

- size of the keyboard terminal input buffer (**3.3.3.5 Input buffers**);

- size of the pictured numeric output string buffer (**3.3.3.6 Other transient regions**);

- size of the scratch area whose address is returned by **6.2.2000** `PAD`
  (**3.3.3.6 Other transient regions**);

- system case-sensitivity characteristics (**3.4.2 Finding definition names**);

- system prompt (**3.3 The Forth dictionary**, **6.1.2050** `QUIT`);

- type of division rounding (**3.2.2.1 Integer division**, **6.1.0100** `*/`, **6.1.0110** `*/MOD`, **6.1.0230** `/`,
  6.1.0240 `/MOD`, **6.1.1890** `MOD`);

- values of **6.1.2250** `STATE` when true;

- values returned after arithmetic overflow (**3.2.2.2 Other integer operations**);

- whether the current definition can be found after **6.1.1250** `DOES>` (**6.1.0450** `:`).

4.1.2 Ambiguous conditions

A system shall document the system action taken upon each of the general or specific ambiguous conditions
identified in this standard. See **3.4.4 Possible actions on an ambiguous condition**.

The following general ambiguous conditions could occur because of a combination of factors:

- a *name* is neither a valid definition name nor a valid number during text interpretation (**3.4 The Forth
  text interpreter**);

- a definition name exceeded the maximum length allowed (**3.3.1.2 Definition names**);

- addressing a region not listed in **3.3.3 Data space**;

- argument type incompatible with specified input parameter, e.g., passing a *flag* to a word expecting
  an *n* (**3.1 Data types**);

- attempting to obtain the execution token, (e.g., with **6.1.0070** `'`, **6.1.1550** `FIND`, etc. of a definition
  with undefined interpretation semantics;

- dividing by zero (**6.1.0100** `*/`, **6.1.0110** `*/MOD`, **6.1.0230** `/`, **6.1.0240** `/MOD`, **6.1.1561** `FM/MOD`,
  6.1.1890 `MOD`, **6.1.2214** `SM/REM`, **6.1.2370** `UM/MOD`, **8.6.1.1820** `M*/`);

- insufficient data-stack space or return-stack space (stack overflow);

- insufficient space for loop-control parameters;

- insufficient space in the dictionary;

- interpreting a word with undefined interpretation semantics;

- modifying the contents of the input buffer or a string literal (**3.3.3.4 Text-literal regions**, **3.3.3.5 Input buffers**);

- overflow of a pictured numeric output string;

- parsed string overflow;

- producing a result out of range, e.g., multiplication (using `*`) results in a value too big to be represented by a single-cell integer (**6.1.0090** `*`, **6.1.0100** `*/`, **6.1.0110** `*/MOD`, **6.1.0570** `>NUMBER`, **6.1.1561** `FM/MOD`, **6.1.2214** `SM/REM`, **6.1.2370** `UM/MOD`, **8.6.1.1820** `M*/`);

- reading from an empty data stack or return stack (stack underflow);

- unexpected end of input buffer, resulting in an attempt to use a zero-length string as a *name*;

The following specific ambiguous conditions are noted in the glossary entries of the relevant words:

- `>IN` greater than size of input buffer (**3.4.1 Parsing**);

- **6.1.2120** `RECURSE` appears after **6.1.1250** `DOES>`;

- argument input source different than current input source for **6.2.2148** `RESTORE-INPUT`;

- data space containing definitions is de-allocated (**3.3.3.2 Contiguous regions**);

- data space read/write with incorrect alignment (**3.3.3.1 Address alignment**);

- data-space pointer not properly aligned (**6.1.0150** `,`, **6.1.0860** `C,`);

- less than *u*+2 stack items (**6.2.2030** `PICK`, **6.2.2150** `ROLL`);

- loop-control parameters not available (**6.1.0140** `+LOOP`, **6.1.1680** `I`, **6.1.1730** `J`, **6.1.1760** `LEAVE`, **6.1.1800** `LOOP`, **6.1.2380** `UNLOOP`);

- most recent definition does not have a *name* (**6.1.1710** `IMMEDIATE`);

- **6.2.2295** `TO` not followed directly by a *name* defined by a word with "`TO` *name* runtime" semantics (**6.2.2405** `VALUE` and **13.6.1.0086** `(LOCAL)`);

- *name* not found **6.1.0070** `'`, **6.1.2033** `POSTPONE`, **6.1.2510** `[']`, **6.2.2530** `[COMPILE]`);

- parameters are not of the same type **6.1.1240** `DO`, **6.2.0620** `?DO`, **6.2.2440** `WITHIN`);

- **6.1.2033** `POSTPONE`, **6.2.2530** `[COMPILE]`, **6.1.0070** `'` or **6.1.2510** `[']` applied to **6.2.2295** `TO`;

- string longer than a counted string returned by **6.1.2450** `WORD`;

- *u* greater than or equal to the number of bits in a cell (**6.1.1805** `LSHIFT`, **6.1.2162** `RSHIFT`);

- word not defined via **6.1.1000** `CREATE` (**6.1.0550** `>BODY`, **6.1.1250** `DOES>`);

- words improperly used outside **6.1.0490** `<#` and **6.1.0040** `#>` (**6.1.0030** `#`, **6.1.0050** `#S`, **6.1.1670** `HOLD`, **6.2.1675** `HOLDS`, **6.1.2210** `SIGN`).

- access to a deferred word, a word defined by **6.2.1173** `DEFER`, which has yet to be assigned to an *xt*.

- access to a deferred word, a word defined by **6.2.1173** `DEFER`, which was not defined by **6.2.1173** `DEFER`.

- **6.1.2033** `POSTPONE`, **6.2.2530** `[COMPILE]`, **6.1.2510** `[']` or **6.1.0070** `'` applied to **6.2.0698** `ACTION-OF` or **6.2.1725** `IS`.

- \x is not followed by two hexadecimal characters (**6.2.2266** `S\"`).

- a \ is placed before any character, other than those defined in **6.2.2266** `S\"`.

4.1.3   Other system documentation

A system shall provide the following information:

- list of non-standard words using **6.2.2000** `PAD` (**3.3.3.6 Other transient regions**);

- operator's terminal facilities available;

- program data space available, in address units;

- return stack space available, in cells;

- stack space available, in cells;

- system dictionary space required, in address units.

4.2   Program documentation

4.2.1   Environmental dependencies

A program shall document the following environmental dependencies, where they apply, and should document other known environmental dependencies:

- considering the pictured numeric output string buffer a fixed area with unchanging access parameters (**3.3.3.6 Other transient regions**);

- depending on the presence or absence of non-graphic characters in a received string (**6.1.0695** `ACCEPT`);

- relying on a particular rounding direction (**3.2.2.1 Integer division**);

- requiring a particular number representation and arithmetic (**3.2.1.1 Internal number representation**);

- requiring non-standard words or techniques (**3 Usage requirements**);

- requiring the ability to send or receive control characters (**3.1.2.2 Control characters**, **6.1.1750** `KEY`);

- using control characters to perform specific functions **6.1.1320** `EMIT`, **6.1.2310** `TYPE`);

- using flags as arithmetic operands (**3.1.3.1 Flags**);

- using lower case for standard definition names or depending on the case sensitivity of a system (**3.3.1.2 Definition names**);

- using definition names of more than 31 characters in length (**3.3.1.2 Definition names**);

- using the graphic character with a value of hex 24 (**3.1.2.1 Graphic characters**).

4.2.2   Other program documentation

A program shall also document:

- minimum operator's terminal facilities required;

- whether a Standard System exists after the program is loaded.

5   Compliance and labeling

5.1   Forth-2012 systems

5.1.1   System compliance

A system that complies with all the system requirements given in sections **3 Usage requirements** and **4.1 System documentation** and their sub-sections is a Standard System. An otherwise Standard System that provides only a portion of the Core words is a Standard System Subset. An otherwise Standard System (Subset) that fails to comply with one or more of the minimum values or ranges specified in **3 Usage requirements** and its sub-sections has environmental restrictions.

5.1.2   System labeling

A Standard System (Subset) shall be labeled a "Forth-2012 System (Subset)". That label, by itself, shall not be applied to Standard Systems or Standard System Subsets that have environmental restrictions.

The phrase "with Environmental Restrictions" shall be appended to the label of a Standard System (Subset) that has environmental restrictions.

The phrase "Providing *name(s)* from the Core Extensions word set" shall be appended to the label of any Standard System that provides portions of the Core Extensions word set.

The phrase "Providing the Core Extensions word set" shall be appended to the label of any Standard System that provides all of the Core Extensions word set.

5.2   Forth-2012 programs

5.2.1   Program compliance

A program that complies with all the program requirements given in sections **3 Usage requirements** and **4.2 Program documentation** and their sub-sections is a Standard Program.

5.2.2   Program labeling

A Standard Program shall be labeled a "Forth-2012 Program". That label, by itself, shall not be applied to Standard Programs that require the system to provide standard words outside the Core word set or that have environmental dependencies.

The phrase "with Environmental Dependencies" shall be appended to the label of Standard Programs that have environmental dependencies.

The phrase "Requiring *name(s)* from the Core Extensions word set" shall be appended to the label of Standard Programs that require the system to provide portions of the Core Extensions word set.

The phrase "Requiring the Core Extensions word set" shall be appended to the label of Standard Programs that require the system to provide all of the Core Extensions word set.

6   Glossary

6.1   Core words

6.1.0010   **!**   "store"   CORE

(*x a-addr* −−)

Store *x* at *a-addr*.

See: **3.3.3.1 Address alignment**.

6.1.0030   **#**   "number-sign"   CORE

(ud_1 −− ud_2)

Divide ud_1 by the number in **BASE** giving the quotient ud_2 and the remainder *n*. (*n* is the least significant digit of ud_1.) Convert *n* to external form and add the resulting character to the beginning of the pictured numeric output string. An ambiguous condition exists if **#** executes outside of a **<# #>** delimited number conversion.

See: **6.1.0040** #>, **6.1.0050** #S, **6.1.0490** <#.

6.1.0040   **#>**   "number-sign-greater"   CORE

(*xd* −− *c-addr u*)

Drop *xd*. Make the pictured numeric output string available as a character string. *c-addr* and *u* specify the resulting character string. A program may replace characters within the string.

See: **6.1.0030** #, **6.1.0050** #S, **6.1.0490** <#.

6.1.0050   **#S**   "number-sign-s"   CORE

(ud_1 −− ud_2)

Convert one digit of ud_1 according to the rule for **#**. Continue conversion until the quotient is zero. ud_2 is zero. An ambiguous condition exists if **#S** executes outside of a **<# #>** delimited number conversion.

See: **6.1.0030** #, **6.1.0040** #>, **6.1.0490** <#.

6.1.0070   **'**   "tick"   **CORE**

(*"⟨spaces⟩name" –– xt*)

Skip leading space delimiters. Parse *name* delimited by a space. Find *name* and return *xt*, the execution token for *name*. An ambiguous condition exists if *name* is not found. When interpreting, **'** xyz EXECUTE is equivalent to xyz.

See: **3.4.3.2 Interpretation semantics**, **3.4.1 Parsing**, **A.6.1.0070 '** , **A.6.1.2033** POSTPONE, **A.6.1.2510** ['].

6.1.0080   **(**   "paren"   **CORE**

Compilation: Perform the execution semantics given below.

Execution: (*"ccc⟨paren⟩" ––*)

Parse *ccc* delimited by **)** (right parenthesis). **(** is an immediate word.

The number of characters in *ccc* may be zero to the number of characters in the parse area.

See: **3.4.1 Parsing**, **11.6.1.0080 (**, **A.6.1.0080 (**.

6.1.0090   *****   "star"   **CORE**

($n_1 \mid u_1 \; n_2 \mid u_2$ –– $n_3 \mid u_3$)

Multiply $n_1 \mid u_1$ by $n_2 \mid u_2$ giving the product $n_3 \mid u_3$.

6.1.0100   ***/**   "star-slash"   **CORE**

($n_1 \; n_2 \; n_3$ –– n_4)

Multiply n_1 by n_2 producing the intermediate double-cell result *d*. Divide *d* by n_3 giving the single-cell quotient n_4. An ambiguous condition exists if n_3 is zero or if the quotient n_4 lies outside the range of a signed number. If *d* and n_3 differ in sign, the implementation-defined result returned will be the same as that returned by either the phrase **>R M* R> FM/MOD SWAP DROP** or the phrase **>R M* R> SM/REM SWAP DROP**.

See: **3.2.2.1 Integer division**.

6.1.0110   ***/MOD**   "star-slash-mod"   **CORE**

($n_1 \; n_2 \; n_3$ –– $n_4 \; n_5$)

Multiply n_1 by n_2 producing the intermediate double-cell result *d*. Divide *d* by n_3 producing the single-cell remainder n_4 and the single-cell quotient n_5. An ambiguous condition

exists if n_3 is zero, or if the quotient n_5 lies outside the range of a single-cell signed integer. If d and n_3 differ in sign, the implementation-defined result returned will be the same as that returned by either the phrase **>R M* R> FM/MOD** or the phrase **>R M* R> SM/REM**.

See: **3.2.2.1 Integer division**.

| 6.1.0120 | **+** | "plus" | CORE |

(n_1 | u_1 n_2 | u_2 −− n_3 | u_3)

Add n_2 | u_2 to n_1 | u_1, giving the sum n_3 | u_3.

See: **3.3.3.1 Address alignment**.

| 6.1.0130 | **+!** | "plus-store" | CORE |

(n | u a-addr −−)

Add n | u to the single-cell number at a-addr.

See: **3.3.3.1 Address alignment**.

| 6.1.0140 | **+LOOP** | "plus-loop" | CORE |

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: do-sys −−)

Append the run-time semantics given below to the current definition. Resolve the destination of all unresolved occurrences of **LEAVE** between the location given by do-sys and the next location for a transfer of control, to execute the words following **+LOOP**.

Run-time: (n −−) (R: loop-sys$_1$ −− | loop-sys$_2$)

An ambiguous condition exists if the loop control parameters are unavailable. Add n to the loop index. If the loop index did not cross the boundary between the loop limit minus one and the loop limit, continue execution at the beginning of the loop. Otherwise, discard the current loop control parameters and continue execution immediately following the loop.

See: **6.1.1240** DO, **6.1.1680** I, **6.1.1760** LEAVE, **A.6.1.0140** +LOOP.

6.1.0150   **,**   "comma"   CORE

(*x* ––)

Reserve one cell of data space and store *x* in the cell. If the data-space pointer is aligned when **,** begins execution, it will remain aligned when **,** finishes execution. An ambiguous condition exists if the data-space pointer is not aligned prior to execution of **,** .

See: **3.3.3 Data space**, **3.3.3.1 Address alignment**, **A.6.1.0150 ,** .

6.1.0160   **–**   "minus"   CORE

(n_1 | u_1 n_2 | u_2 –– n_3 | u_3)

Subtract n_2 | u_2 from n_1 | u_1, giving the difference n_3 | u_3.

See: **3.3.3.1 Address alignment**.

6.1.0180   **.**   "dot"   CORE

(*n* ––)

Display *n* in free field format.

See: **3.2.1.2 Digit conversion**, **3.2.1.3 Free-field number display**.

6.1.0190   **."**   "dot-quote"   CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (*"ccc⟨quote⟩"* ––)

Parse *ccc* delimited by **"** (double-quote). Append the run-time semantics given below to the current definition.

Run-time: (––)

Display *ccc*.

See: **3.4.1 Parsing**, **6.2.0200 . (**, **A.6.1.0190 ." **.

6.1.0230   **/**   "slash"   CORE

(n_1 n_2 –– n_3)

Divide n_1 by n_2, giving the single-cell quotient n_3. An ambiguous condition exists if n_2 is zero. If n_1 and n_2 differ in sign, the implementation-defined result returned will be the same as that returned by either the phrase **>R S>D R> FM/MOD SWAP DROP** or the phrase **>R S>D R> SM/REM SWAP DROP**.

See: **3.2.2.1 Integer division**.

! " # $ % & ' ( ) * + , - . / digits : ; < = > ? @ ALPHA [\ ] ^ _ ` alpha { | } ~   47

6.1.0240   **/MOD**   "slash-mod"   CORE

(n_1 n_2 $--$ n_3 n_4)

Divide n_1 by n_2, giving the single-cell remainder n_3 and the single-cell quotient n_4. An ambiguous condition exists if n_2 is zero. If n_1 and n_2 differ in sign, the implementation-defined result returned will be the same as that returned by either the phrase **>R S>D R> FM/MOD** or the phrase **>R S>D R> SM/REM**.

See: **3.2.2.1 Integer division**.

6.1.0250   **0<**   "zero-less"   CORE

(n $--$ *flag*)

flag is true if and only if n is less than zero.

6.1.0270   **0=**   "zero-equals"   CORE

(x $--$ *flag*)

flag is true if and only if x is equal to zero.

6.1.0290   **1+**   "one-plus"   CORE

(n_1 | u_1 $--$ n_2 | u_2)

Add one (1) to n_1 | u_1 giving the sum n_2 | u_2.

6.1.0300   **1−**   "one-minus"   CORE

(n_1 | u_1 $--$ n_2 | u_2)

Subtract one (1) from n_1 | u_1 giving the difference n_2 | u_2.

6.1.0310   **2!**   "two-store"   CORE

(x_1 x_2 *a-addr* $--$)

Store the cell pair x_1 x_2 at *a-addr*, with x_2 at *a-addr* and x_1 at the next consecutive cell. It is equivalent to the sequence **SWAP OVER ! CELL+ !**.

See: **3.3.3.1 Address alignment**.

6.1.0320   **2***   "two-star"   CORE

(x_1 -- x_2)

x_2 is the result of shifting x_1 one bit toward the most-significant bit, filling the vacated least-significant bit with zero.

6.1.0330   **2/**   "two-slash"   CORE

(x_1 -- x_2)

x_2 is the result of shifting x_1 one bit toward the least-significant bit, leaving the most-significant bit unchanged.

6.1.0350   **2@**   "two-fetch"   CORE

(*a-addr* -- x_1 x_2)

Fetch the cell pair x_1 x_2 stored at *a-addr*. x_2 is stored at *a-addr* and x_1 at the next consecutive cell. It is equivalent to the sequence **DUP CELL+ @ SWAP @**.

See: **3.3.3.1 Address alignment**, **6.1.0310** 2!.

6.1.0370   **2DROP**   "two-drop"   CORE

(x_1 x_2 --)

Drop cell pair x_1 x_2 from the stack.

6.1.0380   **2DUP**   "two-dupe"   CORE

(x_1 x_2 -- x_1 x_2 x_1 x_2)

Duplicate cell pair x_1 x_2.

6.1.0400   **2OVER**   "two-over"   CORE

(x_1 x_2 x_3 x_4 -- x_1 x_2 x_3 x_4 x_1 x_2)

Copy cell pair x_1 x_2 to the top of the stack.

6.1.0430   **2SWAP**   "two-swap"   CORE

(x_1 x_2 x_3 x_4 -- x_3 x_4 x_1 x_2)

Exchange the top two cell pairs.

6.1.0450   **:**   "colon"   CORE

(C: *"⟨spaces⟩name" –– colon-sys*)

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name*, called a "colon definition". Enter compilation state and start the current definition, producing *colon-sys*. Append the initiation semantics given below to the current definition.

The execution semantics of *name* will be determined by the words compiled into the body of the definition. The current definition shall not be findable in the dictionary until it is ended (or until the execution of **DOES>** in some systems).

Initiation: (*i*x –– i*x*) (R: *–– nest-sys*)

Save implementation-dependent information *nest-sys* about the calling definition. The stack effects *i*x* represent arguments to *name*.

name Execution: (*i*x –– j*x*)

Execute the definition *name*. The stack effects *i*x* and *j*x* represent arguments to and results from *name*, respectively.

See: **3.4.3.2 Interpretation semantics**, **3.4.1 Parsing**, **3.4.5 Compilation**, **6.1.1250** DOES>, **6.1.2500** [, **6.1.2540**], **15.6.2.0470** ; CODE, **A.6.1.0450** : .

6.1.0460   **;**   "semicolon"   CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: *colon-sys ––*)

Append the run-time semantics below to the current definition. End the current definition, allow it to be found in the dictionary and enter interpretation state, consuming *colon-sys*. If the data-space pointer is not aligned, reserve enough data space to align it.

Run-time: (*––*) (R: *nest-sys ––*)

Return to the calling definition specified by *nest-sys*.

See: **3.4 The Forth text interpreter**, **3.4.5 Compilation**, **A.6.1.0460** ; .

6.1.0480   **<**   "less-than"   CORE

(*n₁ n₂ –– flag*)

flag is true if and only if n_1 is less than n_2.

See: **6.1.2340** U<.

6.1.0490   **<#**   "less-number-sign"   CORE

(––)

Initialize the pictured numeric output conversion process.

See: **6.1.0030** #, **6.1.0040** #>, **6.1.0050** #S.

6.1.0530   **=**   "equals"   CORE

(x_1 x_2 –– flag)

flag is true if and only if x_1 is bit-for-bit the same as x_2.

6.1.0540   **>**   "greater-than"   CORE

(n_1 n_2 –– flag)

flag is true if and only if n_1 is greater than n_2.

See: **6.2.2350** U>.

6.1.0550   **>BODY**   "to-body"   CORE

(xt –– a-addr)

a-addr is the data-field address corresponding to xt. An ambiguous condition exists if xt
is not for a word defined via **CREATE**.

See: **3.3.3 Data space**, **A.6.1.0550** >BODY.

6.1.0560   **>IN**   "to-in"   CORE

(–– a-addr)

a-addr is the address of a cell containing the offset in characters from the start of the
input buffer to the start of the parse area.

6.1.0570   **>NUMBER**   "to-number"   CORE

(ud_1 c-addr$_1$ u_1 –– ud_2 c-addr$_2$ u_2)

ud_2 is the unsigned result of converting the characters within the string specified by
c-addr$_1$ u_1 into digits, using the number in **BASE**, and adding each into ud_1 after multi-
plying ud_1 by the number in **BASE**. Conversion continues left-to-right until a character
that is not convertible, including any "+" or "-", is encountered or the string is entirely
converted. c-addr$_2$ is the location of the first unconverted character or the first character

---

past the end of the string if the string was entirely converted. u_2 is the number of uncon-
verted characters in the string. An ambiguous condition exists if ud_2 overflows during
the conversion.

See: **3.2.1.2 Digit conversion**.

6.1.0580	**>R**	"to-r"	CORE

Interpretation: Interpretation semantics for this word are undefined.

Execution: (*x* −−) (R: −− *x*)

Move *x* to the return stack.

See: **3.2.3.3 Return stack**, **6.1.2060** R>, **6.1.2070** R@, **6.2.0340** 2>R, **6.2.0410** 2R>, **6.2.0415** 2R@.

6.1.0630	**?DUP**	"question-dupe"	CORE

(*x* −− *0* | *x x*)

Duplicate *x* if it is non-zero.

6.1.0650	**@**	"fetch"	CORE

(*a-addr* −− *x*)

x is the value stored at *a-addr*.

See: **3.3.3.1 Address alignment**.

6.1.0670	**ABORT**		CORE

(*i* * *x* −−) (R: *j* * *x* −−)

Empty the data stack and perform the function of **QUIT**, which includes emptying the
return stack, without displaying a message.

See: **9.6.2.0670** ABORT.

6.1.0680	**ABORT"**	"abort-quote"	CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (*"ccc⟨quote⟩"* −−)

Parse *ccc* delimited by a **"** (double-quote). Append the run-time semantics given below
to the current definition.

Run-time: ($i * x\ x_1$ −− | $i * x$) (R: $j * x$ −− | $j * x$)

Remove x_1 from the stack. If any bit of x_1 is not zero, display *ccc* and perform an implementation-defined abort sequence that includes the function of **ABORT**.

See: **3.4.1 Parsing**, **9.6.2.0680** ABORT", **A.6.1.0680** ABORT".

6.1.0690	**ABS**	"abs"	CORE

(n −− u)

u is the absolute value of *n*.

6.1.0695	**ACCEPT**		CORE

(*c-addr* $+n_1$ −− $+n_2$)

Receive a string of at most $+n_1$ characters. An ambiguous condition exists if $+n_1$ is zero or greater than 32,767. Display graphic characters as they are received. A program that depends on the presence or absence of non-graphic characters in the string has an environmental dependency. The editing functions, if any, that the system performs in order to construct the string are implementation-defined.

Input terminates when an implementation-defined line terminator is received. When input terminates, nothing is appended to the string, and the display is maintained in an implementation-defined way.

$+n_2$ is the length of the string stored at *c-addr*.

See: **A.6.1.0695** ACCEPT.

6.1.0705	**ALIGN**		CORE

(−−)

If the data-space pointer is not aligned, reserve enough space to align it.

See: **3.3.3 Data space**, **3.3.3.1 Address alignment**, **A.6.1.0705** ALIGN.

6.1.0706	**ALIGNED**		CORE

(*addr* −− *a-addr*)

a-addr is the first aligned address greater than or equal to *addr*.

See: **3.3.3.1 Address alignment**, **6.1.0705** ALIGN.

6.1.0710   **ALLOT**   CORE

(*n* −−)

If *n* is greater than zero, reserve *n* address units of data space. If *n* is less than zero, release | *n* | address units of data space. If *n* is zero, leave the data-space pointer unchanged.

If the data-space pointer is aligned and *n* is a multiple of the size of a cell when **ALLOT** begins execution, it will remain aligned when **ALLOT** finishes execution.

If the data-space pointer is character aligned and *n* is a multiple of the size of a character when **ALLOT** begins execution, it will remain character aligned when **ALLOT** finishes execution.

See: **3.3.3 Data space**.

6.1.0720   **AND**   CORE

(x_1 x_2 −− x_3)

x_3 is the bit-by-bit logical "and" of x_1 with x_2.

6.1.0750   **BASE**   CORE

(−− *a-addr*)

a-addr is the address of a cell containing the current number-conversion radix {{2...36}}.

6.1.0760   **BEGIN**   CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: −− *dest*)

Put the next location for a transfer of control, *dest*, onto the control flow stack. Append the run-time semantics given below to the current definition.

Run-time: (−−)

Continue execution.

See: **3.2.3.2 Control-flow stack**, **6.1.2140** REPEAT, **6.1.2390** UNTIL, **6.1.2430** WHILE, **A.6.1.0760** BEGIN.

6.1.0770   **BL**   "b-l"   CORE

(−− *char*)

char is the character value for a space.

See: **A.6.1.0770** BL.

| 6.1.0850 | `C!` | "c-store" | CORE |

(*char c-addr* ––)

Store *char* at *c-addr*. When character size is smaller than cell size, only the number of low-order bits corresponding to character size are transferred.

See: **3.3.3.1 Address alignment**.

| 6.1.0860 | `C,` | "c-comma" | CORE |

(*char* ––)

Reserve space for one character in the data space and store *char* in the space. If the data-space pointer is character aligned when `C,` begins execution, it will remain character aligned when `C,` finishes execution. An ambiguous condition exists if the data-space pointer is not character-aligned prior to execution of `C,` .

See: **3.3.3 Data space**, **3.3.3.1 Address alignment**.

| 6.1.0870 | `C@` | "c-fetch" | CORE |

(*c-addr* –– *char*)

Fetch the character stored at *c-addr*. When the cell size is greater than character size, the unused high-order bits are all zeroes.

See: **3.3.3.1 Address alignment**.

| 6.1.0880 | `CELL+` | "cell-plus" | CORE |

(*a-addr$_1$* –– *a-addr$_2$*)

Add the size in address units of a cell to *a-addr$_1$*, giving *a-addr$_2$*.

See: **3.3.3.1 Address alignment**, **A.6.1.0880** `CELL+`.

| 6.1.0890 | `CELLS` | | CORE |

(*n$_1$* –– *n$_2$*)

n$_2$ is the size in address units of *n$_1$* cells.

See: **A.6.1.0880** `CELL+`, **A.6.1.0890** `CELLS`.

6.1.0895   **CHAR**   "char"   CORE

(*"⟨spaces⟩name"* –– *char*)

Skip leading space delimiters. Parse *name* delimited by a space. Put the value of its first character onto the stack.

See: **3.4.1 Parsing**, **6.1.2520** `[CHAR]`, **A.6.1.0895** `CHAR`.

6.1.0897   **CHAR+**   "char-plus"   CORE

(*c-addr$_1$* –– *c-addr$_2$*)

Add the size in address units of a character to *c-addr$_1$*, giving *c-addr$_2$*.

See: **3.3.3.1 Address alignment**.

6.1.0898   **CHARS**   "chars"   CORE

(*n$_1$* –– *n$_2$*)

n$_2$ is the size in address units of *n$_1$* characters.

6.1.0950   **CONSTANT**   CORE

(*x "⟨spaces⟩name"* ––)

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below.

name is referred to as a "constant".

name Execution: (–– *x*)

Place *x* on the stack.

See: **3.4.1 Parsing**, **A.6.1.0950** `CONSTANT`.

6.1.0980   **COUNT**   CORE

(*c-addr$_1$* –– *c-addr$_2$ u*)

Return the character string specification for the counted string stored at *c-addr$_1$*. *c-addr$_2$* is the address of the first character after *c-addr$_1$*. *u* is the contents of the character at *c-addr$_1$*, which is the length in characters of the string at *c-addr$_2$*.

6.1.0990   **CR**   "c-r"   CORE

(––)

Cause subsequent output to appear at the beginning of the next line.

6.1.1000   **CREATE**   CORE

("⟨spaces⟩name" ––)

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition
for *name* with the execution semantics defined below. If the data-space pointer is not
aligned, reserve enough data space to align it. The new data-space pointer defines *name*'s
data field. **CREATE** does not allocate data space in *name*'s data field.

name Execution: (–– *a-addr*)

a-addr is the address of *name*'s data field. The execution semantics of *name* may be
extended by using **DOES>**.

See: **3.3.3 Data space**, **6.1.1250** DOES>, **A.6.1.1000** CREATE.

6.1.1170   **DECIMAL**   CORE

(––)

Set the numeric conversion radix to ten (decimal).

6.1.1200   **DEPTH**   CORE

(–– +*n*)

+*n* is the number of single-cell values contained in the data stack before +*n* was placed
on the stack.

6.1.1240   **DO**   CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: –– *do-sys*)

Place *do-sys* onto the control-flow stack. Append the run-time semantics given below
to the current definition. The semantics are incomplete until resolved by a consumer of
do-sys such as **LOOP**.

Run-time: (n_1 | u_1 n_2 | u_2 ––) (R: –– *loop-sys*)

Set up loop control parameters with index $n_2 \mid u_2$ and limit $n_1 \mid u_1$. An ambiguous condition exists if $n_1 \mid u_1$ and $n_2 \mid u_2$ are not both the same type. Anything already on the return stack becomes unavailable until the loop-control parameters are discarded.

See: **3.2.3.2 Control-flow stack**, **6.1.0140** +LOOP, **6.1.1800** LOOP, **A.6.1.1240** DO.

6.1.1250   **DOES>**   "does"   CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: *colon-sys$_1$ –– colon-sys$_2$*)

Append the run-time semantics below to the current definition. Whether or not the current definition is rendered findable in the dictionary by the compilation of **DOES>** is implementation defined. Consume *colon-sys$_1$* and produce *colon-sys$_2$*. Append the initiation semantics given below to the current definition.

Run-time: (––) (R: *nest-sys$_1$* ––)

Replace the execution semantics of the most recent definition, referred to as *name*, with the *name* execution semantics given below. Return control to the calling definition specified by *nest-sys$_1$*. An ambiguous condition exists if *name* was not defined with **CREATE** or a user-defined word that calls **CREATE**.

Initiation: ($i*x$ –– $i*x$ *a-addr*) (R: –– *nest-sys$_2$*)

Save implementation-dependent information *nest-sys$_2$* about the calling definition. Place *name*'s data field address on the stack. The stack effects $i*x$ represent arguments to *name*.

name Execution: ($i*x$ –– $j*x$)

Execute the portion of the definition that begins with the initiation semantics appended by the **DOES>** which modified *name*. The stack effects $i*x$ and $j*x$ represent arguments to and results from *name*, respectively.

See: **6.1.1000** CREATE, **A.6.1.1250** DOES>.

6.1.1260   **DROP**   CORE

(x ––)

Remove x from the stack.

6.1.1290   **DUP**   "dupe"   CORE

(x –– $x\,x$)

Duplicate x.

6.1.1310   **ELSE**   CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: *orig₁* −− *orig₂*)

Put the location of a new unresolved forward reference *orig₂* onto the control flow stack. Append the run-time semantics given below to the current definition. The semantics will be incomplete until *orig₂* is resolved (e.g., by **THEN**). Resolve the forward reference *orig₁* using the location following the appended run-time semantics.

Run-time: (−−)

Continue execution at the location given by the resolution of *orig₂*.

See: **6.1.1700** IF, **6.1.2270** THEN, **A.6.1.1310** ELSE.

6.1.1320   **EMIT**   CORE

(*x* −−)

If *x* is a graphic character in the implementation-defined character set, display *x*. The effect of **EMIT** for all other values of *x* is implementation-defined.

When passed a character whose character-defining bits have a value between hex 20 and 7E inclusive, the corresponding standard character, specified by **3.1.2.1 Graphic characters**, is displayed. Because different output devices can respond differently to control characters, programs that use control characters to perform specific functions have an environmental dependency. Each EMIT deals with only one character.

See: **6.1.2310** TYPE.

6.1.1345   **ENVIRONMENT?**   "environment-query"   CORE

(*c-addr u* −− *false* | *i* * *x true*)

c-addr is the address of a character string and *u* is the string's character count. *u* may have a value in the range from zero to an implementation-defined maximum which shall not be less than 31. The character string should contain a keyword from **3.2.6 Environmental queries** or the optional word sets to be checked for correspondence with an attribute of the present environment. If the system treats the attribute as unknown, the returned flag is *false*; otherwise, the flag is *true* and the *i* * *x* returned is of the type specified in the table for the attribute queried.

See: **A.6.1.1345** ENVIRONMENT?.

6.1.1360   **EVALUATE**   CORE

(*i* * *x c-addr u* −− *j* * *x*)

Save the current input source specification. Store minus-one (-1) in **SOURCE-ID** if it is present. Make the string described by *c-addr* and *u* both the input source and input buffer, set **>IN** to zero, and interpret. When the parse area is empty, restore the prior input source specification. Other stack effects are due to the words **EVALUATE**d.

6.1.1370   **EXECUTE**   CORE

(*i* * *x xt* −− *j* * *x*)

Remove *xt* from the stack and perform the semantics identified by it. Other stack effects are due to the word **EXECUTE**d.

See: **6.1.0070 '** , **6.1.2510** ['].

6.1.1380   **EXIT**   CORE

Interpretation: Interpretation semantics for this word are undefined.

Execution: (−−) (R: *nest-sys* −−)

Return control to the calling definition specified by *nest-sys*. Before executing **EXIT** within a do-loop, a program shall discard the loop-control parameters by executing **UNLOOP**.

See: **3.2.3.3 Return stack**, **6.1.2380** UNLOOP, **A.6.1.1380** EXIT.

6.1.1540   **FILL**   CORE

(*c-addr u char* −−)

If *u* is greater than zero, store *char* in each of *u* consecutive characters of memory beginning at *c-addr*.

6.1.1550   **FIND**   CORE

(*c-addr* −− *c-addr 0* | *xt 1* | *xt -1*)

Find the definition named in the counted string at *c-addr*. If the definition is not found, return *c-addr* and zero. If the definition is found, return its execution token *xt*. If the definition is immediate, also return one (*1*), otherwise also return minus-one (-*1*). For a given string, the values returned by **FIND** while compiling may differ from those returned while not compiling.

See: **3.4.2 Finding definition names**, **A.6.1.0070 '** , **A.6.1.1550** FIND, **A.6.1.2033** POSTPONE, **A.6.1.2510** ['].

6.1.1561   **FM/MOD**   "f-m-slash-mod"   CORE

(d_1 n_1 -- n_2 n_3)

Divide d_1 by n_1, giving the floored quotient n_3 and the remainder n_2. Input and output stack arguments are signed. An ambiguous condition exists if n_1 is zero or if the quotient lies outside the range of a single-cell signed integer.

See: **3.2.2.1 Integer division**, **6.1.2214** SM/REM, **6.1.2370** UM/MOD, **A.6.1.1561** FM/MOD.

6.1.1650   **HERE**   CORE

(-- addr)

addr is the data-space pointer.

See: **3.3.3.2 Contiguous regions**.

6.1.1670   **HOLD**   CORE

(char --)

Add char to the beginning of the pictured numeric output string. An ambiguous condition exists if **HOLD** executes outside of a **<# #>** delimited number conversion.

6.1.1680   **I**   CORE

Interpretation: Interpretation semantics for this word are undefined.

Execution: (-- n | u) (R: loop-sys -- loop-sys)

n | u is a copy of the current (innermost) loop index. An ambiguous condition exists if the loop control parameters are unavailable.

