Forth Standards Committee

Forth 200*x* *Draft 21.1*

9th January, 2025

Notice: ***Status of this Document***
This is a draft proposed Standard to replace the Forth 2012 standard. As such, this is not a completed standard. The Forth Standards 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 Standard was published in 1994[1] and was 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. Their work led to the Forth-2012 Standard[3].

The Forth Standards Committee has taken over this work. Changes are proposed and discussed on the forth200x@yahoogroups.com email list and the www.forth-standard.org web site. Annual public meetings are held to review and vote on the proposed changes. This document is the result of these meetings first held on 30 September – 2 October, 2015 (Bath), and subsequently on 7–9 September 2016 (Konstanz), 6–8 September 2017 (Bad Vöslau, Austria) 12–14 September 2018 (Queensferry, Scotland), 11–13 September 2019 (Hamburg, Germany) and 1–3 September 2020 (Online). ed21

[1] ANS X3.215–1994 Information Systems — Programming Language FORTH
[2] ISO/IEC 15145:1997 Information technology. Programming languages. FORTH
[3] www.forth-standard.org

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, see Appendix F.

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 attend a physical meeting as an observer.

Committee membership is open to anybody who can attend physical meetings. On attending a physical meeting, a non-member becomes an observer. An observer becomes a voting member by attending the next physical meeting. An observer will not normally be allowed to vote except at the discretion of the committee. A member will be deemed to have resigned from the committee by failing to attend two consecutive physical meetings.

Electronic meetings of the committee are held between physical meetings as required.

Currently the committee has the following voting members:

~~Paul E. Bennet~~ ~~Paul_E.Bennett@topmail.co.uk~~	~~Independent Member~~ ~~Exeter, UK~~
Willem Botha willem.botha@ccssa.com	Construction Computer Software (Pty) Ltd Cape Town, South Africa
Jermaine Davies jermaine.davies@ccssa.com	Construction Computer Software (Pty) Ltd Cape Town, South Africa
Dr. M. Anton Ertl anton@mips.complang.tuwien.at	Technische Universität Wien Wien, Austria
Andrew Haley aph@redhat.com	Red Hat UK Ltd. Cambridge, UK
Dr. Ulrich Hoffmann uho@forth-ev.de	FH Wedel Wedel, Germany
Dr. Peter Knaggs (Editor) pjk@bcs.org.uk	Independent Member Trowbridge, UK
Krishna Myneni krishna.myneni@ccreweb.org	IndependentMember Huntsville, AL, USA
Howerd Oakford howerd@inventio.co.uk	Inventio Software Ltd Wolfsburg, Germany
Bernd Paysan (Treasurer) bernd.paysan@gmx.de	Net2o Munich, Germany
Stephen Pelc (Chair) stephen@mpeforth.com	MicroProcessor Engineering Ltd. Southampton, UK
Leon Wagner leon@forth.com	FORTH, Inc. Los Angeles, USA
Gerald Wodni (Technical) gerald@wodni.at	Independent Member Wien, Austria

ed25

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

Paul E. Bennet Paul_E.Bennett@topmail.co.uk	Independent Member Exeter, UK
Prof. Sergey Baranov snbaranov@googlemail.com	SPIIRAS St. Petersburg, Russia
Federico de Ceballos federico.ceballos@unican.es	Universidad de Cantabria Santander, Spain
Simon Kaphahn Simon_K99@gmx.de	Independent Member Munich, Germany
Dr. Bill Stoddart w.j.stoddart@gmail.com	Teesside University Middlesbrough, UK
Dr. Willi Stricker stricker_w@t-online.de	Independent Member Springe, Germany
Carsten Strotmann carsten@strotmann.de	Independent Member Neuenkirchen, Germany

ed25

The committee would like to thank the following contributors:

John Hayes
Marcel Hendrix
Gerry Jackson
Alex McDonald
Bruce McFarling
Charles G. Montgomery
Krishna Myneni
Tim Partridge
Elizabeth Rather
David N. Williams

Forth 200*x* 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 F presents a test suite to test the operation of a system complies with the definitions documented in this standard.

Annex H 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 4.1.2Ambiguous conditions for a list of such circumstances and 3.4.4Possible 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 (see 16.2). x:rules-of-find

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 3.4.3Semantics.

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: 3.4.1.3Text interpreter input number conversion, 12.3.7Text 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*, (2.1Definitions of terms);
- ANSI X3.4-1974 *American Standard Code for Information Interchange* (ASCII), (3.1.2.1Graphic characters);
- ISO 646-1983 *ISO 7-bit coded characterset for information interchange, International Reference Version* (IRV) (3.1.2.1Graphic characters);
- ANSI/IEEE 754-1985 *Floating-point Standard*, (12.2.1Definition of terms).

3 Usage requirements

A system shall provide all of the words defined in 6.1Core words and 9The Exception word set. 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* ⇒ *j*" to denote "*i* is a subtype of *j*". The subtype relationship is transitive; if *i* ⇒ *j* and *j* ⇒ *k* then *i* ⇒ *k*:

+n ⇒ *u* ⇒ *x*;
+n ⇒ *n* ⇒ *x*;
char ⇒ *+n*;
a-addr ⇒ *c-addr* ⇒ *addr* ⇒ *u*;
flag ⇒ *x*;
xt ⇒ *x*;
ior ⇒ *n* ⇒ *x*;
+d ⇒ *d* ⇒ *xd*;
+d ⇒ *ud* ⇒ *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 exactly 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.2Hardware 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*. 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 {-32768 ... +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 3.3.3Data 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 3.2.1.1Internal 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 requires two's-complement number representation 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 3.3.3.6Other 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

Dividend	Divisor	Remainder	Quotient
10	7	3	1
-10	7	4	-2
10	-7	-4	-2
-10	-7	-3	1

Table 3.4: Symmetric Division Example

Dividend	Divisor	Remainder	Quotient
10	7	3	1
-10	7	-3	-1
10	-7	3	-1
-10	-7	-3	1

3.2.2.2 Other integer operations

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

- for unsigned results, the exact result modulo 2^n
- for signed results, with the exact result being r, for operations other than division the number x in the range $-2^{n-1} \leq x < 2^{n-1}$ that satisfies x congruent r (mod 2^n).

where n is the number of bits in the result.

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 1.2.2Exclusions.

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?.

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

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 X:wordset-query

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.3.2.7Obsolescent Environmental Queries.

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

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.

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. Defining a name longer than the implementation defined limit will throw a -19 (definition name too long) exception.

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.

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 CREATEd 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 ALLOTted 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.

A system must provide at least two transient buffers for use with C", S" and S\" strings. These buffers shall be no less than 80 characters in length. The system should be able to store two strings defined by sequential use of these words. RAM-limited systems may have environmental restrictions on the number of buffers and their lifetimes.

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 7.3.2Block buffer regions and .3.3Input 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 3.3.3.2Contiguous 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, throw an -13 (undefined word) exception.

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. If the number in >IN is greater than the size of the input buffer a -18 (parsed string overflow) exception is thrown.

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*⟩.

⟨*anynum*⟩ := { ⟨*BASEnum*⟩ | ⟨*decnum*⟩ | ⟨*hexnum*⟩ | ⟨*binnum*⟩ | ⟨*cnum*⟩ }
⟨*BASEnum*⟩ := [**-**]⟨*bdigit*⟩⟨*bdigit*⟩*
⟨*decnum*⟩ := **#**[**-**]⟨*decdigit*⟩⟨*decdigit*⟩*
⟨*hexnum*⟩ := **$**[**-**]⟨*hexdigit*⟩⟨*hexdigit*⟩*
⟨*binnum*⟩ := **%**[**-**]⟨*bindigit*⟩⟨*bindigit*⟩*
⟨*cnum*⟩ := **'**⟨*char*⟩**'**
⟨*bindigit*⟩ := { **0** | **1** }
⟨*decdigit*⟩ := { **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** }
⟨*hexdigit*⟩ := { ⟨*decdigit*⟩ | **a** | **b** | **c** | **d** | **e** | **f** | **A** | **B** | **C** | **D** | **E** | **F** }