6.1.1700   **IF**   CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: -- orig)

Put the location of a new unresolved forward reference orig onto the control flow stack. Append the run-time semantics given below to the current definition. The semantics are incomplete until orig is resolved, e.g., by **THEN** or **ELSE**.

Run-time: (x --)

If all bits of x are zero, continue execution at the location specified by the resolution of orig.

See: **3.2.3.2 Control-flow stack**, **6.1.1310** ELSE, **6.1.2270** THEN, **A.6.1.1700** IF.

6.1.1710   **IMMEDIATE**   CORE

(--)

Make the most recent definition an immediate word. An ambiguous condition exists if the most recent definition does not have a name or if it was defined as a **SYNONYM**.

See: **15.6.2.2264** SYNONYM **A.6.1.1710** IMMEDIATE.

6.1.1720   **INVERT**   CORE

(x_1 -- x_2)

Invert all bits of x_1, giving its logical inverse x_2.

See: **6.1.1910** NEGATE, **6.1.0270** 0=, **A.6.1.1720** INVERT.

6.1.1730   **J**   CORE

Interpretation: Interpretation semantics for this word are undefined.

Execution: (-- $n | u$) (R: *loop-sys$_1$ loop-sys$_2$* -- *loop-sys$_1$ loop-sys$_2$*)

$n | u$ is a copy of the next-outer loop index. An ambiguous condition exists if the loop control parameters of the next-outer loop, *loop-sys$_1$*, are unavailable.

See: **A.6.1.1730** J.

6.1.1750   **KEY**   CORE

(-- *char*)

Receive one character *char*, a member of the implementation-defined character set. Keyboard events that do not correspond to such characters are discarded until a valid character is received, and those events are subsequently unavailable.

All standard characters can be received. Characters received by **KEY** are not displayed.

Any standard character returned by **KEY** has the numeric value specified in **3.1.2.1 Graphic characters**. Programs that require the ability to receive control characters have an environmental dependency.

See: **10.6.2.1305** EKEY, **10.6.2.1307** EKEY?, **10.6.1.1755** KEY?, **A.6.1.1750** KEY.

6.1.1760   **LEAVE**   CORE

Interpretation: Interpretation semantics for this word are undefined.

Execution: (−−) (R: *loop-sys* −−)

Discard the current loop control parameters. An ambiguous condition exists if they are unavailable. Continue execution immediately following the innermost syntactically enclosing **DO**...**LOOP** or **DO**...**+LOOP**.

See: **3.2.3.3 Return stack**, **6.1.0140** +LOOP, **6.1.1800** LOOP, **A.6.1.1760** LEAVE.

6.1.1780   **LITERAL**   CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (*x* −−)

Append the run-time semantics given below to the current definition.

Run-time: (−− *x*)

Place *x* on the stack.

See: **A.6.1.1780** LITERAL.

6.1.1800   **LOOP**   CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: *do-sys* −−)

Append the run-time semantics given below to the current definition. Resolve the destination of all unresolved occurrences of **LEAVE** between the location given by *do-sys* and the next location for a transfer of control, to execute the words following the **LOOP**.

Run-time: (−−) (R: $loop\text{-}sys_1$ −− | $loop\text{-}sys_2$)

An ambiguous condition exists if the loop control parameters are unavailable. Add one to the loop index. If the loop index is then equal to the loop limit, discard the loop parameters and continue execution immediately following the loop. Otherwise continue execution at the beginning of the loop.

See: **6.1.1240** DO, **6.1.1680** I, **6.1.1760** LEAVE**A.6.1.1800** LOOP.

6.1.1805   **LSHIFT**   "l-shift"   CORE

(x_1 *u* −− x_2)

Perform a logical left shift of *u* bit-places on x_1, giving x_2. Put zeroes into the least significant bits vacated by the shift. An ambiguous condition exists if *u* is greater than or equal to the number of bits in a cell.

6.1.1810   **M***   "m-star"   CORE

(n_1 n_2 −− d)

d is the signed product of n_1 times n_2.

See: **A.6.1.1810** M*.

6.1.1870   **MAX**   CORE

(n_1 n_2 −− n_3)

n_3 is the greater of n_1 and n_2.

6.1.1880   **MIN**   CORE

(n_1 n_2 −− n_3)

n_3 is the lesser of n_1 and n_2.

6.1.1890   **MOD**   CORE

(n_1 n_2 −− n_3)

Divide n_1 by n_2, giving the single-cell remainder n_3. An ambiguous condition exists if n_2 is zero. If n_1 and n_2 differ in sign, the implementation-defined result returned will be the same as that returned by either the phrase **>R S>D R> FM/MOD DROP** or the phrase **>R S>D R> SM/REM DROP**.

See: **3.2.2.1 Integer division**.

6.1.1900   **MOVE**   CORE

($addr_1$ $addr_2$ u −−   )

If u is greater than zero, copy the contents of u consecutive address units at $addr_1$ to the u consecutive address units at $addr_2$. After **MOVE** completes, the u consecutive address units at $addr_2$ contain exactly what the u consecutive address units at $addr_1$ contained before the move.

See: **17.6.1.0910** CMOVE, **17.6.1.0920** CMOVE>, **A.6.1.1900** MOVE.

6.1.1910   **NEGATE**   CORE

(n_1 −− n_2)

Negate n_1, giving its arithmetic inverse n_2.

See: **6.1.1720** INVERT, **6.1.0270** 0=.

6.1.1980   **OR**   CORE

(x_1 x_2 $--$ x_3)

x_3 is the bit-by-bit inclusive-or of x_1 with x_2.

6.1.1990   **OVER**   CORE

(x_1 x_2 $--$ x_1 x_2 x_1)

Place a copy of x_1 on top of the stack.

6.1.2033   **POSTPONE**   CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (*"⟨spaces⟩name"* $--$)

Skip leading space delimiters. Parse *name* delimited by a space. Find *name*. Append the compilation semantics of *name* to the current definition. An ambiguous condition exists if *name* is not found.

See: **3.4.1 Parsing**, **A.6.1.2033** POSTPONE.

6.1.2050   **QUIT**   CORE

($--$) (R: $i*x$ $--$)

Empty the return stack, store zero in **SOURCE-ID** if it is present, make the user input device the input source, and enter interpretation state. Do not display a message. Repeat the following:

– Accept a line from the input source into the input buffer, set **>IN** to zero, and interpret.

– Display the implementation-defined system prompt if in interpretation state, all processing has been completed, and no ambiguous condition exists.

See: **3.4 The Forth text interpreter**.

6.1.2060   **R>**   "r-from"   CORE

Interpretation: Interpretation semantics for this word are undefined.

Execution: ($--$ x) (R: x $--$)

Move *x* from the return stack to the data stack.

See: **3.2.3.3 Return stack**, **6.1.0580** >R, **6.1.2070** R@, **6.2.0340** 2>R, **6.2.0410** 2R>, **6.2.0415** 2R@.

6.1.2070   **R@**   "r-fetch"   CORE

Interpretation: Interpretation semantics for this word are undefined.

Execution: ($--$ x) (R: x $--$ x)

Copy x from the return stack to the data stack.

See: **3.2.3.3 Return stack**, **6.1.0580** >R, **6.1.2060** R>, **6.2.0340** 2>R, **6.2.0410** 2R>, **6.2.0415** 2R@.

6.1.2120   **RECURSE**   CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: ($--$)

Append the execution semantics of the current definition to the current definition. An ambiguous condition exists if **RECURSE** appears in a definition after **DOES>**.

See: **6.1.1250** DOES>, **6.1.2120** RECURSE, **A.6.1.2120** RECURSE.

6.1.2140   **REPEAT**   CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: *orig dest* $--$)

Append the run-time semantics given below to the current definition, resolving the backward reference *dest*. Resolve the forward reference *orig* using the location following the appended run-time semantics.

Run-time: ($--$)

Continue execution at the location given by *dest*.

See: **6.1.0760** BEGIN, **6.1.2430** WHILE, **A.6.1.2140** REPEAT.

6.1.2160   **ROT**   "rote"   CORE

(x_1 x_2 x_3 $--$ x_2 x_3 x_1)

Rotate the top three stack entries.

6.1.2162   **RSHIFT**   "r-shift"   CORE

(x_1 u $--$ x_2)

Perform a logical right shift of u bit-places on x_1, giving x_2. Put zeroes into the most significant bits vacated by the shift. An ambiguous condition exists if u is greater than or equal to the number of bits in a cell.

6.1.2165   **S"**   "s-quote"   CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (*"ccc⟨quote⟩"* ––)

Parse *ccc* delimited by **"** (double-quote). Append the run-time semantics given below to the current definition.

Run-time: (–– *c-addr u*)

Return *c-addr* and *u* describing a string consisting of the characters *ccc*. A program shall not alter the returned string.

See: **3.4.1 Parsing**, **6.2.0855** C", **11.6.1.2165** S", **6.2.2266** S\", **A.6.1.2165** S".

6.1.2170   **S>D**   "s-to-d"   CORE

(*n* –– *d*)

Convert the number *n* to the double-cell number *d* with the same numerical value.

6.1.2210   **SIGN**   CORE

(*n* ––)

If *n* is negative, add a minus sign to the beginning of the pictured numeric output string. An ambiguous condition exists if **SIGN** executes outside of a **<# #>** delimited number conversion.

6.1.2214   **SM/REM**   "s-m-slash-rem"   CORE

(*d₁ n₁* –– *n₂ n₃*)

Divide *d₁* by *n₁*, giving the symmetric quotient *n₃* and the remainder *n₂*. Input and output stack arguments are signed. An ambiguous condition exists if *n₁* is zero or if the quotient lies outside the range of a single-cell signed integer.

See: **3.2.2.1 Integer division**, **6.1.1561** FM/MOD, **6.1.2370** UM/MOD, **A.6.1.2214** SM/REM.

6.1.2216   **SOURCE**   CORE

(–– *c-addr u*)

c-addr is the address of, and *u* is the number of characters in, the input buffer.

See: **A.6.1.2216** SOURCE.

6.1.2220   **SPACE**   CORE

(--)

Display one space.

6.1.2230   **SPACES**   CORE

(*n* --)

If *n* is greater than zero, display *n* spaces.

6.1.2250   **STATE**   CORE

(-- *a-addr*)

a-addr is the address of a cell containing the compilation-state flag. **STATE** is *true* when in compilation state, *false* otherwise. The *true* value in **STATE** is non-zero, but is otherwise implementation-defined. Only the following standard words alter the value in **STATE**: **:** (colon), **;** (semicolon), **ABORT**, **QUIT**, **:NONAME**, **[** (left-bracket), **]** (right-bracket).

Note: A program shall not directly alter the contents of **STATE**.

See: **3.4 The Forth text interpreter**, **6.1.0450 :** , **6.1.0460 ;** **6.1.0670** ABORT, **6.1.2050** QUIT, **6.1.2500** [, **6.1.2540**], **6.2.0455** :NONAME, **15.6.2.2250** STATE, **A.6.1.2250** STATE.

6.1.2260   **SWAP**   CORE

(x_1 x_2 -- x_2 x_1)

Exchange the top two stack items.

6.1.2270   **THEN**   CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: *orig* --)

Append the run-time semantics given below to the current definition. Resolve the forward reference *orig* using the location of the appended run-time semantics.

Run-time: (--)

Continue execution.

See: **6.1.1310** ELSE, **6.1.1700** IF, **A.6.1.2270** THEN.

6.1.2310   **TYPE**   CORE

(*c-addr u --*)

If *u* is greater than zero, display the character string specified by *c-addr* and *u*.

When passed a character in a character string whose character-defining bits have a value between hex 20 and 7E inclusive, the corresponding standard character, specified by **3.1.2.1 Graphic characters**, is displayed. Because different output devices can respond differently to control characters, programs that use control characters to perform specific functions have an environmental dependency.

See: **6.1.1320** EMIT.

6.1.2320   **U.**   "u-dot"   CORE

(*u --*)

Display *u* in free field format.

6.1.2340   **U<**   "u-less-than"   CORE

(*u₁ u₂ -- flag*)

flag is true if and only if u_1 is less than u_2.

See: **6.1.0480** <.

6.1.2360   **UM***   "u-m-star"   CORE

(*u₁ u₂ -- ud*)

Multiply u_1 by u_2, giving the unsigned double-cell product *ud*. All values and arithmetic are unsigned.

6.1.2370   **UM/MOD**   "u-m-slash-mod"   CORE

(*ud u₁ -- u₂ u₃*)

Divide *ud* by u_1, giving the quotient u_3 and the remainder u_2. All values and arithmetic are unsigned. An ambiguous condition exists if u_1 is zero or if the quotient lies outside the range of a single-cell unsigned integer.

See: **3.2.2.1 Integer division**, **6.1.1561** FM/MOD, **6.1.2214** SM/REM.

6.1.2380   **UNLOOP**   CORE

Interpretation: Interpretation semantics for this word are undefined.

Execution: (––) (R: *loop-sys* ––)

Discard the loop-control parameters for the current nesting level. An **UNLOOP** is required for each nesting level before the definition may be **EXIT**ed. An ambiguous condition exists if the loop-control parameters are unavailable.

See: **3.2.3.3 Return stack**, **A.6.1.2380** UNLOOP.

6.1.2390   **UNTIL**   CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: *dest* ––)

Append the run-time semantics given below to the current definition, resolving the backward reference *dest*.

Run-time: (*x* ––)

If all bits of *x* are zero, continue execution at the location specified by *dest*.

See: **6.1.0760** BEGIN, **A.6.1.2390** UNTIL.

6.1.2410   **VARIABLE**   CORE

(*"⟨spaces⟩name"* ––)

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Reserve one cell of data space at an aligned address.

name is referred to as a "variable".

name Execution: (–– *a-addr*)

a-addr is the address of the reserved cell. A program is responsible for initializing the contents of the reserved cell.

See: **3.4.1 Parsing**, **A.6.1.2410** VARIABLE.

6.1.2430   **`WHILE`**   CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: *dest* −− *orig dest*)

Put the location of a new unresolved forward reference *orig* onto the control flow stack, under the existing *dest*. Append the run-time semantics given below to the current definition. The semantics are incomplete until *orig* and *dest* are resolved (e.g., by **`REPEAT`**).

Run-time: (*x* −−)

If all bits of *x* are zero, continue execution at the location specified by the resolution of *orig*.

See: **A.6.1.2430** `WHILE`.

6.1.2450   **`WORD`**   CORE

(*char* "⟨*chars*⟩*ccc*⟨*char*⟩" −− *c-addr*)

Skip leading delimiters. Parse characters *ccc* delimited by *char*. An ambiguous condition exists if the length of the parsed string is greater than the implementation-defined length of a counted string.

c-addr is the address of a transient region containing the parsed word as a counted string. If the parse area was empty or contained no characters other than the delimiter, the resulting string has a zero length. A program may replace characters within the string.

See: **3.3.3.6 Other transient regions**, **3.4.1 Parsing**, **A.6.1.2450** `WORD`.

6.1.2490   **`XOR`**   "x-or"   CORE

(x_1 x_2 −− x_3)

x_3 is the bit-by-bit exclusive-or of x_1 with x_2.

6.1.2500   **`[`**   "left-bracket"   CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: Perform the execution semantics given below.

Execution: (−−)

Enter interpretation state. **`[`** is an immediate word.

See: **3.4 The Forth text interpreter**, **3.4.5 Compilation**, **6.1.2540** `]`, **A.6.1.2500** `[`.

6.1.2510   **[']**   "bracket-tick"   CORE

Interpretation: Interpretation semantics for this word are undefined.

  Compilation: (*"⟨spaces⟩name"* −−)

          Skip leading space delimiters. Parse *name* delimited by a space. Find *name*. Append the
          run-time semantics given below to the current definition.

          An ambiguous condition exists if *name* is not found.

    Run-time: (−− *xt*)

          Place *name*'s execution token *xt* on the stack. The execution token returned by the com-
          piled phrase "**[']** X" is the same value returned by "**'** X" outside of compilation state.

      See: **3.4.1 Parsing**, **6.1.1550** FIND, **A.6.1.0070** **'** **A.6.1.2033** POSTPONE,
          A.6.1.2510 ['].

6.1.2520   **[CHAR]**   "bracket-char"   CORE

Interpretation: Interpretation semantics for this word are undefined.

  Compilation: (*"⟨spaces⟩name"* −−)

          Skip leading space delimiters. Parse *name* delimited by a space. Append the run-time
          semantics given below to the current definition.

    Run-time: (−− *char*)

          Place *char*, the value of the first character of *name*, on the stack.

      See: **3.4.1 Parsing**, **6.1.0895** CHAR, **A.6.1.2520** [CHAR].

6.1.2540   **]**   "right-bracket"   CORE

          (−−)

          Enter compilation state.

      See: **3.4 The Forth text interpreter**, **3.4.5 Compilation**, **6.1.2500** [, **A.6.1.2540**].

6.2 Core extension words

6.2.0200   **. (**   "dot-paren"   CORE EXT

Compilation: Perform the execution semantics given below.

Execution: (*"ccc⟨paren⟩"* --)

Parse and display *ccc* delimited by) (right parenthesis). **. (** is an immediate word.

See: **3.4.1 Parsing**, **6.1.0190** **. "**, **A.6.2.0200** **. (**.

6.2.0210   **.R**   "dot-r"   CORE EXT

(n_1 n_2 --)

Display n_1 right aligned in a field n_2 characters wide. If the number of characters required to display n_1 is greater than n_2, all digits are displayed with no leading spaces in a field as wide as necessary.

See: **A.6.2.0210** **.R**.

6.2.0260   **0<>**   "zero-not-equals"   CORE EXT

(*x* -- *flag*)

flag is true if and only if *x* is not equal to zero.

6.2.0280   **0>**   "zero-greater"   CORE EXT

(*n* -- *flag*)

flag is true if and only if *n* is greater than zero.

6.2.0340   **2>R**   "two-to-r"   CORE EXT

Interpretation: Interpretation semantics for this word are undefined.

Execution: (x_1 x_2 --) (R: -- x_1 x_2)

Transfer cell pair x_1 x_2 to the return stack. Semantically equivalent to **SWAP >R >R**.

See: **3.2.3.3 Return stack**, **6.1.0580** >R, **6.1.2060** R>, **6.1.2070** R@, **6.2.0410** 2R>, **6.2.0415** 2R@, **A.6.2.0340** 2>R.

6.2.0410   **2R>**   "two-r-from"   CORE EXT

Interpretation: Interpretation semantics for this word are undefined.

Execution: ($--$ x_1 x_2) (R: x_1 x_2 $--$)

Transfer cell pair x_1 x_2 from the return stack. Semantically equivalent to **R> R> SWAP**.

See: **3.2.3.3 Return stack**, **6.1.0580** >R **6.1.2060** R> **6.1.2070** R@ **6.2.0340** 2>R, **6.2.0415** 2R@, **A.6.2.0410** 2R>.

6.2.0415   **2R@**   "two-r-fetch"   CORE EXT

Interpretation: Interpretation semantics for this word are undefined.

Execution: ($--$ x_1 x_2) (R: x_1 x_2 $--$ x_1 x_2)

Copy cell pair x_1 x_2 from the return stack. Semantically equivalent to **R> R> 2DUP >R >R SWAP**.

See: **3.2.3.3 Return stack**, **6.1.0580** >R, **6.1.2060** R>, **6.1.2070** R@, **6.2.0340** 2>R, **6.2.0410** 2R>.

6.2.0455   **:NONAME**   "colon-no-name"   CORE EXT

(C: $--$ *colon-sys*) (S: $--$ *xt*)

Create an execution token *xt*, enter compilation state and start the current definition, producing *colon-sys*. Append the initiation semantics given below to the current definition.

The execution semantics of *xt* will be determined by the words compiled into the body of the definition. This definition can be executed later by using *xt* **EXECUTE**.

If the control-flow stack is implemented using the data stack, *colon-sys* shall be the topmost item on the data stack. See **3.2.3.2 Control-flow stack**.

Initiation: ($i*x$ $--$ $i*x$) (R: $--$ *nest-sys*)

Save implementation-dependent information *nest-sys* about the calling definition. The stack effects $i*x$ represent arguments to *xt*.

xt Execution: ($i*x$ $--$ $j*x$)

Execute the definition specified by *xt*. The stack effects $i*x$ and $j*x$ represent arguments to and results from *xt*, respectively.

See: **A.6.2.0455** :NONAME.

6.2.0500   **<>**   "not-equals"   CORE EXT

(x_1 x_2 $--$ *flag*)

flag is true if and only if x_1 is not bit-for-bit the same as x_2.

6.2.0620   **?DO**   "question-do"   CORE EXT

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: $--$ *do-sys*)

Put *do-sys* onto the control-flow stack. Append the run-time semantics given below to the current definition. The semantics are incomplete until resolved by a consumer of *do-sys* such as **LOOP**.

Run-time: (n_1 | u_1 n_2 | u_2 $--$) (R: $--$ *loop-sys*)

If n_1 | u_1 is equal to n_2 | u_2, continue execution at the location given by the consumer of *do-sys*. Otherwise set up loop control parameters with index n_2 | u_2 and limit n_1 | u_1 and continue executing immediately following **?DO**. Anything already on the return stack becomes unavailable until the loop control parameters are discarded. An ambiguous condition exists if n_1 | u_1 and n_2 | u_2 are not both of the same type.

See: **3.2.3.2 Control-flow stack**, **6.1.0140** +LOOP, **6.1.1240** DO, **6.1.1680** I, **6.1.1760** LEAVE, **6.1.1800** LOOP, **6.1.2380** UNLOOP, **A.6.2.0620** ?DO.

6.2.0698   **ACTION-OF**   CORE EXT

Interpretation: (*"⟨spaces⟩name"* $--$ *xt*)

Skip leading spaces and parse *name* delimited by a space. *xt* is the execution token that *name* is set to execute. An ambiguous condition exists if *name* was not defined by **DEFER**, or if the *name* has not been set to execute an *xt*.

Compilation: (*"⟨spaces⟩name"* $--$)

Skip leading spaces and parse *name* delimited by a space. Append the run-time semantics given below to the current definition. An ambiguous condition exists if *name* was not defined by **DEFER**.

Run-time: ($--$ *xt*)

xt is the execution token that *name* is set to execute. An ambiguous condition exists if *name* has not been set to execute an *xt*.

An ambiguous condition exists if **POSTPONE**, **[COMPILE]**, **[']** or **'** is applied to **ACTION-OF**.

See: **6.2.1173** DEFER, **6.2.1175** DEFER!, **6.2.1177** DEFER@, **6.2.1725** IS.

6.2.0700   **AGAIN**   CORE EXT

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: *dest* ––)

Append the run-time semantics given below to the current definition, resolving the backward reference *dest*.

Run-time: (––)

Continue execution at the location specified by *dest*. If no other control flow words are used, any program code after **AGAIN** will not be executed.

See: **6.1.0760** BEGIN, **A.6.2.0700** AGAIN.

6.2.0825   **BUFFER:**   "buffer-colon"   CORE EXT

(*u* "⟨*spaces*⟩*name*" ––)

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name*, with the execution semantics defined below. Reserve *u* address units at an aligned address. Contiguity of this region with any other region is undefined.

name Execution: (–– *a-addr*)

a-addr is the address of the space reserved by **BUFFER:** when it defined *name*. The program is responsible for initializing the contents.

See: **A.6.2.0825** BUFFER:.

6.2.0855   **C"**   "c-quote"   CORE EXT

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (*"ccc⟨quote⟩"* ––)

Parse *ccc* delimited by " (double-quote) and append the run-time semantics given below to the current definition.

Run-time: (–– *c-addr*)

Return *c-addr*, a counted string consisting of the characters *ccc*. A program shall not alter the returned string.

See: **3.4.1 Parsing**, **6.1.2165** S", **11.6.1.2165** S", **A.6.2.0855** C".

6.2.0873   **CASE**   CORE EXT

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: −− *case-sys*)

Mark the start of the **CASE**. . . **OF**. . . **ENDOF**. . . **ENDCASE** structure. Append the run-time semantics given below to the current definition.

Run-time: (−−)

Continue execution.

See: **6.2.1342** ENDCASE, **6.2.1343** ENDOF, **6.2.1950** OF, **A.6.2.0873** CASE.

6.2.0945   **COMPILE,**   "compile-comma"   CORE EXT

Interpretation: Interpretation semantics for this word are undefined.

Execution: (*xt* −−)

Append the execution semantics of the definition represented by *xt* to the execution semantics of the current definition.

See: **A.6.2.0945** COMPILE,.

6.2.1173   **DEFER**   CORE EXT

(*"⟨spaces⟩name"* −−)

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below.

name Execution: ($i * x$ −− $j * x$)

Execute the *xt* that *name* is set to execute. An ambiguous condition exists if *name* has not been set to execute an *xt*.

See: **6.2.0698** ACTION-OF, **6.2.1175** DEFER!, **6.2.1177** DEFER@, **6.2.1725** IS.

6.2.1175   **DEFER!**   "defer-store"   CORE EXT

(xt_2 xt_1 −−)

Set the word xt_1 to execute xt_2. An ambiguous condition exists if xt_1 is not for a word defined by **DEFER**.

See: **6.2.0698** ACTION-OF, **6.2.1173** DEFER, **6.2.1177** DEFER@, **6.2.1725** IS.

6.2.1177   **DEFER@**   "defer-fetch"   CORE EXT

(xt_1 −− xt_2)

xt_2 is the execution token xt_1 is set to execute. An ambiguous condition exists if xt_1 is not the execution token of a word defined by **DEFER**, or if xt_1 has not been set to execute an xt.

See: **6.2.0698** ACTION-OF, **6.2.1173** DEFER, **6.2.1175** DEFER!, **6.2.1725** IS.

6.2.1342   **ENDCASE**   "end-case"   CORE EXT

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: *case-sys* −−)

Mark the end of the **CASE**...**OF**...**ENDOF**...**ENDCASE** structure. Use *case-sys* to resolve the entire structure. Append the run-time semantics given below to the current definition.

Run-time: (x −−)

Discard the case selector x and continue execution.

See: **6.2.0873** CASE, **6.2.1343** ENDOF, **6.2.1950** OF, **A.6.2.1342** ENDCASE.

6.2.1343   **ENDOF**   "end-of"   CORE EXT

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: *case-sys$_1$ of-sys* −− *case-sys$_2$*)

Mark the end of the **OF**...**ENDOF** part of the **CASE** structure. The next location for a transfer of control resolves the reference given by *of-sys*. Append the run-time semantics given below to the current definition. Replace *case-sys$_1$* with *case-sys$_2$* on the control-flow stack, to be resolved by **ENDCASE**.

Run-time: (−−)

Continue execution at the location specified by the consumer of *case-sys$_2$*.

See: **6.2.0873** CASE, **6.2.1342** ENDCASE, **6.2.1950** OF, **A.6.2.1343** ENDOF.

6.2.1350   **ERASE**   CORE EXT

(*addr u* −−)

If u is greater than zero, clear all bits in each of u consecutive address units of memory beginning at *addr*.

6.2.1485   **FALSE**   CORE EXT

(-- *false*)

Return a *false* flag.

See: **3.1.3.1 Flags**

6.2.1660   **HEX**   CORE EXT

(--)

Set contents of **BASE** to sixteen.

6.2.1675   **HOLDS**   CORE EXT

(*c-addr u* --)

Adds the string represented by *c-addr u* to the pictured numeric output string. An ambiguous condition exists if **HOLDS** executes outside of a **<# #>** delimited number conversion.

See: **6.1.1670** HOLD.

6.2.1725   **IS**   CORE EXT

Interpretation: (*xt* "⟨*spaces*⟩*name*" --)

Skip leading spaces and parse *name* delimited by a space. Set *name* to execute *xt*.

An ambiguous condition exists if *name* was not defined by **DEFER**.

Compilation: ("⟨*spaces*⟩*name*" --)

Skip leading spaces and parse *name* delimited by a space. Append the run-time semantics given below to the current definition. An ambiguous condition exists if *name* was not defined by **DEFER**.

Run-time: (*xt* --)

Set *name* to execute *xt*.

An ambiguous condition exists if **POSTPONE**, **[COMPILE]**, **[']** or **'** is applied to **IS**.

See: **6.2.0698** ACTION-OF, **6.2.1173** DEFER, **6.2.1175** DEFER!, **6.2.1177** DEFER@.

6.2.1850   **MARKER**   CORE EXT

(*"⟨spaces⟩name"* −−)

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below.

name Execution: (−−)

Restore all dictionary allocation and search order pointers to the state they had just prior to the definition of *name*. Remove the definition of *name* and all subsequent definitions. Restoration of any structures still existing that could refer to deleted definitions or deallocated data space is not necessarily provided. No other contextual information such as numeric base is affected.

See: **3.4.1 Parsing**, **15.6.2.1580** FORGET, **A.6.2.1850** MARKER.

6.2.1930   **NIP**   CORE EXT

(x_1 x_2 −− x_2)

Drop the first item below the top of stack.

6.2.1950   **OF**   CORE EXT

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: −− *of-sys*)

Put *of-sys* onto the control flow stack. Append the run-time semantics given below to the current definition. The semantics are incomplete until resolved by a consumer of *of-sys* such as **ENDOF**.

Run-time: (x_1 x_2 −− | x_1)

If the two values on the stack are not equal, discard the top value and continue execution at the location specified by the consumer of *of-sys*, e.g., following the next **ENDOF**. Otherwise, discard both values and continue execution in line.

See: **6.2.0873** CASE, **6.2.1342** ENDCASE, **6.2.1343** ENDOF, **A.6.2.1950** OF.

6.2.2000   **PAD**   CORE EXT

(−− *c-addr*)

c-addr is the address of a transient region that can be used to hold data for intermediate processing.

See: **3.3.3.6 Other transient regions**, **A.6.2.2000** PAD.

6.2.2008   **PARSE**   CORE EXT

(*char* "*ccc⟨char⟩*" *−−* *c-addr u*)

Parse *ccc* delimited by the delimiter *char*.

c-addr is the address (within the input buffer) and *u* is the length of the parsed string. If the parse area was empty, the resulting string has a zero length.

See: **3.4.1 Parsing**, **A.6.2.2008** PARSE.

6.2.2020   **PARSE−NAME**   CORE EXT

("⟨*spaces*⟩*name*⟨*space*⟩" *−−* *c-addr u*)

Skip leading space delimiters. Parse *name* delimited by a space.

c-addr is the address of the selected string within the input buffer and *u* is its length in characters. If the parse area is empty or contains only white space, the resulting string has length zero.

6.2.2030   **PICK**   CORE EXT

($x_u \ldots x_1 \ x_0 \ u \ -- \ x_u \ldots x_1 \ x_0 \ x_u$)

Remove *u*. Copy the x_u to the top of the stack. An ambiguous condition exists if there are less than *u+2* items on the stack before **PICK** is executed.

See: **A.6.2.2030** PICK.

6.2.2125   **REFILL**   CORE EXT

(*−−* *flag*)

Attempt to fill the input buffer from the input source, returning a true flag if successful.

When the input source is the user input device, attempt to receive input into the terminal input buffer. If successful, make the result the input buffer, set **>IN** to zero, and return *true*. Receipt of a line containing no characters is considered successful. If there is no input available from the current input source, return *false*.

When the input source is a string from **EVALUATE**, return *false* and perform no other action.

See: **7.6.2.2125** REFILL, **11.6.2.2125** REFILL, **A.6.2.2125** REFILL.

6.2.2148 **RESTORE-INPUT**   CORE EXT

($x_n \ldots x_1 \, n$ $--$ *flag*)

Attempt to restore the input source specification to the state described by x_1 through x_n. *flag* is true if the input source specification cannot be so restored.

An ambiguous condition exists if the input source represented by the arguments is not the same as the current input source.

See: **15.6.2.1908** N>R, **15.6.2.1940** NR>, **A.6.2.2182** SAVE-INPUT.

6.2.2150 **ROLL**   CORE EXT

($x_u \, x_{u-1} \ldots x_0 \, u$ $--$ $x_{u-1} \ldots x_0 \, x_u$)

Remove u. Rotate $u+1$ items on the top of the stack. An ambiguous condition exists if there are less than $u+2$ items on the stack before **ROLL** is executed.

See: **A.6.2.2150** ROLL.

6.2.2266 **S\"**   "s-backslash-quote"   CORE EXT

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (*"ccc⟨quote⟩"* $--$)
Parse *ccc* delimited by **"** (double-quote), using the translation rules below. Append the run-time semantics given below to the current definition.

Translation rules: Characters are processed one at a time and appended to the compiled string. If the character is a '\' character it is processed by parsing and substituting one or more characters as follows, where the character after the backslash is case sensitive:

\a	BEL	(alert,	ASCII 7)
\b	BS	(backspace,	ASCII 8)
\e	ESC	(escape,	ASCII 27)
\f	FF	(form feed,	ASCII 12)
\l	LF	(line feed,	ASCII 10)
\m	CR/LF	pair	(ASCII 13, 10)
\n	newline	(implementation dependent , e.g., CR/LF, CR, LF, LF/CR)	
\q	double-quote		(ASCII 34)
\r	CR	(carriage return,	ASCII 13)
\t	HT	(horizontal tab,	ASCII 9)
\v	VT	(vertical tab,	ASCII 11)
\z	NUL	(no character,	ASCII 0)
\"	double-quote		(ASCII 34)

\x⟨*hexdigit*⟩⟨*hexdigit*⟩
The resulting character is the conversion of these two hexadecimal digits. An ambiguous conditions exists if \x is not followed by two hexadecimal characters.

\\	backslash itself	(ASCII 92)

An ambiguous condition exists if a \ is placed before any character, other than those defined in here.

Run-time: (−− *c-addr u*)

Return *c-addr* and *u* describing a string consisting of the translation of the characters *ccc*. A program shall not alter the returned string.

See: **3.4.1 Parsing**, **6.2.0855** C", **11.6.1.2165** S", **A.6.1.2165** S".

6.2.2182   **SAVE−INPUT**   CORE EXT

(−− $x_n \ldots x_1 \ n$)

x_1 through x_n describe the current state of the input source specification for later use by **RESTORE−INPUT**.

See: **15.6.2.1908** N>R, **15.6.2.1940** NR>, **A.6.2.2182** SAVE-INPUT.

6.2.2218   **SOURCE−ID**   "source-i-d"   CORE EXT

(−− *0 | -1*)

Identifies the input source as follows:

SOURCE−ID	Input source
-1	String (via **EVALUATE**)
0	User input device

See: **11.6.1.2218** SOURCE-ID.

6.2.2295   **TO**   CORE EXT

Interpretation: ($i * x$ "⟨*spaces*⟩*name*" −−)

Skip leading spaces and parse *name* delimited by a space. Perform the "TO *name* run-time" semantics given in the definition for the defining word of *name*. An ambiguous condition exists if *name* was not defined by a word with "TO *name* run-time" semantics.

Compilation: ("⟨*spaces*⟩*name*" −−)

Skip leading spaces and parse *name* delimited by a space. Append the "TO *name* run-time" semantics given in the definition for the defining word of *name* to the current definition. An ambiguous condition exists if *name* was not defined by a word with "TO *name* run-time" semantics.

Note: An ambiguous condition exists if any of **POSTPONE**, **[COMPILE]**, **'** or **[']** are applied to **TO**.

See: **6.2.2405** VALUE, **8.6.2.0435** 2VALUE, **12.6.2.1628** FVALUE, **13.6.1.0086** (LOCAL), **A.6.2.2295** TO.

6.2.2298   **TRUE**   CORE EXT

$(-- \textit{true})$

Return a *true* flag, a single-cell value with all bits set.

See: **3.1.3.1 Flags**, **A.6.2.2298** TRUE.

6.2.2300   **TUCK**   CORE EXT

$(x_1\ x_2 -- x_2\ x_1\ x_2)$

Copy the first (top) stack item below the second stack item.

6.2.2330   **U.R**   "u-dot-r"   CORE EXT

$(u\ n --)$

Display *u* right aligned in a field *n* characters wide. If the number of characters required to display *u* is greater than *n*, all digits are displayed with no leading spaces in a field as wide as necessary.

6.2.2350   **U>**   "u-greater-than"   CORE EXT

$(u_1\ u_2 -- \textit{flag})$

flag is true if and only if u_1 is greater than u_2.

See: **6.1.0540** >.

6.2.2395   **UNUSED**   CORE EXT

$(-- u)$

u is the amount of space remaining in the region addressed by **HERE**, in address units.

6.2.2405   **VALUE**   CORE EXT

$(x\ \text{"}\langle spaces\rangle name\text{"} --)$

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below, with an initial value equal to *x*.

name is referred to as a "value".

name Execution: $(-- x)$

Place *x* on the stack. The value of *x* is that given when *name* was created, until the phrase *x* **TO** *name* is executed, causing a new value of *x* to be assigned to *name*.

TO *name* Run-time: (*x* −−)

> Assign the value *x* to *name*.

> See: **3.4.1 Parsing**, **6.2.2295** TO, **A.6.2.2405** VALUE.

6.2.2440　　**WITHIN**　　　　　　　　　　　　　　　　　　　　　　　**CORE EXT**

> (n_1 | u_1 n_2 | u_2 n_3 | u_3 −− *flag*)

> Perform a comparison of a test value n_1 | u_1 with a lower limit n_2 | u_2 and an upper limit n_3 | u_3, returning *true* if either (n_2 | $u_2 < n_3$ | u_3 and (n_2 | $u_2 <= n_1$ | u_1 and n_1 | $u_1 < n_3$ | u_3)) or (n_2 | $u_2 > n_3$ | u_3 and (n_2 | $u_2 <= n_1$ | u_1 or n_1 | $u_1 < n_3$ | u_3)) is true, returning *false* otherwise. An ambiguous condition exists n_1 | u_1, n_2 | u_2, and n_3 | u_3 are not all the same type.

> See: **A.6.2.2440** WITHIN.

6.2.2530　　**[COMPILE]**　　　　　　　　　"bracket-compile"　　　　　　　　**CORE EXT**

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (*"⟨spaces⟩name"* −−)

> Skip leading space delimiters. Parse *name* delimited by a space. Find *name*. If *name* has other than default compilation semantics, append them to the current definition; otherwise append the execution semantics of *name*. An ambiguous condition exists if *name* is not found.

Note: This word is obsolescent and is included as a concession to existing implementations.

> See: **3.4.1 Parsing**, **A.6.2.2530** [COMPILE].

6.2.2535　　****　　　　　　　　　　　　　　"backslash"　　　　　　　　　　　**CORE EXT**

Compilation: Perform the execution semantics given below.

Execution: (*"ccc⟨eol⟩"* −−)

> Parse and discard the remainder of the parse area. **** is an immediate word.

> See: **7.6.2.2535** \\, **A.6.2.2535** \\.

! " # $ % & ' ( ) * + , - . / digits : ; < = > ? @ ALPHA [\\ ] ^ _ ` alpha { | } ~

7   The optional Block word set

7.1   Introduction

7.2   Additional terms

block: 1024 characters of data on mass storage, designated by a block number.

block buffer: A block-sized region of data space where a block is made temporarily available for use. The current block buffer is the block buffer most recently accessed by **BLOCK**, **BUFFER**, **LOAD**, **LIST**, or **THRU**.

7.3   Additional usage requirements

7.3.1   Data space

A program may access memory within a valid block buffer.

See: **3.3.3 Data space**.

7.3.2   Block buffer regions

The address of a block buffer returned by **BLOCK** or **BUFFER** is transient. A call to **BLOCK** or **BUFFER** may render a previously-obtained block-buffer address invalid, as may a call to any word that:

- parses:

- displays characters on the user output device, such as **TYPE** or **EMIT**;

- controls the user output device, such as **CR** or **AT-XY**;

- receives or tests for the presence of characters from the user input device such as **ACCEPT** or **KEY**;

- waits for a condition or event, such as **MS** or **EKEY**;

- manages the block buffers, such as **FLUSH**, **SAVE-BUFFERS**, or **EMPTY-BUFFERS**;

- performs any operation on a file or file-name directory that implies I/O, such as **REFILL** or any word that returns an *ior*;

- implicitly performs I/O, such as text interpreter nesting and un-nesting when files are being used (including un-nesting implied by **THROW**).

If the input source is a block, these restrictions also apply to the address returned by **SOURCE**. Block buffers are uniquely assigned to blocks.

See **A.7.3.2 Block buffer regions**.

7.3.3   Parsing

The Block word set implements an alternative input source for the text interpreter. When the input source is a block, **BLK** shall contain the non-zero block number and the input buffer is the 1024-character buffer containing that block.

A block is conventionally displayed as 16 lines of 64 characters.

A program may switch the input source to a block by using **LOAD** or **THRU**. Input sources may be nested using **LOAD** and **EVALUATE** in any order.

A program may reposition the parse area within a block by manipulating **>IN**. More extensive repositioning can be accomplished using **SAVE−INPUT** and **RESTORE−INPUT**.

See: **3.4.1 Parsing**.

7.3.4 Possible action on an ambiguous condition

See: **3.4.4 Possible actions on an ambiguous condition**.

 – A system with the Block word set may set interpretation state and interpret a block.

7.4   Additional documentation requirements

7.4.1 System documentation

7.4.1.1 Implementation-defined options

 – the format used for display by **7.6.2.1770** LIST (if implemented);

 – the length of a line affected by **7.6.2.2535** \ (if implemented).

7.4.1.2 Ambiguous conditions

 – Correct block read was not possible;

 – I/O exception in block transfer;

 – Invalid block number (**7.6.1.0800** BLOCK, **7.6.1.0820** BUFFER, **7.6.1.1790** LOAD);

 – A program directly alters the contents of **7.6.1.0790** BLK;

 – No current block buffer for **7.6.1.2400** UPDATE.

7.4.1.3 Other system documentation

 – any restrictions a multiprogramming system places on the use of buffer addresses;

 – the number of blocks available for source text and data.

7.4.2 Program documentation

 – the number of blocks required by the program.

7.5   Compliance and labeling

7.5.1 Forth-2012 systems

The phrase "Providing the Block word set" shall be appended to the label of any Standard System that provides all of the Block word set.

The phrase "Providing *name(s)* from the Block Extensions word set" shall be appended to the label of any Standard System that provides portions of the Block Extensions word set.

The phrase "Providing the Block Extensions word set" shall be appended to the label of any Standard System that provides all of the Block and Block Extensions word sets.

7.5.2 Forth-2012 programs

The phrase "Requiring the Block word set" shall be appended to the label of Standard Programs that require the system to provide the Block word set.

The phrase "Requiring *name(s)* from the Block Extensions word set" shall be appended to the label of Standard Programs that require the system to provide portions of the Block Extensions word set.

The phrase "Requiring the Block Extensions word set" shall be appended to the label of Standard Programs that require the system to provide all of the Block and Block Extensions word sets.

7.6 Glossary

7.6.1 Block words

7.6.1.0790   BLK   "b-l-k"   BLOCK

($--$ *a-addr*)

a-addr is the address of a cell containing zero or the number of the mass-storage block being interpreted. If **BLK** contains zero, the input source is not a block and can be identified by **SOURCE-ID**, if **SOURCE-ID** is available. An ambiguous condition exists if a program directly alters the contents of **BLK**.

See: **7.3.2 Block buffer regions**.

7.6.1.0800   BLOCK   BLOCK

(*u* $--$ *a-addr*)

a-addr is the address of the first character of the block buffer assigned to mass-storage block *u*. An ambiguous condition exists if *u* is not an available block number.

If block *u* is already in a block buffer, *a-addr* is the address of that block buffer.

If block *u* is not already in memory and there is an unassigned block buffer, transfer block *u* from mass storage to an unassigned block buffer. *a-addr* is the address of that block buffer.

If block *u* is not already in memory and there are no unassigned block buffers, unassign a block buffer. If the block in that buffer has been **UPDATE**d, transfer the block to mass storage and transfer block *u* from mass storage into that buffer. *a-addr* is the address of that block buffer.

At the conclusion of the operation, the block buffer pointed to by *a-addr* is the current block buffer and is assigned to *u*.

7.6.1.0820   **BUFFER**   BLOCK

(*u* − − *a-addr*)

a-addr is the address of the first character of the block buffer assigned to block *u*. The contents of the block are unspecified. An ambiguous condition exists if *u* is not an available block number.

If block *u* is already in a block buffer, *a-addr* is the address of that block buffer.

If block *u* is not already in memory and there is an unassigned buffer, *a-addr* is the address of that block buffer.

If block *u* is not already in memory and there are no unassigned block buffers, unassign a block buffer. If the block in that buffer has been **UPDATE**d, transfer the block to mass storage. *a-addr* is the address of that block buffer.

At the conclusion of the operation, the block buffer pointed to by *a-addr* is the current block buffer and is assigned to *u*.

See: **7.6.1.0800** BLOCK.

7.6.1.1360   **EVALUATE**   BLOCK

Extend the semantics of **6.1.1360** EVALUATE to include: Store zero in **BLK**.

7.6.1.1559   **FLUSH**   BLOCK

(− −)

Perform the function of **SAVE−BUFFERS**, then unassign all block buffers.

7.6.1.1790   **LOAD**   BLOCK

(*i* * *x* *u* − − *j* * *x*)

Save the current input-source specification. Store *u* in **BLK** (thus making block *u* the input source and setting the input buffer to encompass its contents), set **>IN** to zero, and interpret. When the parse area is exhausted, restore the prior input source specification. Other stack effects are due to the words **LOAD**ed.

An ambiguous condition exists if *u* is zero or is not a valid block number.

See: **3.4 The Forth text interpreter**.

7.6.1.2180   **SAVE-BUFFERS**   BLOCK

(--)

Transfer the contents of each **UPDATE**d block buffer to mass storage. Mark all buffers as unmodified.

7.6.1.2400   **UPDATE**   BLOCK

(--)

Mark the current block buffer as modified. An ambiguous condition exists if there is no current block buffer.

UPDATE does not immediately cause I/O.

See: **7.6.1.0800** BLOCK, **7.6.1.0820** BUFFER, **7.6.1.1559** FLUSH, **7.6.1.2180** SAVE-BUFFERS.

7.6.2   Block extension words

7.6.2.1330   **EMPTY-BUFFERS**   BLOCK EXT

(--)

Unassign all block buffers. Do not transfer the contents of any **UPDATE**d block buffer to mass storage.

See: **7.6.1.0800** BLOCK.

7.6.2.1770   **LIST**   BLOCK EXT

(u --)

Display block *u* in an implementation-defined format. Store *u* in **SCR**.

See: **7.6.1.0800** BLOCK.

7.6.2.2125   **REFILL**   BLOCK EXT

(-- *flag*)

Extend the execution semantics of **6.2.2125** REFILL with the following:

When the input source is a block, make the next block the input source and current input buffer by adding one to the value of **BLK** and setting **>IN** to zero. Return *true* if the new value of BLK is a valid block number, otherwise *false*.

See: **6.2.2125** REFILL, **11.6.2.2125** REFILL.

7.6.2.2190   **SCR**   "s-c-r"   BLOCK EXT

($--$ *a-addr*)

a-addr is the address of a cell containing the block number of the block most recently **LIST**ed.

See: **A.7.6.2.2190** SCR.

7.6.2.2280   **THRU**   BLOCK EXT

($i * x\, u_1\, u_2 -- j * x$)

LOAD the mass storage blocks numbered u_1 through u_2 in sequence. Other stack effects are due to the words **LOAD**ed.

7.6.2.2535   ****   "backslash"   BLOCK EXT

Extend the semantics of **6.2.2535** \ to be:

Compilation: Perform the execution semantics given below.

Execution: (*"ccc⟨eol⟩"* $--$)

If **BLK** contains zero, parse and discard the remainder of the parse area; otherwise parse and discard the portion of the parse area corresponding to the remainder of the current line. **** is an immediate word.

8 The optional Double-Number word set

8.1 Introduction

Sixteen-bit Forth systems often use double-length numbers. However, 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 use of double-length numbers is much diminished. Therefore, the words that manipulate double-length entities have been placed in this optional word set.

8.2 Additional terms and notation

None.

8.3 Additional usage requirements

8.3.1 Text interpreter input number conversion

When the text interpreter processes a number, except a ⟨*cnum*⟩, that is immediately followed by a decimal point and is not found as a definition name, the text interpreter shall convert it to a double-cell number.

For example, entering **DECIMAL** 1234 leaves the single-cell number 1234 on the stack, and entering **DECIMAL** 1234. leaves the double-cell number 1234 0 on the stack.

See: **3.4.1.3 Text interpreter input number conversion**.

8.4 Additional documentation requirements

8.4.1 System documentation

8.4.1.1 Implementation-defined options

– no additional requirements.

8.4.1.2 Ambiguous conditions

– *d* outside range of *n* in **8.6.1.1140** D>S.

8.4.1.3 Other system documentation

– no additional requirements.

8.4.2 Program documentation

– no additional requirements.

8.5 Compliance and labeling

8.5.1 Forth-2012 systems

The phrase "Providing the Double-Number word set" shall be appended to the label of any Standard System that provides all of the Double-Number word set.

The phrase "Providing *name(s)* from the Double-Number Extensions word set" shall be appended to the label of any Standard System that provides portions of the Double-Number Extensions word set.

The phrase "Providing the Double-Number Extensions word set" shall be appended to the label of any Standard System that provides all of the Double-Number and Double-Number Extensions word sets.

8.5.2 Forth-2012 programs

The phrase "Requiring the Double-Number word set" shall be appended to the label of Standard Programs that require the system to provide the Double-Number word set.

The phrase "Requiring *name(s)* from the Double-Number Extensions word set" shall be appended to the label of Standard Programs that require the system to provide portions of the Double-Number Extensions word set.

The phrase "Requiring the Double-Number Extensions word set" shall be appended to the label of Standard Programs that require the system to provide all of the Double-Number and Double-Number Extensions word sets.

8.6 Glossary

8.6.1 Double-Number words

8.6.1.0360   **2CONSTANT**   "two-constant"   DOUBLE

$(x_1 \, x_2 \, "\langle spaces \rangle name" -- \ )$

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below.

name is referred to as a "two-constant".

name Execution: $(-- \ x_1 \, x_2 \ )$

Place cell pair $x_1 \, x_2$ on the stack.

See: **3.4.1 Parsing**, **A.8.6.1.0360** 2CONSTANT.

8.6.1.0390   **2LITERAL**   "two-literal"   DOUBLE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: $(x_1 \, x_2 -- \ )$

Append the run-time semantics below to the current definition.

Run-time: $(-- \ x_1 \, x_2 \ )$

Place cell pair $x_1 \, x_2$ on the stack.

See: **A.8.6.1.0390** 2LITERAL.

8.6.1.0440   **2VARIABLE**   "two-variable"   DOUBLE

(*"⟨spaces⟩name" −−*)

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Reserve two consecutive cells of data space.

name is referred to as a "two-variable".

name Execution: (*−− a-addr*)

a-addr is the address of the first (lowest address) cell of two consecutive cells in data space reserved by **2VARIABLE** when it defined *name*. A program is responsible for initializing the contents.

See: **3.4.1 Parsing**, **6.1.2410** VARIABLE, **A.8.6.1.0440** 2VARIABLE.

8.6.1.1040   **D+**   "d-plus"   DOUBLE

(*d₁ | ud₁ d₂ | ud₂ −− d₃ | ud₃*)

Add $d_2 | ud_2$ to $d_1 | ud_1$, giving the sum $d_3 | ud_3$.

8.6.1.1050   **D−**   "d-minus"   DOUBLE

(*d₁ | ud₁ d₂ | ud₂ −− d₃ | ud₃*)

Subtract $d_2 | ud_2$ from $d_1 | ud_1$, giving the difference $d_3 | ud_3$.

8.6.1.1060   **D.**   "d-dot"   DOUBLE

(*d −−*)

Display *d* in free field format.

8.6.1.1070   **D.R**   "d-dot-r"   DOUBLE

(*d n −−*)

Display *d* right aligned in a field *n* characters wide. If the number of characters required to display *d* is greater than *n*, all digits are displayed with no leading spaces in a field as wide as necessary.

See: **A.8.6.1.1070** D.R.

8.6.1.1075   **D0<**   "d-zero-less"   DOUBLE

(*d* −− *flag*)

flag is true if and only if *d* is less than zero.

8.6.1.1080   **D0=**   "d-zero-equals"   DOUBLE

(*xd* −− *flag*)

flag is true if and only if *xd* is equal to zero.

8.6.1.1090   **D2***   "d-two-star"   DOUBLE

(*xd₁* −− *xd₂*)

xd₂ is the result of shifting *xd₁* one bit toward the most-significant bit, filling the vacated least-significant bit with zero.

8.6.1.1100   **D2/**   "d-two-slash"   DOUBLE

(*xd₁* −− *xd₂*)

xd₂ is the result of shifting *xd₁* one bit toward the least-significant bit, leaving the most-significant bit unchanged.

8.6.1.1110   **D<**   "d-less-than"   DOUBLE

(*d₁ d₂* −− *flag*)

flag is true if and only if *d₁* is less than *d₂*.

8.6.1.1120   **D=**   "d-equals"   DOUBLE

(*xd₁ xd₂* −− *flag*)

flag is true if and only if *xd₁* is bit-for-bit the same as *xd₂*.

8.6.1.1140   **D>S**   "d-to-s"   DOUBLE

(*d* −− *n*)

n is the equivalent of *d*. An ambiguous condition exists if *d* lies outside the range of a signed single-cell number.

See: **A.8.6.1.1140** D>S.

8.6.1.1160   **DABS**   "d-abs"   DOUBLE

(*d* −− *ud*)

ud is the absolute value of *d*.

8.6.1.1210   **DMAX**   "d-max"   DOUBLE

(d_1 d_2 −− d_3)

d_3 is the greater of d_1 and d_2.

8.6.1.1220   **DMIN**   "d-min"   DOUBLE

(d_1 d_2 −− d_3)

d_3 is the lesser of d_1 and d_2.

8.6.1.1230   **DNEGATE**   "d-negate"   DOUBLE

(d_1 −− d_2)

d_2 is the negation of d_1.

8.6.1.1820   **M*/**   "m-star-slash"   DOUBLE

(d_1 n_1 +n_2 −− d_2)

Multiply d_1 by n_1 producing the triple-cell intermediate result *t*. Divide *t* by +n_2 giving the double-cell quotient d_2. An ambiguous condition exists if +n_2 is zero or negative, or the quotient lies outside of the range of a double-precision signed integer.

See: **A.8.6.1.1820** M*/.

8.6.1.1830   **M+**   "m-plus"   DOUBLE

(d_1 | ud_1 *n* −− d_2 | ud_2)

Add *n* to d_1 | ud_1, giving the sum d_2 | ud_2.

See: **A.8.6.1.1830** M+.

8.6.2   Double-Number extension words

8.6.2.0420   **2ROT**   "two-rote"   DOUBLE EXT

(x_1 x_2 x_3 x_4 x_5 x_6 $--$ x_3 x_4 x_5 x_6 x_1 x_2)

Rotate the top three cell pairs on the stack bringing cell pair x_1 x_2 to the top of the stack.

8.6.2.0435   **2VALUE**   "two-value"   DOUBLE EXT

(x_1 x_2 "⟨spaces⟩name" $--$)

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below, with an initial value of x_1 x_2.

name is referred to as a "two-value".

name Execution: ($--$ x_1 x_2)

Place cell pair x_1 x_2 on the stack. The value of x_1 x_2 is that given when *name* was created, until the phrase "x_1 x_2 **TO** *name*" is executed, causing a new cell pair x_1 x_2 to be assigned to *name*.

TO *name* Run-time: (x_1 x_2 $--$)

Assign the cell pair x_1 x_2 to *name*.

See: **3.4.1 Parsing** and **6.2.2295** TO, **A.8.6.2.0435** 2VALUE.

8.6.2.1270   **DU<**   "d-u-less"   DOUBLE EXT

(ud_1 ud_2 $--$ *flag*)

flag is true if and only if ud_1 is less than ud_2.

---

9 The optional Exception word set

9.1 Introduction

9.2 Additional terms and notation

None.

9.3 Additional usage requirements

9.3.1 THROW values

The **THROW** values {-255. . .-1} shall be used only as assigned by this standard. The values {-4095. . .-256} shall be used only as assigned by a system.

Programs shall not define values for use with **THROW** in the range {-4095. . .-1}.

9.3.2 Exception frame

An exception frame is the implementation-dependent set of information recording the current execution state necessary for the proper functioning of **CATCH** and **THROW**. It often includes the depths of the data stack and return stack.

9.3.3 Exception stack

A stack used for the nesting of exception frames by **CATCH** and **THROW**. It may be, but need not be, implemented using the return stack.

9.3.4 Possible actions on an ambiguous condition

A system choosing to execute **THROW** when detecting one of the ambiguous conditions listed in table 9.1 shall use the throw code listed there.

See: **3.4.4 Possible actions on an ambiguous condition**.

9.3.5 Exception handling

There are several methods of coupling **CATCH** and **THROW** to other procedural nestings. The usual nestings are the execution of definitions, use of the return stack, use of loops, instantiation of locals and nesting of input sources (i.e., with **LOAD**, **EVALUATE**, or **INCLUDE-FILE**).

When a **THROW** returns control to a **CATCH**, the system shall un-nest not only definitions, but also, if present, locals and input source specifications, to return the system to its proper state for continued execution past the **CATCH**.

Table 9.1: **THROW** code assignments

Code	Reserved for	Code	Reserved for
-1	**ABORT**	-40	invalid **BASE** for floating point conversion
-2	**ABORT"**	-41	loss of precision
-3	stack overflow	-42	floating-point divide by zero
-4	stack underflow	-43	floating-point result out of range
-5	return stack overflow	-44	floating-point stack overflow
-6	return stack underflow	-45	floating-point stack underflow
-7	do-loops nested too deeply during execution	-46	floating-point invalid argument
-8	dictionary overflow	-47	compilation word list deleted
-9	invalid memory address	-48	invalid **POSTPONE**
-10	division by zero	-49	search-order overflow
-11	result out of range	-50	search-order underflow
-12	argument type mismatch	-51	compilation word list changed
-13	undefined word	-52	control-flow stack overflow
-14	interpreting a compile-only word	-53	exception stack overflow
-15	invalid **FORGET**	-54	floating-point underflow
-16	attempt to use zero-length string as a name	-55	floating-point unidentified fault
-17	pictured numeric output string overflow	-56	**QUIT**
-18	parsed string overflow	-57	exception in sending or receiving a character
-19	definition name too long	-58	**[IF]**, **[ELSE]**, or **[THEN]** exception
-20	write to a read-only location	-59	**ALLOCATE**
-21	unsupported operation	-60	**FREE**
	(e.g., **AT-XY** on a too-dumb terminal)	-61	**RESIZE**
-22	control structure mismatch	-62	**CLOSE-FILE**
-23	address alignment exception	-63	**CREATE-FILE**
-24	invalid numeric argument	-64	**DELETE-FILE**
-25	return stack imbalance	-65	**FILE-POSITION**
-26	loop parameters unavailable	-66	**FILE-SIZE**
-27	invalid recursion	-67	**FILE-STATUS**
-28	user interrupt	-68	**FLUSH-FILE**
-29	compiler nesting	-69	**OPEN-FILE**
-30	obsolescent feature	-70	**READ-FILE**
-31	**>BODY** used on non-**CREATE**d definition	-71	**READ-LINE**
-32	invalid *name* argument (e.g., **TO** *name*)	-72	**RENAME-FILE**
-33	block read exception	-73	**REPOSITION-FILE**
-34	block write exception	-74	**RESIZE-FILE**
-35	invalid block number	-75	**WRITE-FILE**
-36	invalid file position	-76	**WRITE-LINE**
-37	file I/O exception	-77	Malformed xchar
-38	non-existent file	-78	**SUBSTITUTE**
-39	unexpected end of file	-79	**REPLACES**

9.4   Additional documentation requirements

9.4.1   System documentation

9.4.1.1   Implementation-defined options

– Values used in the system by **9.6.1.0875** `CATCH` and **9.6.1.2275** `THROW` (**9.3.1 THROW values**, **9.3.4 Possible actions on an ambiguous condition**).

9.4.1.2   Ambiguous conditions

– no additional requirements.

9.4.1.3   Other system documentation

– no additional requirements.

9.4.2   Program documentation

– no additional requirements.

9.5   Compliance and labeling

9.5.1   Forth-2012 systems

The phrase "Providing the Exception word set" shall be appended to the label of any Standard System that provides all of the Exception word set.

The phrase "Providing *name(s)* from the Exception Extensions word set" shall be appended to the label of any Standard System that provides portions of the Exception Extensions word set.

The phrase "Providing the Exception Extensions word set" shall be appended to the label of any Standard System that provides all of the Exception and Exception Extensions word sets.

9.5.2   Forth-2012 programs

The phrase "Requiring the Exception word set" shall be appended to the label of Standard Programs that require the system to provide the Exception word set.

The phrase "Requiring *name(s)* from the Exception Extensions word set" shall be appended to the label of Standard Programs that require the system to provide portions of the Exception Extensions word set.

The phrase "Requiring the Exception Extensions word set" shall be appended to the label of Standard Programs that require the system to provide all of the Exception and Exception Extensions word sets.

9.6   Glossary

9.6.1   Exception words

9.6.1.0875   CATCH   EXCEPTION

> (*i*x xt -- j*x 0 | i*x n*)
>
> Push an exception frame on the exception stack and then execute the execution token *xt* (as with **EXECUTE**) in such a way that control can be transferred to a point just after **CATCH** if **THROW** is executed during the execution of *xt*.
>
> If the execution of *xt* completes normally (i.e., the exception frame pushed by this **CATCH** is not popped by an execution of **THROW**) pop the exception frame and return zero on top of the data stack, above whatever stack items would have been returned by *xt* **EXECUTE**. Otherwise, the remainder of the execution semantics are given by **THROW**.

See: **A.9.6.1.2275** THROW.

9.6.1.2275   THROW   EXCEPTION

> (*k*x n -- k*x | i*x n*)
>
> If any bits of *n* are non-zero, pop the topmost exception frame from the exception stack, along with everything on the return stack above that frame. Then restore the input source specification in use before the corresponding **CATCH** and adjust the depths of all stacks defined by this standard so that they are the same as the depths saved in the exception frame (*i* is the same number as the *i* in the input arguments to the corresponding **CATCH**), put *n* on top of the data stack, and transfer control to a point just after the **CATCH** that pushed that exception frame.
>
> If the top of the stack is non zero and there is no exception frame on the exception stack, the behavior is as follows:
>
> If *n* is minus-one (-1), perform the function of **6.1.0670** ABORT (the version of **ABORT** in the Core word set), displaying no message.
>
> If *n* is minus-two, perform the function of **6.1.0680** ABORT" (the version of **ABORT"** in the Core word set), displaying the characters *ccc* associated with the **ABORT"** that generated the **THROW**.
>
> Otherwise, the system may display an implementation-dependent message giving information about the condition associated with the **THROW** code *n*. Subsequently, the system shall perform the function of **6.1.0670** ABORT (the version of **ABORT** in the Core word set).

See: **A.9.6.1.2275** THROW.

9.6.2   Exception extension words

9.6.2.0670   ABORT   EXCEPTION EXT

        Extend the semantics of **6.1.0670** ABORT to be:

        ($i * x$ −−) (R: $j * x$ −−)

        Perform the function of −1 **THROW**.

    See: **6.1.0670** ABORT.

9.6.2.0680   ABORT"   "abort-quote"   EXCEPTION EXT

        Extend the semantics of **6.1.0680** ABORT" to be:

Interpretation: Interpretation semantics for this word are undefined.

  Compilation: (*"ccc⟨quote⟩"* −−)

        Parse *ccc* delimited by a **"** (double-quote). Append the run-time semantics given below to the current definition.

  Run-time: ($i * x\ x_1$ −− | $i * x$) (R: $j * x$ −− | $j * x$)

        Remove x_1 from the stack. If any bit of x_1 is not zero, perform the function of −2 **THROW**, displaying *ccc* if there is no exception frame on the exception stack.

    See: **3.4.1 Parsing**, **6.1.0680** ABORT".

10   The optional Facility word set

10.1   Introduction

10.2   Additional terms and notation

None.

10.3   Additional usage requirements

10.3.1   Data types

Append table 10.1 to table 3.1.

Table 10.1: Data types

Symbol	Data type	Size on stack
struct-sys	data structures	implementation dependent

10.3.1.1   Structure type

The implementation-dependent data generated upon beginning to compile a **BEGIN-STRUCTURE** ... **END-STRUCTURE** structure and consumed at its close is represented by the symbol *struct-sys* throughout this standard.

10.3.1.2   Character types

Programs that use more than seven bits of a character by **EKEY** have an environmental dependency.

See: **3.1.2 Character types**.

10.4   Additional documentation requirements

10.4.1   System documentation

10.4.1.1   Implementation-defined options

  – encoding of keyboard events **10.6.2.1305** EKEY);

  – duration of a system clock tick;

  – repeatability to be expected from execution of **10.6.2.1905** MS.

10.4.1.2   Ambiguous conditions

  – **10.6.1.0742** AT-XY operation can't be performed on user output device.

  – A *name* defined by **10.6.2.0763** BEGIN-STRUCTURE is executed before the corresponding **10.6.2.1336** END-STRUCTURE has been executed.

10.4.1.3 Other system documentation

– no additional requirements.

10.4.2 Program documentation

10.4.2.1 Environmental dependencies

– using more than seven bits of a character in **10.6.2.1305** `EKEY`.

10.4.2.2 Other program documentation

– no additional requirements.

10.5 Compliance and labeling

10.5.1 Forth-2012 systems

The phrase "Providing the Facility word set" shall be appended to the label of any Standard System that provides all of the Facility word set.

The phrase "Providing *name(s)* from the Facility Extensions word set" shall be appended to the label of any Standard System that provides portions of the Facility Extensions word set.

The phrase "Providing the Facility Extensions word set" shall be appended to the label of any Standard System that provides all of the Facility and Facility Extensions word sets.

10.5.2 Forth-2012 programs

The phrase "Requiring the Facility word set" shall be appended to the label of Standard Programs that require the system to provide the Facility word set.

The phrase "Requiring *name(s)* from the Facility Extensions word set" shall be appended to the label of Standard Programs that require the system to provide portions of the Facility Extensions word set.

The phrase "Requiring the Facility Extensions word set" shall be appended to the label of Standard Programs that require the system to provide all of the Facility and Facility Extensions word sets.

10.6 Glossary

10.6.1 Facility words

10.6.1.0742   `AT-XY`   "at-x-y"   FACILITY

$(\ u_1\ u_2\ --\ \ )$

Perform implementation-dependent steps so that the next character displayed will appear in column u_1, row u_2 of the user output device, the upper left corner of which is column zero, row zero. An ambiguous condition exists if the operation cannot be performed on the user output device with the specified parameters.

10.6.1.1755   **KEY?**   "key-question"   FACILITY

(-- *flag*)

If a character is available, return *true*. Otherwise, return *false*. If non-character keyboard events are available before the first valid character, they are discarded and are subsequently unavailable. The character shall be returned by the next execution of **KEY**.

After **KEY?** returns with a value of *true*, subsequent executions of **KEY?** prior to the execution of **KEY** or **EKEY** also return *true*, without discarding keyboard events.

See: **A.10.6.1.1755** KEY?.

10.6.1.2005   **PAGE**   FACILITY

(--)

Move to another page for output. Actual function depends on the output device. On a terminal, **PAGE** clears the screen and resets the cursor position to the upper left corner. On a printer, **PAGE** performs a form feed.

10.6.2   Facility extension words

10.6.2.0135   **+FIELD**   "plus-field"   FACILITY EXT

(n_1 n_2 "⟨spaces⟩name" -- n_3)

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Return $n_3 = n_1 + n_2$ where n_1 is the offset in the data structure before **+FIELD** executes, and n_2 is the size of the data to be added to the data structure. n_1 and n_2 are in address units.

name Execution: ($addr_1$ -- $addr_2$)

Add n_1 to $addr_1$ giving $addr_2$.

See: **10.6.2.0763** BEGIN-STRUCTURE, **10.6.2.1336** END-STRUCTURE,
10.6.2.0893 CFIELD:, **10.6.2.1518** FIELD:, **12.6.2.1517** FFIELD:,
12.6.2.2206.40 SFFIELD:, **12.6.2.1207.40** DFFIELD:, **A.10.6.2.0135** +FIELD.

10.6.2.0763   **BEGIN-STRUCTURE**   FACILITY EXT

("⟨spaces⟩name" -- *struct-sys 0*)

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Return a *struct-sys* (zero or more implementation dependent items) that will be used by **END-STRUCTURE** and an initial offset of 0.

name Execution: (-- +*n*)

> +*n* is the size in memory expressed in address units of the data structure. An ambiguous condition exists if *name* is executed prior to the associated **END-STRUCTURE** being executed.

> See: **10.6.2.0135** +FIELD, **10.6.2.1336** END-STRUCTURE,
> **A.10.6.2.0763** BEGIN-STRUCTURE.

10.6.2.0893   **CFIELD:**   "c-field-colon"   FACILITY EXT

> (*n₁* "⟨spaces⟩name" -- *n₂*)

> Skip leading space delimiters. Parse *name* delimited by a space. *Offset* is the first character aligned value greater than or equal to *n₁*. *n₂* = *offset* + *1* character.

> Create a definition for *name* with the execution semantics given below.

name Execution: (*addr₁* -- *addr₂*)

> Add the *offset* calculated during the compile-time action to *addr₁* giving the address *addr₂*.

> See: **10.6.2.0135** +FIELD, **10.6.2.0763** BEGIN-STRUCTURE,
> **10.6.2.1336** END-STRUCTURE, **A.10.6.2.1518** FIELD:.

10.6.2.1305   **EKEY**   "e-key"   FACILITY EXT

> (-- *x*)

> Receive one keyboard event *x*. The encoding of keyboard events is implementation defined.

> See: **6.1.1750** KEY, **10.6.1.1755** KEY?, **A.10.6.2.1305** EKEY.

10.6.2.1306   **EKEY>CHAR**   "e-key-to-char"   FACILITY EXT

> (*x* -- *x false* | *char true*)

> If the keyboard event *x* corresponds to a character in the implementation-defined character set, return that character and *true*. Otherwise return *x* and *false*.

> See: **A.10.6.2.1306** EKEY>CHAR.

10.6.2.1306.40 EKEY>FKEY   "e-key-to-f-key"   FACILITY EXT

> (*x* -- *u flag*)

> If the keyboard event *x* corresponds to a keypress in the implementation-defined special key set, return that key's id *u* and *true*. Otherwise return *x* and *false*.

Note: The keyboard may lack some of the keys, or the capability to report them. Programs
should be written such that they also work (although less conveniently or with less func-
tionality) if these key numbers cannot be produced.

See: **10.6.2.1305** EKEY, **10.6.2.1740.01** K-ALT-MASK, **10.6.2.1740.02** K-CTRL-MASK,
10.6.2.1740.03 K-DELETE, **10.6.2.1740.04** K-DOWN, **10.6.2.1740.05** K-END,
10.6.2.1740.06 K-F1, **10.6.2.1740.07** K-F10, **10.6.2.1740.08** K-F11,
10.6.2.1740.09 K-F12, **10.6.2.1740.10** K-F2, **10.6.2.1740.11** K-F3,
10.6.2.1740.12 K-F4, **10.6.2.1740.13** K-F5, **10.6.2.1740.14** K-F6,
10.6.2.1740.15 K-F7, **10.6.2.1740.16** K-F8, **10.6.2.1740.17** K-F9,
10.6.2.1740.18 K-HOME, **10.6.2.1740.19** K-INSERT, **10.6.2.1740.20** K-LEFT,
10.6.2.1740.21 K-NEXT, **10.6.2.1740.22** K-PRIOR, **10.6.2.1740.23** K-RIGHT,
10.6.2.1740.24 K-SHIFT-MASK, **10.6.2.1740.25** K-UP,
A.10.6.2.1306.40 EKEY>FKEY.

10.6.2.1307   **EKEY?**   "e-key-question"   FACILITY EXT

(-- *flag*)

If a keyboard event is available, return *true*. Otherwise return *false*. The event shall be
returned by the next execution of **EKEY**.

After **EKEY?** returns with a value of *true*, subsequent executions of **EKEY?** prior to the
execution of **KEY**, **KEY?** or **EKEY** also return *true*, referring to the same event.

10.6.2.1325   **EMIT?**   "emit-question"   FACILITY EXT

(-- *flag*)

flag is true if the user output device is ready to accept data and the execution of **EMIT**
in place of **EMIT?** would not have suffered an indefinite delay. If the device status is
indeterminate, *flag* is true.

See: **A.10.6.2.1325** EMIT?.

10.6.2.1336   **END-STRUCTURE**   FACILITY EXT

(*struct-sys +n* --)

Terminate definition of a structure started by **BEGIN-STRUCTURE**.

See: **10.6.2.0135** +FIELD, **10.6.2.0763** BEGIN-STRUCTURE.

10.6.2.1518   **FIELD:**   "field-colon"   FACILITY EXT

(n_1 "⟨spaces⟩name" –– n_2)

Skip leading space delimiters. Parse *name* delimited by a space. *Offset* is the first cell aligned value greater than or equal to n_1. n_2 = *offset* + *1* cell.

Create a definition for *name* with the execution semantics given below.

name Execution: ($addr_1$ –– $addr_2$)

Add the *offset* calculated during the compile-time action to $addr_1$ giving the address $addr_2$.