⟨*bdigit*⟩ represents a digit according to the value of BASE (see 3.2.1.2Digit 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 2.2.5BNF 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 3Usage 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 3.1.3.3Addresses;
- behavior of 6.1.1320 `EMIT` for non-graphic characters;
- character editing of 6.1.0695 `ACCEPT`;
- character set (3.1.2Character types, 6.1.1320 `EMIT`, 6.1.1750 `KEY`);
- character-aligned address requirements (3.1.3.3Addresses);
- character-set-extensions matching characteristics (3.4.2Finding definition names);
- conditions under which control characters match a space delimiter (3.4.1.1Delimiters);
- format of the control-flow stack (3.2.3.2Control-flow stack);
- conversion of digits larger than thirty-five (3.2.1.2Digit conversion);
- display after input terminates in 6.1.0695 `ACCEPT`;
- exception abort sequence (as in 6.1.0680 `ABORT"`);
- input line terminator (3.2.4.1User input device);
- maximum size of a counted string, in characters (3.1.3.4Counted strings, 6.1.2450 `WORD`);
- maximum size of a parsed string (3.4.1Parsing);
- maximum size of a definition name, in characters (3.3.1.2Definition names);
- maximum string length for 6.1.1345 `ENVIRONMENT?`, in characters;
- method of selecting 3.2.4.1User input device;
- method of selecting 3.2.4.2User output device;
- methods of dictionary compilation (3.3The Forth dictionary);
- number of bits in one address unit (3.1.3.3Addresses);
- number representation and arithmetic (3.2.1.1Internal number representation);
- ranges for *n*, *+n*, *u*, *d*, *+d*, and *ud* (3.1.3Single-cell types, 3.1.4Cell-pair types);
- read-only data-space regions (3.3.3Data space);

- size of buffer at 6.1.2450 WORD (3.3.3.6Other transient regions);
- size of one cell in address units (3.1.3Single-cell types);
- size of one character in address units (3.1.2Character types);
- number of string buffers provided (3.3.3.4Text-literal regions);
- size of string buffer used by 3.3.3.4Text-literal regions;
- size of the keyboard terminal input buffer (3.3.3.5Input buffers);
- size of the pictured numeric output string buffer (3.3.3.6Other transient regions);
- size of the scratch area whose address is returned by 6.2.2000 PAD (3.3.3.6Other transient regions);
- system case-sensitivity characteristics (3.4.2Finding definition names);
- system prompt (3.3The Forth dictionary, 6.1.2050 QUIT);
- type of division rounding (3.2.2.1Integer 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;
- 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 3.4.4Possible actions on an ambiguous condition.

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

- addressing a region not listed in 3.3.3Data space;
- argument type incompatible with specified input parameter, (3.1Data 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;
- interpreting a word with undefined interpretation semantics;
- modifying the contents of the input buffer or a string literal (3.3.3.4Text-literal regions, 3.3.3.5Input 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 (3.4.1Parsing);
- 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 (3.3.3.2Contiguous regions);
- data space read/write with incorrect alignment (3.3.3.1Address 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 (3.3.3.6Other 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 (3.3.3.6Other 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 (3.2.2.1Integer division);
- requiring a particular number representation and arithmetic (3.2.1.1Internal number representation);
- requiring non-standard words or techniques (3Usage requirements);
- requiring the ability to send or receive control characters (3.1.2.2Control 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 (3.1.3.1Flags);
- using lower case for standard definition names or depending on the case sensitivity of a system (3.3.1.2Definition names);
- using definition names of more than 31 characters in length (3.3.1.2Definition names);
- using the graphic character with a value of hex 24 (3.1.2.1Graphic 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 3Usage requirements and 4.1System 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 3Usage 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 3Usage requirements and 4.2Program 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 also: 3.3.3.1 Address alignment (page 29), F.6.1.0010 ! (page 283).

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 also: 6.1.0040 #> (page 40), 6.1.0050 #S (page 40), 6.1.0490 <# (page 47), F.6.1.0030 # (page 284).

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 also: 6.1.0030 # (page 40), 6.1.0050 #S (page 40), 6.1.0490 <# (page 47), F.6.1.0040 #> (page 284).

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 also: 6.1.0030 # (page 40), 6.1.0040 #> (page 40), 6.1.0490 <# (page 47), F.6.1.0050 #S (page 284).

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 also: 3.4.1 Parsing (page 31), 3.4.3.2 Interpretation semantics (page 33), A.6.1.0070 `'` (page 193), A.6.1.2033 `POSTPONE` (page 199), A.6.1.2510 `[']` (page 202), F.6.1.0070 `'` (page 284).

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 also: 3.4.1 Parsing (page 31), 11.6.1.0080 `(` (page 112), A.6.1.0080 `(` (page 193), F.6.1.0080 `(` (page 285).

6.1.0090 ***** "star" CORE

(n_1 | u_1 n_2 | u_2 -- n_3 | u_3)

Multiply n_1 | u_1 by n_2 | u_2 giving the product n_3 | u_3.

See also: F.6.1.0090 `*` (page 285).

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 also: 3.2.2.1 Integer division (page 25), F.6.1.0100 `*/` (page 285).

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 also: 3.2.2.1 Integer division (page 25), F.6.1.0110 `*/MOD` (page 286).

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 also: 3.3.3.1 Address alignment (page 29), F.6.1.0120 + (page 286).

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

(n | u *a-addr* --)

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

See also: 3.3.3.1 Address alignment (page 29), F.6.1.0130 +! (page 286).

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: $loop\text{-}sys_1$ -- | $loop\text{-}sys_2$)

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 also: 6.1.1240 `DO` (page 54), 6.1.1680 `I` (page 58), 6.1.1760 `LEAVE` (page 59), A.6.1.0140 `+LOOP` (page 193), F.6.1.0140 `+LOOP` (page 286).

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 also: 3.3.3 Data space (page 29), 3.3.3.1 Address alignment (page 29), A.6.1.0150 **,** (page 193), F.6.1.0150 **,** (page 288).

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 also: 3.3.3.1 Address alignment (page 29), F.6.1.0160 - (page 288).

6.1.0180 **.** "dot" CORE

(*n* --)

Display *n* in free field format.

See also: 3.2.1.2 Digit conversion (page 25), 3.2.1.3 Free-field number display (page 25), F.6.1.0180 **.** (page 288).

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 also: 3.4.1 Parsing (page 31), 6.2.0200 **.(** (page 70), A.6.1.0190 **."** (page 193), F.6.1.0190 **."** (page 288).

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 also: 3.2.2.1 Integer division (page 25), F.6.1.0230 / (page 288).

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 also: 3.2.2.1 Integer division (page 25), F.6.1.0240 `/MOD` (page 289).

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

(*n* -- *flag*)

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

See also: F.6.1.0250 `0<` (page 289).

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

(*x* -- *flag*)

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

See also: F.6.1.0270 `0=` (page 290).

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.

See also: F.6.1.0290 `1+` (page 290).

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.

See also: F.6.1.0300 `1-` (page 290).

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 also: 3.3.3.1 Address alignment (page 29), F.6.1.0310 `2!` (page 290).

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.

See also: F.6.1.0320 2* (page 290).

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.

See also: F.6.1.0330 2/ (page 290).

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 @.

2!.

See also: 3.3.3.1 Address alignment (page 29), 6.1.0310 2! (page 44), F.6.1.0350 2@ (page 290) .

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

(x_1 x_2 --)

Drop cell pair x_1 x_2 from the stack.

See also: F.6.1.0370 2DROP (page 290).

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.

See also: F.6.1.0380 2DUP (page 291).

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.

See also: F.6.1.0400 2OVER (page 291).

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.

See also: F.6.1.0430 2SWAP (page 291).

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. x:revise-colon

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 also: 3.4.1 Parsing (page 31), 3.4.3.2 Interpretation semantics (page 33), 3.4.5 Compilation (page 34), 6.1.1250 DOES> (page 54), 6.1.2500 [(page 69), 6.1.2540] (page 70), 15.6.2.0470 ;CODE (page 151), A.6.1.0450 : (page 194), F.6.1.0450 : (page 291).

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.

An ambiguous condition exists if the compilation semantics of ; are preformed inside a quotation ([: ... ;] block).

Run-time: (--) (R: *nest-sys* --)

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

See also: 3.4 The Forth text interpreter (page 31), 3.4.5 Compilation (page 34), A.6.1.0460 ; (page 194), F.6.1.0460 ; (page 291).

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

(n_1 n_2 -- *flag*)

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

See also: 6.1.2340 U< (page 66), F.6.1.0480 < (page 291).

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

(--)

Initialize the pictured numeric output conversion process.

See also: 6.1.0030 # (page 40), 6.1.0040 #> (page 40), 6.1.0050 #S (page 40), F.6.1.0490 <# (page 291).

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.

See also: F.6.1.0530 = (page 291).

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 also: 6.2.2350 U> (page 81), F.6.1.0540 > (page 292).

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 also: 3.3.3 Data space (page 29), A.6.1.0550 >BODY (page 194), F.6.1.0550 >BODY (page 292).

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.

See also: F.6.1.0560 >IN (page 292).

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 multiplying 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 unconverted characters in the string. An ambiguous condition exists if ud_2 overflows during the conversion.

See also: 3.2.1.2 Digit conversion (page 25), F.6.1.0570 >NUMBER (page 292).

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 also: 3.2.3.3 Return stack (page 26), 6.1.2060 R> (page 62), 6.1.2070 R@ (page 62), 6.2.0340 2>R (page 70), 6.2.0410 2R> (page 71), 6.2.0415 2R@ (page 71), F.6.1.0580 >R (page 293).

6.1.0630 **?DUP** “question-dupe” CORE

(*x* -- *0* | *x x*)

Duplicate *x* if it is non-zero.

See also: F.6.1.0630 ?DUP (page 293).

6.1.0650 **@** “fetch” CORE

(*a-addr* -- *x*)

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

See also: 3.3.3.1 Address alignment (page 29), F.6.1.0650 @ (page 293).

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 also: 9.6.2.0670 ABORT (page 98).

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 also: 3.4.1 Parsing (page 31), 9.6.2.0680 ABORT" (page 98), A.6.1.0680 ABORT" (page 194).

6.1.0690 ABS “abs” CORE

(*n* –– *u*)

u is the absolute value of *n*.

See also: F.6.1.0690 ABS (page 294).

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 also: A.6.1.0695 ACCEPT (page 194), F.6.1.0695 ACCEPT (page 294).

6.1.0705 ALIGN CORE

(––)

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

See also: 3.3.3 Data space (page 29), 3.3.3.1 Address alignment (page 29), A.6.1.0705 ALIGN (page 194), F.6.1.0705 ALIGN (page 294).

6.1.0706 ALIGNED CORE

(*addr* -- *a-addr*)

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

See also: 3.3.3.1 Address alignment (page 29), 6.1.0705 ALIGN (page 49).

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 also: 3.3.3 Data space (page 29), F.6.1.0710 ALLOT (page 294).

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.

See also: F.6.1.0720 AND (page 294).

6.1.0750 BASE CORE

(-- *a-addr*)

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

See also: F.6.1.0750 BASE (page 295).

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 also: 3.2.3.2 Control-flow stack (page 26), 6.1.2140 REPEAT (page 63), 6.1.2390 UNTIL (page 67), 6.1.2430 WHILE (page 68), A.6.1.0760 BEGIN (page 195), F.6.1.0760 BEGIN (page 295).

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

(-- *char*)

char is the character value for a space.

See also: A.6.1.0770 BL (page 195), F.6.1.0770 BL (page 295).

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 also: 3.3.3.1 Address alignment (page 29), F.6.1.0850 C! (page 295).

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 also: 3.3.3 Data space (page 29), 3.3.3.1 Address alignment (page 29), F.6.1.0860 C, (page 295).

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 also: 3.3.3.1 Address alignment (page 29), F.6.1.0870 C@ (page 295).

6.1.0880 **CELL+** "cell-plus" CORE

($a\text{-}addr_1$ -- $a\text{-}addr_2$)

Add the size in address units of a cell to $a\text{-}addr_1$, giving $a\text{-}addr_2$.

See also: 3.3.3.1 Address alignment (page 29), A.6.1.0880 CELL+ (page 195), F.6.1.0880 CELL+ (page 295).

6.1.0890 CELLS CORE

(n_1 -- n_2)

n_2 is the size in address units of n_1 cells.

See also: A.6.1.0880 CELL+ (page 195), A.6.1.0890 CELLS (page 195), F.6.1.0890 CELLS (page 295).

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 also: 3.4.1 Parsing (page 31), 6.1.2520 [CHAR] (page 69), A.6.1.0895 CHAR (page 195), F.6.1.0895 CHAR (page 296).

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 also: 3.3.3.1 Address alignment (page 29), F.6.1.0897 CHAR+ (page 296).

6.1.0898 CHARS "chars" CORE

(n_1 -- n_2)

n_2 is the size in address units of n_1 characters.

See also: F.6.1.0898 CHARS (page 296).

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 also: 3.4.1 Parsing (page 31), A.6.1.0950 CONSTANT (page 195), F.6.1.0950 CONSTANT (page 296).

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*$_2$.

See also: F.6.1.0980 COUNT (page 296).

6.1.0990 CR "c-r" CORE

(--)

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

See also: F.6.1.0990 CR (page 296).

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 also: 3.3.3 Data space (page 29), 6.1.1250 DOES> (page 54), A.6.1.1000 CREATE (page 195), F.6.1.1000 CREATE (page 296).

6.1.1170 DECIMAL CORE

(--)

Set the numeric conversion radix to ten (decimal).

See also: F.6.1.1170 DECIMAL (page 296).

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.

See also: F.6.1.1200 DEPTH (page 296).

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 | u_2 and limit n_1 | u_1. An ambiguous condition exists if n_1 | u_1 and n_2 | u_2 are not both the same type. Anything already on the return stack becomes unavailable until the loop-control parameters are discarded.

See also: 3.2.3.2 Control-flow stack (page 26), 6.1.0140 +LOOP (page 42), 6.1.1800 LOOP (page 60), A.6.1.1240 DO (page 196), F.6.1.1240 DO (page 296).

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.

Quotation: The DOES> part inside a quotation end with the terminating ;].

See also: 6.1.1000 CREATE (page 53), 15.6.2.0 ;] (page 152), A.6.1.1250 DOES> (page 196), F.6.1.1250 DOES> (page 296).

6.1.1260 **DROP** CORE

(*x* ––)

Remove *x* from the stack.

See also: F.6.1.1260 DROP (page 297).

6.1.1290 **DUP** "dupe" CORE

(*x* –– *x x*)

Duplicate *x*.

See also: F.6.1.1290 DUP (page 297).

6.1.1310 **ELSE** CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: $orig_1$ –– $orig_2$)

Put the location of a new unresolved forward reference $orig_2$ onto the control flow stack. Append the run-time semantics given below to the current definition. The semantics will be incomplete until $orig_2$ is resolved (e.g., by THEN). Resolve the forward reference $orig_1$ using the location following the appended run-time semantics.

Run-time: (––)

Continue execution at the location given by the resolution of $orig_2$.

See also: 6.1.1700 IF (page 58), 6.1.2270 THEN (page 66), A.6.1.1310 ELSE (page 196), F.6.1.1310 ELSE (page 297).

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 3.1.2.1Graphic 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 also: 6.1.2310 TYPE (page 66), F.6.1.1320 EMIT (page 297).

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 3.2.6Environmental 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 also: A.6.1.1345 ENVIRONMENT? (page 196), F.6.1.1345 ENVIRONMENT? (page 298).

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 EVALUATEd.

See also: F.6.1.1360 EVALUATE (page 298).

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 EXECUTEd.

See also: 6.1.0070 ' (page 41), 6.1.2510 ['] (page 69), F.6.1.1370 EXECUTE (page 298).

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 also: 3.2.3.3 Return stack (page 26), 6.1.2380 UNLOOP (page 67), A.6.1.1380 EXIT (page 196), F.6.1.1380 EXIT (page 298).

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*.

See also: F.6.1.1540 FILL (page 298).

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 also: 3.4.2 Finding definition names (page 32), 15.6.2.2530.30 [DEFINED] (page 158), 16.6.1.1550 FIND (page 162), A.6.1.0070 ' (page 193), A.6.1.1550 FIND (page 196), A.6.1.2033 POSTPONE (page 199), A.6.1.2510 ['] (page 202), F.6.1.1550 FIND (page 298). x:rule-of-find

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 also: 3.2.2.1 Integer division (page 25), 6.1.2214 SM/REM (page 64), 6.1.2370 UM/MOD (page 67), A.6.1.1561 FM/MOD (page 197), F.6.1.1561 FM/MOD (page 298).