See: **10.6.2.0135** +FIELD, **10.6.2.0763** BEGIN-STRUCTURE, **10.6.2.1336** END-STRUCTURE, **A.10.6.2.1518** FIELD:.

10.6.2.1740.01 K-ALT-MASK   FACILITY EXT

(–– u)

Mask for the ALT key, that can be **OR**ed with the key value to produce a value that the sequence **EKEY EKEY>FKEY** may produce when the user presses the corresponding key combination.

See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.02 K-CTRL-MASK   FACILITY EXT

(–– u)

Mask for the CTRL key, that can be **OR**ed with the key value to produce a value that the sequence **EKEY EKEY>FKEY** may produce when the user presses the corresponding key combination.

See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.03 K-DELETE   FACILITY EXT

(–– u)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "Delete" key.

See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.04 `K-DOWN`   FACILITY EXT

(*−− u*)

Leaves the value *u* that the sequence **`EKEY EKEY>FKEY`** would produce when the user presses the "cursor down" key.

See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.05 `K-END`   FACILITY EXT

(*−− u*)

Leaves the value *u* that the sequence **`EKEY EKEY>FKEY`** would produce when the user presses the "End" key.

See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.06 `K-F1`   "k-f-1"   FACILITY EXT

(*−− u*)

Leaves the value *u* that the sequence **`EKEY EKEY>FKEY`** would produce when the user presses the "F1" key.

See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.07 `K-F10`   "k-f-10"   FACILITY EXT

(*−− u*)

Leaves the value *u* that the sequence **`EKEY EKEY>FKEY`** would produce when the user presses the "F10" key.

See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.08 `K-F11`   "k-f-11"   FACILITY EXT

(*−− u*)

Leaves the value *u* that the sequence **`EKEY EKEY>FKEY`** would produce when the user presses the "F11" key.

See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.09 K-F12   "k-f-12"   FACILITY EXT

    (*-- u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "F12" key.

See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.10 K-F2   "k-f-2"   FACILITY EXT

    (*-- u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "F2" key.

See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.11 K-F3   "k-f-3"   FACILITY EXT

    (*-- u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "F3" key.

See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.12 K-F4   "k-f-4"   FACILITY EXT

    (*-- u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "F4" key.

See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.13 K-F5   "k-f-5"   FACILITY EXT

    (*-- u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "F5" key.

See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.14 K-F6   "k-f-6"   FACILITY EXT

      ($--$ u)

      Leaves the value u that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "F6" key.

    See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.15 K-F7   "k-f-7"   FACILITY EXT

      ($--$ u)

      Leaves the value u that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "F7" key.

    See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.16 K-F8   "k-f-8"   FACILITY EXT

      ($--$ u)

      Leaves the value u that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "F8" key.

    See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.17 K-F9   "k-f-9"   FACILITY EXT

      ($--$ u)

      Leaves the value u that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "F9" key.

    See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.18 K-HOME   FACILITY EXT

      ($--$ u)

      Leaves the value u that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "home" or "Pos1" key.

    See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.19 K-INSERT   FACILITY EXT

> (-- *u*)
>
> Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "Insert" key.

See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.20 K-LEFT   FACILITY EXT

> (-- *u*)
>
> Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "cursor left" key.

See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.21 K-NEXT   FACILITY EXT

> (-- *u*)
>
> Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "PgDn" (Page Down) or "Next" key.

See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.22 K-PRIOR   FACILITY EXT

> (-- *u*)
>
> Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "PgUp" (Page Up) or "Prior" key.

See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.23 K-RIGHT   FACILITY EXT

> (-- *u*)
>
> Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "cursor right" key.

See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.24 K-SHIFT-MASK   FACILITY EXT

(-- *u*)

Mask for the SHIFT key, that can be **OR**ed with the key value to produce a value that the sequence **EKEY EKEY>FKEY** may produce when the user presses the corresponding key combination.

See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1740.25 K-UP   FACILITY EXT

(-- *u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "cursor up" key.

See: **10.6.2.1306.40** EKEY>FKEY, **A.10.6.2.1306.40** EKEY>FKEY.

10.6.2.1905   MS   FACILITY EXT

(*u* --)

Wait at least *u* milliseconds.

Note: The actual length and variability of the time period depends upon the implementation-defined resolution of the system clock and upon other system and computer characteristics beyond the scope of this standard.

See: **A.10.6.2.1905** MS.

10.6.2.2292   TIME&DATE   "time-and-date"   FACILITY EXT

(-- $+n_1$ $+n_2$ $+n_3$ $+n_4$ $+n_5$ $+n_6$)

Return the current time and date. $+n_1$ is the second $\{0\dots59\}$, $+n_2$ is the minute $\{0\dots59\}$, $+n_3$ is the hour $\{0\dots23\}$, $+n_4$ is the day $\{1\dots31\}$, $+n_5$ is the month $\{1\dots12\}$ and $+n_6$ is the year (e.g., 1991).

See: **A.10.6.2.2292** TIME&DATE.

11 The optional File-Access word set

11.1 Introduction

These words provide access to mass storage in the form of "files" under the following assumptions:

- files are provided by a host operating system;

- file names are represented as character strings;

- the format of file names is determined by the host operating system;

- an open file is identified by a single-cell file identifier (*fileid*);

- file-state information (e.g., position, size) is managed by the host operating system;

- file contents are accessed as a sequence of characters;

- file read operations return an actual transfer count, which can differ from the requested transfer count.

11.2 Additional terms

file-access method: A permissible means of accessing a file, such as "read/write" or "read only".

file position: The character offset from the start of the file.

input file: The file, containing a sequence of lines, that is the input source.

11.3 Additional usage requirements

11.3.1 Data types

Append table 11.1 to table 3.1.

Table 11.1: Data types

Symbol	Data type	Size on stack
fam	file access method	1 cell
fileid	file identifier	1 cell

11.3.1.1 File identifiers

File identifiers are implementation-dependent single-cell values that are passed to file operators to designate specific files. Opening a file assigns a file identifier, which remains valid until closed.

11.3.1.3 File access methods (11.3.1.3)

File access methods are implementation-defined single-cell values.

11.3.1.4 File names

A character string containing the name of the file. The file name may include an implementation-dependent path name. The format of file names is implementation defined.

11.3.2   Blocks in files

Blocks may, but need not, reside in files. When they do:

- – Block numbers may be mapped to one or more files by implementation-defined means. An ambiguous condition exists if a requested block number is not currently mapped;

- – An **UPDATE**d block that came from a file shall be transferred back to the same file.

11.3.3   Input source

The File-Access word set creates another input source for the text interpreter. When the input source is a text file, **BLK** shall contain zero, **SOURCE-ID** shall contain the *fileid* of that text file, and the input buffer shall contain one line of the text file. During text interpretation from a text file, the value returned by **FILE-POSITION** for the *fileid* returned by **SOURCE-ID** is undefined. A standard program shall not call **REPOSITION-FILE** on the *fileid* returned by **SOURCE-ID**.

Input with **INCLUDED**, **INCLUDE-FILE**, **LOAD** and **EVALUATE** shall be nestable in any order to at least eight levels.

A program that uses more than eight levels of input-file nesting has an environmental dependency. See: **3.3.3.5 Input buffers**, **9 The optional Exception word set**.

11.3.4   Other transient regions

The system provides transient buffers for **S"** and **S\"** strings. These buffers shall be no less than 80 characters in length, and there shall be at least two buffers. The system should be able to store two strings defined by sequential use of **S"** or **S\"**. RAM-limited systems may have environmental restrictions on the number of buffers and their lifetimes.

11.3.5   Parsing

When parsing from a text file using a space delimiter, control characters shall be treated the same as the space character.

Lines of at least 128 characters shall be supported. A program that requires lines of more than 128 characters has an environmental dependency.

A program may reposition the parse area within the input buffer by manipulating the contents of **>IN**. More extensive repositioning can be accomplished using **SAVE-INPUT** and **RESTORE-INPUT**.

See: **3.4.1 Parsing**.

11.4   Additional documentation requirements

11.4.1   System documentation

11.4.1.1   Implementation-defined options

- – file access methods used by **11.6.1.0765** BIN, **11.6.1.1010** CREATE-FILE, **11.6.1.1970** OPEN-FILE, **11.6.1.2054** R/O, **11.6.1.2056** R/W and **11.6.1.2425** W/O;

- – file exceptions;

- file line terminator (**11.6.1.2090** `READ-LINE`);

- file name format (**11.3.1.4 File names**);

- information returned by **11.6.2.1524** `FILE-STATUS`;

- input file state after an exception (**11.6.1.1717** `INCLUDE-FILE`, **11.6.1.1718** `INCLUDED`);

- maximum depth of file input nesting (**11.3.3 Input source**);

- maximum size of input line (**11.3.5 Parsing**);

- methods for mapping block ranges to files (**11.3.2 Blocks in files**);

- number of string buffers provided (**11.3.4 Other transient regions**);

- size of string buffer used by **11.3.4 Other transient regions**.

11.4.1.2 Ambiguous conditions

- attempting to position a file outside its boundaries (**11.6.1.2142** `REPOSITION-FILE`);

- attempting to read from file positions not yet written (**11.6.1.2080** `READ-FILE`, **11.6.1.2090** `READ-LINE`);

- *fileid* is invalid (**11.6.1.1717** `INCLUDE-FILE`);

- I/O exception reading or closing *fileid* (**11.6.1.1717** `INCLUDE-FILE`, **11.6.1.1718** `INCLUDED`);

- named file cannot be opened (**11.6.1.1718** `INCLUDED`);

- requesting an unmapped block number (**11.3.2 Blocks in files**);

- using **11.6.1.2218** `SOURCE-ID` when **7.6.1.0790** `BLK` is not zero.

- a file is required while it is being **REQUIRED** (11.6.2.2144.50) or **INCLUDED** (11.6.1.1718).

- a marker is defined outside and executed inside a file or vice versa, and the file is **REQUIRED** (11.6.2.2144.50) again.

- the same file is required twice using different names (e.g., through symbolic links), or different files with the same name are provided to **11.6.2.2144.50** `REQUIRED` (by doing some renaming between the invocations of **REQUIRED**).

- the stack effect of including with **11.6.2.2144.50** `REQUIRED` the file is not ($i*x--$ $i*x$) .

11.4.1.3 Other system documentation

- no additional requirements.

11.4.2 Program documentation

11.4.2.1 Environmental dependencies

- requiring lines longer than 128 characters (**11.3.5 Parsing**);

- using more than eight levels of input-file nesting (**11.3.3 Input source**).

! " # $ % & ' ( ) * + , - . / digits : ; < = > ? @ ALPHA [\ ] ^ _ ` alpha { | } ~

11.4.2.2 Other program documentation

– no additional requirements.

11.5 Compliance and labeling

11.5.1 Forth-2012 systems

The phrase "Providing the File Access word set" shall be appended to the label of any Standard System that provides all of the File Access word set.

The phrase "Providing *name(s)* from the File Access Extensions word set" shall be appended to the label of any Standard System that provides portions of the File Access Extensions word set.

The phrase "Providing the File Access Extensions word set" shall be appended to the label of any Standard System that provides all of the File Access and File Access Extensions word sets.

11.5.2 Forth-2012 programs

The phrase "Requiring the File Access word set" shall be appended to the label of Standard Programs that require the system to provide the File Access word set.

The phrase "Requiring *name(s)* from the File Access Extensions word set" shall be appended to the label of Standard Programs that require the system to provide portions of the File Access Extensions word set.

The phrase "Requiring the File Access Extensions word set" shall be appended to the label of Standard Programs that require the system to provide all of the File Access and File Access Extensions word sets.

11.6 Glossary

11.6.1 File Access words

11.6.1.0080   ("paren"   FILE

(*"ccc⟨paren⟩"* ––)

Extend the semantics of **6.1.0080** `(` to include:

When parsing from a text file, if the end of the parse area is reached before a right parenthesis is found, refill the input buffer from the next line of the file, set **>IN** to zero, and resume parsing, repeating this process until either a right parenthesis is found or the end of the file is reached.

11.6.1.0765   BIN   FILE

(*fam₁* –– *fam₂*)

Modify the implementation-defined file access method *fam₁* to additionally select a "binary", i.e., not line oriented, file access method, giving access method *fam₂*.

See: **11.6.1.2054** `R/O`, **11.6.1.2056** `R/W`, **11.6.1.2425** `W/O`, **A.11.6.1.0765** `BIN`.

11.6.1.0900   **CLOSE−FILE**   FILE

(*fileid −− ior*)

Close the file identified by *fileid*. *ior* is the implementation-defined I/O result code.

11.6.1.1010   **CREATE−FILE**   FILE

(*c-addr u fam −− fileid ior*)

Create the file named in the character string specified by *c-addr* and *u*, and open it with file access method *fam*. The meaning of values of *fam* is implementation defined. If a file with the same name already exists, recreate it as an empty file.

If the file was successfully created and opened, *ior* is zero, *fileid* is its identifier, and the file has been positioned to the start of the file.

Otherwise, *ior* is the implementation-defined I/O result code and *fileid* is undefined.

See: **A.11.6.1.1010** CREATE-FILE.

11.6.1.1190   **DELETE−FILE**   FILE

(*c-addr u −− ior*)

Delete the file named in the character string specified by *c-addr u*. *ior* is the implementation-defined I/O result code.

11.6.1.1520   **FILE−POSITION**   FILE

(*fileid −− ud ior*)

ud is the current file position for the file identified by *fileid*. *ior* is the implementation-defined I/O result code. *ud* is undefined if *ior* is non-zero.

11.6.1.1522   **FILE−SIZE**   FILE

(*fileid −− ud ior*)

ud is the size, in characters, of the file identified by *fileid*. *ior* is the implementation-defined I/O result code. This operation does not affect the value returned by **FILE−POSITION**. *ud* is undefined if *ior* is non-zero.

11.6.1.1717   **INCLUDE-FILE**   FILE

(*i* * *x fileid* −− *j* * *x*)

Remove *fileid* from the stack. Save the current input source specification, including the current value of **SOURCE-ID**. Store *fileid* in **SOURCE-ID**. Make the file specified by *fileid* the input source. Store zero in **BLK**. Other stack effects are due to the words included.

Repeat until end of file: read a line from the file, fill the input buffer from the contents of that line, set **>IN** to zero, and interpret.

Text interpretation begins at the file position where the next file read would occur.

When the end of the file is reached, close the file and restore the input source specification to its saved value.

An ambiguous condition exists if *fileid* is invalid, if there is an I/O exception reading *fileid*, or if an I/O exception occurs while closing *fileid*. When an ambiguous condition exists, the status (open or closed) of any files that were being interpreted is implementation-defined.

See: **11.3.3 Input source**, **A.11.6.1.1717** INCLUDE-FILE.

11.6.1.1718   **INCLUDED**   FILE

(*i* * *x c-addr u* −− *j* * *x*)

Remove *c-addr u* from the stack. Save the current input source specification, including the current value of **SOURCE-ID**. Open the file specified by *c-addr u*, store the resulting *fileid* in **SOURCE-ID**, and make it the input source. Store zero in **BLK**. Other stack effects are due to the words included.

Repeat until end of file: read a line from the file, fill the input buffer from the contents of that line, set **>IN** to zero, and interpret.

Text interpretation begins at the start of the file.

When the end of the file is reached, close the file and restore the input source specification to its saved value.

An ambiguous condition exists if the named file can not be opened, if an I/O exception occurs reading the file, or if an I/O exception occurs while closing the file. When an ambiguous condition exists, the status (open or closed) of any files that were being interpreted is implementation-defined.

INCLUDED may allocate memory in data space before it starts interpreting the file.

See: **11.6.1.1717** INCLUDE-FILE, **A.11.6.1.1718** INCLUDED.

11.6.1.1970   **OPEN-FILE**   FILE

(*c-addr u fam −− fileid ior*)

Open the file named in the character string specified by *c-addr u*, with file access method indicated by *fam*. The meaning of values of *fam* is implementation defined.

If the file is successfully opened, *ior* is zero, *fileid* is its identifier, and the file has been positioned to the start of the file.

Otherwise, *ior* is the implementation-defined I/O result code and *fileid* is undefined.

See: **A.11.6.1.1970** OPEN-FILE.

11.6.1.2054   **R/O**   "r-o"   FILE

(*−− fam*)

fam is the implementation-defined value for selecting the "read only" file access method.

See: **11.6.1.1010** CREATE-FILE, **11.6.1.1970** OPEN-FILE.

11.6.1.2056   **R/W**   "r-w"   FILE

(*−− fam*)

fam is the implementation-defined value for selecting the "read/write" file access method.

See: **11.6.1.1010** CREATE-FILE, **11.6.1.1970** OPEN-FILE.

11.6.1.2080   **READ-FILE**   FILE

(*c-addr u_1 fileid −− u_2 ior*)

Read u_1 consecutive characters to *c-addr* from the current position of the file identified by *fileid*.

If u_1 characters are read without an exception, *ior* is zero and u_2 is equal to u_1.

If the end of the file is reached before u_1 characters are read, *ior* is zero and u_2 is the number of characters actually read.

If the operation is initiated when the value returned by **FILE-POSITION** is equal to the value returned by **FILE-SIZE** for the file identified by *fileid*, *ior* is zero and u_2 is zero.

If an exception occurs, *ior* is the implementation-defined I/O result code, and u_2 is the number of characters transferred to *c-addr* without an exception.

An ambiguous condition exists if the operation is initiated when the value returned by **FILE-POSITION** is greater than the value returned by **FILE-SIZE** for the file identified by *fileid*, or if the requested operation attempts to read portions of the file not written.

At the conclusion of the operation, **FILE-POSITION** returns the next file position after the last character read.

See: **A.11.6.1.2080** READ-FILE.

11.6.1.2090   **READ-LINE**   FILE

(*c-addr u₁ fileid* −− *u₂ flag ior*)

Read the next line from the file specified by *fileid* into memory at the address *c-addr*. At most u_1 characters are read. Up to two implementation-defined line-terminating characters may be read into memory at the end of the line, but are not included in the count u_2. The line buffer provided by *c-addr* should be at least u_1+2 characters long.

If the operation succeeded, *flag* is true and *ior* is zero. If a line terminator was received before u_1 characters were read, then u_2 is the number of characters, not including the line terminator, actually read ($0 <= u_2 <= u_1$). When $u_1 = u_2$ the line terminator has yet to be reached.

If the operation is initiated when the value returned by **FILE-POSITION** is equal to the value returned by **FILE-SIZE** for the file identified by *fileid*, *flag* is false, *ior* is zero, and u_2 is zero. If *ior* is non-zero, an exception occurred during the operation and *ior* is the implementation-defined I/O result code.

An ambiguous condition exists if the operation is initiated when the value returned by **FILE-POSITION** is greater than the value returned by **FILE-SIZE** for the file identified by *fileid*, or if the requested operation attempts to read portions of the file not written.

At the conclusion of the operation, **FILE-POSITION** returns the next file position after the last character read.

See: **A.11.6.1.2090** READ-LINE.

11.6.1.2142   **REPOSITION-FILE**   FILE

(*ud fileid* −− *ior*)

Reposition the file identified by *fileid* to *ud*. *ior* is the implementation-defined I/O result code. An ambiguous condition exists if the file is positioned outside the file boundaries.

At the conclusion of the operation, **FILE-POSITION** returns the value *ud*.

11.6.1.2147   **RESIZE-FILE**   FILE

(*ud fileid* −− *ior*)

Set the size of the file identified by *fileid* to *ud*. *ior* is the implementation-defined I/O result code.

If the resultant file is larger than the file before the operation, the portion of the file added as a result of the operation might not have been written.

At the conclusion of the operation, **FILE-SIZE** returns the value *ud* and **FILE-POSITION** returns an unspecified value.

See: **11.6.1.2080** READ-FILE, **11.6.1.2090** READ-LINE.

11.6.1.2165   **S"**   "s-quote"   FILE

Extend the semantics of **6.1.2165** S" to be:

Interpretation: (*"ccc⟨quote⟩"* −− *c-addr u*)

Parse *ccc* delimited by **"** (double quote). Store the resulting string in a transient buffer *c-addr u*.

Compilation: (*"ccc⟨quote⟩"* −−)

Parse *ccc* delimited by **"** (double quote). Append the run-time semantics given below to the current definition.

Run-time: (−− *c-addr u*)

Return *c-addr* and *u* that describe a string consisting of the characters *ccc*.

See: **3.4.1 Parsing**, **6.2.0855** C", **6.1.2165** S", **11.3.4 Other transient regions**, **11.6.2.2266** S\", **A.11.3.4 Other transient regions**, **A.11.6.1.2165** S".

11.6.1.2218   **SOURCE-ID**   "source-i-d"   FILE

(−− *0 | -1 | fileid*)

Extend **6.2.2218** SOURCE-ID to include text-file input as follows:

SOURCE-ID	Input source
fileid	Text file "*fileid*"
-1	String (via **EVALUATE**)
0	User input device

An ambiguous condition exists if **SOURCE-ID** is used when **BLK** contains a non-zero value.

11.6.1.2425   **W/O**   "w-o"   FILE

(−− *fam*)

fam is the implementation-defined value for selecting the "write only" file access method.

See: **11.6.1.1010** CREATE-FILE, **11.6.1.1970** OPEN-FILE.

11.6.1.2480   WRITE-FILE   FILE

(*c-addr u fileid — ior*)

Write *u* characters from *c-addr* to the file identified by *fileid* starting at its current position. *ior* is the implementation-defined I/O result code.

At the conclusion of the operation, **FILE-POSITION** returns the next file position after the last character written to the file, and **FILE-SIZE** returns a value greater than or equal to the value returned by **FILE-POSITION**.

See: **11.6.1.2080** READ-FILE, **11.6.1.2090** READ-LINE.

11.6.1.2485   WRITE-LINE   FILE

(*c-addr u fileid — ior*)

Write *u* characters from *c-addr* followed by the implementation-dependent line terminator to the file identified by *fileid* starting at its current position. *ior* is the implementation-defined I/O result code.

At the conclusion of the operation, **FILE-POSITION** returns the next file position after the last character written to the file, and **FILE-SIZE** returns a value greater than or equal to the value returned by **FILE-POSITION**.

See: **11.6.1.2080** READ-FILE, **11.6.1.2090** READ-LINE.

11.6.2   File-Access extension words

11.6.2.1524   FILE-STATUS   FILE EXT

(*c-addr u — x ior*)

Return the status of the file identified by the character string *c-addr u*. If the file exists, *ior* is zero; otherwise *ior* is the implementation-defined I/O result code. *x* contains implementation-defined information about the file.

11.6.2.1560   FLUSH-FILE   FILE EXT

(*fileid — ior*)

Attempt to force any buffered information written to the file referred to by *fileid* to be written to mass storage, and the size information for the file to be recorded in the storage directory if changed. If the operation is successful, *ior* is zero. Otherwise, it is an implementation-defined I/O result code.

11.6.2.1714 **`INCLUDE`**   FILE EXT

(*i* * *x* "*name*" −− *j* * *x*)

Skip leading white space and parse *name* delimited by a white space character. Push the address and length of the *name* on the stack and perform the function of **`INCLUDED`**.

See: **11.6.1.1718** `INCLUDED`, **A.11.6.2.1714** `INCLUDE`.

11.6.2.2125 **`REFILL`**   FILE EXT

(−− *flag*)

Extend the execution semantics of **6.2.2125** `REFILL` with the following:

When the input source is a text file, attempt to read the next line from the text-input file. If successful, make the result the current input buffer, set **`>IN`** to zero, and return *true*. Otherwise return *false*.

See: **6.2.2125** `REFILL`, **7.6.2.2125** `REFILL`.

11.6.2.2130 **`RENAME-FILE`**   FILE EXT

(*c-addr$_1$* *u$_1$* *c-addr$_2$* *u$_2$* −− *ior*)

Rename the file named by the character string *c-addr$_1$* *u$_1$* to the name in the character string *c-addr$_2$* *u$_2$*. *ior* is the implementation-defined I/O result code.

11.6.2.2144.10 REQUIRE   FILE EXT

(*i* * *x* "*name*" −− *i* * *x*)

Skip leading white space and parse *name* delimited by a white space character. Push the address and length of the *name* on the stack and perform the function of **`REQUIRED`**.

See: **11.6.2.2144.50** `REQUIRED`, **A.11.6.2.2144.10** `REQUIRE`.

11.6.2.2144.50 REQUIRED   FILE EXT

(*i* * *x* *c-addr* *u* −− *i* * *x*)

If the file specified by *c-addr u* has been **`INCLUDED`** or **`REQUIRED`** already, but not between the definition and execution of a marker (or equivalent usage of **`FORGET`**), discard *c-addr u*; otherwise, perform the function of **`INCLUDED`**.

An ambiguous condition exists if a file is **`REQUIRED`** while it is being **`REQUIRED`** or **`INCLUDED`**.

An ambiguous condition exists, if a marker is defined outside and executed inside a file or vice versa, and the file is **`REQUIRED`** again.

An ambiguous condition exists if the same file is **REQUIRED** twice using different names (e.g., through symbolic links), or different files with the same name are **REQUIRED** (by doing some renaming between the invocations of **REQUIRED**).

An ambiguous condition exists if the stack effect of including the file is not ($i*x$ -- $i*x$).

See: **A.11.6.2.2144.50** REQUIRED.

11.6.2.2266   **S\"**   "s-backslash-quote"   FILE EXT

Extend the semantics of **6.2.2266** S\" to be:

Interpretation: (*"ccc⟨quote⟩"* -- *c-addr u*)

Parse *ccc* delimited by **"** (double quote) according to the translation rules given in **6.2.2266** S\". Store the resulting string in a transient buffer *c-addr u*.

Compilation: (*"ccc⟨quote⟩"* --)

Parse *ccc* delimited by **"** (double quote) according to the translation rules given in **6.2.2266** S\". Append the run-time semantics given below to the current definition.

Run-time: (-- *c-addr u*)

Return a string *c-addr u* describing the translation of *ccc*. A program shall not alter the returned string.

See: **6.2.2266** S\" **11.3.4 Other transient regions**, **11.6.1.2165** S", **A.11.3.4 Other transient regions**.

---

12   The optional Floating-Point word set

12.1   Introduction

12.2   Additional terms and notation

12.2.1   Definition of terms

float-aligned address: The address of a memory location at which a floating-point number can be accessed.

double-float-aligned address: The address of a memory location at which a 64-bit IEEE double-precision floating-point number can be accessed.

single-float-aligned address: The address of a memory location at which a 32-bit IEEE single-precision floating-point number can be accessed.

IEEE floating-point number: A single- or double-precision floating-point number as defined in **ANSI/IEEE 754-1985**.

12.2.2   Notation

12.2.2.2   Stack notation

Floating-point stack notation is:

(F: *before* −− *after*)

A unified stack notation is provided for systems with the environmental restriction that the floating-point numbers are kept on the data stack.

12.3   Additional usage requirements

12.3.1   Data types

Append table 12.1 to table 3.1.

<div align="center">

Table 12.1: Data Types

Symbol	Data type	Size on stack
df-addr	double-float-aligned address	1 cell
f-addr	float-aligned address	1 cell
r	floating-point number	implementation-defined
sf-addr	single-float-aligned address	1 cell

</div>

12.3.1.1   Addresses

The set of float-aligned addresses is an implementation-defined subset of the set of aligned addresses. Adding the size of a floating-point number to a float-aligned address shall produce a float-aligned address.

The set of double-float-aligned addresses is an implementation-defined subset of the set of aligned addresses. Adding the size of a 64-bit IEEE double-precision floating-point number to a double-float-aligned address shall produce a double-float-aligned address.

The set of single-float-aligned addresses is an implementation-defined subset of the set of aligned addresses. Adding the size of a 32-bit IEEE single-precision floating-point number to a single-float-aligned address shall produce a single-float-aligned address.

12.3.1.2 Floating-point numbers

The internal representation of a floating-point number, including the format and precision of the significand and the format and range of the exponent, is implementation defined.

Any rounding or truncation of floating-point numbers is implementation defined.

12.3.2 Floating-point operations

"Round to nearest" means round the result of a floating-point operation to the representable value nearest the result. If the two nearest representable values are equally near the result, the one having zero as its least significant bit shall be delivered.

"Round toward negative infinity" means round the result of a floating-point operation to the representable value nearest to and no greater than the result.

"Round toward zero" means round the result of a floating-point operation to the representable value nearest to zero, frequently referred to as "truncation".

12.3.3 Floating-point stack

A last in, first out list that shall be used by all floating-point operators.

The width of the floating-point stack is implementation-defined. The floating-point stack shall be separate from the data and return stacks.

The size of a floating-point stack shall be at least 6 items.

A program that depends on the floating-point stack being larger than six items has an environmental dependency.

12.3.4 Environmental queries

Append table 12.2 to table 3.5.

See: **3.2.6 Environmental queries**.

12.3.5 Address alignment

Since the address returned by a **CREATE**d word is not necessarily aligned for any particular class of floating-point data, a program shall align the address (to be float aligned, single-float aligned, or double-float aligned) before accessing floating-point data at the address.

See: **3.3.3.1 Address alignment**, **12.3.1.1 Addresses**.

Table 12.2: Environmental Query Strings

String	Value data type	Constant?	Meaning
FLOATING-STACK	*n*	yes	the maximum depth of the separate floating-point stack. On systems with the environmental restriction of keeping floating-point items on the data stack, $n = 0$.
MAX-FLOAT	*r*	yes	largest usable floating-point number

12.3.6 Variables

A program may address memory in data space regions made available by **FVARIABLE**. These regions may be non-contiguous with regions subsequently allocated with **,** (comma) or **ALLOT**. See: **3.3.3.3 Variables**.

12.3.7 Text interpreter input number conversion

If the Floating-Point word set is present in the dictionary and the current base is **DECIMAL**, the input number-conversion algorithm shall be extended to recognize floating-point numbers in this form:

$$\text{Convertible string} := \langle significand \rangle \langle exponent \rangle$$
$$\langle significand \rangle := [\langle sign \rangle]\langle digits \rangle[\textbf{.}\langle digits0 \rangle]$$
$$\langle exponent \rangle := \textbf{E}[\langle sign \rangle]\langle digits0 \rangle$$
$$\langle sign \rangle := \{ \textbf{+} | \textbf{-} \}$$
$$\langle digits \rangle := \langle digit \rangle \langle digits0 \rangle$$
$$\langle digits0 \rangle := \langle digit \rangle *$$
$$\langle digit \rangle := \{ \textbf{0} | \textbf{1} | \textbf{2} | \textbf{3} | \textbf{4} | \textbf{5} | \textbf{6} | \textbf{7} | \textbf{8} | \textbf{9} \}$$

These are examples of valid representations of floating-point numbers in program source:

```
        1E      1.E      1.E0      +1.23E-1      -1.23E+1
```

See: **3.4.1.3 Text interpreter input number conversion**, **12.6.1.0558** >FLOAT.

12.4 Additional documentation requirements

12.4.1 System documentation

12.4.1.1 Implementation-defined options

- format and range of floating-point numbers (**12.3.1 Data types**, **12.6.1.2143** REPRESENT)

- results of **12.6.1.2143** REPRESENT when *float* is out of range;

- rounding or truncation of floating-point numbers (**12.3.1.2 Floating-point numbers**);

- size of floating-point stack (**12.3.3 Floating-point stack**);

- width of floating-point stack (**12.3.3 Floating-point stack**).

12.4.1.2 Ambiguous conditions

- **DF@** or **DF!** is used with an address that is not double-float aligned;

- **F@** or **F!** is used with an address that is not float aligned;

- floating point result out of range (e.g., in **12.6.1.1430** F/);

- **SF@** or **SF!** is used with an address that is not single-float aligned;

- **BASE** is not decimal (**12.6.1.2143** REPRESENT, **12.6.2.1427** F., **12.6.2.1513** FE., **12.6.2.1613** FS.);

- both arguments equal zero (**12.6.2.1489** FATAN2);

- cosine of argument is zero for **12.6.2.1625** FTAN;

- *d* can't be precisely represented as *float* in **12.6.1.1130** D>F;

- dividing by zero (**12.6.1.1430** F/);

- exponent too big for conversion (**12.6.2.1203** DF!, **12.6.2.1204** DF@, **12.6.2.2202** SF!, **12.6.2.2203** SF@);

- *float* less than one (**12.6.2.1477** FACOSH);

- *float* less than or equal to minus-one (**12.6.2.1554** FLNP1);

- *float* less than or equal to zero (**12.6.2.1553** FLN, **12.6.2.1557** FLOG);

- *float* less than zero (**12.6.2.1618** FSQRT);

- *float* magnitude greater than one (**12.6.2.1476** FACOS, **12.6.2.1486** FASIN, **12.6.2.1491** FATANH);

- integer part of *float* can't be represented by *d* in **12.6.1.1470** F>D;

- string larger than pictured-numeric output area (**12.6.2.1427** F., **12.6.2.1513** FE., **12.6.2.1613** FS.).

- *n* can't be precisely represented as *float* in **12.6.2.2175** S>F;

- integer part of *float* can't be represented by *n* in **12.6.2.1471** F>S.

12.4.1.3   Other system documentation

- no additional requirements.

12.4.1.4   Environmental restrictions

- Keeping floating-point numbers on the data stack.

12.4.2   Program documentation

12.4.2.1   Environmental dependencies

- requiring the floating-point stack to be larger than six items (**12.3.3 Floating-point stack**).

- requiring floating-point numbers to be kept on the data stack, with *n* cells per floating point number.

12.4.2.2   Other program documentation

- no additional requirements.

12.5   Compliance and labeling

12.5.1   Forth-2012 systems

The phrase "Providing the Floating-Point word set" shall be appended to the label of any Standard System that provides all of the Floating-Point word set.

The phrase "Providing *name(s)* from the Floating-Point Extensions word set" shall be appended to the label of any Standard System that provides portions of the Floating-Point Extensions word set.

The phrase "Providing the Floating-Point Extensions word set" shall be appended to the label of any Standard System that provides all of the Floating-Point and Floating-Point Extensions word sets.

12.5.2   Forth-2012 programs

The phrase "Requiring the Floating-Point word set" shall be appended to the label of Standard Programs that require the system to provide the Floating-Point word set.

The phrase "Requiring *name(s)* from the Floating-Point Extensions word set" shall be appended to the label of Standard Programs that require the system to provide portions of the Floating-Point Extensions word set.

The phrase "Requiring the Floating-Point Extensions word set" shall be appended to the label of Standard Programs that require the system to provide all of the Floating-Point and Floating-Point Extensions word sets.

12.6   Glossary

12.6.1   Floating-Point words

12.6.1.0558   >FLOAT   "to-float"   FLOATING

(*c-addr u −− true | false*) (F: *−− r |*) or (*c-addr u −− r true | false*)

An attempt is made to convert the string specified by *c-addr* and *u* to internal floating-point representation. If the string represents a valid floating-point number in the syntax below, its value *r* and *true* are returned. If the string does not represent a valid floating-point number only *false* is returned.

A string of blanks should be treated as a special case representing zero.

The syntax of a convertible string

$$
\begin{aligned}
&:= \langle significand\rangle[\langle exponent\rangle] \\
\langle significand\rangle &:= [\langle sign\rangle]\{ \langle digits\rangle[.\langle digits0\rangle] | .\langle digits\rangle \} \\
\langle exponent\rangle &:= \langle marker\rangle\langle digits0\rangle \\
\langle marker\rangle &:= \{\langle e\text{-}form\rangle | \langle sign\text{-}form\rangle \} \\
\langle e\text{-}form\rangle &:= \langle e\text{-}char\rangle[\langle sign\text{-}form\rangle] \\
\langle sign\text{-}form\rangle &:= \{ + | - \} \\
\langle e\text{-}char\rangle &:= \{ D | d | E | e \}
\end{aligned}
$$

See: **A.12.6.1.0558** >FLOAT.

12.6.1.1130   **D>F**   "d-to-f"   FLOATING

(*d* −−) (F: −− *r*) or (*d* −− *r*)

r is the floating-point equivalent of *d*. An ambiguous condition exists if *d* cannot be precisely represented as a floating-point value.

12.6.1.1400   **F!**   "f-store"   FLOATING

(*f-addr* −−) (F: *r* −−) or (*r f-addr* −−)

Store *r* at *f-addr*.

12.6.1.1410   **F***   "f-star"   FLOATING

(F: r_1 r_2 −− r_3) or (r_1 r_2 −− r_3)

Multiply r_1 by r_2 giving r_3.

12.6.1.1420   **F+**   "f-plus"   FLOATING

(F: r_1 r_2 −− r_3) or (r_1 r_2 −− r_3)

Add r_1 to r_2 giving the sum r_3.

12.6.1.1425   **F−**   "f-minus"   FLOATING

(F: r_1 r_2 −− r_3) or (r_1 r_2 −− r_3)

Subtract r_2 from r_1, giving r_3.

12.6.1.1430   **F/**   "f-slash"   FLOATING

(F: r_1 r_2 −− r_3) or (r_1 r_2 −− r_3)

Divide r_1 by r_2, giving the quotient r_3. An ambiguous condition exists if r_2 is zero, or the quotient lies outside of the range of a floating-point number.

12.6.1.1440   **F0<**   "f-zero-less-than"   FLOATING

(−− *flag*) (F: *r* −−) or (*r* −− *flag*)

flag is true if and only if *r* is less than zero.

12.6.1.1450   **F0=**   "f-zero-equals"   FLOATING

  (-- *flag*) (F: *r* --) or (*r* -- *flag*)

  flag is true if and only if *r* is equal to zero.

12.6.1.1460   **F<**   "f-less-than"   FLOATING

  (-- *flag*) (F: r_1 r_2 --) or (r_1 r_2 -- *flag*)

  flag is true if and only if r_1 is less than r_2.

12.6.1.1470   **F>D**   "f-to-d"   FLOATING

  (-- *d*) (F: *r* --) or (*r* -- *d*)

  d is the double-cell signed-integer equivalent of the integer portion of *r*. The fractional portion of *r* is discarded. An ambiguous condition exists if the integer portion of *r* cannot be represented as a double-cell signed integer.

  Note: Rounding the floating-point value prior to calling **F>D** is advised, because **F>D** rounds towards zero.

12.6.1.1472   **F@**   "f-fetch"   FLOATING

  (*f-addr* --) (F: -- *r*) or (*f-addr* -- *r*)

  r is the value stored at *f-addr*.

12.6.1.1479   **FALIGN**   "f-align"   FLOATING

  (--)

  If the data-space pointer is not float aligned, reserve enough data space to make it so.

12.6.1.1483   **FALIGNED**   "f-aligned"   FLOATING

  (*addr* -- *f-addr*)

  f-addr is the first float-aligned address greater than or equal to *addr*.

12.6.1.1492   **FCONSTANT**   "f-constant"   FLOATING

  ("⟨spaces⟩name" --) (F: *r* --) or (*r* "⟨spaces⟩name" --)

  Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below.

name is referred to as an "f-constant".

name Execution: (‒‒) (F: ‒‒ *r*) or (‒‒ *r*)

Place *r* on the floating-point stack.

See: **3.4.1 Parsing**, **A.12.6.1.1492** FCONSTANT.

12.6.1.1497   **FDEPTH**   "f-depth"   FLOATING

(‒‒ +*n*)

+*n* is the number of values contained on the floating-point stack. If the system has an environmental restriction of keeping the floating-point numbers on the data stack, +*n* is the current number of possible floating-point values contained on the data stack.

12.6.1.1500   **FDROP**   "f-drop"   FLOATING

(F: *r* ‒‒) or (*r* ‒‒)

Remove *r* from the floating-point stack.

12.6.1.1510   **FDUP**   "f-dupe"   FLOATING

(F: *r* ‒‒ *r r*) or (*r* ‒‒ *r r*)

Duplicate *r*.

12.6.1.1552   **FLITERAL**   "f-literal"   FLOATING

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (F: *r* ‒‒) or (*r* ‒‒)

Append the run-time semantics given below to the current definition.

Run-time: (F: ‒‒ *r*) or (‒‒ *r*)

Place *r* on the floating-point stack.

See: **A.12.6.1.1552** FLITERAL.

12.6.1.1555   **FLOAT+**   "float-plus"   FLOATING

(*f-addr₁* ‒‒ *f-addr₂*)

Add the size in address units of a floating-point number to *f-addr₁*, giving *f-addr₂*.

12.6.1.1556   **FLOATS**   FLOATING

   $(n_1 -- n_2)$

   n_2 is the size in address units of n_1 floating-point numbers.

12.6.1.1558   **FLOOR**   FLOATING

   $(F: r_1 -- r_2)$ or $(r_1 -- r_2)$

   Round r_1 to an integral value using the "round toward negative infinity" rule, giving r_2.

   See: **12.3.2 Floating-point operations**, **12.6.1.1612** FROUND, **12.6.2.1627** FTRUNC.

12.6.1.1562   **FMAX**   "f-max"   FLOATING

   $(F: r_1 r_2 -- r_3)$ or $(r_1 r_2 -- r_3)$

   r_3 is the greater of r_1 and r_2.

12.6.1.1565   **FMIN**   "f-min"   FLOATING

   $(F: r_1 r_2 -- r_3)$ or $(r_1 r_2 -- r_3)$

   r_3 is the lesser of r_1 and r_2.

12.6.1.1567   **FNEGATE**   "f-negate"   FLOATING

   $(F: r_1 -- r_2)$ or $(r_1 -- r_2)$

   r_2 is the negation of r_1.

12.6.1.1600   **FOVER**   "f-over"   FLOATING

   $(F: r_1 r_2 -- r_1 r_2 r_1)$ or $(r_1 r_2 -- r_1 r_2 r_1)$

   Place a copy of r_1 on top of the floating-point stack.

12.6.1.1610   **FROT**   "f-rote"   FLOATING

   $(F: r_1 r_2 r_3 -- r_2 r_3 r_1)$ or $(r_1 r_2 r_3 -- r_2 r_3 r_1)$

   Rotate the top three floating-point stack entries.

12.6.1.1612 **FROUND**   "f-round"   FLOATING

(F: r_1 −− r_2) or (r_1 −− r_2)

Round r_1 to an integral value using the "round to nearest" rule, giving r_2.

See: **12.3.2 Floating-point operations**, **12.6.1.1558** FLOOR, **12.6.2.1627** FTRUNC.

12.6.1.1620 **FSWAP**   "f-swap"   FLOATING

(F: r_1 r_2 −− r_2 r_1) or (r_1 r_2 −− r_2 r_1)

Exchange the top two floating-point stack items.

12.6.1.1630 **FVARIABLE**   "f-variable"   FLOATING

("⟨spaces⟩name" −−)

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for
name with the execution semantics defined below. Reserve 1 **FLOATS** address units of
data space at a float-aligned address.

name is referred to as an "f-variable".

name Execution: (−− *f-addr*)

f-addr is the address of the data space reserved by **FVARIABLE** when it created *name*.
A program is responsible for initializing the contents of the reserved space.

See: **3.4.1 Parsing**, **A.12.6.1.1630** FVARIABLE.

12.6.1.2143 **REPRESENT**   FLOATING

(*c-addr u* −− *n flag$_1$ flag$_2$*) (F: *r* −−) or (*r c-addr u* −− *n flag$_1$ flag$_2$*)

At *c-addr*, place the character-string external representation of the significand of the
floating-point number *r*. Return the decimal-base exponent as *n*, the sign as *flag$_1$* and
"valid result" as *flag$_2$*. The character string shall consist of the *u* most significant digits
of the significand represented as a decimal fraction with the implied decimal point to the
left of the first digit, and the first digit zero only if all digits are zero. The significand
is rounded to *u* digits following the "round to nearest" rule; *n* is adjusted, if necessary,
to correspond to the rounded magnitude of the significand. If *flag$_2$* is *true* then *r* was
in the implementation-defined range of floating-point numbers. If *flag$_1$* is *true* then *r* is
negative.

An ambiguous condition exists if the value of **BASE** is not decimal ten.

When *flag$_2$* is *false*, *n* and *flag$_1$* are implementation defined, as are the contents of *c-addr*.
Under these circumstances, the string at *c-addr* shall consist of graphic characters.

See: **3.2.1.2 Digit conversion**, **6.1.0750** BASE, **12.3.2 Floating-point operations**,
A.12.6.1.2143 REPRESENT.

12.6.2   Floating-Point extension words

12.6.2.1203   DF !   "d-f-store"   FLOATING EXT

(*df-addr −−*) (F: *r −−*) or (*r df-addr −−*)

Store the floating-point number *r* as a 64-bit IEEE double-precision number at *df-addr*.
If the significand of the internal representation of *r* has more precision than the IEEE
double-precision format, it will be rounded using the "round to nearest" rule. An am-
biguous condition exists if the exponent of *r* is too large to be accommodated in IEEE
double-precision format.

See: **12.3.1.1 Addresses**, **12.3.2 Floating-point operations**.

12.6.2.1204   DF@   "d-f-fetch"   FLOATING EXT

(*df-addr −−*) (F: *−− r*) or (*df-addr −− r*)

Fetch the 64-bit IEEE double-precision number stored at *df-addr* to the floating-point
stack as *r* in the internal representation. If the IEEE double-precision significand has
more precision than the internal representation it will be rounded to the internal repre-
sentation using the "round to nearest" rule. An ambiguous condition exists if the expo-
nent of the IEEE double-precision representation is too large to be accommodated by the
internal representation.

See: **12.3.1.1 Addresses**, **12.3.2 Floating-point operations**.

12.6.2.1205   DFALIGN   "d-f-align"   FLOATING EXT

(*−−*)

If the data-space pointer is not double-float aligned, reserve enough data space to make
it so.

See: **12.3.1.1 Addresses**.

12.6.2.1207   DFALIGNED   "d-f-aligned"   FLOATING EXT

(*addr −− df-addr*)

df-addr is the first double-float-aligned address greater than or equal to *addr*.

See: **12.3.1.1 Addresses**.

12.6.2.1207.40 DFFIELD: "d-f-field-colon"   FLOATING EXT

(n_1 "⟨spaces⟩name" –– n_2)

Skip leading space delimiters. Parse *name* delimited by a space. *Offset* is the first double-float aligned value greater than or equal to n_1. n_2 = *offset + 1* double-float.

Create a definition for *name* with the execution semantics given below.

name Execution: ($addr_1$ –– $addr_2$)

Add the *offset* calculated during the compile-time action to $addr_1$ giving the address $addr_2$.

See: **10.6.2.0135** +FIELD, **10.6.2.0763** BEGIN-STRUCTURE, **10.6.2.1336** END-STRUCTURE, **A.10.6.2.1518** FIELD:.

12.6.2.1208   DFLOAT+   "d-float-plus"   FLOATING EXT

(*df-addr$_1$* –– *df-addr$_2$*)

Add the size in address units of a 64-bit IEEE double-precision number to *df-addr$_1$*, giving *df-addr$_2$*.

See: **12.3.1.1 Addresses**.

12.6.2.1209   DFLOATS   "d-floats"   FLOATING EXT

(n_1 –– n_2)

n_2 is the size in address units of n_1 64-bit IEEE double-precision numbers.

12.6.2.1415   F**   "f-star-star"   FLOATING EXT

(F: r_1 r_2 –– r_3) or (r_1 r_2 –– r_3)

Raise r_1 to the power r_2, giving the product r_3.

12.6.2.1427   F.   "f-dot"   FLOATING EXT

(––) (F: r ––) or (r ––)

Display, with a trailing space, the top number on the floating-point stack using fixed-point notation:

[-] ⟨digits⟩.⟨digits0⟩

An ambiguous condition exists if the value of **BASE** is not (decimal) ten or if the character string representation exceeds the size of the pictured numeric output string buffer.

See: **12.6.1.0558** >FLOAT, **A.12.6.2.1427** F.

12.6.2.1471   **F>S**   "F to S"   FLOATING EXT

($--$ n) (F: r $--$) or (r $--$ n)

n is the single-cell signed-integer equivalent of the integer portion of r. The fractional portion of r is discarded. An ambiguous condition exists if the integer portion of r cannot be represented as a single-cell signed integer.

Note: Rounding the floating-point value prior to calling **F>S** is advised, because **F>S** rounds towards zero.

See: **12.6.2.2175** S>F.

12.6.2.1474   **FABS**   "f-abs"   FLOATING EXT

(F: r_1 $--$ r_2) or (r_1 $--$ r_2)

r_2 is the absolute value of r_1.

12.6.2.1476   **FACOS**   "f-a-cos"   FLOATING EXT

(F: r_1 $--$ r_2) or (r_1 $--$ r_2)

r_2 is the principal radian angle whose cosine is r_1. An ambiguous condition exists if | r_1 | is greater than one.

12.6.2.1477   **FACOSH**   "f-a-cosh"   FLOATING EXT

(F: r_1 $--$ r_2) or (r_1 $--$ r_2)

r_2 is the floating-point value whose hyperbolic cosine is r_1. An ambiguous condition exists if r_1 is less than one.

12.6.2.1484   **FALOG**   "f-a-log"   FLOATING EXT

(F: r_1 $--$ r_2) or (r_1 $--$ r_2)

Raise ten to the power r_1, giving r_2.

12.6.2.1486   **FASIN**   "f-a-sine"   FLOATING EXT

(F: r_1 $--$ r_2) or (r_1 $--$ r_2)

r_2 is the principal radian angle whose sine is r_1. An ambiguous condition exists if | r_1 | is greater than one.

12.6.2.1487   **FASINH**   "f-a-cinch"   FLOATING EXT

(F: r_1 −− r_2) or (r_1 −− r_2)

r_2 is the floating-point value whose hyperbolic sine is r_1.

12.6.2.1488   **FATAN**   "f-a-tan"   FLOATING EXT

(F: r_1 −− r_2) or (r_1 −− r_2)

r_2 is the principal radian angle whose tangent is r_1.

12.6.2.1489   **FATAN2**   "f-a-tan-two"   FLOATING EXT

(F: r_1 r_2 −− r_3) or (r_1 r_2 −− r_3)

r_3 is the principal radian angle (between -π and π) whose tangent is r_1/r_2. A system that returns false for "−0E 0E 0E **F~**" shall return a value (approximating) −π when $r_1 =$ 0E and r_2 is negative. An ambiguous condition exists if r_1 and r_2 are zero.

See: **A.12.6.2.1489** FATAN2.

12.6.2.1491   **FATANH**   "f-a-tan-h"   FLOATING EXT

(F: r_1 −− r_2) or (r_1 −− r_2)

r_2 is the floating-point value whose hyperbolic tangent is r_1. An ambiguous condition exists if r_1 is outside the range of -1E0 to 1E0.

12.6.2.1493   **FCOS**   "f-cos"   FLOATING EXT

(F: r_1 −− r_2) or (r_1 −− r_2)

r_2 is the cosine of the radian angle r_1.

12.6.2.1494   **FCOSH**   "f-cosh"   FLOATING EXT

(F: r_1 −− r_2) or (r_1 −− r_2)

r_2 is the hyperbolic cosine of r_1.

12.6.2.1513   **FE.**   "f-e-dot"   FLOATING EXT

($--$) (F: $r --$) or ($r --$)

Display, with a trailing space, the top number on the floating-point stack using engineering notation, where the significand is greater than or equal to 1.0 and less than 1000.0 and the decimal exponent is a multiple of three.

An ambiguous condition exists if the value of **BASE** is not (decimal) ten or if the character string representation exceeds the size of the pictured numeric output string buffer.

See: **6.1.0750** BASE, **12.3.2 Floating-point operations**, **12.6.1.2143** REPRESENT.

12.6.2.1515   **FEXP**   "f-e-x-p"   FLOATING EXT

(F: $r_1 -- r_2$) or ($r_1 -- r_2$)

Raise e to the power r_1, giving r_2.

12.6.2.1516   **FEXPM1**   "f-e-x-p-m-one"   FLOATING EXT

(F: $r_1 -- r_2$) or ($r_1 -- r_2$)

Raise e to the power r_1 and subtract one, giving r_2.

See: **A.12.6.2.1516** FEXPM1.

12.6.2.1517   **FFIELD:**   "f-field-colon"   FLOATING EXT

(n_1 "⟨spaces⟩name" $-- n_2$)

Skip leading space delimiters. Parse *name* delimited by a space. *Offset* is the first float aligned value greater than or equal to n_1. $n_2 = offset + 1$ float.

Create a definition for *name* with the execution semantics given below.

name Execution: ($addr_1 -- addr_2$)

Add the *offset* calculated during the compile-time action to $addr_1$ giving the address $addr_2$.

See: **10.6.2.0135** +FIELD, **10.6.2.0763** BEGIN-STRUCTURE,
10.6.2.1336 END-STRUCTURE, **A.10.6.2.1518** FIELD:.

12.6.2.1553   **FLN**   "f-l-n"   FLOATING EXT

(F: $r_1 -- r_2$) or ($r_1 -- r_2$)

r_2 is the natural logarithm of r_1. An ambiguous condition exists if r_1 is less than or equal to zero.

12.6.2.1554   **FLNP1**   "f-l-n-p-one"   FLOATING EXT

(F: r_1 –– r_2) or (r_1 –– r_2)

r_2 is the natural logarithm of the quantity r_1 plus one. An ambiguous condition exists if r_1 is less than or equal to negative one.

See: **A.12.6.2.1554** FLNP1.

12.6.2.1557   **FLOG**   "f-log"   FLOATING EXT

(F: r_1 –– r_2) or (r_1 –– r_2)

r_2 is the base-ten logarithm of r_1. An ambiguous condition exists if r_1 is less than or equal to zero.

12.6.2.1613   **FS.**   "f-s-dot"   FLOATING EXT

(––) (F: r ––) or (r ––)

Display, with a trailing space, the top number on the floating-point stack in scientific notation: ⟨*significand*⟩⟨*exponent*⟩ where:

$$\langle significand \rangle := \text{[-]}\langle digit \rangle \textbf{.} \langle digits0 \rangle$$
$$\langle exponent \rangle := \textbf{E}\text{[-]}\langle digits \rangle$$

An ambiguous condition exists if the value of **BASE** is not (decimal) ten or if the character string representation exceeds the size of the pictured numeric output string buffer.

See: **6.1.0750** BASE, **12.3.2 Floating-point operations**, **12.6.1.2143** REPRESENT.

12.6.2.1614   **FSIN**   "f-sine"   FLOATING EXT

(F: r_1 –– r_2) or (r_1 –– r_2)

r_2 is the sine of the radian angle r_1.

12.6.2.1616   **FSINCOS**   "f-sine-cos"   FLOATING EXT

(F: r_1 –– r_2 r_3) or (r_1 –– r_2 r_3)

r_2 is the sine of the radian angle r_1. r_3 is the cosine of the radian angle r_1.

See: **A.12.6.2.1489** FATAN2.

12.6.2.1617   **FSINH**   "f-cinch"   FLOATING EXT

(F: r_1 –– r_2) or (r_1 –– r_2)

r_2 is the hyperbolic sine of r_1.

12.6.2.1618   **FSQRT**   "f-square-root"   FLOATING EXT

   (F: r_1 –– r_2) or (r_1 –– r_2)

   r_2 is the square root of r_1. An ambiguous condition exists if r_1 is less than zero.

12.6.2.1625   **FTAN**   "f-tan"   FLOATING EXT

   (F: r_1 –– r_2) or (r_1 –– r_2)

   r_2 is the tangent of the radian angle r_1. An ambiguous condition exists if $\cos(r_1)$ is zero.

12.6.2.1626   **FTANH**   "f-tan-h"   FLOATING EXT

   (F: r_1 –– r_2) or (r_1 –– r_2)

   r_2 is the hyperbolic tangent of r_1.

12.6.2.1627   **FTRUNC**   "f-trunc"   FLOATING EXT

   (F: r_1 –– r_2) or (r_1 –– r_2)

   Round r_1 to an integral value using the "round towards zero" rule, giving r_2.

   See: **12.3.2 Floating-point operations**, **12.6.1.1612** FROUND, **12.6.1.1558** FLOOR.

12.6.2.1628   **FVALUE**   "f-value"   FLOATING EXT

   (F: r ––) ("⟨spaces⟩name" ––) or (r "⟨spaces⟩name" ––)

   Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below, with an initial value equal to *r*.

   name is referred to as a "f-value".

name Execution: (F: –– r) or (–– r)

   Place *r* on the floating point stack. The value of *r* is that given when *name* was created, until the phrase "*r* **TO** *name*" is executed, causing a new value of *r* to be assigned to *name*.

TO *name* Run-time: (F: r ––) or (r ––)

   Assign the value *r* to *name*.

   See: **3.4.1 Parsing**, **6.2.2295** TO

12.6.2.1640   **F~**   "f-proximate"   FLOATING EXT

($--$ *flag*) (F: r_1 r_2 r_3 $--$) or (r_1 r_2 r_3 $--$ *flag*)

If r_3 is positive, *flag* is true if the absolute value of (r_1 minus r_2) is less than r_3.

If r_3 is zero, *flag* is true if the implementation-dependent encoding of r_1 and r_2 are exactly identical (positive and negative zero are unequal if they have distinct encodings).

If r_3 is negative, *flag* is true if the absolute value of (r_1 minus r_2) is less than the absolute value of r_3 times the sum of the absolute values of r_1 and r_2.

See: **A.12.6.2.1640** F~.

12.6.2.2035   **PRECISION**   FLOATING EXT

($--$ *u*)

Return the number of significant digits currently used by **F.**, **FE.**, or **FS.** as *u*.

12.6.2.2175   **S>F**   "S to F"   FLOATING EXT

(*n* $--$) (F: $--$ *r*) or (*n* $--$ *r*)

r is the floating-point equivalent of the single-cell value *n*. An ambiguous condition exists if *n* can not be precisely represented as a floating-point value.

See: **12.6.2.1471** F>S.

12.6.2.2200   **SET-PRECISION**   FLOATING EXT

(*u* $--$)

Set the number of significant digits currently used by **F.**, **FE.**, or **FS.** to *u*.

12.6.2.2202   **SF!**   "s-f-store"   FLOATING EXT

(*sf-addr* $--$) (F: *r* $--$) or (*r* *sf-addr* $--$)

Store the floating-point number *r* as a 32-bit IEEE single-precision number at *sf-addr*. If the significand of the internal representation of *r* has more precision than the IEEE single-precision format, it will be rounded using the "round to nearest" rule. An ambiguous condition exists if the exponent of *r* is too large to be accommodated by the IEEE single-precision format.

See: **12.3.1.1 Addresses**, **12.3.2 Floating-point operations**.

12.6.2.2203   **SF@**   "s-f-fetch"   FLOATING EXT

(*sf-addr* −−) (F: −− *r*) or (*sf-addr* −− *r*)

Fetch the 32-bit IEEE single-precision number stored at *sf-addr* to the floating-point stack as *r* in the internal representation. If the IEEE single-precision significand has more precision than the internal representation, it will be rounded to the internal representation using the "round to nearest" rule. An ambiguous condition exists if the exponent of the IEEE single-precision representation is too large to be accommodated by the internal representation.

See: **12.3.1.1 Addresses**, **12.3.2 Floating-point operations**.

12.6.2.2204   **SFALIGN**   "s-f-align"   FLOATING EXT

(−−)

If the data-space pointer is not single-float aligned, reserve enough data space to make it so.

See: **12.3.1.1 Addresses**.

12.6.2.2206   **SFALIGNED**   "s-f-aligned"   FLOATING EXT

(*addr* −− *sf-addr*)

sf-addr is the first single-float-aligned address greater than or equal to *addr*.

See: **12.3.1.1 Addresses**.

12.6.2.2206.40 SFFIELD: "s-f-field-colon"   FLOATING EXT

(n_1 "⟨*spaces*⟩*name*" −− n_2)

Skip leading space delimiters. Parse *name* delimited by a space. *Offset* is the first single-float aligned value greater than or equal to n_1. $n_2 = \textit{offset} + 1$ single-float.

Create a definition for *name* with the execution semantics given below.

name Execution: ($addr_1$ −− $addr_2$)

Add the *offset* calculated during the compile-time action to $addr_1$ giving the address $addr_2$.

See: **10.6.2.0135** +FIELD, **10.6.2.0763** BEGIN-STRUCTURE,
10.6.2.1336 END-STRUCTURE, **A.10.6.2.1518** FIELD:.

12.6.2.2207   **SFLOAT+**   "s-float-plus"   FLOATING EXT

(*sf-addr$_1$* −− *sf-addr$_2$*)

Add the size in address units of a 32-bit IEEE single-precision number to *sf-addr$_1$*, giving *sf-addr$_2$*.

See: **12.3.1.1 Addresses**.

12.6.2.2208   **SFLOATS**   "s-floats"   FLOATING EXT

(*n$_1$* −− *n$_2$*)

n$_2$ is the size in address units of *n$_1$* 32-bit IEEE single-precision numbers.

See: **12.3.1.1 Addresses**.

13 The optional Locals word set

13.1 Introduction

13.2 Additional terms and notation

None.

13.3 Additional usage requirements

13.3.1 Locals

A local is a data object whose execution semantics shall return its value, whose scope shall be limited to the definition in which it is declared, and whose use in a definition shall not preclude reentrancy or recursion.

13.3.2 Environmental queries

Append table 13.1 to table 3.5.

See: **3.2.6 Environmental queries**.

Table 13.1: Environmental Query Strings

String	Value data type	Constant?	Meaning
#LOCALS	n	yes	maximum number of local variables in a definition

13.3.3 Processing locals

To support the locals word set, a system shall provide a mechanism to receive the messages defined by **(LOCAL)** and respond as described here.

During the compilation of a definition after **:** (colon), **:NONAME**, or **DOES>**, a program may begin sending local identifier messages to the system. The process shall begin when the first message is sent. The process shall end when the "last local" message is sent. The system shall keep track of the names, order, and number of identifiers contained in the complete sequence.

13.3.3.1 Compilation semantics

The system, upon receipt of a sequence of local-identifier messages, shall take the following actions at compile time:

a) Create temporary dictionary entries for each of the identifiers passed to **(LOCAL)**, such that each identifier will behave as a *local*. These temporary dictionary entries shall vanish at the end of the definition, denoted by **;** (semicolon), **;CODE**, or **DOES>**. The system need not maintain these identifiers in the same way it does other dictionary entries as long as they can be found by normal dictionary searching processes. Furthermore, if the Search-Order word set is present, local identifiers shall

always be searched before any of the word lists in any definable search order, and none of the Search-Order words shall change the locals' privileged position in the search order. Local identifiers may reside in mass storage.

b) For each identifier passed to **(LOCAL)**, the system shall generate an appropriate code sequence that does the following at execution time:

   1) Allocate a storage resource adequate to contain the value of a local. The storage shall be allocated in a way that does not preclude re-entrancy or recursion in the definition using the local.

   2) Initialize the value using the top item on the data stack. If more than one local is declared, the top item on the stack shall be moved into the first local identified, the next item shall be moved into the second, and so on.

   The storage resource may be the return stack or may be implemented in other ways, such as in registers. The storage resource shall not be the data stack. Use of locals shall not restrict use of the data stack before or after the point of declaration.

c) Arrange that any of the legitimate methods of terminating execution of a definition, specifically **;** (semicolon), **;CODE**, **DOES>** or **EXIT**, will release the storage resource allocated for the locals, if any, declared in that definition. **ABORT** shall release all local storage resources, and **CATCH** / **THROW** (if implemented) shall release such resources for all definitions whose execution is being terminated.

d) Separate sets of locals may be declared in defining words before **DOES>** for use by the defining word, and after **DOES>** for use by the word defined.

A system implementing the Locals word set shall support the declaration of at least sixteen locals in a definition.

13.3.3.2 Syntax restrictions

Immediate words in a program may use **(LOCAL)** to implement syntaxes for local declarations with the following restrictions:

a) A program shall not compile any executable code into the current definition between the time **(LOCAL)** is executed to identify the first local for that definition and the time of sending the single required "last local" message;

b) The position in program source at which the sequence of **(LOCAL)** messages is sent, referred to here as the point at which locals are declared, shall not lie within the scope of any control structure;

c) Locals shall not be declared until values previously placed on the return stack within the definition have been removed;

d) After a definition's locals have been declared, a program may place data on the return stack. However, if this is done, locals shall not be accessed until those values have been removed from the return stack;

e) Words that return execution tokens, such as **'** (tick), **[']**, or **FIND**, shall not be used with local names;

---

f) A program that declares more than sixteen locals in a single definition has an environmental dependency;

g) Locals may be accessed or updated within control structures, including do-loops;

h) Local names shall not be referenced by **POSTPONE** and **[COMPILE]**.

See: **3.4 The Forth text interpreter**.

13.4 Additional documentation requirements

13.4.1 System documentation

13.4.1.1 Implementation-defined options

– maximum number of locals in a definition (**13.3.3 Processing locals**, **13.6.2.1795** LOCALS|).

13.4.1.2 Ambiguous conditions

– executing a named *local* while in interpretation state (**13.6.1.0086** (LOCAL));

– a local name ends in "`:`", "`[`", "`^`";

– a local name is a single non-alphabetic character;

– the text between **{:** and **:}** extends over more than one line;

– **{:** … **:}** is used more than once in a word.

13.4.1.3 Other system documentation

– no additional requirements.

13.4.2 Program documentation

13.4.2.1 Environmental dependencies

– declaring more than sixteen locals in a single definition (**13.3.3 Processing locals**).

13.4.2.2 Other program documentation

– no additional requirements.

13.5 Compliance and labeling

13.5.1 Forth-2012 systems

The phrase "Providing the Locals word set" shall be appended to the label of any Standard System that provides all of the Locals word set.

The phrase "Providing *name(s)* from the Locals Extensions word set" shall be appended to the label of any Standard System that provides portions of the Locals Extensions word set.

The phrase "Providing the Locals Extensions word set" shall be appended to the label of any Standard System that provides all of the Locals and Locals Extensions word sets.

13.5.2 Forth-2012 programs

The phrase "Requiring the Locals word set" shall be appended to the label of Standard Programs that require the system to provide the Locals word set.

The phrase "Requiring *name(s)* from the Locals Extensions word set" shall be appended to the label of Standard Programs that require the system to provide portions of the Locals Extensions word set.

The phrase "Requiring the Locals Extensions word set" shall be appended to the label of Standard Programs that require the system to provide all of the Locals and Locals Extensions word sets.

13.6 Glossary

13.6.1 Locals words

13.6.1.0086   **(LOCAL)**   "paren-local-paren"   LOCAL

   Interpretation: Interpretation semantics for this word are undefined.

      Execution: (*c-addr u --*)

            When executed during compilation, **(LOCAL)** passes a message to the system that has one of two meanings. If *u* is non-zero, the message identifies a new *local* whose definition name is given by the string of characters identified by *c-addr u*. If *u* is zero, the message is "last local" and *c-addr* has no significance.

            The result of executing **(LOCAL)** during compilation of a definition is to create a set of named local identifiers, each of which is a definition name, that only have execution semantics within the scope of that definition's source.

local Execution: (*-- x*)

            Push the local's value, *x*, onto the stack. The local's value is initialized as described in **13.3.3 Processing locals** and may be changed by preceding the local's name with **TO**. An ambiguous condition exists when local is executed while in interpretation state.

TO *local* Run-time: (*x --*)

            Assign the value *x* to the local value *local*.

         Note: This word does not have special compilation semantics in the usual sense because it provides access to a system capability for use by other user-defined words that do have them. However, the locals facility as a whole and the sequence of messages passed defines specific usage rules with semantic implications that are described in detail in section **13.3.3 Processing locals**.

         Note: This word is not intended for direct use in a definition to declare that definition's locals. It is instead used by system or user compiling words. These compiling words in turn define their own syntax, and may be used directly in definitions to declare locals. In this context, the syntax for **(LOCAL)** is defined in terms of a sequence of compile-time messages and is described in detail in section **13.3.3 Processing locals**.

         See: **3.4 The Forth text interpreter** and **6.2.2295** TO.

13.6.2   Locals extension words

13.6.2.1795   `LOCALS|`   "locals-bar"   LOCAL EXT

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (*"⟨spaces⟩name₁" "⟨spaces⟩name₂" … "⟨spaces⟩nameₙ" "|"* $--$)

Create up to eight local identifiers by repeatedly skipping leading spaces, parsing *name*, and executing **13.6.1.0086** `(LOCAL)`. The list of locals to be defined is terminated by `|`. Append the run-time semantics given below to the current definition.

Run-time: (x_n … $x_2\,x_1$ $--$)

Initialize up to eight local identifiers as described in **13.6.1.0086** `(LOCAL)`, each of which takes as its initial value the top stack item, removing it from the stack. Identifier *name₁* is initialized with x_1, identifier *name₂* with x_2, etc. When invoked, each local will return its value. The value of a local may be changed using **6.2.2295** `TO`.

Note: This word is obsolescent and is included as a concession to existing implementations.

13.6.2.2550   `{:`   "brace-colon"   LOCAL EXT

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (*i*x* "⟨spaces⟩ccc :}" $--$)

Parse *ccc* according to the following syntax:

$$\texttt{\{:} \ \langle arg\rangle\texttt{*} \ [\texttt{|} \ \langle val\rangle\texttt{*}] \ [\texttt{--} \ \langle out\rangle\texttt{*}] \ \texttt{:\}}$$

where ⟨*arg*⟩, ⟨*val*⟩ and ⟨*out*⟩ are local names, and *i* is the number of ⟨*arg*⟩ names given.

The following ambiguous conditions exist when:

- a local name ends in ":", "[", "^";

- a local name is a single non-alphabetic character;

- the text between `{:` and `:}` extends over more than one line;

- `{:` … `:}` is used more than once in a word.

Append the run-time semantics below.

Run-time: (x_1 … x_n $--$)

Create locals for ⟨*arg*⟩s and ⟨*val*⟩s. ⟨*out*⟩s are ignored.

⟨*arg*⟩ names are initialized from the data stack, with the top of the stack being assigned to the right most ⟨*arg*⟩ name.

⟨*val*⟩ names are uninitialized.

⟨*val*⟩ and ⟨*arg*⟩ names have the execution semantics given below.

name Execution: (−− *x*)

> Place the value currently assigned to *name* on the stack. An ambiguous condition exists
> when *name* is executed while in interpretation state.

TO *name* Run-time: (*x* −−)

> Set *name* to the value *x*.

> See: **2.2.5 BNF notation**, **6.2.2405** VALUE, **6.2.2295** TO, **A.13.6.2.2550** { : .

14   The optional Memory-Allocation word set

14.1   Introduction

14.2   Additional terms and notation

None.

14.3   Additional usage requirements

14.3.3   Allocated regions (14.3.3)

A program may address memory in data space regions made available by **ALLOCATE** or **RESIZE** and not yet released by **FREE**.

See: **3.3.3 Data space**.

14.4   Additional documentation requirements

None.

14.5   Compliance and labeling

14.5.1   Forth-2012 systems

The phrase "Providing the Memory-Allocation word set" shall be appended to the label of any Standard System that provides all of the Memory-Allocation word set.

The phrase "Providing *name(s)* from the Memory-Allocation Extensions word set" shall be appended to the label of any Standard System that provides portions of the Memory-Allocation Extensions word set.

The phrase "Providing the Memory-Allocation Extensions word set" shall be appended to the label of any Standard System that provides all of the Memory-Allocation and Memory-Allocation Extensions word sets.

14.5.2   Forth-2012 programs

The phrase "Requiring the Memory-Allocation word set" shall be appended to the label of Standard Programs that require the system to provide the Memory-Allocation word set.

The phrase "Requiring *name(s)* from the Memory-Allocation Extensions word set" shall be appended to the label of Standard Programs that require the system to provide portions of the Memory-Allocation Extensions word set.

The phrase "Requiring the Memory-Allocation Extensions word set" shall be appended to the label of Standard Programs that require the system to provide all of the Memory-Allocation and Memory-Allocation Extensions word sets.

14.6 Glossary

14.6.1 Memory-Allocation words

14.6.1.0707   `ALLOCATE`   MEMORY

(*u − − a-addr ior*)

Allocate *u* address units of contiguous data space. The data-space pointer is unaffected by this operation. The initial content of the allocated space is undefined.

If the allocation succeeds, *a-addr* is the aligned starting address of the allocated space and *ior* is zero.

If the operation fails, *a-addr* does not represent a valid address and *ior* is the implementation-defined I/O result code.

See: **6.1.1650** HERE, **14.6.1.1605** FREE, **14.6.1.2145** RESIZE.

14.6.1.1605   `FREE`   MEMORY

(*a-addr − − ior*)

Return the contiguous region of data space indicated by *a-addr* to the system for later allocation. *a-addr* shall indicate a region of data space that was previously obtained by **ALLOCATE** or **RESIZE**. The data-space pointer is unaffected by this operation.

If the operation succeeds, *ior* is zero. If the operation fails, *ior* is the implementation-defined I/O result code.

See: **6.1.1650** HERE, **14.6.1.0707** ALLOCATE, **14.6.1.2145** RESIZE.

14.6.1.2145   `RESIZE`   MEMORY

(*a-addr₁ u − − a-addr₂ ior*)

Change the allocation of the contiguous data space starting at the address *a-addr₁*, previously allocated by **ALLOCATE** or **RESIZE**, to *u* address units. *u* may be either larger or smaller than the current size of the region. The data-space pointer is unaffected by this operation.

If the operation succeeds, *a-addr₂* is the aligned starting address of *u* address units of allocated memory and *ior* is zero. *a-addr₂* may be, but need not be, the same as *a-addr₁*. If they are not the same, the values contained in the region at *a-addr₁* are copied to *a-addr₂*, up to the minimum size of either of the two regions. If they are the same, the values contained in the region are preserved to the minimum of *u* or the original size. If *a-addr₂* is not the same as *a-addr₁*, the region of memory at *a-addr₁* is returned to the system according to the operation of **FREE**.