6.1.1650 HERE CORE

(–– *addr*)

addr is the data-space pointer.

See also: 3.3.3.2 Contiguous regions (page 29), F.6.1.1650 HERE (page 299).

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.

See also: F.6.1.1670 HOLD (page 299).

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.

See also: F.6.1.1680 I (page 299).

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 also: 3.2.3.2 Control-flow stack (page 26), 6.1.1310 ELSE (page 55), 6.1.2270 THEN (page 66), A.6.1.1700 IF (page 197), F.6.1.1700 IF (page 299).

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 also: 15.6.2.2264 SYNONYM (page 156), A.6.1.1710 IMMEDIATE (page 197), F.6.1.1710 IMMEDIATE (page 300).

6.1.1720 INVERT CORE

(x_1 -- x_2)

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

See also: 6.1.0270 0= (page 44), 6.1.1910 NEGATE (page 61), A.6.1.1720 INVERT (page 197), F.6.1.1720 INVERT (page 300).

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 also: A.6.1.1730 J (page 198), F.6.1.1730 J (page 300).

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 3.1.2.1Graphic characters. Programs that require the ability to receive control characters have an environmental dependency.

See also: 10.6.1.1755 KEY? (page 100), 10.6.2.1305 EKEY (page 102), 10.6.2.1307 EKEY? (page 102), A.6.1.1750 KEY (page 198).

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 also: 3.2.3.3 Return stack (page 26), 6.1.0140 +LOOP (page 42), 6.1.1800 LOOP (page 60), A.6.1.1760 LEAVE (page 198), F.6.1.1760 LEAVE (page 300).

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 also: A.6.1.1780 LITERAL (page 198), F.6.1.1780 LITERAL (page 301).

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-sys*$_1$ –– | *loop-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 also: 6.1.1240 DO (page 54), 6.1.1680 I (page 58), 6.1.1760 LEAVE (page 59), A.6.1.1800 LOOP (page 198), F.6.1.1800 LOOP (page 301).

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.

See also: F.6.1.1805 LSHIFT (page 301).

6.1.1810 M* "m-star" CORE

(n_1 n_2 –– d)

d is the signed product of n_1 times n_2.

See also: A.6.1.1810 M* (page 198), F.6.1.1810 M* (page 301).

6.1.1870 MAX CORE

(n_1 n_2 –– n_3)

n_3 is the greater of n_1 and n_2.

See also: F.6.1.1870 MAX (page 301).

6.1.1880 MIN CORE

(n_1 n_2 –– n_3)

n_3 is the lesser of n_1 and n_2.

See also: F.6.1.1880 MIN (page 302).

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 also: 3.2.2.1 Integer division (page 25), F.6.1.1890 MOD (page 302).

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 also: 17.6.1.0910 CMOVE (page 167), 17.6.1.0920 CMOVE> (page 168), A.6.1.1900 MOVE (page 199), F.6.1.1900 MOVE (page 303).

6.1.1910 NEGATE CORE

(n_1 -- n_2)

Negate n_1, giving its arithmetic inverse n_2.

See also: 6.1.0270 0= (page 44), 6.1.1720 INVERT (page 58), E.6.1.1910 NEGATE (page 244), F.6.1.1910 NEGATE (page 303).

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.

See also: F.6.1.1980 OR (page 303).

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.

See also: F.6.1.1990 OVER (page 303).

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 also: 3.4.1 Parsing (page 31), A.6.1.2033 POSTPONE (page 199), F.6.1.2033 POSTPONE (page 304).

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 also: 3.4 The Forth text interpreter (page 31), E.6.1.2050 QUIT (page 244).

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 also: 3.2.3.3 Return stack (page 26), 6.1.0580 >R (page 48), 6.1.2070 R@ (page 62), 6.2.0340 2>R (page 70), 6.2.0410 2R> (page 71), 6.2.0415 2R@ (page 71), F.6.1.2060 R> (page 304).

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 also: 3.2.3.3 Return stack (page 26), 6.1.0580 >R (page 48), 6.1.2060 R> (page 62), 6.2.0340 2>R (page 70), 6.2.0410 2R> (page 71), 6.2.0415 2R@ (page 71), F.6.1.2070 R@ (page 304).

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>.

If inside a quotation, calls the quotation directly surrounding the RECURSE, not the containing colon definition.

See also: 6.1.1250 DOES> (page 54), 6.1.2120 RECURSE (page 63), 15.6.2.0 [: (page 157), A.6.1.2120 RECURSE (page 199), F.6.1.2120 RECURSE (page 304).

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 also: 6.1.0760 BEGIN (page 50), 6.1.2430 WHILE (page 68), A.6.1.2140 REPEAT (page 200), F.6.1.2140 REPEAT (page 304).

6.1.2160 **ROT** “rote” CORE

(x_1 x_2 x_3 –– x_2 x_3 x_1)

Rotate the top three stack entries.

See also: F.6.1.2160 ROT (page 305).

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.

See also: F.6.1.2162 RSHIFT (page 305).

6.1.2165 S" "s-quote" CORE

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* describing a string consisting of the characters *ccc*. A program shall not alter the returned string.

See also: 3.3.3.4 Text-literal regions (page 30), 3.4.1 Parsing (page 31), 6.2.0855 C" (page 73), 6.2.2266 S\" (page 79), A.6.1.2165 S" (page 200), F.6.1.2165 S" (page 305).

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.

See also: F.6.1.2170 S>D (page 305).

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.

See also: F.6.1.2210 SIGN (page 305).

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

(*d_1 n_1 -- n_2 n_3*)

Divide d_1 by n_1, giving the symmetric 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 also: 3.2.2.1 Integer division (page 25), 6.1.1561 FM/MOD (page 57), 6.1.2370 UM/MOD (page 67), A.6.1.2214 SM/REM (page 200), F.6.1.2214 SM/REM (page 305).

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 also: A.6.1.2216 SOURCE (page 200), F.6.1.2216 SOURCE (page 306).

6.1.2220 SPACE CORE

(--)

Display one space.

See also: F.6.1.2220 SPACE (page 306).

6.1.2230 SPACES CORE

(*n* --)

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

See also: F.6.1.2230 SPACES (page 306).

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 also: 3.4 The Forth text interpreter (page 31), 6.1.0450 : (page 46), 6.1.0460 ; (page 46), 6.1.0670 ABORT (page 48), 6.1.2050 QUIT (page 62), 6.1.2500 [(page 69), 6.1.2540] (page 70), 6.2.0455 :NONAME (page 71), 15.6.2.2250 STATE (page 156), A.6.1.2250 STATE (page 201), F.6.1.2250 STATE (page 306).

6.1.2260 SWAP CORE

(*x1 x2* -- *x2 x1*)

Exchange the top two stack items.

See also: F.6.1.2260 SWAP (page 306).

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 also: 6.1.1310 ELSE (page 55), 6.1.1700 IF (page 58), A.6.1.2270 THEN (page 201), F.6.1.2270 THEN (page 306).

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 3.1.2.1Graphic 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 also: 6.1.1320 EMIT (page 55), F.6.1.2310 TYPE (page 306).

6.1.2320 U. "u-dot" CORE

(*u* ––)

Display *u* in free field format.

See also: F.6.1.2320 U. (page 307).

6.1.2340 U< "u-less-than" CORE

(u_1 u_2 –– *flag*)

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

See also: 6.1.0480 < (page 47), F.6.1.2340 U< (page 307).

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

(u_1 u_2 -- ud)

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

See also: F.6.1.2360 UM* (page 307).

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

(ud u_1 -- u_2 u_3)

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 also: 3.2.2.1 Integer division (page 25), 6.1.1561 FM/MOD (page 57), 6.1.2214 SM/REM (page 64), F.6.1.2370 UM/MOD (page 307).

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 EXITed. An ambiguous condition exists if the loop-control parameters are unavailable.

See also: 3.2.3.3 Return stack (page 26), A.6.1.2380 UNLOOP (page 201), F.6.1.2380 UNLOOP (page 307).

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 also: 6.1.0760 BEGIN (page 50), A.6.1.2390 UNTIL (page 201), F.6.1.2390 UNTIL (page 308).

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 also: 3.4.1 Parsing (page 31), A.6.1.2410 VARIABLE (page 201), F.6.1.2410 VARIABLE (page 308).

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 also: A.6.1.2430 WHILE (page 202), F.6.1.2430 WHILE (page 308).

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 also: 3.3.3.6 Other transient regions (page 30), 3.4.1 Parsing (page 31), A.6.1.2450 WORD (page 202), F.6.1.2450 WORD (page 308).

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.

See also: F.6.1.2490 XOR (page 308).

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 also: 3.4 The Forth text interpreter (page 31), 3.4.5 Compilation (page 34), 6.1.2540] (page 70), A.6.1.2500 [(page 202), F.6.1.2500 [(page 308).

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 compiled phrase "['] X" is the same value returned by "' X" outside of compilation state.

See also: 3.4.1 Parsing (page 31), 6.1.1550 FIND (page 57), A.6.1.0070 ' (page 193), A.6.1.2033 POSTPONE (page 199), A.6.1.2510 ['] (page 202), F.6.1.2510 ['] (page 309).

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 also: 3.4.1 Parsing (page 31), 6.1.0895 CHAR (page 52), A.6.1.2520 [CHAR] (page 202), F.6.1.2520 [CHAR] (page 309).

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

(--)

Enter compilation state.

See also: 3.4 The Forth text interpreter (page 31), 3.4.5 Compilation (page 34), 6.1.2500 [(page 69), A.6.1.2540] (page 202), F.6.1.2540] (page 309).

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 also: 3.4.1 Parsing (page 31), 6.1.0190 ." (page 43), A.6.2.0200 .((page 202).

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 also: A.6.2.0210 .R (page 202).

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 also: 3.2.3.3 Return stack (page 26), 6.1.0580 >R (page 48), 6.1.2060 R> (page 62), 6.1.2070 R@ (page 62), 6.2.0410 2R> (page 71), 6.2.0415 2R@ (page 71), A.6.2.0340 2>R (page 202).

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 also: 3.2.3.3 Return stack (page 26), 6.1.0580 >R (page 48), 6.1.2060 R> (page 62), 6.1.2070 R@ (page 62), 6.2.0340 2>R (page 70), 6.2.0415 2R@ (page 71), A.6.2.0410 2R> (page 202).

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 also: 3.2.3.3 Return stack (page 26), 6.1.0580 >R (page 48), 6.1.2060 R> (page 62), 6.1.2070 R@ (page 62), 6.2.0340 2>R (page 70), 6.2.0410 2R> (page 71).

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 3.2.3.2Control-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 also: A.6.2.0455 :NONAME (page 203), F.6.2.0455 :NONAME (page 309).

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 also: 3.2.3.2 Control-flow stack (page 26), 6.1.0140 +LOOP (page 42), 6.1.1240 DO (page 54), 6.1.1680 I (page 58), 6.1.1760 LEAVE (page 59), 6.1.1800 LOOP (page 60), 6.1.2380 UNLOOP (page 67), A.6.2.0620 ?DO (page 203), F.6.2.0620 ?DO (page 309).

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 also: 6.2.1173 DEFER (page 74), 6.2.1175 DEFER! (page 74), 6.2.1177 DEFER@ (page 75), 6.2.1725 IS (page 76), E.6.2.0698 ACTION-OF (page 245), F.6.2.0698 ACTION-OF (page 310).

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 also: 6.1.0760 BEGIN (page 50), A.6.2.0700 AGAIN (page 203).

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 also: A.6.2.0825 BUFFER: (page 203), E.6.2.0825 BUFFER: (page 245), F.6.2.0825 BUFFER: (page 310).

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 also: 3.3.3.4 Text-literal regions (page 30), 3.4.1 Parsing (page 31), 6.1.2165 S" (page 64), 6.2.2266 S\" (page 79), A.6.2.0855 C" (page 203), F.6.2.0855 C" (page 311).

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 also: 6.2.1342 ENDCASE (page 75), 6.2.1343 ENDOF (page 75), 6.2.1950 OF (page 77), A.6.2.0873 CASE (page 203), F.6.2.0873 CASE (page 311).

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 also: A.6.2.0945 COMPILE, (page 204), F.6.2.0945 COMPILE, (page 312).

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 also: 6.2.0698 ACTION-OF (page 72), 6.2.1175 DEFER! (page 74), 6.2.1177 DEFER@ (page 75), 6.2.1725 IS (page 76), E.6.2.1173 DEFER (page 245), F.6.2.1173 DEFER (page 312).

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 also: 6.2.0698 ACTION-OF (page 72), 6.2.1173 DEFER (page 74), 6.2.1177 DEFER@ (page 75), 6.2.1725 IS (page 76), E.6.2.1175 DEFER! (page 245), F.6.2.1175 DEFER! (page 312).

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 also: 6.2.0698 ACTION-OF (page 72), 6.2.1173 DEFER (page 74), 6.2.1175 DEFER! (page 74), 6.2.1725 IS (page 76), E.6.2.1177 DEFER@ (page 245), F.6.2.1177 DEFER@ (page 312).

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 also: 6.2.0873 CASE (page 74), 6.2.1343 ENDOF (page 75), 6.2.1950 OF (page 77), A.6.2.1342 ENDCASE (page 204), F.6.2.1342 ENDCASE (page 312).

6.2.1343 ENDOF "end-of" CORE EXT

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: $case\text{-}sys_1$ *of-sys* -- $case\text{-}sys_2$)

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 $case\text{-}sys_1$ with $case\text{-}sys_2$ on the control-flow stack, to be resolved by ENDCASE.

Run-time: (--)

Continue execution at the location specified by the consumer of $case\text{-}sys_2$.

See also: 6.2.0873 CASE (page 74), 6.2.1342 ENDCASE (page 75), 6.2.1950 OF (page 77), A.6.2.1343 ENDOF (page 204), F.6.2.1343 ENDOF (page 312).

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 also: 3.1.3.1 Flags (page 23), F.6.2.1485 FALSE (page 313).

6.2.1660 HEX CORE EXT

(--)

Set contents of BASE to sixteen.

See also: F.6.2.1660 HEX (page 313).

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 also: 6.1.1670 HOLD (page 57), E.6.2.1675 HOLDS (page 245), F.6.2.1675 HOLDS (page 313).

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 also: 6.2.0698 ACTION-OF (page 72), 6.2.1173 DEFER (page 74), 6.2.1175 DEFER! (page 74), 6.2.1177 DEFER@ (page 75), E.6.2.1725 IS (page 245), F.6.2.1725 IS (page 313) .

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 also: 3.4.1 Parsing (page 31), 15.6.2.1580 FORGET (page 155), A.6.2.1850 MARKER (page 205).

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 also: 6.2.0873 CASE (page 74), 6.2.1342 ENDCASE (page 75), 6.2.1343 ENDOF (page 75), A.6.2.1950 OF (page 205), F.6.2.1950 OF (page 313).

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 also: 3.3.3.6 Other transient regions (page 30), A.6.2.2000 PAD (page 205).

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 also: 3.4.1 Parsing (page 31), A.6.2.2008 PARSE (page 205).

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.

See also: E.6.2.2020 PARSE-NAME (page 245), F.6.2.2020 PARSE-NAME (page 313).

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 also: A.6.2.2030 PICK (page 206).

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 also: 7.6.2.2125 REFILL (page 88), 11.6.2.2125 REFILL (page 118), A.6.2.2125 REFILL (page 206).

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 also: 15.6.2.1908 N>R (page 155), 15.6.2.1940 NR> (page 156).

6.2.2150 ROLL CORE EXT

($x_u\ x_{u\text{-}1} \ldots x_0\ u$ -- $x_{u\text{-}1} \ldots x_0\ x_u$)

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 also: A.6.2.2150 ROLL (page 206).

6.2.2266 S\" "s-backslash-quote" CORE EXT

Interpretation: (*"ccc⟨quote⟩"* -- *c-addr u*)

Parse *ccc* delimited by " (double quote) according to the translation rules below. Store the resulting string in a transient buffer *c-addr u*.