If the operation fails, *a-addr₂* equals *a-addr₁*, the region of memory at *a-addr₁* is unaffected, and *ior* is the implementation-defined I/O result code.

See: **6.1.1650** HERE, **14.6.1.0707** ALLOCATE, **14.6.1.1605** FREE.

! " # $ % & ' ( ) * + , - . / digits : ; < = > ? @ ALPHA [\ ] ^ _ ` alpha { | } ~

14.6.2   Memory-Allocation extension words

None

15   The optional Programming-Tools word set

15.1   Introduction

This optional word set contains words most often used during the development of applications.

15.2   Additional terms and notation

None.

15.3   Additional usage requirements

15.3.1   Data types

A name token is a single-cell value that identifies a named word.

Append table 15.1 to table 3.1.

Table 15.1: Data types

Symbol	Data type	Size on stack
nt	name token	1 cell

See: **A.15.3.1 Name tokens**.

15.3.2   The Forth dictionary

A program using the words **CODE** or **; CODE** associated with assembler code has an environmental dependency on that particular instruction set and assembler notation.

Programs using the words **EDITOR** or **ASSEMBLER** require the Search Order word set or an equivalent implementation-defined capability.

See: **3.3 The Forth dictionary**.

15.4   Additional documentation requirements

15.4.1   System documentation

15.4.1.1   Implementation-defined options

– ending sequence for input following **15.6.2.0470** `; CODE` and **15.6.2.0930** `CODE`;

– manner of processing input following **15.6.2.0470** `; CODE` and **15.6.2.0930** `CODE`;

– search-order capability for **15.6.2.1300** `EDITOR` and **15.6.2.0740** `ASSEMBLER` (**15.3.2 The Forth dictionary**);

– source and format of display by **15.6.1.2194** `SEE`.

15.4.1.2 Ambiguous conditions

– deleting the compilation word-list (**15.6.2.1580** `FORGET`);

– fewer than $u + 1$ items on control-flow stack (**15.6.2.1015** `CS-PICK`, **15.6.2.1020** `CS-ROLL`);

– *name* can't be found (**15.6.2.1580** `FORGET`, **15.6.2.2264** `SYNONYM`);

– *name* not defined via **6.1.1000** `CREATE` (**15.6.2.0470** `;CODE`);

– **6.1.2033** `POSTPONE` applied to **15.6.2.2532** `[IF]`;

– reaching the end of the input source before matching **15.6.2.2531** `[ELSE]` or **15.6.2.2533** `[THEN]` (**15.6.2.2532** `[IF]`);

– removing a needed definition (**15.6.2.1580** `FORGET`).

– **6.1.1710** `IMMEDIATE` is applied to a word defined by **15.6.2.2264** `SYNONYM`.

– **15.6.2.1940** `NR>` is used with data not stored by **15.6.2.1908** `N>R`.

– adding to or deleting from the wordlist during the execution of **15.6.2.2297** `TRAVERSE-WORDLIST`

15.4.1.3 Other system documentation

– no additional requirements.

15.4.2 Program documentation

15.4.2.1 Environmental dependencies

– using the words **15.6.2.0470** `;CODE` or **15.6.2.0930** `CODE`.

15.4.2.2 Other program documentation

– no additional requirements.

15.5   Compliance and labeling

15.5.1   Forth-2012 systems

The phrase "Providing the Programming-Tools word set" shall be appended to the label of any Standard System that provides all of the Programming-Tools word set.

The phrase "Providing *name(s)* from the Programming-Tools Extensions word set" shall be appended to the label of any Standard System that provides portions of the Programming-Tools Extensions word set.

The phrase "Providing the Programming-Tools Extensions word set" shall be appended to the label of any Standard System that provides all of the Programming-Tools and Programming-Tools Extensions word sets.

15.5.2   Forth-2012 programs

The phrase "Requiring the Programming-Tools word set" shall be appended to the label of Standard Programs that require the system to provide the Programming-Tools word set.

The phrase "Requiring *name(s)* from the Programming-Tools Extensions word set" shall be appended to the label of Standard Programs that require the system to provide portions of the Programming-Tools Extensions word set.

The phrase "Requiring the Programming-Tools Extensions word set" shall be appended to the label of Standard Programs that require the system to provide all of the Programming-Tools and Programming-Tools Extensions word sets.

15.6   Glossary

15.6.1   Programming-Tools words

15.6.1.0220   **.S**   "dot-s"   TOOLS

(––)

Copy and display the values currently on the data stack. The format of the display is implementation-dependent.

.S may be implemented using pictured numeric output words. Consequently, its use may corrupt the transient region identified by **#>**.

See: **3.3.3.6 Other transient regions**, **A.15.6.1.0220** .S.

15.6.1.0600   **?**   "question"   TOOLS

(*a-addr* ––)

Display the value stored at *a-addr*.

? may be implemented using pictured numeric output words. Consequently, its use may corrupt the transient region identified by **#>**.

See: **3.3.3.6 Other transient regions**.

15.6.1.1280   **DUMP**   TOOLS

(*addr u* ––)

Display the contents of *u* consecutive addresses starting at *addr*. The format of the display is implementation dependent.

DUMP may be implemented using pictured numeric output words. Consequently, its use may corrupt the transient region identified by **#>**.

See: **3.3.3.6 Other transient regions**.

15.6.1.2194   **SEE**   TOOLS

(*"⟨spaces⟩name"* ––)

Display a human-readable representation of the named word's definition. The source of the representation (object-code decompilation, source block, etc.) and the particular form of the display is implementation defined.

SEE may be implemented using pictured numeric output words. Consequently, its use may corrupt the transient region identified by **#>**.

See: **3.3.3.6 Other transient regions**, **A.15.6.1.2194** SEE.

15.6.1.2465   **WORDS**   TOOLS

(––)

List the definition names in the first word list of the search order. The format of the display is implementation-dependent.

WORDS may be implemented using pictured numeric output words. Consequently, its use may corrupt the transient region identified by **#>**.

See: **3.3.3.6 Other transient regions**, **A.15.6.1.2465** WORDS.

15.6.2   Programming-Tools extension words

15.6.2.0470   **;CODE**   "semicolon-code"   TOOLS EXT

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: *colon-sys* ––)

Append the run-time semantics below to the current definition. End the current definition, allow it to be found in the dictionary, and enter interpretation state, consuming *colon-sys*.

Subsequent characters in the parse area typically represent source code in a programming language, usually some form of assembly language. Those characters are processed in an implementation-defined manner, generating the corresponding machine code. The process continues, refilling the input buffer as needed, until an implementation-defined ending sequence is processed.

Run-time: (––) (R: *nest-sys* ––)

Replace the execution semantics of the most recent definition with the *name* execution semantics given below. Return control to the calling definition specified by *nest-sys*. An ambiguous condition exists if the most recent definition was not defined with **CREATE** or a user-defined word that calls **CREATE**.

name Execution: (*i* * *x* −− *j* * *x*)

>Perform the machine code sequence that was generated following **; CODE**.

>See: **6.1.1250** DOES>, **A.15.6.2.0470** ; CODE.

15.6.2.0702   AHEAD   TOOLS EXT

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: −− *orig*)

>Put the location of a new unresolved forward reference *orig* onto the control flow stack. Append the run-time semantics given below to the current definition. The semantics are incomplete until *orig* is resolved (e.g., by **THEN**).

Run-time: (−−)

>Continue execution at the location specified by the resolution of *orig*.

15.6.2.0740   ASSEMBLER   TOOLS EXT

>(−−)

>Replace the first word list in the search order with the **ASSEMBLER** word list.

>See: **16 The optional Search-Order word set**.

15.6.2.0830   BYE   TOOLS EXT

>(−−)

>Return control to the host operating system, if any.

15.6.2.0930   CODE   TOOLS EXT

>(*"⟨spaces⟩name"* −−)

>Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name*, called a "code definition", with the execution semantics defined below.

>Subsequent characters in the parse area typically represent source code in a programming language, usually some form of assembly language. Those characters are processed in an implementation-defined manner, generating the corresponding machine code. The process continues, refilling the input buffer as needed, until an implementation-defined ending sequence is processed.

name Execution: (*i* * *x* −− *j* * *x*)

>Execute the machine code sequence that was generated following **CODE**.

>See: **3.4.1 Parsing**, **A.15.6.2.0930** CODE.

15.6.2.1015   **CS-PICK**   "c-s-pick"   TOOLS EXT

Interpretation: Interpretation semantics for this word are undefined.

Execution: (C: $dest_u \ldots orig_0 \mid dest_0 -- dest_u \ldots orig_0 \mid dest_0 \, dest_u$) (S: $u --$)

Remove u. Copy $dest_u$ to the top of the control-flow stack. An ambiguous condition exists if there are less than $u+1$ items, each of which shall be an *orig* or *dest*, on the control-flow stack before **CS-PICK** is executed.

If the control-flow stack is implemented using the data stack, u shall be the topmost item on the data stack.

See: **A.15.6.2.1015** CS-PICK.

15.6.2.1020   **CS-ROLL**   "c-s-roll"   TOOLS EXT

Interpretation: Interpretation semantics for this word are undefined.

Execution: (C: $orig_u \mid dest_u \, orig_{u-1} \mid dest_{u-1} \ldots orig_0 \mid dest_0 -- orig_{u-1} \mid dest_{u-1} \ldots orig_0 \mid dest_0$ $orig_u \mid dest_u$) (S: $u --$)

Remove u. Rotate $u+1$ elements on top of the control-flow stack so that $orig_u \mid dest_u$ is on top of the control-flow stack. An ambiguous condition exists if there are less than $u+1$ items, each of which shall be an *orig* or *dest*, on the control-flow stack before **CS-ROLL** is executed.

If the control-flow stack is implemented using the data stack, u shall be the topmost item on the data stack.

See: **A.15.6.2.1020** CS-ROLL.

15.6.2.1300   **EDITOR**   TOOLS EXT

(--)

Replace the first word list in the search order with the **EDITOR** word list.

See: **16 The optional Search-Order word set**.

15.6.2.1580   **FORGET**   TOOLS EXT

(*"⟨spaces⟩name"* --)

Skip leading space delimiters. Parse *name* delimited by a space. Find *name*, then delete *name* from the dictionary along with all words added to the dictionary after *name*. An ambiguous condition exists if *name* cannot be found.

If the Search-Order word set is present, **FORGET** searches the compilation word list. An ambiguous condition exists if the compilation word list is deleted.

An ambiguous condition exists if **FORGET** removes a word required for correct execution.

Note: This word is obsolescent and is included as a concession to existing implementations.

See: **3.4.1 Parsing**, **A.15.6.2.1580** FORGET.

15.6.2.1908	**N>R**	"n-to-r"	TOOLS EXT

Interpretation: Interpretation semantics for this word are undefined.

Execution: ($i * n + n --$) (R: $-- j * x + n$)

Remove $n+1$ items from the data stack and store them for later retrieval by **NR>**. The return stack may be used to store the data. Until this data has been retrieved by **NR>**:

- this data will not be overwritten by a subsequent invocation of **N>R** and

- a program may not access data placed on the return stack before the invocation of **N>R**.

See: **15.6.2.1940** NR>, **A.15.6.2.1908** N>R.

15.6.2.1909.10	**NAME>COMPILE**	"name-to-compile"	TOOLS EXT

($nt --$ x xt)

x xt represents the compilation semantics of the word nt. The returned xt has the stack effect ($i * x$ $x --$ $j * x$) . Executing xt consumes x and performs the compilation semantics of the word represented by nt.

See: **A.15.6.2.1909.10** NAME>COMPILE, **15.6.2.2297** TRAVERSE-WORDLIST.

15.6.2.1909.20	**NAME>INTERPRET**	"name-to-interpret"	TOOLS EXT

($nt --$ xt | 0)

xt represents the interpretation semantics of the word nt. If nt has no interpretation semantics, **NAME>INTERPRET** returns 0.

Note: This standard does not define the interpretation semantics of some words, but systems are allowed to do so.

See: **15.6.2.2297 TRAVERSE-WORDLIST**.

15.6.2.1909.40	**NAME>STRING**	"name-to-string"	TOOLS EXT

($nt --$ c-$addr$ u)

NAME>STRING returns the name of the word nt in the character string c-$addr$ u. The case of the characters in the string is implementation-dependent. The buffer containing

c-addr u may be transient and valid until the next invocation of **NAME>STRING**. A program shall not write into the buffer containing the resulting string.

See: **15.6.2.2297** TRAVERSE-WORDLIST.

15.6.2.1940   **NR>**   "n-r-from"   TOOLS EXT

Interpretation: Interpretation semantics for this word are undefined.

Execution: ($--\ i*x\ +n$) (R: $j*x\ +n\ --$)

Retrieve the items previously stored by an invocation of **N>R**. *n* is the number of items placed on the data stack. It is an ambiguous condition if **NR>** is used with data not stored by **N>R**.

See: **15.6.2.1908** N>R, **A.15.6.2.1908** N>R.

15.6.2.2250   **STATE**   TOOLS EXT

($--\ a\text{-}addr$)

Extend the semantics of **6.1.2250** STATE to allow **;CODE** to change the value in **STATE**. A program shall not directly alter the contents of **STATE**.

See: **3.4 The Forth text interpreter**, **6.1.0450** :, **6.1.0460** ;, **6.1.0670** ABORT, **6.1.2050** QUIT, **6.1.2250** STATE, **6.1.2500** [, **6.1.2540**], **6.2.0455** :NONAME, **15.6.2.0470** ;CODE.

15.6.2.2264   **SYNONYM**   TOOLS EXT

(*"⟨spaces⟩newname" "⟨spaces⟩oldname" --*)

For both strings skip leading space delimiters. Parse *newname* and *oldname* delimited by a space. Create a definition for *newname* with the semantics defined below. *Newname* may be the same as *oldname*; when looking up *oldname*, *newname* shall not be found.

An ambiguous conditions exists if *oldname* can not be found or **IMMEDIATE** is applied to *newname*.

newname interpretation: ($i*x -- j*x$)
Perform the interpretation semantics of *oldname*.

newname compilation: ($i*x -- j*x$)
Perform the compilation semantics of *oldname*.

See: **6.1.1710** IMMEDIATE.

15.6.2.2297   **TRAVERSE-WORDLIST**   TOOLS EXT

($i*x\ xt\ wid -- j*x$)

Remove *wid* and *xt* from the stack. Execute *xt* once for every word in the wordlist *wid*, passing the name token *nt* of the word to *xt*, until the wordlist is exhausted or until *xt* returns false.

The invoked *xt* has the stack effect (*k* * *x nt* −− *l* * *x flag*) .

If *flag* is true, **TRAVERSE−WORDLIST** will continue with the next name, otherwise it will return. **TRAVERSE−WORDLIST** does not put any items other than *nt* on the stack when calling *xt*, so that *xt* can access and modify the rest of the stack.

TRAVERSE−WORDLIST may visit words in any order, with one exception: words with the same name are called in the order newest-to-oldest (possibly with other words in between).

An ambiguous condition exists if words are added to or deleted from the wordlist *wid* during the execution of **TRAVERSE−WORDLIST**.

See: **A.15.6.2.2297** TRAVERSE-WORDLIST, **15.6.2.1909.40** NAME>STRING, **15.6.2.1909.20** NAME>INTERPRET, **15.6.2.1909.10** NAME>COMPILE.

15.6.2.2530.30 [DEFINED]   "bracket-defined"   TOOLS EXT

Compilation: Perform the execution semantics given below.

Execution: (*"⟨spaces⟩name . . . "* −− *flag*)

Skip leading space delimiters. Parse name delimited by a space. Return a true flag if *name* is the name of a word that can be found (according to the rules in the system's **FIND**); otherwise return a false flag. **[DEFINED]** is an immediate word.

15.6.2.2531   [ELSE]   "bracket-else"   TOOLS EXT

Compilation: Perform the execution semantics given below.

Execution: (*"⟨spaces⟩name . . . "* −−)

Skipping leading spaces, parse and discard space-delimited words from the parse area, including nested occurrences of **[IF]** . . . **[THEN]** and **[IF]** . . . **[ELSE]** . . . **[THEN]**, until the word **[THEN]** has been parsed and discarded. If the parse area becomes exhausted, it is refilled as with **REFILL**. **[ELSE]** is an immediate word.

See: **3.4.1 Parsing**, **A.15.6.2.2531** [ELSE].

15.6.2.2532   [IF]   "bracket-if"   TOOLS EXT

Compilation: Perform the execution semantics given below.

Execution: (*flag* | *flag* "⟨*spaces*⟩*name* . . . " −−)

> If *flag* is true, do nothing. Otherwise, skipping leading spaces, parse and discard space-delimited words from the parse area, including nested occurrences of **[IF]** . . . **[THEN]** and **[IF]** . . . **[ELSE]** . . . **[THEN]**, until either the word **[ELSE]** or the word **[THEN]** has been parsed and discarded. If the parse area becomes exhausted, it is refilled as with **REFILL**. **[IF]** is an immediate word.

> An ambiguous condition exists if **[IF]** is **POSTPONE**d, or if the end of the input buffer is reached and cannot be refilled before the terminating **[ELSE]** or **[THEN]** is parsed.

> See: **3.4.1 Parsing**, **A.15.6.2.2532** [IF].

15.6.2.2533   **[THEN]**   "bracket-then"   TOOLS EXT

Compilation: Perform the execution semantics given below.

Execution: (−−)

> Does nothing. **[THEN]** is an immediate word.

> See: **A.15.6.2.2533** [THEN].

15.6.2.2534   **[UNDEFINED]**   "bracket-undefined"   TOOLS EXT

Compilation: Perform the execution semantics given below.

Execution: ("⟨*spaces*⟩*name* . . . " −− *flag*)

> Skip leading space delimiters. Parse name delimited by a space. Return a false flag if *name* is the name of a word that can be found (according to the rules in the system's **FIND**); otherwise return a true flag. **[UNDEFINED]** is an immediate word.

---

16   The optional Search-Order word set

16.1   Introduction

16.2   Additional terms and notation

compilation word list: The word list into which new definition names are placed.

search order: A list of word lists specifying the order in which the dictionary will be searched.

16.3   Additional usage requirements

16.3.1   Data types

Word list identifiers are implementation-dependent single-cell values that identify word lists.

Append table 16.1 to table 3.1.

Table 16.1: Data types

Symbol	Data type	Size on stack
wid	word list identifiers	1 cell

See: **3.1 Data types**, **3.4.2 Finding definition names**, **3.4 The Forth text interpreter**.

16.3.2   Environmental queries

Append table 16.2 to table 3.5.

See: **3.2.6 Environmental queries**.

Table 16.2: Environmental Query Strings

String	Value data type	Constant?	Meaning
WORDLISTS	*n*	yes	maximum number of word lists usable in the search order

16.3.3   Finding definition names

When searching a word list for a definition name, the system shall search each word list from its last definition to its first. The search may encompass only a single word list, as with **SEARCH-WORDLIST**, or all the word lists in the search order, as with the text interpreter and **FIND**.

Changing the search order shall only affect the subsequent finding of definition names in the dictionary. A system with the Search-Order word set shall allow at least eight word lists in the search order.

An ambiguous condition exists if a program changes the compilation word list during the compilation of a definition or before modification of the behavior of the most recently compiled definition with **;CODE**, **DOES>**, or **IMMEDIATE**.

A program that requires more than eight word lists in the search order has an environmental dependency.

See: **3.4.2 Finding definition names**.

16.3.4 Contiguous regions

The regions of data space produced by the operations described in **3.3.3.2 Contiguous regions** may be non-contiguous if `WORDLIST` is executed between allocations.

16.4 Additional documentation requirements

16.4.1 System documentation

16.4.1.1 Implementation-defined options

– maximum number of word lists in the search order (**16.3.3 Finding definition names**, **16.6.1.2197** `SET-ORDER`);

– minimum search order (**16.6.1.2197** `SET-ORDER`, **16.6.2.1965** `ONLY`).

16.4.1.2 Ambiguous conditions

– changing the compilation word list (**16.3.3 Finding definition names**);

– search order empty (**16.6.2.2037** `PREVIOUS`);

– too many word lists in search order (**16.6.2.0715** `ALSO`).

16.4.1.3 Other system documentation

– no additional requirements.

16.4.2 Program documentation

16.4.2.1 Environmental dependencies

– requiring more than eight word-lists in the search order (**16.3.3 Finding definition names**).

16.4.2.2 Other program documentation

– no additional requirements.

16.5 Compliance and labeling

16.5.1 Forth-2012 systems

The phrase "Providing the Search-Order word set" shall be appended to the label of any Standard System that provides all of the Search-Order word set.

The phrase "Providing *name(s)* from the Search-Order Extensions word set" shall be appended to the label of any Standard System that provides portions of the Search-Order Extensions word set.

The phrase "Providing the Search-Order Extensions word set" shall be appended to the label of any Standard System that provides all of the Search-Order and Search-Order Extensions word sets.

16.5.2 Forth-2012 programs

The phrase "Requiring the Search-Order word set" shall be appended to the label of Standard Programs that require the system to provide the Search-Order word set.

The phrase "Requiring *name(s)* from the Search-Order Extensions word set" shall be appended to the label of Standard Programs that require the system to provide portions of the Search-Order Extensions word set.

The phrase "Requiring the Search-Order Extensions word set" shall be appended to the label of Standard Programs that require the system to provide all of the Search-Order and Search-Order Extensions word sets.

16.6 Glossary

16.6.1 Search-Order words

16.6.1.1180   DEFINITIONS   SEARCH

(--)

Make the compilation word list the same as the first word list in the search order. Specifies that the names of subsequent definitions will be placed in the compilation word list. Subsequent changes in the search order will not affect the compilation word list.

See: **16.3.3 Finding definition names**.

16.6.1.1550   FIND   SEARCH

Extend the semantics of **6.1.1550** FIND to be:

(*c-addr* -- *c-addr 0* | *xt 1* | *xt -1*)

Find the definition named in the counted string at *c-addr*. If the definition is not found after searching all the word lists in the search order, return *c-addr* and zero. If the definition is found, return *xt*. If the definition is immediate, also return one (*1*); otherwise also return minus-one (*-1*). For a given string, the values returned by **FIND** while compiling may differ from those returned while not compiling.

See: **3.4.2 Finding definition names**, **6.1.0070 '**, **6.1.1550** FIND, **6.1.2033** POSTPONE, **6.1.2510** [']**.

16.6.1.1595   FORTH-WORDLIST   SEARCH

(-- *wid*)

Return *wid*, the identifier of the word list that includes all standard words provided by the implementation. This word list is initially the compilation word list and is part of the initial search order.

16.6.1.1643   **GET−CURRENT**   SEARCH

(−− *wid*)

Return *wid*, the identifier of the compilation word list.

16.6.1.1647   **GET−ORDER**   SEARCH

(−− *wid$_n$* ... *wid$_1$* *n*)

Returns the number of word lists *n* in the search order and the word list identifiers *wid$_n$* ... *wid$_1$* identifying these word lists. *wid$_1$* identifies the word list that is searched first, and *wid$_n$* the word list that is searched last. The search order is unaffected.

16.6.1.2192   **SEARCH−WORDLIST**   SEARCH

(*c-addr u wid* −− *0* | *xt 1* | *xt -1*)

Find the definition identified by the string *c-addr u* in the word list identified by *wid*. If the definition is not found, return zero. If the definition is found, return its execution token *xt* and one (*1*) if the definition is immediate, minus-one (*-1*) otherwise.

See: **A.16.6.1.2192** SEARCH-WORDLIST.

16.6.1.2195   **SET−CURRENT**   SEARCH

(*wid* −−)

Set the compilation word list to the word list identified by *wid*.

16.6.1.2197   **SET−ORDER**   SEARCH

(*wid$_n$* ... *wid$_1$* *n* −−)

Set the search order to the word lists identified by *wid$_n$* ... *wid$_1$*. Subsequently, word list *wid$_1$* will be searched first, and word list *wid$_n$* searched last. If *n* is zero, empty the search order. If *n* is minus one, set the search order to the implementation-defined minimum search order. The minimum search order shall include the words **FORTH−WORDLIST** and **SET−ORDER**. A system shall allow *n* to be at least eight.

16.6.1.2460   **WORDLIST**   SEARCH

(−− *wid*)

Create a new empty word list, returning its word list identifier *wid*. The new word list may be returned from a pool of preallocated word lists or may be dynamically allocated in data space. A system shall allow the creation of at least 8 new word lists in addition to any provided as part of the system.

16.6.2   Search-Order extension words

16.6.2.0715   `ALSO`   SEARCH EXT

   (--)

Transform the search order consisting of wid_n, ... wid_2, wid_1 (where wid_1 is searched first) into wid_n, ... wid_2, wid_1, wid_1. An ambiguous condition exists if there are too many word lists in the search order.

16.6.2.1590   `FORTH`   SEARCH EXT

   (--)

Transform the search order consisting of wid_n, ... wid_2, wid_1 (where wid_1 is searched first) into wid_n, ... wid_2, $wid_{FORTH-WORDLIST}$.

16.6.2.1965   `ONLY`   SEARCH EXT

   (--)

Set the search order to the implementation-defined minimum search order. The minimum search order shall include the words **FORTH-WORDLIST** and **SET-ORDER**.

16.6.2.1985   `ORDER`   SEARCH EXT

   (--)

Display the word lists in the search order in their search order sequence, from first searched to last searched. Also display the word list into which new definitions will be placed. The display format is implementation dependent.

ORDER may be implemented using pictured numeric output words. Consequently, its use may corrupt the transient region identified by **#>**.

See: **3.3.3.6 Other transient regions**.

16.6.2.2037   `PREVIOUS`   SEARCH EXT

   (--)

Transform the search order consisting of wid_n, ... wid_2, wid_1 (where wid_1 is searched first) into wid_n, ... wid_2. An ambiguous condition exists if the search order was empty before **PREVIOUS** was executed.

17 The optional String word set

17.1 Introduction

17.2 Additional terms and notation

None.

17.3 Additional usage requirements

None.

17.4 Additional documentation requirements

17.4.1 System documentation

17.4.1.1 Implementation-defined options

– no additional options.

17.4.1.2 Ambiguous conditions

– The substitution cannot be created (**REPLACES**);

– The name of a substitution contains the '`%`' delimiter character (**REPLACES**);

– Result of a substitution is too long to fit into the given buffer (**SUBSTITUTE** and **UNESCAPE**);

– Source and destination buffers for **SUBSTITUTE** are the same.

17.4.1.3 Other system documentation

– no additional requirements.

17.4.2 Program documentation

17.4.2.1 Environmental dependencies

– no additional dependencies.

17.4.2.2 Other program documentation

– no additional requirements.

17.5 Compliance and labeling

17.5.1 Forth-2012 systems

The phrase "Providing the String word set" shall be appended to the label of any Standard System that provides all of the String word set.

The phrase "Providing *name(s)* from the String Extensions word set" shall be appended to the label of any Standard System that provides portions of the String Extensions word set.

The phrase "Providing the String Extensions word set" shall be appended to the label of any Standard System that provides all of the String and String Extensions word sets.

17.5.2 Forth-2012 programs

The phrase "Requiring the String word set" shall be appended to the label of Standard Programs that require the system to provide the String word set.

The phrase "Requiring *name(s)* from the String Extensions word set" shall be appended to the label of Standard Programs that require the system to provide portions of the String Extensions word set.

The phrase "Requiring the String Extensions word set" shall be appended to the label of Standard Programs that require the system to provide all of the String and String Extensions word sets.

17.6 Glossary

17.6.1 String words

17.6.1.0170   −TRAILING   "dash-trailing"   STRING

($c\text{-}addr\ u_1$ −− $c\text{-}addr\ u_2$)

If u_1 is greater than zero, u_2 is equal to u_1 less the number of spaces at the end of the character string specified by $c\text{-}addr\ u_1$. If u_1 is zero or the entire string consists of spaces, u_2 is zero.

17.6.1.0245   /STRING   "slash-string"   STRING

($c\text{-}addr_1\ u_1\ n$ −− $c\text{-}addr_2\ u_2$)

Adjust the character string at $c\text{-}addr_1$ by n characters. The resulting character string, specified by $c\text{-}addr_2\ u_2$, begins at $c\text{-}addr_1$ plus n characters and is u_1 minus n characters long.

See: **A.17.6.1.0245** /STRING.

17.6.1.0780   BLANK   STRING

($c\text{-}addr\ u$ −−   )

If u is greater than zero, store the character value for space in u consecutive character positions beginning at $c\text{-}addr$.

17.6.1.0910   **CMOVE**   "c-move"   STRING

(*c-addr$_1$ c-addr$_2$ u* −−)

If *u* is greater than zero, copy *u* consecutive characters from the data space starting at *c-addr$_1$* to that starting at *c-addr$_2$*, proceeding character-by-character from lower addresses to higher addresses.

See: **17.6.1.0920** CMOVE>, **A.17.6.1.0910** CMOVE.

17.6.1.0920   **CMOVE>**   "c-move-up"   STRING

(*c-addr$_1$ c-addr$_2$ u* −−)

If *u* is greater than zero, copy *u* consecutive characters from the data space starting at *c-addr$_1$* to that starting at *c-addr$_2$*, proceeding character-by-character from higher addresses to lower addresses.

See: **17.6.1.0910** CMOVE, **A.17.6.1.0920** CMOVE>.

17.6.1.0935   **COMPARE**   STRING

(*c-addr$_1$ u$_1$ c-addr$_2$ u$_2$* −− *n*)

Compare the string specified by *c-addr$_1$ u$_1$* to the string specified by *c-addr$_2$ u$_2$*. The strings are compared, beginning at the given addresses, character by character, up to the length of the shorter string or until a difference is found. If the two strings are identical, *n* is zero. If the two strings are identical up to the length of the shorter string, *n* is minus-one (-1) if *u$_1$* is less than *u$_2$* and one (1) otherwise. If the two strings are not identical up to the length of the shorter string, *n* is minus-one (-1) if the first non-matching character in the string specified by *c-addr$_1$ u$_1$* has a lesser numeric value than the corresponding character in the string specified by *c-addr$_2$ u$_2$* and one (1) otherwise.

17.6.1.2191   **SEARCH**   STRING

(*c-addr$_1$ u$_1$ c-addr$_2$ u$_2$* −− *c-addr$_3$ u$_3$ flag*)

Search the string specified by *c-addr$_1$ u$_1$* for the string specified by *c-addr$_2$ u$_2$*. If *flag* is true, a match was found at *c-addr$_3$* with *u$_3$* characters remaining. If *flag* is false there was no match and *c-addr$_3$* is *c-addr$_1$* and *u$_3$* is *u$_1$*.

17.6.1.2212   **SLITERAL**   STRING

Interpretation: Interpretation semantics for this word are undefined.

 Compilation: (*c-addr$_1$ u* −−)

Append the run-time semantics given below to the current definition.

Run-time: ($--$ $c\text{-}addr_2\ u$)

Return $c\text{-}addr_2\ u$ describing a string consisting of the characters specified by $c\text{-}addr_1\ u$ during compilation. A program shall not alter the returned string.

See: **A.17.6.1.2212** `SLITERAL`.

17.6.2   String extension words

17.6.2.2141   **REPLACES**   STRING EXT

($c\text{-}addr_1\ u_1\ c\text{-}addr_2\ u_2 --$)

Set the string $c\text{-}addr_1\ u_1$ as the text to substitute for the substitution named by $c\text{-}addr_2$ u_2. If the substitution does not exist it is created. The program may then reuse the buffer $c\text{-}addr_1\ u_1$ without affecting the definition of the substitution.

Ambiguous conditions occur as follows:

- The substitution cannot be created.

- The name of a substitution contains the '%' delimiter character.

REPLACES may allot data space and create a definition. This breaks the contiguity of the current region and is not allowed during compilation of a colon definition

See: **3.3.3.2 Contiguous regions**, **3.4.5 Compilation**, **17.6.2.2255** `SUBSTITUTE`.

17.6.2.2255   **SUBSTITUTE**   STRING EXT

($c\text{-}addr_1\ u_1\ c\text{-}addr_2\ u_2 -- c\text{-}addr_2\ u_3\ n$)

Perform substitution on the string $c\text{-}addr_1\ u_1$ placing the result at string $c\text{-}addr_2\ u_3$, where $u3$ is the length of the resulting string. An error occurs if the resulting string will not fit into $c\text{-}addr_2\ u_2$ or if $c\text{-}addr_2$ is the same as $c\text{-}addr_1$. The return value n is positive or 0 on success and indicates the number of substitutions made. A negative value for n indicates that an error occurred, leaving $c\text{-}addr_2\ u_3$ undefined. Negative values of n are implementation defined except for values in table **9.1 THROW code assignments**.

Substitution occurs left to right from the start of $c\text{-}addr_1$ in one pass and is non-recursive.

When text of a potential substitution name, surrounded by '%' (ASCII $25) delimiters is encountered by **SUBSTITUTE**, the following occurs:

a) If the name is null, a single delimiter character is passed to the output, i.e., `%%` is replaced by `%`. The current number of substitutions is not changed.

b) If the text is a valid substitution name acceptable to **17.6.2.2141** `REPLACES`, the leading and trailing delimiter characters and the enclosed substitution name are replaced by the substitution text. The current number of substitutions is incremented.

    c) If the text is not a valid substitution name, the name with leading and trailing delimiters is passed unchanged to the output. The current number of substitutions is not changed.

    d) Parsing of the input string resumes after the trailing delimiter.

If after processing any pairs of delimiters, the residue of the input string contains a single delimiter, the residue is passed unchanged to the output.

See: **17.6.2.2141** REPLACES, **17.6.2.2375** UNESCAPE, **A.17.6.2.2255** SUBSTITUTE.

17.6.2.2375   **UNESCAPE**   STRING EXT

($c\text{-}addr_1$ u_1 $c\text{-}addr_2$ $--$ $c\text{-}addr_2$ u_2)

Replace each '%' character in the input string $c\text{-}addr_1$ u_1 by two '%' characters. The output is represented by $c\text{-}addr_2$ u_2. The buffer at $c\text{-}addr_2$ shall be big enough to hold the unescaped string. An ambiguous condition occurs if the resulting string will not fit into the destination buffer ($c\text{-}addr_2$).

See: **17.6.2.2255** SUBSTITUTE.

18 The optional Extended-Character word set

18.1 Introduction

This word set deals with variable width character encodings. It also works with fixed width encodings.

Since the standard specifies ASCII encoding for characters, only ASCII-compatible encodings may be used. Because ASCII compatibility has so many benefits, most encodings actually are ASCII compatible. The characters beyond the ASCII encoding are called "extended characters" (xchars).

All words dealing with strings shall handle xchars when the xchar word set is present. This includes dictionary definitions. White space parsing does not have to treat code points greater than $20 as white space.

18.2 Additional terms and notation

18.2.1 Definition of Terms

code point: A member of an extended character set.

18.2.2 Parsed-text notation

Append table 18.1 to table 2.1.

Table 18.1: Parsed text abbreviations

Abbreviation	Description
⟨xchar⟩	the delimiting extended character

See: **2.2.3 Parsed-text notation**.

18.3 Additional usage requirements

18.3.1 Data types

Append table 18.2 to table 3.1.

Table 18.2: Data Types

Symbol	Data type	Size on stack
pchar	primitive character	1 cell
xchar	extended character	1 cell
xc-addr	xchar-aligned address	1 cell

See: **3.1 Data types**.

18.3.1.1 Extended Characters

An extended character (xchar) is the code point of a character within an extended character set; on the stack it is a subset of *u*. Extended characters are stored in memory encoded as one or more primitive characters (pchars).

18.3.2 Environmental queries

Append table 18.3 to table 3.5.

Table 18.3: Environmental Query Strings

String	Value data type	Constant?	Meaning
XCHAR-ENCODING	*c-addr u*	no	Returns a printable ASCII string that represents the encoding, and use the preferred MIME name (if any) or the name in the IANA character-set register[1] (RFC-1700) such as "ISO-LATIN-1" or "UTF-8", with the exception of "ASCII", where the alias "ASCII" is preferred.
MAX-XCHAR	*u*	no	Maximal value for *xchar*
XCHAR-MAXMEM	*u*	no	Maximal memory consumed by an *xchar* in address units

[1] http://www.iana.org/assignments/character-sets

See: **3.2.6 Environmental queries**.

18.3.3 Common encodings

Input and files are often encoded iso–latin–1 or utf–8. The encoding depends on settings of the computer system such as the LANG environment variable on Unix. You can use the system consistently only when you do not change the encoding, or only use the ASCII subset. The typical practice in environments requiring more than one encoding is that the base system is ASCII only, and the character set is then extended to specify the required encoding.

18.3.4 The Forth text interpreter

In section **3.4.1.3 Text interpreter input number conversion**, ⟨*cnum*⟩ should be redefined to be:

⟨*cnum*⟩   the number is the value of ⟨*xchar*⟩

18.3.5 Input and Output

IO words such as **KEY**, **EMIT**, **TYPE**, **READ-FILE**, **READ-LINE**, **WRITE-FILE**, and **WRITE-LINE** operate on *pchars*. Therefore, it is possible that these words read or write incomplete *xchars*, which are completed in the next consecutive operation(s). The IO system shall combine these *pchars* into a complete *xchars* on output, or split an *xchars* into *pchars* on input, and shall not throw a "malformed *xchars*" exception when the combination of these *pchars* form a valid *xchars*. **-TRAILING-GARBAGE** can be used to process an incomplete *xchars* at the end of such an IO operation.

ACCEPT as input editor may be aware of *xchars* to provide comfort like backspace or cursor movement.

18.4   Additional documentation requirements

18.4.1   System documentation

18.4.1.1   Implementation-defined options

Since Unicode input and display poses a number of challenges like input method editors for different languages, left-to-right and right-to-left writing, and most fonts contain only a subset of Unicode glyphs, systems should document their capabilities. File IO and in-memory string handling should work transparently with *xchars*.

18.4.1.2   Ambiguous conditions

- the data in memory does not encode a valid xchar (**18.6.1.2486.50** X-SIZE);

- the *xchars* value is outside the range of allowed code points of the current character set used.

- words improperly used outside **6.1.0490** <# and **6.1.0040** #> (**18.6.2.2488.20** XHOLD).

18.4.1.3   Other system documentation

- no additional requirements.

18.4.2   Program documentation

- no additional requirements.

18.5   Compliance and labeling

18.5.1   Forth-2012 systems

The phrase "Providing the Extended-Character word set" shall be appended to the label of any Standard System that provides all of the Extended-Character word set.

The phrase "Providing *name(s)* from the Extended-Character Extensions word set" shall be appended to the label of any Standard System that provides portions of the Extended-Character Extensions word set.

The phrase "Providing the Extended-Character Extensions word set" shall be appended to the label of any Standard System that provides all of the Extended-Character and Extended-Character Extensions word sets.

18.5.2   Forth-2012 programs

The phrase "Requiring the Extended-Character word set" shall be appended to the label of Standard Programs that require the system to provide the Extended-Character word set.

The phrase "Requiring *name(s)* from the Extended-Character Extensions word set" shall be appended to the label of Standard Programs that require the system to provide portions of the Extended-Character Extensions word set.

The phrase "Requiring the Extended-Character Extensions word set" shall be appended to the label of Standard Programs that require the system to provide all of the Extended-Character Exception and Extended-Character Extensions word sets.

18.6 Glossary

18.6.1 Extended-Character words

18.6.1.2486.50 X-SIZE   XCHAR

(*xc-addr* u_1 −− u_2)

u_2 is the number of pchars used to encode the first xchar stored in the string *xc-addr u1*. To calculate the size of the xchar, only the bytes inside the buffer may be accessed. An ambiguous condition exists if the xchar is incomplete or malformed.

18.6.1.2487.10 XC!+   "x-c-store-plus"   XCHAR

(*xchar xc-addr₁* −− *xc-addr₂*)

Stores the *xchar* at *xc-addr₁*. *xc-addr₂* points to the first memory location after the stored *xchar*.

18.6.1.2487.15 XC!+?   "x-c-store-plus-query"   XCHAR

(*xchar xc-addr₁ u₁* −− *xc-addr₂ u₂ flag*)

Stores the *xchar* into the string buffer specified by *xc-addr₁ u₁*. *xc-addr₂ u₂* is the remaining string buffer. If the *xchar* did fit into the buffer, *flag* is true, otherwise *flag* is false, and *xc-addr₂ u₂* equal *xc-addr₁ u₁*. **XC!+?** is safe for buffer overflows.

18.6.1.2487.20 XC,   "x-c-comma"   XCHAR

(*xchar* −−)

Append the encoding of *xchar* to the dictionary.

   See: **6.1.0860** C, .

18.6.1.2487.25 XC-SIZE   "x-c-size"   XCHAR

(*xchar* −− *u*)

u is the number of pchars used to encode *xchar* in memory.

18.6.1.2487.35 `XC@+`   "x-c-fetch-plus"   XCHAR

($xc\text{-}addr_1$ −− $xc\text{-}addr_2$ $xchar$)

Fetches the *xchar* at $xc\text{-}addr_1$. $xc\text{-}addr_2$ points to the first memory location after the retrieved *xchar*.

18.6.1.2487.40 `XCHAR+`   "x-char-plus"   XCHAR

($xc\text{-}addr_1$ −− $xc\text{-}addr_2$)

Adds the size of the xchar stored at $xc\text{-}addr_1$ to this address, giving $xc\text{-}addr_2$.

See: **6.1.0897** `CHAR+`.

18.6.1.2488.10 `XEMIT`   "x-emit"   XCHAR

($xchar$ −−)

Prints an *xchar* on the terminal.

See: **6.1.1320** `EMIT`

18.6.1.2488.30 `XKEY`   "x-key"   XCHAR

(−− $xchar$)

Reads an *xchar* from the terminal. This will discard all input events up to the completion of the *xchar*.

See: **6.1.1750** `KEY`.

18.6.1.2488.35 `XKEY?`   "x-key-query"   XCHAR

(−− *flag*)

Flag is true when it's possible to do **`XKEY`** without blocking. Subsequent **`KEY?`**, **`KEY`**, **`EKEY?`**, and **`EKEY`** may be affected by **`XKEY?`**.

See: **10.6.1.1755** `KEY?`.

18.6.2   Extended-Character extension words

18.6.2.0145   `+X/STRING`   "plus-x-string"   XCHAR EXT

($xc\text{-}addr_1$ u_1 −− $xc\text{-}addr_2$ u_2)

Step forward by one xchar in the buffer defined by $xc\text{-}addr_1$ u_1. $xc\text{-}addr_2$ u_2 is the remaining buffer after stepping over the first xchar in the buffer.

18.6.2.0175   −TRAILING−GARBAGE   "minus-trailing-garbage"   XCHAR EXT

(*xc-addr u₁* −− *xc-addr u₂*)

Examine the last xchar in the string *xc-addr u₁* — if the encoding is correct and it represents a full xchar, u_2 equals u_1, otherwise, u_2 represents the string without the last (garbled) xchar. **−TRAILING−GARBAGE** does not change this garbled xchar.

18.6.2.0895   CHAR   XCHAR EXT

(*"⟨spaces⟩name"* −− *xchar*)

Skip leading space delimiters. Parse *name* delimited by a space. Put the value of its first *xchar* onto the stack.

See: **6.1.0895** CHAR

18.6.2.1306.60 EKEY>XCHAR   "e-key-to-x-char"   XCHAR EXT

(*x* −− *xchar true | x false*)

If the keyboard event *x* corresponds to an xchar, return the *xchar* and *true*. Otherwise, return *x* and *false*.

See: **10.6.2.1305** EKEY, **10.6.2.1306** EKEY>CHAR, **10.6.2.1306.40** EKEY>FKEY.

18.6.2.2008   PARSE   XCHAR EXT

(*xchar "ccc⟨xchar⟩"* −− *c-addr u*)

Parse *ccc* in the input stream delimited by *xchar*.

c-addr is the address (within the input buffer) and *u* is the length of the parsed string. If the parse area was empty, the resulting string has a zero length.

See: **3.4.1 Parsing**, **6.2.2008** PARSE, **A.6.2.2008** PARSE.

18.6.2.2486.70 X−WIDTH   XCHAR EXT

(*xc-addr u* −− *n*)

n is the number of monospace ASCII characters that take the same space to display as the xchar string *xc-addr u*; assuming a monospaced display font, i.e., xchar width is always an integer multiple of the width of an ASCII character.

18.6.2.2487.30 XC-WIDTH   "x-c-width"   XCHAR EXT

(*xchar* −− *n*)

n is the number of monospace ASCII characters that take the same space to display as the *xchar*; i.e., *xchar* width is always an integer multiple of the width of an ASCII char.

18.6.2.2487.45 XCHAR-   "x-char-minus"   XCHAR EXT

(*xc-addr₁* −− *xc-addr₂*)

Goes backward from *xc-addr₁* until it finds an xchar so that the size of this xchar added to *xc-addr₂* gives *xc-addr1*. There is an ambiguous condition when the encoding doesn't permit reliable backward stepping through the text.

18.6.2.2488.20 XHOLD   "x-hold"   XCHAR EXT

(*xchar* −−)

Adds *xchar* to the picture numeric output string. An ambiguous condition exists if **XHOLD** executes outside of a **<# #>** delimited number conversion.

See: **6.1.1670** HOLD.

18.6.2.2495   X\STRING-   "x-string-minus"   XCHAR EXT

(*xc-addr u₁* −− *xc-addr u₂*)

Search for the penultimate xchar in the string *xc-addr u₁*. The string *xc-addr u₂* contains all xchars of *xc-addr u₁*, but the last. Unlike **XCHAR-**, **X\STRING-** can be implemented in encodings where xchar boundaries can only reliably detected when scanning in forward direction.

18.6.2.2520   [CHAR]   "bracket-char"   XCHAR EXT

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (*"⟨spaces⟩name"* −−)

Skip leading space delimiters. Parse *name* delimited by a space. Append the run-time semantics given below to the current definition.

Run-time: (−− *xchar*)

Place *xchar*, the value of the first xchar of *name*, on the stack.

See: **6.1.2520** [CHAR]

Annex A
(informative)
Rationale

A.1 Introduction

A.1.1 Purpose

A.1.2 Scope

When judging relative merits of proposed changes to the standard, the members of the 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.2 Terms and notation

A.2.1 Definitions of terms

aligned

Data can only be loaded from and stored to addresses that are aligned according to the alignment requirements of the accessed type. Field offsets that are added to structure addresses also need to be aligned.

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 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 subrange 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
```

185

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

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

Forth 94 moved 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.3.1.3.6   Error results

The term *ior* was originally defined to describe the result of an input/output operation. This was extended to include other operations.

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, a Forth-2012 system is permitted to supply either floored or symmetric operators. In addition, a standard system 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 Forth 94 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 Forth 94 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.3.2.3.2   Control-flow stack   The simplest use of control-flow words is to implement the basic control structures shown in figure **A.1**.
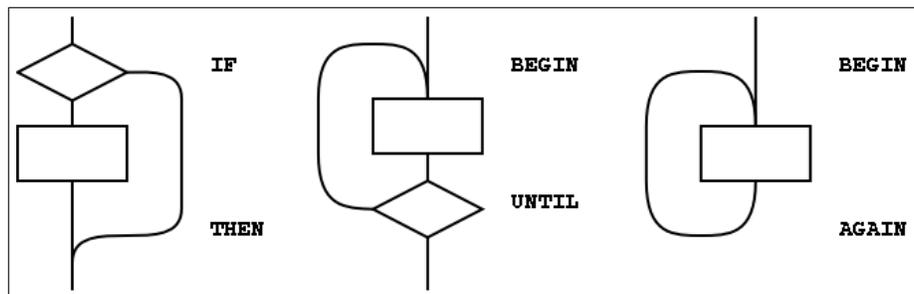
Figure A.1: The basic control-flow patterns

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 brach
  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
```
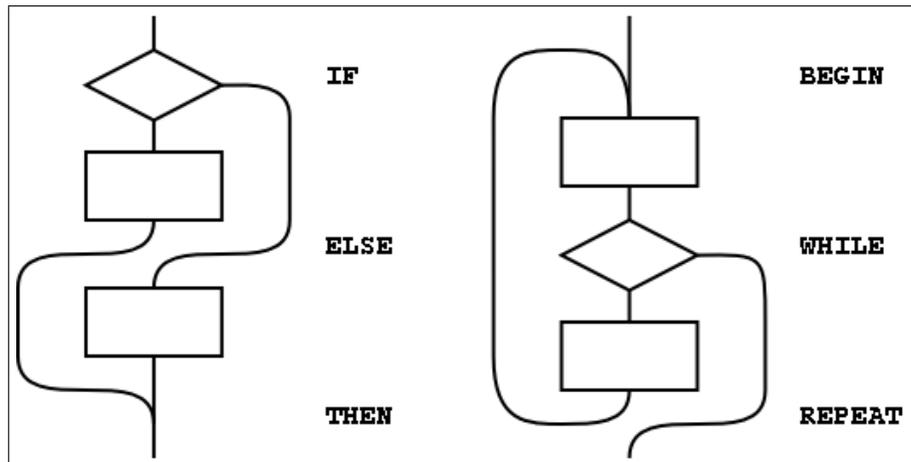

Figure A.2: Additional basic control-flow patterns

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.

Note that **REPEAT** creates an anomaly when matching the **WHILE** with **ELSE** or **THEN**, most notably 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 **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 -- )
```
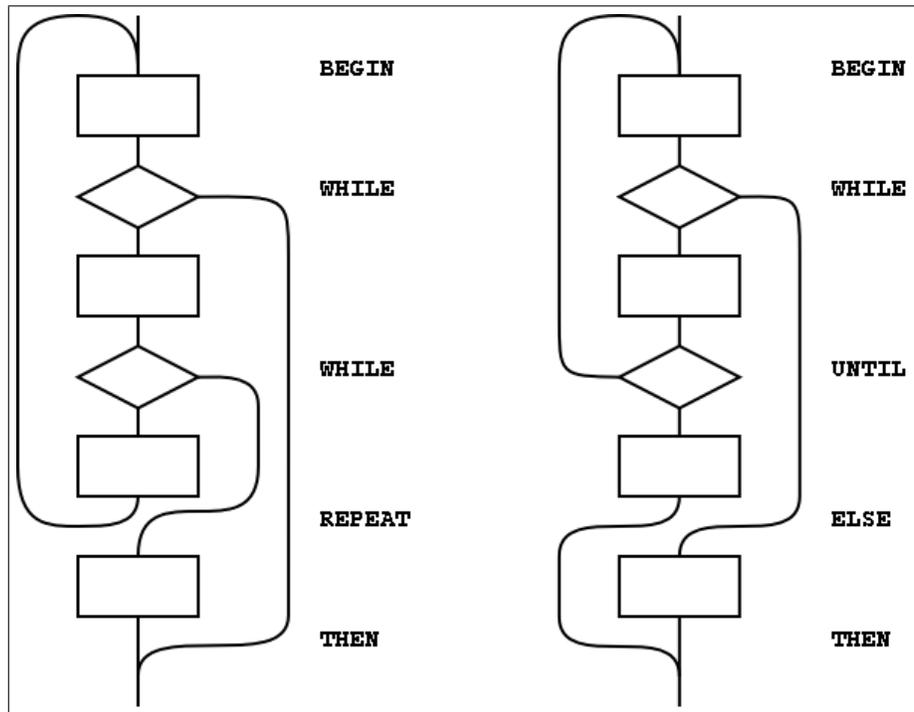
Figure A.3: Extended control-flow patterns

```
  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, now obsolescent, 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.3.2.7 Obsolescent Environmental Queries

When reviewing the Forth 94 Standard, the question of adapting the word set queries had to be addressed. Despite the recommendation in Forth 94, word set queries have not been supported in a meaningful way by many systems. Consequently, these queries are not used by many programmers. The committee was unwilling to exacerbate the problem by introducing additional queries for the revised word sets. The committee has therefore declared the word set environment queries (see table 3.6) as obsolescent with the intention of removing them altogether in the next revision. They are retained in this standard to support existing Forth 94 programs. New programs should not use them.

A.3.2.8 Extension queries

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 **>IN**, **BASE**, **BLK**, **SCR**, **SOURCE**, **SOURCE-ID**, **STATE** 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

Some processors have restrictions on the addresses that can be used by memory access instructions. For example, some architectures require 16-bit data to be loaded or stored only at even addresses and 32-bit data only at addresses that are multiples of four.

An implementor 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 system with 16-bit cells, **@** could be implemented with two 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 could define each memory-access word using the native instructions that most closely match the word's function. The 16-bit cell system could implement **@** using the processor's 16-bit fetch instruction, in this case, the responsibility for giving **@** a correctly-aligned address falls on the programmer. A portable 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 discontiguous 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 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   Forth-2012 systms

Section 5.1 defines the criteria that a system must meet in order to justify the label "Forth-2012 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 a standard system, in that both use similar compiling tools to process a program. However, in order to fully specify a Forth-2012 standard 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 a Forth-2012 standard program for execution in a target environment, it is inappropriate for a cross compiler to be labeled a Forth-2012 standard system.

A.5.2   Forth-2012 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 cells 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 a "Standard Forth-2012 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.6.1.0070   *'*

    Typical use: . . . *' name*.

    Many Forth systems use a state-smart tick. Many do not. Forth-2012 follows the usage of Forth 94.

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

A.6.1.0080   **(**

    Typical use: . . . **(** *ccc*) . . .

A.6.1.0140 **+LOOP**

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

A.6.1.0150 **,**

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

See: **6.2.0945** COMPILE,.

A.6.1.0190 **."**

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

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

A.6.1.0450 **:**

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**

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 committee doesn't wish to preclude it unnecessarily. Because actual values are almost always small integers, no functionality is impaired by this restriction.

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

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.6.1.0760 **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 **CELLS** 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**

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]).

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 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.6.1.1750 `KEY`

Use of `KEY` indicates that the application is processing primitive characters. Some input devices, e.g., keyboards, may provide more information than can be represented as a primitive character and such an event may be received as an implementation-specific sequence of primitive characters.

See **A.10.6.2.1305** EKEY.

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; Forth needs a move instruction capable of dealing with address units. Thus MOVE has been defined and added to the Core word set, and CMOVE and CMOVE> have been moved to the String word set.

A.6.1.2033 POSTPONE

Typical use:

```
: ENDIF POSTPONE THEN ; IMMEDIATE

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

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

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

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

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

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

A.6.1.2120 RECURSE

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

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

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

$n_2 = n_1(n_1 - 1)(n_1 - 2) \cdots (2)(1)$, the product of n_1 with all positive integers less than itself (as a special case, zero factorial equals one). While beloved of computer scientists, recursion makes

unusually heavy use of both stacks and should therefore be used with caution. See alternate definition in **A.6.1.2140** REPEAT.

A.6.1.2140 **REPEAT**

Typical use:

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

A.6.1.2165 **S"**

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

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.

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-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.

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] c ... ;`

A.6.1.2540   `]`

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

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

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`

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`

Typical use:

    DEFER print

    :NONAME (n --) . ; IS print

Note: `RECURSE` and `DOES>` are allowed within a `:NONAME` definition.

A.6.2.0620   `?DO`

Typical use:

    : X ... ?DO ... LOOP ... ;

A.6.2.0700   `AGAIN`

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

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

A.6.2.0825 `BUFFER:`

`BUFFER:` provides a means of defining an uninitialized buffer. In systems that use a single memory space, this can effectively be defined as:

```
: BUFFER: ( u "<name>" -- ; -- addr )
  CREATE ALLOT
;
```

However, many systems profit from a separation of uninitialized and initialized data areas. Such systems can implement `BUFFER:` so that it allocates memory from a separate uninitialized memory area. Embedded systems can take advantage of the lack of initialization of the memory area while hosted systems are permitted to `ALLOCATE` a buffer. A system may select a region of memory for performance reasons. A detailed knowledge of the memory allocation within the system is required to provide a version of `BUFFER:` that can take advantage of the system.

It should be noted that the memory buffer provided by `BUFFER:` is not initialized by the system and that if the application requires it to be initialized, it is the responsibility of the application to initialize it.

A.6.2.0855 `C"`

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

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 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.

The intention is that `COMPILE,` can be used as follows to write the classic interpreter/compiler loop:

```
...                                                    ( c-addr )
FIND ?DUP IF                                           ( xt +-1 )
  STATE @ IF                                           ( xt +-1 )
    0> IF  EXECUTE  ELSE  COMPILE,  THEN               ( ???  )
  ELSE                                                 ( xt +-1 )
    DROP EXECUTE                                       ( ???  )
  THEN
ELSE                                                   ( c-addr )
  ( whatever you do for an undefined word )
THEN
...
```

Thus the interpretation semantics are left undefined, as **COMPILE,** will not be executed during interpretation.

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.6.2.1850 **MARKER**

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

A.6.2.1950 **OF**

Typical use:

```
: X ...
  CASE
```

```
        test1 OF ... ENDOF
        testn OF ... ENDOF
        ... ( default )
      ENDCASE ...
  ;
```

A.6.2.2000 PAD

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

A.6.2.2008 PARSE

Typical use: *char* **PARSE** *ccc⟨char⟩*

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

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

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

This works fine when used in a line like:

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

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

```
.( )   5 .
```

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

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

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

This definition avoids the "empty string" anomaly.

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

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

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

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

4) The transient buffer is typically overwritten by the next use of **WORD**.

The need for **WORD** has largely been eliminated by **PARSE** and **PARSE-NAME**. **WORD** is retained for backward compatibility.

A.6.2.2030 **PICK**

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

A.6.2.2125 **REFILL**

REFILL is designed to behave reasonably for all possible input sources. If the input source is coming from the user, **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.6.2.2295   **TO**

Typical use: x **TO** *name*

Some implementations of **TO** do not parse; instead they set a mode flag that is tested by the subsequent execution of *name*. Standard programs must use **TO** as if it parses. Therefore **TO** and *name* must be contiguous and on the same line in the source text.

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:

$$u\text{———————————————————}o$$

There are two cases to consider: either the $n_2 \mid u_2 \ldots n_3 \mid u_3$ range straddles the overflow/underflow points or it does not. Lets examine the non-straddle case first:

$$u\text{—————}[\ldots \ldots)\text{————}o$$

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

$$n_2 \mid u_2 \leq n_1 \mid u_1 \text{ and } n_1 \mid u_1 < n_3 \mid u_3.$$

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

$$u \ldots \ldots)\text{————}[\ldots \ldots o$$

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

$$n_2 \mid u_2 \leq n_1 \mid u_1 \text{ or } n_1 \mid u_1 < n_3 \mid u_3.$$

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

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

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

```
 33000   32000   34000   WITHIN
```

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

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

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

```
1   -5   5   WITHIN
```

would give an incorrect answer.

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

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

A.6.2.2530   [COMPILE]

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

A.6.2.2535   \

Typical use:

```
5 CONSTANT THAT   \     This is a comment about THAT
```

A.7   The optional Block word set

Early Forth systems ran without a host OS; these are known as native systems. Such systems provide mass storage in blocks of 1024 bytes. The Block Word set includes the most common words for accessing program source and data on disk.

A.7.2   Additional terms

block

Forth systems may 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.7.3   Additional usage requirements

A.7.3.2   Block buffer regions

While the standard does not address multitasking per se, the items listed in **7.3.2 Block buffer regions** that may render block-buffer addresses invalid are due to multitasking considerations. The standard restricts programs such that items that could fail on multitasking systems are not standard usage. It also permits multitasking systems to be declared standard systems.

A.7.6   Glossary

A.7.6.2.2190   SCR

SCR is short for screen.

A.8   The optional 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 2*xxx* 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 D*xxx* deal specifically with double-length integers.

3. Words with names of the form M*xxx* 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.8.6.1.1140 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.8.6.2.0435 2VALUE

Typical use:

```
: fn1 S" filename" ;
fn1 2VALUE myfile
myfile INCLUDED
```

```
: fn2 S" filename2" ;
fn2 TO myfile
myfile INCLUDED
```

A.9   The optional 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**.

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 optional Facility word set

A.10.6   Glossary

A.10.6.1.1755   **KEY?**

The 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 process 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.10.6.2.0135   **+FIELD**

+FIELD is not required to align items. This is deliberate and allows the construction of unaligned data structures for communication with external elements such as a hardware register map or protocol packet. Field alignment has been left to the appropriate xFIELD: definition.

A.10.6.2.0763   **BEGIN-STRUCTURE**

There are two schools of thought regarding named data structures: name first and name last. The name last school can define a named data structure as follows:

```
0                          \ initial total byte count
  1 CELLS +FIELD p.x  \ A single cell filed named p.x
  1 CELLS +FIELD p.y  \ A single cell field named p.y
CONSTANT point           \ save structure size
```

While the name first school would define the same data structure as:

```
BEGIN-STRUCTURE point \ create the named structure
  1 CELLS +FIELD p.x  \ A single cell filed named p.x
  1 CELLS +FIELD p.y  \ A single cell field named p.y
END-STRUCTURE
```

Although many systems provide a name first structure there is no common practice to the words used. The words **BEGIN-STRUCTURE** and **END-STRUCTURE** have been defied as a means of providing a portable notation that does not conflict with existing systems.

The field defining words (*x*FIELD: and **+FIELD**) are defined so they can be used by both schools. Compatibility between the two schools comes from defining a new stack item *struct-sys*, which is implementation dependent and can be 0 or more cells. The name first school would provide an address (*addr*) as the *struct-sys* parameter, while the name last school would defined *struct-sys* as being empty.

Executing the name of the data structure, returns the size of the data structure. This allows the data stricture to be used within another data structure:

```
BEGIN-STRUCTURE point \ -- a-addr 0 ; -- lenp
  FIELD: p.x             \ -- a-addr cell
  FIELD: p.y             \ -- a-addr cell*2
END-STRUCTURE


BEGIN-STRUCTURE rect  \ -- a-addr 0 ; -- lenr
  point +FIELD r.tlhc \ -- a-addr cell*2
  point +FIELD r.brhc \ -- a-addr cell*4
END-STRUCTURE
```

Alignment:

In practice, structures are used for two different purposes with incompatible requirements:

a) For collecting related internal-use data into a convenient "package" that can be referred to by a single "handle". For this use, alignment is important, so that efficient native fetch and store instructions can be used.

b) For mapping external data structures like hardware register maps and protocol packets. For this use, automatic alignment is inappropriate, because the alignment of the external data structure often doesn't match the rules for a given processor.

Many languages cater for the first use, but ignore the second. This leads to various customized solutions, usage requirements, portability problems, bugs, etc. **+FIELD** is defined to be non-aligning, while the named field defining words (*x*FIELD:) are aligning. This is intentional and allows for both uses.

The standard currently defines an aligned field defining word for each of the standard data types:

CFIELD:	a character
FIELD:	a native integer (single cell)
FFIELD:	a native float
SFFIELD:	a 32 bit float
DFFIELD:	a 64 bit float

Although this is a sufficient set, most systems provide facilities to define field defining words for standard data types.

Future:

The following cannot be defined until the required addressing has been defined. The names should be considered reserved until then.

BFIELD:	1 byte (8 bit) field
WFIELD:	16 bit field
LFIELD:	32 bit field
XFIELD:	64 bit field

A.10.6.2.1305 **EKEY**

For some input devices, such as keyboards, more information is available than can be returned by a single execution of **KEY**. **EKEY** provides a standard word to access a system-dependent set of events.

EKEY and **EKEY?** are implementation specific; no assumption can be made regarding the interaction between the pairs **EKEY**/**EKEY?** and **KEY**/**KEY?**. This standard does not define a timing relationship between **KEY?** and **EKEY?**. Undefined results may be avoided by using only one pairing of **KEY**/**KEY?** or **EKEY**/**EKEY?** in a program for each input stream.

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. A standard program may only interpret the results of **EKEY** via the translation words provided for that purpose (**EKEY>CHAR** and **EKEY>FKEY**).

See: **A.6.1.1750** KEY, **10.6.2.1306** EKEY>CHAR and **10.6.2.1306.40** EKEY>FKEY.

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.10.6.2.1306.40 **EKEY>FKEY**

EKEY produces an abstract cell type for a keyboard event (e.g., a keyboard scan code). **EKEY>FKEY** checks if such an event corresponds to a special (non-graphic) key press, and if so, returns a code for the special key press. The encoding of special keys (returned by **EKEY>FKEY**) may be different from the encoding of these keys as keyboard events (input to **EKEY>FKEY**).

Typical Use:

```
... EKEY EKEY>FKEY IF
  CASE
    K-UP OF ... ENDOF
    K-F1 OF ... ENDOF
    K-LEFT K-SHIFT-MASK OR K-CTRL-MASK OR OF ... ENDOF
    ...
  ENDCASE
ELSE
  ...
THEN
```

The codes for the special keys are system-dependent, but this standard provides words for getting the key codes for a number of keys:

Word	Key	Word	Key
K-F1	F1	K-LEFT	cursor left
K-F2	F2	K-RIGHT	cursor right
K-F3	F3	K-UP	cursor up
K-F4	F4	K-DOWN	cursor down
K-F5	F5	K-HOME	home or Pos1
K-F6	F6	K-END	End
K-F7	F7	K-PRIOR	PgUp or Prior
K-F8	F8	K-NEXT	PgDn or Next
K-F9	F9	K-INSERT	Insert
K-F10	F10	K-DELETE	Delete
K-F11	F11		
K-F12	F12		

In addition, you can get codes for shifted variants of these keys by ORing with K-SHIFT-MASK, K-CTRL-MASK and/or K-ALT-MASK, e.g. K-CTRL-MASK K-ALT-MASK OR K-DELETE OR. The masks for the shift keys are:

Word	Key
K-SHIFT-MASK	Shift
K-CTRL-MASK	Ctrl
K-ALT-MASK	Alt

Note that not all of these keys are available on all systems, and not all combinations of keys and shift keys are available. Therefore programs should be written such that they continue to work (although less conveniently or with less functionality) if these key combinations cannot be produced.

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.10.6.2.1518   FIELD:

Create an aligned single-cell field in a data structure.

The various xFIELD: words provide for different alignment and size allocation.

The *x*`FIELD:` words could be defined as:

```
: FIELD:    ( n1 "name" -- n2 ; addr1 -- addr2 )    ALIGNED  1 CELLS   +FIELD ;
: CFIELD:   ( n1 "name" -- n2 ; addr1 -- addr2 )             1 CHARS   +FIELD ;
: FFIELD:   ( n1 "name" -- n2 ; addr1 -- addr2 )   FALIGNED  1 FLOATS  +FIELD ;
: SFFIELD:  ( n1 "name" -- n2 ; addr1 -- addr2 )  SFALIGNED  1 SFLOATS +FIELD ;
: DFFIELD:  ( n1 "name" -- n2 ; addr1 -- addr2 )  DFALIGNED  1 DFLOATS +FIELD ;
```

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 optional File-Access word set

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.11.3.4   Other transient regions

Additional transient buffers are provided for use by `S"` and `S\"`. The buffers should be able to store two consecutive strings, thus allowing the command line:

```
S" name1" S" name2" RENAME-FILE
```

The buffers may be implemented in a circular arrangement, where a string is placed into the next available buffer. When there are no buffers available, the oldest buffer is overwritten.

`S"` and `S\"` may share the same buffers.

The list of words using memory in transient regions is extended to include **11.6.1.2165** `S"` and **11.6.2.2266** `S\"`. See **3.3.3.6 Other transient regions**.

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 committee intends that, in **BIN** mode, the contents of these storage elements can be written to a file and later read back without alteration.

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 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 transient 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                  ( )
```

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.

A.11.6.2.1714 **INCLUDE**

Typical use:

      INCLUDE filename

A.11.6.2.2144.10 **REQUIRE**

Typical use:

      REQUIRE filename

A.11.6.2.2144.50 **REQUIRED**

Typical use:

      S" filename**"** **REQUIRED**

A.12   The optional Floating-Point word set

The current base for floating-point input must be **DECIMAL**. Floating-point input is not allowed in an arbitrary base. All floating-point numbers to be interpreted by a standard system must contain the exponent indicator "E" (see **12.3.7 Text interpreter input number conversion**). Consensus in the 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.

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 Forth-2012, 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 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 Forth-2012 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 this standard may include such a requirement.

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.12.6.2.1427   **F.**

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

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.

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 $\sinh(x)$ 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 calculation method allows the hyperbolic functions to be computed with six-digit accuracy. For example, sinh 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.12.6.2.1516** FEXPM1.

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 optional Locals word set

A.13.3   Additional usage requirements

Rule 13.3.3.2d could be relaxed without affecting the integrity of the rest of this structure. 13.3.3.2c could not be.

13.3.3.2b 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.

Access to previously declared local variables is prohibited by Section 13.3.3.2d 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 13.3.3.2g. Although words, such as **(LOCAL)**, 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.13.6.2.2550 {:

The Forth 94 Technical Committee was unable to identify any common practice for locals. It provided a way to define locals and a method of parsing them in the hope that a common practice would emerge.

Since then, common practice has emerged. Most implementations that provide **(LOCAL)** and **LOCALS|** also provide some form of the { ... } notation; however, the phrase { ...} conflicts with other systems. The **{:** ... **:}** notation is a compromise to avoid name conflicts.

The notation provides for different kinds of local: those that are initialized from the data stack at run-time, uninitialized locals, and outputs. Initialized locals are separated from uninitialized locals by '|'. The definition of locals is terminated by '--' or ':}'.

All text between '--' and ':}' is ignored. This eases documentation by allowing a complete stack comment in the locals definition.

The '|' (ASCII $7C) character is widely used as the separator between local arguments and local values. Some implementations have used '\' (ASCII $5C) or '¦' ($A6). Systems are free to continue to provide these alternative separators. However, only the recognition of the '|' separator is mandatory. Therefore portable programs must use the '|' separator.

A number of systems extend the locals notation in various ways. Some of these extensions may emerge as common practice. This standard has reserved the notation used by these extensions to avoid difficulties when porting code to these systems. In particular local names ending in ':' (colon), '[' (open bracket), or '^' (caret) are reserved.

A.14 The optional 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 optional 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 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.15.3.1 Name tokens

Name tokens are an abstract data type identifying named words. You can use words such as **NAME>STRING** to get information out of name tokens.

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 copy 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 *orig*s and *dest*s 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 *name* tries to infer the previous dictionary state from *name*; this is not always possible. As a consequence, **FORGET** *name* removes *name* and all following words in the name space.

See **A.6.2.1850** MARKER.

A.15.6.2.1908 N>R

An implementation may store the stack items in any manner. It may store them on the return stack, in any order. A stack-constrained system may prefer to use a buffer to store the items and place a reference to the buffer on the return stack.

 See: **6.2.2182** SAVE-INPUT, **6.2.2148** RESTORE-INPUT, **16.6.1.1647** GET-ORDER, **16.6.1.2197** SET-ORDER.

A.15.6.2.1909.10 NAME>COMPILE

In a traditional *xt*+immediate-flag system, the *x xt* returned by **NAME>COMPILE** is typically *xt1 xt2*, where *xt1* is the *xt* of the word under consideration, and *xt2* is the *xt* of **EXECUTE** (for immediate words) or **COMPILE,** (for words with default compilation semantics).

If you want to **POSTPONE** *nt*, you can do so with

```
NAME>COMPILE SWAP POSTPONE LITERAL COMPILE,
```

A.15.6.2.2297 TRAVERSE-WORDLIST

Typical use:

```
: WORDS-COUNT ( x nt - x' f ) DROP 1+ TRUE ;
0 ' WORDS-COUNT FORTH-WORDLIST TRAVERSE-WORDLIST .
```

prints a count of the number of words in the **FORTH-WORDLIST**.

```
: ALL-WORDS NAME>STRING CR TYPE TRUE ;
' ALL-WORDS GET-CURRENT TRAVERSE-WORDLIST
```

prints the names of words in the current compilation wordlist.

```
: CONTAINS-STRING
  NAME>STRING 2OVER SEARCH IF CR TYPE THEN FALSE ;
S" COM" ' CONTAINS-STRING GET-CURRENT TRAVERSE-WORDLIST
```

prints the name of a word containing the string "COM", if it exists, and then terminates.

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 optional Programming tools extensions word set are useful for this purpose.

A.16   The optional Search-Order word set

The search-order word set is intended to be a portable "construction set" from which search-order words may be built. **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 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.

A.16.2   Additional terms and notation

search order

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

A.16.3   Additional usage requirements

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`

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

A.17   The optional String word set

A.17.6   Glossary

A.17.6.1.0245   `/STRING`

/STRING is used to remove or add characters relative to the current position in 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.

```
S" ABC" 2 /STRING 2DUP TYPE \ outputs "C"
-1 /STRING TYPE \ outputs "BC"
```

A.17.6.1.0910   `CMOVE`

If $c\text{-}addr_2$ lies within the source region (i.e., when $c\text{-}addr_2$ is not less than $c\text{-}addr_1$ and $c\text{-}addr_2$ is less than the quantity $c\text{-}addr_1$ *u* **CHARS +**), memory propagation occurs.

Assume a character string at address 100: "ABCD". Then after

```
    100 DUP   CHAR+   3 CMOVE
```

the string at address 100 is "AAAA".

See **A.6.1.1900** MOVE.

A.17.6.1.0920   `CMOVE>`

If $c\text{-}addr_1$ lies within the destination region (i.e., when $c\text{-}addr_1$ is greater than or equal to $c\text{-}addr_2$ and $c\text{-}addr_2$ is less than the quantity $c\text{-}addr_1$ *u* **CHARS +**), memory propagation occurs.

Assume a character string at address 100: "ABCD". Then after

```
    100 DUP CHAR+ SWAP 3 CMOVE>
```

the string at address 100 is "DDDD".