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~~ given in table 6.1 , where the character after the backslash is case sensitive. ed21

An ambiguous condition exists if a \ is placed before any character, other than those defined ~~in here~~ in the table .

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 also: 3.3.3.4 Text-literal regions (page 30), 3.4.1 Parsing (page 31), 6.1.2165 S" (page 64), 6.2.0855 C" (page 73), A.6.1.2165 S" (page 200).

Table 6.1: S\" escape sequences

\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
\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.	
\z	NUL	no character	ASCII 0
\"		double-quote	ASCII 34
\\		backslash itself	ASCII 92

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 also: 15.6.2.1908 N>R (page 155), 15.6.2.1940 NR> (page 156), A.6.2.2182 SAVE-INPUT (page 206), F.6.2.2182 SAVE-INPUT (page 313).

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 also: 11.6.1.2218 SOURCE-ID (page 116).

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 also: 6.2.2405 `VALUE` (page 82), 8.6.2.0435 `2VALUE` (page 94), 12.6.2.1628 `FVALUE` (page 135), 13.6.1.0086 `(LOCAL)` (page 142), A.6.2.2295 `TO` (page 208), F.6.2.2295 `TO` (page 315).

6.2.2298 TRUE CORE EXT

(-- *true*)

Return a *true* flag, a single-cell value with all bits set.

See also: 3.1.3.1 Flags (page 23), A.6.2.2298 `TRUE` (page 208), F.6.2.2298 `TRUE` (page 315).

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 -- *flag*)

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

See also: 6.1.0540 > (page 47).

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* "⟨*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 *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 also: 3.4.1 Parsing (page 31), 6.2.2295 TO (page 81), A.6.2.2405 VALUE (page 208), F.6.2.2405 VALUE (page 315).

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 also: A.6.2.2440 WITHIN (page 208), E.6.2.2440 WITHIN (page 246).

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 also: 3.4.1 Parsing (page 31), A.6.2.2530 [COMPILE] (page 209), F.6.2.2530 [COMPILE] (page 315).

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 also: 7.6.2.2535 \ (page 88), A.6.2.2535 \ (page 209).

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: 3.3.3Data 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.7.3.2Block 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: 3.4.1Parsing.

7.3.4 Possible action on an ambiguous condition

See: 3.4.4Possible 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 also: 7.3.2 Block buffer regions (page 84).

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 UPDATEd, 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 UPDATEd, 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 also: 7.6.1.0800 BLOCK (page 86).

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 LOADed.

An ambiguous condition exists if *u* is zero or is not a valid block number.

See also: 3.4 The Forth text interpreter (page 31).

7.6.1.2180 SAVE-BUFFERS BLOCK

(––)

Transfer the contents of each UPDATEd 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 also: 7.6.1.0800 BLOCK (page 86), 7.6.1.0820 BUFFER (page 86), 7.6.1.1559 FLUSH (page 87), 7.6.1.2180 SAVE-BUFFERS (page 87).

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 UPDATEd block buffer to mass storage.

See also: 7.6.1.0800 BLOCK (page 86).

7.6.2.1770 LIST BLOCK EXT

(*u* --)

Display block *u* in an implementation-defined format. Store *u* in SCR.

See also: 7.6.1.0800 BLOCK (page 86).

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 also: 6.2.2125 REFILL (page 78), 11.6.2.2125 REFILL (page 118).

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 LISTed.

See also: A.7.6.2.2190 SCR (page 210).

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 LOADed.

**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: 3.4.1.3Text 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 "⟨*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 "two-constant".

name Execution: (-- x_1 x_2)

Place cell pair x_1 x_2 on the stack.

See also: 3.4.1 Parsing (page 31), A.8.6.1.0360 2CONSTANT (page 210), F.8.6.1.0360 2CONSTANT (page 316).

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 also: A.8.6.1.0390 2LITERAL (page 210), F.8.6.1.0390 2LITERAL (page 316).

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 also: 3.4.1 Parsing (page 31), 6.1.2410 VARIABLE (page 68), A.8.6.1.0440 2VARIABLE (page 210), F.8.6.1.0440 2VARIABLE (page 317).

8.6.1.1040 D+ "d-plus" DOUBLE

(d_1 | ud_1 d_2 | ud_2 -- d_3 | ud_3)

Add d_2 | ud_2 to d_1 | ud_1, giving the sum d_3 | ud_3.

See also: F.8.6.1.1040 D+ (page 317).

8.6.1.1050 D- "d-minus" DOUBLE

(d_1 | ud_1 d_2 | ud_2 -- d_3 | ud_3)

Subtract d_2 | ud_2 from d_1 | ud_1, giving the difference d_3 | ud_3.

See also: F.8.6.1.1050 D- (page 317).

8.6.1.1060 D. "d-dot" DOUBLE

(*d* --)

Display *d* in free field format.

See also: F.8.6.1.1060 D. (page 318).

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 also: A.8.6.1.1070 D.R (page 210), F.8.6.1.1070 D.R (page 318).

8.6.1.1075 D0< "d-zero-less" DOUBLE

(*d* -- *flag*)

flag is true if and only if *d* is less than zero.

See also: F.8.6.1.1075 D0< (page 319).

8.6.1.1080 **D0=** “d-zero-equals” DOUBLE

(*xd* –– *flag*)

flag is true if and only if *xd* is equal to zero.

See also: F.8.6.1.1080 D0= (page 319).

8.6.1.1090 **D2*** “d-two-star” DOUBLE

(xd_1 –– xd_2)

xd_2 is the result of shifting xd_1 one bit toward the most-significant bit, filling the vacated least-significant bit with zero.

See also: F.8.6.1.1090 D2* (page 319).

8.6.1.1100 **D2/** “d-two-slash” DOUBLE

(xd_1 –– xd_2)

xd_2 is the result of shifting xd_1 one bit toward the least-significant bit, leaving the most-significant bit unchanged.

See also: F.8.6.1.1100 D2/ (page 319).

8.6.1.1110 **D<** “d-less-than” DOUBLE

(d_1 d_2 –– *flag*)

flag is true if and only if d_1 is less than d_2.

See also: F.8.6.1.1110 D< (page 319).

8.6.1.1120 **D=** “d-equals” DOUBLE

(xd_1 xd_2 –– *flag*)

flag is true if and only if xd_1 is bit-for-bit the same as xd_2.

See also: F.8.6.1.1120 D= (page 320).

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 also: A.8.6.1.1140 D>S (page 210), E.8.6.1.1140 D>S (page 246), F.8.6.1.1140 D>S (page 320).

8.6.1.1160 **DABS** "d-abs" DOUBLE

(*d* -- *ud*)

ud is the absolute value of *d*.

See also: F.8.6.1.1160 DABS (page 320).

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.

See also: F.8.6.1.1210 DMAX (page 321).

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.

See also: F.8.6.1.1220 DMIN (page 321).

8.6.1.1230 **DNEGATE** "d-negate" DOUBLE

(d_1 -- d_2)

d_2 is the negation of d_1.

See also: F.8.6.1.1230 DNEGATE (page 321).

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 also: A.8.6.1.1820 M*/ (page 210), F.8.6.1.1820 M*/ (page 322).

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 also: A.8.6.1.1830 M+ (page 211), F.8.6.1.1830 M+ (page 322).

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.

See also: F.8.6.2.0420 2ROT (page 322).

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 also: 3.4.1 Parsing (page 31), 6.2.2295 TO (page 81), A.8.6.2.0435 2VALUE (page 211), E.8.6.2.0435 2VALUE (page 246), F.8.6.2.0435 2VALUE (page 322).

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.

See also: F.8.6.2.1270 DU< (page 322).

9 The 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: 3.4.4Possible 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.

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 (9.3.1THROW values, 9.3.4Possible actions on an ambiguous condition).

9.4.1.2 Ambiguous conditions

- no additional requirements.

9.4.1.3 Other system documentation

- no additional requirements.

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-CREATEd 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

9.4.2 Program documentation

– no additional requirements.

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 also: A.9.6.1.2275 THROW (page 212), E.9.6.1.0875 CATCH (page 247), F.9.6.1.0875 CATCH (page 323).

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 also: A.9.6.1.2275 THROW (page 212), E.9.6.1.2275 THROW (page 247), F.9.6.1.2275 THROW (page 323).

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 also: 6.1.0670 ABORT (page 48), E.9.6.2.0670 ABORT (page 248), F.9.6.2.0670 ABORT (page 323).

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 also: 3.4.1 Parsing (page 31), 6.1.0680 ABORT" (page 49), F.9.6.2.0680 ABORT" (page 323) .

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: 3.1.2Character 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 also: A.10.6.1.1755 KEY? (page 212).

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 also: 10.6.2.0763 BEGIN-STRUCTURE (page 101), 10.6.2.0893 CFIELD: (page 101), 10.6.2.1336 END-STRUCTURE (page 103), 10.6.2.1518 FIELD: (page 103), 12.6.2.1207.40 DFFIELD: (page 130), 12.6.2.1517 FFIELD: (page 133), 12.6.2.2206.40 SFFIELD: (page 137), A.10.6.2.0135 +FIELD (page 213), E.10.6.2.0135 +FIELD (page 248).

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 also: 10.6.2.0135 +FIELD (page 101), 10.6.2.1336 END-STRUCTURE (page 103), A.10.6.2.0763 BEGIN-STRUCTURE (page 213), E.10.6.2.0763 BEGIN-STRUCTURE (page 248).

10.6.2.0893 CFIELD: "c-field-colon" FACILITY EXT

(n_1 "⟨spaces⟩name" -- n_2)

Skip leading space delimiters. Parse *name* delimited by a space. *Offset* is the first character aligned value greater than or equal to n_1. n_2 = *offset* + *1* character.

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 also: 10.6.2.0135 +FIELD (page 101), 10.6.2.0763 BEGIN-STRUCTURE (page 101), 10.6.2.1336 END-STRUCTURE (page 103), A.10.6.2.1518 FIELD: (page 216).

10.6.2.1305 EKEY "e-key" FACILITY EXT

(-- *x*)

Receive one keyboard event *x*. The encoding of keyboard events is implementation defined.

See also: 6.1.1750 KEY (page 59), 10.6.1.1755 KEY? (page 100), A.10.6.2.1305 EKEY (page 214).

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 also: A.10.6.2.1306 EKEY>CHAR (page 214).

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 functionality) if these key numbers cannot be produced.

See also: 10.6.2.1305 EKEY (page 102), 10.6.2.1740.01 K-ALT-MASK (page 103), 10.6.2.1740.02 K-CTRL-MASK (page 103), 10.6.2.1740.03 K-DELETE (page 104), 10.6.2.1740.04 K-DOWN (page 104), 10.6.2.1740.05 K-END (page 104), 10.6.2.1740.06 K-F1 (page 104), 10.6.2.1740.07 K-F10 (page 104), 10.6.2.1740.08 K-F11 (page 105), 10.6.2.1740.09 K-F12 (page 105), 10.6.2.1740.10 K-F2 (page 105), 10.6.2.1740.11 K-F3 (page 105), 10.6.2.1740.12 K-F4 (page 105), 10.6.2.1740.13 K-F5 (page 106), 10.6.2.1740.14 K-F6 (page 106), 10.6.2.1740.15 K-F7 (page 106), 10.6.2.1740.16 K-F8 (page 106), 10.6.2.1740.17 K-F9 (page 106), 10.6.2.1740.18 K-HOME (page 107), 10.6.2.1740.19 K-INSERT (page 107), 10.6.2.1740.20 K-LEFT (page 107), 10.6.2.1740.21 K-NEXT (page 107), 10.6.2.1740.22 K-PRIOR (page 107), 10.6.2.1740.23 K-RIGHT (page 108), 10.6.2.1740.24 K-SHIFT-MASK (page 108), 10.6.2.1740.25 K-UP (page 108), A.10.6.2.1306.40 EKEY>FKEY (page 215), E.10.6.2.1306.40 EKEY>FKEY (page 248), F.10.6.2.1306.40 EKEY>FKEY (page 324).

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 also: A.10.6.2.1325 EMIT? (page 216).

10.6.2.1336 END-STRUCTURE FACILITY EXT

(*struct-sys* +*n* --)

Terminate definition of a structure started by BEGIN-STRUCTURE.

See also: 10.6.2.0135 +FIELD (page 101), 10.6.2.0763 BEGIN-STRUCTURE (page 101), E.10.6.2.1336 END-STRUCTURE (page 248).

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 also: 10.6.2.0135 +FIELD (page 101), 10.6.2.0763 BEGIN-STRUCTURE (page 101), 10.6.2.1336 END-STRUCTURE (page 103), A.10.6.2.1518 FIELD: (page 216).

10.6.2.1740.01 K-ALT-MASK FACILITY EXT

(-- *u*)

Mask for the ALT key, that can be ORed 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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

10.6.2.1740.02 K-CTRL-MASK FACILITY EXT

(-- *u*)

Mask for the CTRL key, that can be ORed 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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

10.6.2.1740.24 K-SHIFT-MASK FACILITY EXT

(-- *u*)

Mask for the SHIFT key, that can be ORed 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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: A.10.6.2.1905 MS (page 216).

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 $\{0\ldots59\}$, $+n_2$ is the minute $\{0\ldots59\}$, $+n_3$ is the hour $\{0\ldots23\}$, $+n_4$ is the day $\{1\ldots31\}$, $+n_5$ is the month $\{1\ldots12\}$ and $+n_6$ is the year (e.g., 1991).

See also: A.10.6.2.2292 TIME&DATE (page 216).

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 UPDATEd 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: 3.3.3.5Input buffers, 9The Exception word set.

11.3.4 Other transient regions

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: 3.4.1Parsing.

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 (.3.1.4File 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 (.3.3Input source);

– maximum size of input line (.3.5Parsing);

– methods for mapping block ranges to files (.3.2Blocks in files);

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 (.3.2Blocks 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 (.3.5Parsing);
- using more than eight levels of input-file nesting (.3.3Input source).

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.

See also: F.11.6.1.0080 ((page 325).

11.6.1.0765 BIN FILE

(*fam*$_1$ –– *fam*$_2$)

Modify the implementation-defined file access method *fam*$_1$ to additionally select a "binary", i.e., not line oriented, file access method, giving access method *fam*$_2$.

See also: 11.6.1.2054 R/O (page 114), 11.6.1.2056 R/W (page 114), 11.6.1.2425 W/O (page 116), A.11.6.1.0765 BIN (page 216).

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 also: A.11.6.1.1010 CREATE-FILE (page 216), F.11.6.1.1010 CREATE-FILE (page 325).

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.

See also: F.11.6.1.1190 DELETE-FILE (page 325).

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.

See also: F.11.6.1.1522 FILE-SIZE (page 325).

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 also: 11.3.3 Input source (page 110), A.11.6.1.1717 INCLUDE-FILE (page 216).

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 also: 11.6.1.1717 INCLUDE-FILE (page 113), A.11.6.1.1718 INCLUDED (page 217), F.11.6.1.1718 INCLUDED (page 326).

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 also: A.11.6.1.1970 OPEN-FILE (page 217).

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 also: 11.6.1.1010 CREATE-FILE (page 112), 11.6.1.1970 OPEN-FILE (page 114).

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 also: 11.6.1.1010 CREATE-FILE (page 112), 11.6.1.1970 OPEN-FILE (page 114).

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 also: A.11.6.1.2080 READ-FILE (page 217).

11.6.1.2090 READ-LINE FILE

(*c-addr* u_1 *fileid* -- u_2 *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 also: A.11.6.1.2090 READ-LINE (page 217), F.11.6.1.2090 READ-LINE (page 326).

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*.

See also: F.11.6.1.2142 REPOSITION-FILE (page 326).

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 also: 11.6.1.2080 READ-FILE (page 115), 11.6.1.2090 READ-LINE (page 115), F.11.6.1.2147 RESIZE-FILE (page 327).

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.

See also: F.11.6.1.2218 SOURCE-ID (page 327).

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 also: 11.6.1.1010 CREATE-FILE (page 112), 11.6.1.1970 OPEN-FILE (page 114).

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 also: 11.6.1.2080 READ-FILE (page 115), 11.6.1.2090 READ-LINE (page 115).

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 also: 11.6.1.2080 READ-FILE (page 115), 11.6.1.2090 READ-LINE (page 115), F.11.6.1.2485 WRITE-LINE (page 327).

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.