See **A.6.1.1900** MOVE.

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
```

A.17.6.2.2255 **SUBSTITUTE**

Many applications need to be able to perform text substitution, for example:

> Your balance at ⟨*time*⟩ on ⟨*date*⟩ is ⟨*currencyvalue*⟩.

Translation of a sentence or message from one language to another may result in changes to the displayed parameter order. The example, the Afrikaans translation of this sentence requires a different order:

> Jou balans op ⟨*date*⟩ om ⟨*time*⟩ is ⟨*currencyvalue*⟩.

The words **SUBSTITUTE** and **REPLACES** provide for this requirements by defining a text substitution facility. For example, we can provide an initial string in the form:

```
Your balance at %time% on %date% is %currencyvalue%.
```

The % is used as delimiters for the substitution name. The text "currencyvalue", "date" and "time" are text substitutions, where the replacement text is defined by **REPLACES**:

```
: date S" 10/Nov/2014" ;
: time S" 14:52" ;
date S" date" REPLACES
time S" time" REPLACES
```

The substitution name "date" is defined to be replaced with the string "10/Nov/2014" and "time" to be replaced with "14:52". Thus **SUBSTITUTE** would produce the string:

```
Your balance at 14:52 on 10/Nov/2014 is %currencyvalue%.
```

As the substitution name "currencyvalue" has not been defined, it is left unchanged in the resulting string.

The return value *n* is nonnegative on success and indicates the number of substitutions made. In the above example, this would be two. A negative value indicates that an error occurred. As substitution is not recursive, the return value could be used to provide a recursive substitution.

Implementation of **SUBSTITUTE** may be considered as being equivalent to a wordlist which is searched. If the substitution name is found, the word is executed, returning a substitution string. Such words can be deferred or multiple wordlists can be used. The implementation techniques required are similar to those used by **ENVIRONMENT?**. There is no provision for changing the delimiter character, although a system may provide system-specific extensions.

A.18   The optional Extended-Character word set

Forth defines graphic and control characters from the ASCII character set. ASCII was originally designed for the English language. However, most western languages fit somewhat into the Forth framework, since

a single byte is sufficient to encode all characters in each language, although different languages may use different encodings; Latin-1 and Latin-15 are widely used. For other languages, different character sets have to be used, several of which are variable-width. Currently, the most popular representative of the variable-width character sets is UTF-8.

Many Forth systems today are case insensitive, to accept lower case standard words. It is sufficient to be case insensitive for the ASCII subset to make this work — this saves a large code mapping table for comparison of other symbols. Case is mostly an issue of European languages (Latin, Greek, and Cyrillic), but similar issues exist between traditional and simplified Chinese (finally being dealt with by Unihan), and between different Latin code pages in the Universal Character Set (UCS), e.g., full width vs. normal half width Latin letters.

Furthermore, UCS allows composition of glyphs and has direct encoding for composed glyphs, which look the same, but have different encodings. This is not a problem for a Forth system to solve.

Some encodings (not UTF-8) might give surprises when you use a case insensitive ASCII-compare that's 8-bit safe, but not aware of the current encoding.

The xchar word set does not address problems that come from using multiple different encodings and switching or converting between them. It is good practice for a system implementing xchar to support UTF–8.

A.18.6   Glossary

A.18.6.2.0895   CHAR

The behavior of the extended version of **CHAR** is fully backward compatible with **6.1.0895** CHAR.

Annex B
(informative)
Bibliography

Industry standards

Forth-77 Standard, Forth Users Group, FST-780314.
Forth-78 Standard, Forth International Standards Team.
Forth-79 Standard, Forth Standards Team.
Forth-83 Standard and Appendices, Forth Standards Team.

The standards referenced in this section were developed by the Forth Standards Team, a volunteer group which included both implementors and users. This was a volunteer organization operating under its own charter and without any formal ties to ANSI, IEEE or any similar standards body.

The following standards where developed under the auspices of ANSI. The committee drawing up the ANSI standard included several members of the Forth Standards Team.

ANSI X3.215-1994 Information Systems — Programming Language FORTH
ISO/IEC 15145:1997 Information technology. Programming languages. FORTH

Books

Brodie, L. *Thinking FORTH*. Englewood Cliffs, NJ: Prentice Hall, 1984. Now available from http://thinking-forth.sourceforge.net/

Brodie, L. *Starting FORTH* (2nd edition). Englewood Cliffs, NJ: Prentice Hall, 1987.

Feierbach, G. and Thomas, P. *Forth Tools & Applications*. Reston, VA: Reston Computer Books, 1985.

Haydon, Dr. Glen B. *All About FORTH* (3rd edition). La Honda, CA: 1990.

Kelly, Mahlon G. and Spies, N. *FORTH: A Text and Reference*. Englewood Cliffs, NJ: Prentice Hall, 1986.

Knecht, K. *Introduction to Forth*. Indiana: Howard Sams & Co., 1982.

Koopman, P. *Stack Computers, The New Wave*. Chichester, West Sussex, England: Ellis Horwood Ltd. 1989.

Martin, Thea, editor. *A Bibliography of Forth References* (3rd edition). Rochester, New York: Institute of Applied Forth Research, 1987.

McCabe, C. K. *Forth Fundamentals* (2 volumes). Oregon: Dilithium Press, 1983.

Ouverson, Marlin, editor. *Dr. Dobbs Toolbook of Forth*. Redwood City, CA: M&T Press, Vol. 1, 1986; Vol. 2, 1987.

Pelc, Stephen. *Programming Forth*. Southampton, England: MicroProcessor Engineering Limited, 2005. http://www.mpeforth.com/arena/ProgramForth.pdf.

Pountain, R. *Object Oriented Forth*. London, England: Academic Press, 1987.

Rather, Elizabeth D. *Forth Application Techniques*. FORTH, Inc., 2006. ISBN: 978-0966215618.

Rather, Elizabeth D. and Conklin, Edward K. *Forth Programmer's Handbook* (3rd edition). BookSurge Publishing, 2007. ISBN: 978-1419675492.

Terry, J. D. *Library of Forth Routines and Utilities*. New York: Shadow Lawn Press, 1986.

Tracy, M. and Anderson, A. *Mastering FORTH* (revised edition). New York: Brady Books, 1989.

Winfield, A. *The Complete Forth*. New York: Wiley Books, 1983.

Journals, magazines and newsletters

Forsley, Lawrence P., Conference Chairman. *Rochester Forth Conference Proceedings*. Rochester, New York: Institute of Applied Forth Research, 1981 to present.

Forsley, Lawrence P., Editor-in-Chief. *The Journal of Forth Application and Research*. Rochester, New York: Institute of Applied Forth Research, 1983 to present.

Frenger, Paul, editor. *SIGForth Newsletter*. New York, NY: Association for Computing Machinery, 1989 to present.

Ouverson, Marlin, editor. *Forth Dimensions*. San Jose, CA: The Forth Interest Group, 1978 to present.

Reiling, Robert, editor. *FORML Conference Proceedings*. San Jose, CA: The Forth Interest Group, 1980 to present.

Ting, Dr. C. H., editor. *More on Forth Engines*. San Mateo, CA: Offete Enterprises, 1986 to present.

Selected articles

Hayes, J.R. "Postpone" *Proceedings of the 1989 Rochester Forth Conference*. Rochester, New York: Institute for Applied Forth Research, 1989.

Kelly, Guy M. "Forth". *McGraw-Hill Personal Computer Programming Encyclopedia — Languages and Operation Systems*. New York: McGraw-Hill, 1985.

Kogge, P. M. "An Architectural Trail to Threaded Code Systems". *IEEE Computer* (March, 1982).

Moore, C. H. "The Evolution of FORTH — An Unusual Language". *Byte* (August 1980).

Rather, E. D. "Forth Programming Language". *Encyclopedia of Physical Science & Technology* (Vol. 5). New York: Academic Press, 1987.

Rather, E. D. "FORTH". *Computer Programming Management*. Auerbach Publishers, Inc., 1985.

Rather, E. D.; Colburn, D. R.; Moore, C. H. "The Evolution of Forth". *ACM SIGPLAN Notices* (Vol. 28, No. 3, March 1993).

Annex C
(informative)
Compatibility analysis

Before this standard, there were several industry standards for Forth. The most influential are listed here in chronological order, along with the major differences between this standard and the most recent, Forth 94.

C.1   FIG Forth (circa 1978)

FIG Forth was a "model" implementation of the Forth language developed by the Forth Interest Group (FIG). In FIG Forth, a relatively small number of words were implemented in processor-dependent machine language and the rest of the words were implemented in Forth. The FIG model was placed in the public domain, and was ported to a wide variety of computer systems. Because the bulk of the FIG Forth implementation was the same across all machines, programs written in FIG Forth enjoyed a substantial degree of portability, even for "system-level" programs that directly manipulate the internals of the Forth system implementation.

FIG Forth implementations were influential in increasing the number of people interested in using Forth. Many people associate the implementation techniques embodied in the FIG Forth model with "the nature of Forth".

However, FIG Forth was not necessarily representative of commercial Forth implementations of the same era. Some of the most successful commercial Forth systems used implementation techniques different from the FIG Forth "model".

C.2   Forth 79

The Forth-79 Standard resulted from a series of meetings from 1978 to 1980, by the Forth Standards Team, an international group of Forth users and vendors (interim versions known as Forth 77 and Forth 78 were also released by the group).

Forth 79 described a set of words defined on a 16-bit, twos-complement, unaligned, linear byte-addressing virtual machine. It prescribed an implementation technique known as "indirect threaded code", and used the ASCII character set.

The Forth-79 Standard served as the basis for several public domain and commercial implementations, some of which are still available and supported today.

C.3   Forth 83

The Forth-83 Standard, also by the Forth Standards Team, was released in 1983. Forth 83 attempted to fix some of the deficiencies of Forth 79.

Forth 83 was similar to Forth 79 in most respects. However, Forth 83 changed the definition of several well-defined features of Forth 79. For example, the rounding behavior of integer division, the base value of the operands of **PICK** and **ROLL**, the meaning of the address returned by **'**, the compilation behavior of **'**, the value of a "true" flag, the meaning of NOT, and the "chaining" behavior of words defined by VOCABULARY were all changed. Forth 83 relaxed the implementation restrictions of Forth 79 to allow

any kind of threaded code, but it did not fully allow compilation to native machine code (this was not specifically prohibited, but rather was an indirect consequence of another provision).

Many new Forth implementations were based on the Forth-83 Standard, but few "strictly compliant" Forth-83 implementations exist.

Although the incompatibilities resulting from the changes between Forth 79 and Forth 83 were usually relatively easy to fix, a number of successful Forth vendors did not convert their implementations to be Forth 83 compliant. For example, the most successful commercial Forth for Apple Macintosh computers is based on Forth 79.

C.4   ANS Forth (1994)

In the mid to late 1980s the computer industry underwent a rapid and profound change. The personal-computer market matured into a business and commercial market, while the market for ROM-based embedded control computers grew substantially. Improvements in custom processor design allowed for the development of numerous "Forth chips," customized for the execution of the Forth language.

In order to take full advantage of evolving technology, many Forth implementations ignored some of the restrictions imposed by the implied "virtual machine" of previous standards. The ANS Forth committee was formed in 1987 to address the fragmentation within the Forth community caused not only by the difference between Forth 79 and Forth 83 but the exploitation of technical developments.

The committee undertook a comprehensive review of a variety of existing implementations, especially those with substantial user bases and/or considerable success in the market place. This allowed them to identify and document features common to these systems, many of which had not been included in any previous standard. This was the most comprehensive review of Forth systems to date, taking eighty-seven days covering twenty-three meetings over eight years. The inclusive nature of the standard allowed the various factions within the community to unify in support of ANS Forth, with many systems providing a compatibility layer.

The committee chose to move away from prescribing stringent requirements as previous standards had, with the specification of a virtual machine. It preferred to describe the operation of the virtual machine, without reference to its implementation, thus allowing an implementor to take full advantage of any technical developments while providing the developer with a complete list of entitlements.

This required the identification of implicit assumptions made by the previous standards, making them explicit and abstracting them into more general concepts where possible. A good example of this is the size of an item on the stack. In previous standards this was assumed to be 16 bits wide. This was no longer a valid assumption. ANS Forth introduced the concept of the *cell*, allowing an implementation to use a stack size most suited to the environment.

The American National Standards Institution (ANSI) published the ANS Forth Standard in 1994 with the title "*ANSI X3.215-1994 Information Systems — Programming Language FORTH*". This is referenced throughout this document as Forth 94.

C.5   ISO Forth (1997)

ANSI submitted the Forth 94 Standard to the ISO (International Organization for Standardization) and IEC (International Electrotechnical Commission) joint committee for consideration as an international standard.

The ISO/IEC adopted the Forth 94 document as an international standard in 1997, publishing it under the title "*ISO/IEC 15145:1997 Information technology. Programming languages. FORTH*".

C.6   Approach of this standard

During a workshop on the Forth standard at the EuroForth conference in 2004 it was agreed that Forth 94 required updating.

A committee was formed and agreed that the process should be as open as possible, adopting the Usenet RfD/CfV (Request for Discussion/Call for Votes) process to produce semi-formal proposals for changes to the standard. In addition to general discussion on the `comp.lang.forth` usenet news group, a moderated mailing list (with public archive) was created for those who do not follow the news group. Standards meetings to discuss CfVs were held in public in conjunction with the EuroForth conference.

The work of the Forth 94 committee was the basis of this standard, informally called Forth 200*x*. The aim of the Forth 200*x* committee is to produce a rolling document, with the standard constantly being updated based on discussion of proposals and the corresponding votes. A snapshot document is occasionally produced, with this document being the first.

The Forth 200*x* committee defined a procedure for proposals. In addition to the formal text of the proposal, they had to include: the rationale behind the change; a reference implementation, or a description of the reason a reference implementation cannot be presented; unit testing for the proposed change, especially for border conditions. See **Proposals Process** (page vii) for a full description.

C.7   Differences from Forth 94

C.7.1   Removed Obsolete Words

Forth 94 declared seven words as 'obsolescent', all but **FORGET** have been removed from this standard.

Words affected:
> #TIB, CONVERT, EXPECT, QUERY, SPAN, TIB, **WORD**.

Reason:
> Obsolescent words have been removed.

Impact:
> **WORD** is no longer required to leave a space at the end of the returned string.

> It is recommended that, should the obsolete words be included, they have the behaviour described in Forth 94. The names should not be reused for other purposes.

Transition/Conversion:
> The functions of TIB and #TIB have been superseded by **SOURCE**.

> The function of CONVERT has been superseded by **>NUMBER**.

> The functions of EXPECT and SPAN have been superseded by **ACCEPT**.

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

C.7.2   Separate Floating-point Stack is now Standard

Previously systems could implement either a separate floating-point stack or a combined floating-point/data stack; programs were required to cater for both (or declare an environmental dependency on a particular variant).

Words Affected:
> All floating-point words.

Reason:
> The developing of software that may be used with either a combined stack or a separate stack is extremely difficult and costly. While some of the systems surveyed provide a combined floating-point/data stack, they all provide a separate floating-point stack.

Impact:
> Forth 94 programs with an environmental dependency on a separate floating-point stack become standard programs.

> Forth 94 programs with an environmental dependency on a combined stack retain the environmental dependency.

> Forth 94 programs (without environmental dependency, i.e., those working on either kind of system) remain standard programs.

> Forth 94 systems that implement a separate floating-point stack continue to be standard systems.

> Forth 94 systems that implement a combined stack become systems with an environmental restriction of not providing a separate floating-point stack, but a combined stack.

Transition/Conversion:
> Any code that has an environmental dependency on the use of a combined floating-point/data stack should be ported to use a separate floating-point stack.

> A system that has an environmental restriction on using a combined floating-point/data stack should consider providing a separate floating-point stack.

C.7.3   Using ENVIRONMENT? to inquire whether a word set is present

With the advent of a new standard, it was necessary to review the meaning of word set queries. Compatibility with Forth 94 demands that a word set query produce the same result as for Forth 94; i.e., querying for CORE-EXT returns true only if all the Forth 94 CORE EXT words are present. The question was how to distinguish between word sets described by this and subsequent standards.

The committee considered adding a year indicator to the word set name ("CORE-EXT-2012") or a providing a general query ("Forth-2012") which could be combined with the word-set query. As the committee could find very few examples of the word-set queries being used, it chose not to update the word set-query mechanism, but rather to mark it as obsolescent.

Words Affected:
> ENVIRONMENT?

Reason:

> The use of the word-set query to inquire whether a word set is present in the system has been marked obsolescent. If present the query indicates the word set, as documented in Forth 94, is available.

Impact:

> Forth 94 did not guarantee the presence of these queries. Many systems that provided all the words in a particular word set did not provide the corresponding query. Portable programs are not affected as they could not rely on this function.

Transition/Conversion:

> There is no direct equivalent to determine the presence of a whole word set. The **15.6.2.2530.30** `[DEFINED]` and **15.6.2.2534** `[UNDEFINED]` words can be used to detect the availability (or otherwise) of individual words.

C.7.4   Additional **TO** targets

6.2.2295 `TO` has been extended to act on targets defined with **12.6.2.1628** `FVALUE` and **8.6.2.0435** `2VALUE`.

Words affected:

> **TO**

C.7.5   Input/Output return values

Words affected:

> All words that return an *ior*.

Reason:

> Forth 94 left the error code (*ior*) implementation-defined, although it did recommend an *ior* to be a **THROW** code. Forth 2012 now requires an *ior* to be a **THROW** code.

Transition/Conversion:

> Forth 94 programs are not affected. Programs that are dependent on *iors* being throwable are no longer required to document the dependency.

> Forth 94 systems that abided by the recommendation are not affected. Systems that did not heed this advice are required to do so. A number of **THROW** codes were added to table 9.1 to ease this transition.

C.7.6   Minimum number of locals

Words affected:

> **(LOCAL)** , **LOCALS|**

Reason:

> Some programs require more than eight locals.

Transition/Conversion:

> Existing programs are unaffected. Systems implementing the locals word set have to be changed to support at least 16 (previously 8) locals.

C.7.7 Number prefixes

Decimal, hexadecimal, binary number literals can now be written irrespective of **BASE** by using the prefix #, $, %. Also, character literals can be written as 'c'.

Standard programs are unaffected. Systems have to be changed to recognize these forms.

See **3.4.1.3 Text interpreter input number conversion**.

C.7.8 **SOURCE-ID** Clarification

When interpreting text from a file, the relationship between the position in the file returned by **SOURCE-ID**, and the current interpretation position is undefined.

C.7.9 **FASINH**

An ambiguous condition on *r1* being less than 0 was removed.

Existing programs are not affected. Existing systems are unlikely to be affected.

C.7.10 **FATAN2**

Words affected:
    FATAN2

Reason:
    The result is now specified more tightly: it is the principal angle (between -pi and pi).

Impact:
    Forth 94 compliant programs are not affected.

Transition/Conversion:
    Systems may have to change **FATAN2** to return the principal angle.

C.8 Additional words

The following words have been added to the standard:

C.8.6 Core word sets

The following words have been added to **6.2 Core extension words**:

6.2.0698 ACTION-OF	**6.2.1175** DEFER!	**6.2.1725** IS
6.2.0825 BUFFER:	**6.2.1177** DEFER@	**6.2.2020** PARSE-NAME
6.2.1173 DEFER	**6.2.1675** HOLDS	**6.2.2266** S\"

C.8.8 Double-Number word sets

The following words have been added to **8.6.2 Double-Number extension words**:

8.6.2.0435 2VALUE

C.8.10   Facility word sets

The following words have been added to **10.6.2 Facility extension words**:

10.6.2.0135 `+FIELD`	**10.6.2.1740.06** `K-F1`	**10.6.2.1740.17** `K-F9`
10.6.2.0763 `BEGIN-STRUCTURE`	**10.6.2.1740.07** `K-F10`	**10.6.2.1740.18** `K-HOME`
10.6.2.0893 `CFIELD:`	**10.6.2.1740.08** `K-F11`	**10.6.2.1740.19** `K-INSERT`
10.6.2.1306.40 `EKEY>FKEY`	**10.6.2.1740.09** `K-F12`	**10.6.2.1740.20** `K-LEFT`
10.6.2.1336 `END-STRUCTURE`	**10.6.2.1740.10** `K-F2`	**10.6.2.1740.21** `K-NEXT`
10.6.2.1518 `FIELD:`	**10.6.2.1740.11** `K-F3`	**10.6.2.1740.22** `K-PRIOR`
10.6.2.1740.01 `K-ALT-MASK`	**10.6.2.1740.12** `K-F4`	**10.6.2.1740.23** `K-RIGHT`
10.6.2.1740.02 `K-CTRL-MASK`	**10.6.2.1740.13** `K-F5`	**10.6.2.1740.24** `K-SHIFT-MASK`
10.6.2.1740.03 `K-DELETE`	**10.6.2.1740.14** `K-F6`	**10.6.2.1740.25** `K-UP`
10.6.2.1740.04 `K-DOWN`	**10.6.2.1740.15** `K-F7`	
10.6.2.1740.05 `K-END`	**10.6.2.1740.16** `K-F8`	

C.8.11   File-Access word sets

The following words have been added to **11.6.2 File-Access extension words**:

11.6.2.1714 `INCLUDE`	**11.6.2.2144.10** `REQUIRE`	**11.6.2.2144.50** `REQUIRED`

C.8.12   Floating-Point word sets

The following words have been added to **12.6.2 Floating-Point extension words**:

12.6.2.1207.40 `DFFIELD:`	**12.6.2.1627** `FTRUNC`	**12.6.2.2175** `S>F`
12.6.2.1471 `F>S`	**12.6.2.1628** `FVALUE`	**12.6.2.2206.40** `SFFIELD:`
12.6.2.1517 `FFIELD:`		

C.8.13   Locals word sets

The following words have been added to **13.6.2 Locals extension words**:

13.6.2.2550 `{:`

C.8.15   Programming-Tools word sets

The following words have been added to the **15.6.2 Programming-Tools extension words**:

15.6.2.1908 `N>R`	**15.6.2.2264** `SYNONYM`
15.6.2.1909.10 `NAME>COMPILE`	**15.6.2.2297** `TRAVERSE-WORDLIST`
15.6.2.1909.20 `NAME>INTERPRET`	**15.6.2.2530.30** `[DEFINED]`
15.6.2.1909.40 `NAME>STRING`	**15.6.2.2534** `[UNDEFINED]`
15.6.2.1940 `NR>`	

C.8.17   String word sets

The following words have been added to the **17.6.2 String extension words**:

17.6.2.2141 `REPLACES`	**17.6.2.2255** `SUBSTITUTE`	**17.6.2.2375** `UNESCAPE`

C.8.18   Extended-Character word sets

The Extended Character word set was introduced by Forth-2012.

The following words make up **18 The optional Extended-Character word set**:

18.6.1.2486.50 X-SIZE	**18.6.1.2487.25** XC-SIZE	**18.6.1.2488.30** XKEY
18.6.1.2487.10 XC!+	**18.6.1.2487.35** XC@+	**18.6.1.2488.35** XKEY?
18.6.1.2487.15 XC!+?	**18.6.1.2487.40** XCHAR+	**18.6.2.0145** +X/STRING
18.6.1.2487.20 XC,	**18.6.1.2488.10** XEMIT	**18.6.2.0175** -TRAILING-GARBAGE

The following words make up **18.6.2 Extended-Character extension words**:

18.6.2.0895 CHAR	**18.6.2.2486.70** X-WIDTH	**18.6.2.2488.20** XHOLD
18.6.2.1306.60 EKEY>XCHAR	**18.6.2.2487.30** XC-WIDTH	**18.6.2.2495** X\STRING-
18.6.2.2008 PARSE	**18.6.2.2487.45** XCHAR-	**18.6.2.2520** [CHAR]

Annex D
(informative)
Portability guide

D.1   Introduction

A primary goal of Forth 94 was to enable a programmer to write Forth programs that work on a wide variety of machines, Forth-2012 continues this practice. This goal is accomplished by allowing some key Forth terms to be implementation defined (e.g., cell size) and by providing Forth operators (words) that conceal the implementation. This allows the implementor to produce the Forth system that most effectively uses the native hardware. The machine independent operators, together with some programmer discipline, support program portability.

It can be difficult for someone familiar with only one machine architecture to imagine the problems caused by transporting programs between dissimilar machines. This Annex provides guidelines for writing portable Forth programs. The first section describes ways to make a program hardware independent.

The second section describes assumptions about Forth implementations that many programmers make, but can't be relied upon in a portable program.

D.2   Hardware peculiarities

D.2.1   Data/memory abstraction

This standard gives definitions for data and memory that apply to a wide variety of computers. These definitions give us a way to talk about the common elements of data and memory while ignoring the details of specific hardware. Similarly, Forth programs that use data and memory in ways that conform to these definitions can also ignore hardware details. The following sections discuss the definitions and describe how to write programs that are independent of the data and memory peculiarities of different computers.

D.2.2   Definitions

Three terms defined by this standard are address unit, cell, and character.

The address space of a Forth system is divided into an array of address units; an address unit is the smallest collection of bits that can be addressed. In other words, an address unit is the number of bits spanned by the addresses *addr* and *addr*+1. The most prevalent machines use 8-bit address units, but other address unit sizes exist.

In this standard, the size of a cell is an implementation-defined number of address units. Forth implemented on a 16-bit microprocessor could use a 16-bit cell and an implementation on a 32-bit machine could use a 32-bit cell. Less common cell sizes (e.g., 18-bit or 36-bit machines, etc.) could implement Forth systems with their native cell sizes. In all of these systems, Forth words such as **DUP** and **!** do the same things (duplicate the top cell on the stack and store the second cell into the address given by the first cell, respectively).

Similarly, the definition of a character has been generalized to be an implementation-defined number of address units (but at least eight bits). This removes the need for a Forth implementor to provide 8-bit characters on processors where it is inappropriate. For example, on an 18-bit machine with a 9-bit address

unit, a 9-bit character would be most convenient. Since, by definition, you can't address anything smaller than an address unit, a character must be at least as big as an address unit. This will result in big characters on machines with large address units. An example is a 16-bit cell addressed machine where a 16-bit character makes the most sense.

D.2.3   Addressing memory

One of the most common portability problems is the addressing of successive cells in memory. Given the memory address of a cell, how do you find the address of the next cell? On a byte-addressed machine with 32-bit cells the code to find the next cell would be `4 +`. The code would be **1+** on a cell-addressed processor and `16 +` on a bit-addressed processor with 16-bit cells. This standard provides a next-cell operator named **CELL+** that can be used in all of these cases. Given an address, **CELL+** adjusts the address by the size of a cell (measured in address units).

A related problem is that of addressing an array of cells in an arbitrary order. This standard provides a portable scaling operator named **CELLS**. Given a number *n*, **CELLS** returns the number of address units needed to hold *n* cells. Using **CELLS**, we can make a portable definition of an ARRAY defining word:

```
: ARRAY ( u -- ) CREATE   CELLS ALLOT
  DOES> ( u -- addr ) SWAP CELLS + ;
```

There are also portability problems with addressing arrays of characters. In a byte-addressed machine, the size of a character equals the size of an address unit. Addresses of successive characters in memory can be found using **1+** and scaling indices into a character array is a no-op (i.e., `1 *`). However, there could be implementations where a character is larger than an address unit. The **CHAR+** and **CHARS** operators, analogous to **CELL+** and **CELLS** are available to allow maximum portability.

This standard generalizes the definition of some Forth words that operate on regions of memory to use address units. One example is **ALLOT**. By prefixing **ALLOT** with the appropriate scaling operator (**CELLS**, **CHARS**, etc.), space for any desired data structure can be allocated (see definition of array above). For example:

```
CREATE ABUFFER 5 CHARS ALLOT ( allot 5 character buffer)
```

D.2.4   Alignment problems

Some processors have restrictions on the addresses that can be used by memory access instructions. This standard does not require an implementor of a Forth to make alignment transparent; on the contrary, it requires (in Section **3.3.3.1 Address alignment**) that a standard Forth program assume that character and cell alignment may be required. One pitfall caused by alignment restrictions is in creating tables containing both characters and cells. When **,** (comma) or **C,** is used to initialize a table, data are stored at the data-space pointer. Consequently, it must be suitably aligned. For example, a non-portable table definition would be:

```
CREATE ATABLE 1 C, X , 2 C, Y ,
```

On a machine that restricts memory fetches to aligned addresses, **CREATE** would leave the data space pointer at an aligned address. However, the first **C,** would leave the data space pointer at an unaligned address, and the subsequent **,** (comma) would violate the alignment restriction by storing X at an unaligned address. A portable way to create the table is:

```
    CREATE ATABLE 1 C, ALIGN X , 2 C, ALIGN Y ,
```

ALIGN adjusts the data space pointer to the first aligned address greater than or equal to its current address. An aligned address is suitable for storing or fetching characters, cells, cell pairs, or double-cell numbers. After initializing the table, we would also like to read values from the table. For example, assume we want to fetch the first cell, X, from the table. ATABLE **CHAR+** gives the address of the first thing after the character. However this may not be the address of X since we aligned the dictionary pointer between the **C,** and the **,** . The portable way to get the address of X is:

```
    ATABLE CHAR+ ALIGNED
```

ALIGNED adjusts the address on top of the stack to the first aligned address greater than or equal to its current value.

D.3   Number representation

D.3.1   Big endian vs. little endian

The constituent bits of a number in memory are kept in different orders on different machines. Some machines place the most-significant part of a number at an address in memory with less-significant parts following it at higher addresses; this is known as big-endian ording. Other machines do the opposite; the least-significant part is stored at the lowest address (little-endian ordering).

For example, the following code for a 16-bit little endian Forth would produce the answer 1:

```
    VARIABLE FOO   1 FOO !   FOO C@
```

The same code on a 16-bit big-endian Forth would produce the answer 0. A portable program cannot exploit the representation of a number in memory.

A related issue is the representation of cell pairs and double-cell numbers in memory. When a cell pair is moved from the stack to memory with **2!**, the cell that was on top of the stack is placed at the lower memory address. It is useful and reasonable to manipulate the individual cells when they are in memory.

D.3.2   ALU organization

Different computers use different bit patterns to represent integers. Possibilities include binary representations (two's complement, one's complement, sign magnitude, etc.) and decimal representations (BCD, etc.). Each of these formats creates advantages and disadvantages in the design of a computer's arithmetic logic unit (ALU). The most commonly used representation, two's complement, is popular because of the simplicity of its addition and subtraction operations.

Programmers who have grown up on two's complement machines tend to become intimate with their representation of numbers and take some properties of that representation for granted. For example, a trick to find the remainder of a number divided by a power of two is to mask off some bits with **AND**. A common application of this trick is to test a number for oddness using 1 **AND**. However, this will not work on a one's complement machine if the number is negative (a portable technique is 2 **MOD**).

The remainder of this section is a (non-exhaustive) list of things to watch for when portability between machines with binary representations other than two's complement is desired.

To convert a single-cell number to a double-cell number, Forth 94 provided the operator **S>D**. To convert a double-cell number to single-cell, Forth programmers have traditionally used **DROP**. However, this trick doesn't work on sign-magnitude machines. For portability a **D>S** operator is available. Converting an unsigned single-cell number to a double-cell number can be done portably by pushing a zero on the stack.

D.4   Forth system implementation

During Forth's history, an amazing variety of implementation techniques have been developed. The ANS Forth Standard encourages this diversity and consequently restricts the assumptions a user can make about the underlying implementation of an ANS Forth system. Users of a particular Forth implementation frequently become accustomed to aspects of the implementation and assume they are common to all Forths. This section points out many of these incorrect assumptions.

D.4.1   Definitions

Traditionally, Forth definitions have consisted of the name of the Forth word, a dictionary search link, data describing how to execute the definition, and parameters describing the definition itself. These components have historically been referred to as the name, link, code, and parameter fields. No method for accessing these fields has been found that works across all of the Forth implementations currently in use. Therefore, a portable Forth program may not use the name, link, or code field in any way. Use of the parameter field (renamed to data field for clarity) is limited to the operations described below.

Only words defined with **CREATE** or with other defining words that call **CREATE** have data fields. The other defining words in the standard (**VARIABLE**, **CONSTANT**, **:**, etc.) might not be implemented with **CREATE**. Consequently, a Standard Program must assume that words defined by **VARIABLE**, **CONSTANT**, **:**, etc., may have no data fields. There is no portable way for a Standard Program to modify the value of a constant or to "patch" a colon definition at run time. The **DOES>** part of a defining word operates on a data field, so **DOES>** may only be used on words ultimately defined by **CREATE**.

In standard Forth, **FIND**, **[']** and **'** (tick) return an unspecified entity called an execution token. There are only a few things that may be done with an execution token. The token may be passed to **EXECUTE** to execute the word ticked or compiled into the current definition with **COMPILE,**. The token can also be stored in a variable or other data structure and used later. Finally, if the word ticked was defined via **CREATE**, **>BODY** converts the execution token into the word's data-field address.

An execution token cannot be assumed to be an address and may not be used as one.

D.4.2   Stacks

In some Forth implementations, it is possible to find the address of a stack in memory and manipulate the stack as an array of cells. This technique is not portable. On some systems, especially Forth-in-hardware systems, the stacks might be in memory that can't be addressed by the program or might not be in memory at all. Forth's parameter and return stacks must be treated as stacks.

A Standard Program may use the return stack directly only for temporarily storing values. Every value examined or removed from the return stack using **R@**, **R>**, or **2R>** must have been put on the stack explicitly using **>R** or **2>R**. Even this must be done carefully because the system may use the return stack to hold return addresses and loop-control parameters. Section **3.2.3.3 Return stack** of the standard has a list of restrictions.

D.5 Summary

The Forth Standard does not force anyone to write a portable program. In situations where performance is paramount, the programmer is encouraged to use every trick available. On the other hand, if portability to a wide variety of systems is needed(or anticipated), this standard provides the tools to accomplish this. There might be no such thing as a completely portable program. A programmer, using this guide, should intelligently weigh the tradeoffs of providing portability to specific machines. For example, machines that use sign-magnitude numbers are rare and probably don't deserve much thought. But, systems with different cell sizes will certainly be encountered and should be provided for. In general, making a program portable clarifies both the programmer's thinking process and the final program.

Annex E

(informative)

Alphabetic list of words

In the following list, the last, four-digit, part of the reference number establishes a sequence corresponding to the alphabetic ordering of all standard words. The first two or three parts indicate the word set and glossary section in which the word is defined.

6.1.0010	`!`		"store"	CORE	44
6.1.0030	`#`		"number-sign"	CORE	44
6.1.0040	`#>`		"number-sign-greater"	CORE	44
6.1.0050	`#S`		"number-sign-s"	CORE	44
6.1.0070	`'`		"tick"	CORE	45
6.1.0080	`(`		"paren"	CORE	45
11.6.1.0080	`(`		"paren"	FILE	117
13.6.1.0086	`(LOCAL)`		"paren-local-paren"	LOCAL	149
6.1.0090	`*`		"star"	CORE	45
6.1.0100	`*/`		"star-slash"	CORE	45
6.1.0110	`*/MOD`		"star-slash-mod"	CORE	45
6.1.0120	`+`		"plus"	CORE	46
6.1.0130	`+!`		"plus-store"	CORE	46
10.6.2.0135	`+FIELD`		"plus-field"	FACILITY EXT	105
6.1.0140	`+LOOP`		"plus-loop"	CORE	46
18.6.2.0145	`+X/STRING`		"plus-x-string"	XCHAR EXT	179
6.1.0150	`,`		"comma"	CORE	47
6.1.0160	`−`		"minus"	CORE	47
17.6.1.0170	`−TRAILING`		"dash-trailing"	STRING	171
18.6.2.0175	`−TRAILING−GARBAGE`		"minus-trailing-garbage"	XCHAR EXT	180
6.1.0180	`.`		"dot"	CORE	47
6.1.0190	`."`		"dot-quote"	CORE	47
6.2.0200	`.(`		"dot-paren"	CORE EXT	73
6.2.0210	`.R`		"dot-r"	CORE EXT	73
15.6.1.0220	`.S`		"dot-s"	TOOLS	157
6.1.0230	`/`		"slash"	CORE	47
6.1.0240	`/MOD`		"slash-mod"	CORE	48
17.6.1.0245	`/STRING`		"slash-string"	STRING	171
6.1.0250	`0<`		"zero-less"	CORE	48
6.2.0260	`0<>`		"zero-not-equals"	CORE EXT	73
6.1.0270	`0=`		"zero-equals"	CORE	48
6.2.0280	`0>`		"zero-greater"	CORE EXT	73
6.1.0290	`1+`		"one-plus"	CORE	48
6.1.0300	`1−`		"one-minus"	CORE	48
6.1.0310	`2!`		"two-store"	CORE	48
6.1.0320	`2*`		"two-star"	CORE	49
6.1.0330	`2/`		"two-slash"	CORE	49