INCLUDED

See also: 11.6.1.1718 INCLUDED (page 114), A.11.6.2.1714 INCLUDE (page 218), E.11.6.2.1714 INCLUDE (page 248), F.11.6.2.1714 INCLUDE (page 327).

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 also: 6.2.2125 REFILL (page 78), 7.6.2.2125 REFILL (page 88).

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.

See also: F.11.6.2.2130 RENAME-FILE (page 327).

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 also: 11.6.2.2144.50 REQUIRED (page 119), A.11.6.2.2144.10 REQUIRE (page 218), E.11.6.2.2144.10 REQUIRE (page 248), F.11.6.2.2144.10 REQUIRE (page 328).

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 also: A.11.6.2.2144.50 REQUIRED (page 218), E.11.6.2.2144.50 REQUIRED (page 249), F.11.6.2.2144.50 REQUIRED (page 328).

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.

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

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: 3.2.6Environmental queries.

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

12.3.5 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: 3.3.3.1Address alignment, 12.3.1.1Addresses.

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: 3.3.3.3Variables.

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:

Convertible string := ⟨*significand*⟩⟨*exponent*⟩
⟨*significand*⟩ := [⟨*sign*⟩]⟨*digits*⟩[**.**⟨*digits0*⟩]
⟨*exponent*⟩ := **E**[⟨*sign*⟩]⟨*digits0*⟩
⟨*sign*⟩ := { **+** | **-** }
⟨*digits*⟩ := ⟨*digit*⟩⟨*digits0*⟩
⟨*digits0*⟩ := ⟨*digit*⟩*
⟨*digit*⟩ := { **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **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: 3.4.1.3Text 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 (12.3.1Data 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 (12.3.1.2Floating-point numbers);
- size of floating-point stack (12.3.3Floating-point stack);
- width of floating-point stack (12.3.3Floating-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;
- 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.);
- 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 (12.3.3Floating-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

⟨*significand*⟩[⟨*exponent*⟩]
⟨*significand*⟩ := [⟨*sign*⟩]{⟨*digits*⟩[**.**⟨*digits0*⟩] | **.**⟨*digits*⟩ }
⟨*exponent*⟩ := ⟨*marker*⟩⟨*digits0*⟩
⟨*marker*⟩ := {⟨*e-form*⟩ | ⟨*sign-form*⟩}
⟨*e-form*⟩ := ⟨*e-char*⟩[⟨*sign-form*⟩]
⟨*sign-form*⟩ := { **+** | **-** }
⟨*e-char*⟩ := { **D** | **d** | **E** | **e** }

See also: A.12.6.1.0558 >FLOAT (page 219).

12.6.1.1130 **D>F** "d-to-f" FLOATING

(*d* ––) (F: –– *r*) or (*d* –– *r*)

r is the floating-point value which is nearest to *d* in the sense of "round to nearest".

See also: 12.3.2 Floating-point operations (page 121).

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 also: 3.4.1 Parsing (page 31), A.12.6.1.1492 FCONSTANT (page 219).

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 also: A.12.6.1.1552 FLITERAL (page 219).

12.6.1.1555 FLOAT+ "float-plus" FLOATING

(*f-addr*$_1$ -- *f-addr*$_2$)

Add the size in address units of a floating-point number to *f-addr*$_1$, giving *f-addr*$_2$.

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 also: 12.3.2 Floating-point operations (page 121), 12.6.1.1612 FROUND (page 128), 12.6.2.1627 FTRUNC (page 135).

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 also: 12.3.2 Floating-point operations (page 121), 12.6.1.1558 FLOOR (page 127), 12.6.2.1627 FTRUNC (page 135).

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 also: 3.4.1 Parsing (page 31), A.12.6.1.1630 FVARIABLE (page 219).

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 also: 3.2.1.2 Digit conversion (page 25), 6.1.0750 BASE (page 50), 12.3.2 Floating-point operations (page 121), A.12.6.1.2143 REPRESENT (page 219).

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 ambiguous condition exists if the exponent of *r* is too large to be accommodated in IEEE double-precision format.

See also: 12.3.1.1 Addresses (page 120), 12.3.2 Floating-point operations (page 121).

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 representation using the "round to nearest" rule. An ambiguous condition exists if the exponent of the IEEE double-precision representation is too large to be accommodated by the internal representation.

See also: 12.3.1.1 Addresses (page 120), 12.3.2 Floating-point operations (page 121).

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 also: 12.3.1.1 Addresses (page 120).

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 also: 12.3.1.1 Addresses (page 120).

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 also: 10.6.2.0135 +FIELD (page 101), 10.6.2.0763 BEGIN-STRUCTURE (page 101), 10.6.2.1336 END-STRUCTURE (page 103).

12.6.2.1208 DFLOAT+ “d-float-plus” FLOATING EXT

($df\text{-}addr_1$ –– $df\text{-}addr_2$)

Add the size in address units of a 64-bit IEEE double-precision number to $df\text{-}addr_1$, giving $df\text{-}addr_2$.

See also: 12.3.1.1 Addresses (page 120).

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. ed25

See also: 12.3.1.1 Addresses (page 120).

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 also: 12.6.1.0558 >FLOAT (page 124), A.12.6.2.1427 F. (page 219).

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 also: 12.6.2.2175 S>F (page 136), E.12.6.2.1471 F>S (page 250).

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) $-\pi$ when $r_1 = -0E$ and r_2 is negative. An ambiguous condition exists if r_1 and r_2 are zero.

See also: A.12.6.2.1489 `FATAN2` (page 219), F.12.6.2.1489 `FATAN2` (page 328).

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 also: 6.1.0750 BASE (page 50), 12.3.2 Floating-point operations (page 121), 12.6.1.2143 REPRESENT (page 129).

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 also: A.12.6.2.1516 FEXPM1 (page 220).

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 also: 10.6.2.0135 +FIELD (page 101), 10.6.2.0763 BEGIN-STRUCTURE (page 101), 10.6.2.1336 END-STRUCTURE (page 103).

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 also: A.12.6.2.1554 FLNP1 (page 220).

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:

⟨*significand*⟩ := [**-**]⟨*digit*⟩**.**⟨*digits0*⟩
⟨*exponent*⟩ := **E**[**-**]⟨*digits*⟩

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 also: 6.1.0750 BASE (page 50), 12.3.2 Floating-point operations (page 121), 12.6.1.2143 REPRESENT (page 129).

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 also: A.12.6.2.1489 FATAN2 (page 219).

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 also: 12.3.2 Floating-point operations (page 121), 12.6.1.1558 FLOOR (page 127), 12.6.1.1612 FROUND (page 128), E.12.6.2.1627 FTRUNC (page 250), F.12.6.2.1627 FTRUNC (page 331).

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 also: 3.4.1 Parsing (page 31), 6.2.2295 TO (page 81), E.12.6.2.1628 FVALUE (page 250), F.12.6.2.1628 FVALUE (page 331).

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 also: A.12.6.2.1640 F~ (page 220).

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 value which is nearest to *n* in the sense of "round to nearest".

See also: 12.6.2.1471 F>S (page 131), E.12.6.2.2175 S>F (page 250).

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 also: 12.3.1.1 Addresses (page 120), 12.3.2 Floating-point operations (page 121).

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 also: 12.3.1.1 Addresses (page 120), 12.3.2 Floating-point operations (page 121).

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 also: 12.3.1.1 Addresses (page 120).

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 also: 12.3.1.1 Addresses (page 120).

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 = *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 also: 10.6.2.0135 +FIELD (page 101), 10.6.2.0763 BEGIN-STRUCTURE (page 101), 10.6.2.1336 END-STRUCTURE (page 103), A.10.6.2.1518 FIELD: (page 216).

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 also: 12.3.1.1 Addresses (page 120).

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 also: 12.3.1.1 Addresses (page 120).

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: 3.2.6Environmental 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: 3.4The 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 (13.3.3Processing 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 (13.3.3Processing 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 13.3.3Processing 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.

local Interpretation: Interpretation semantics for this word are undefined. x:local-interpret

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 13.3.3Processing 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 13.3.3Processing locals.

See also: 3.4 The Forth text interpreter (page 31), 6.2.2295 TO (page 81).

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*⟩*name*$_1$" "⟨*spaces*⟩*name*$_2$" ... "⟨*spaces*⟩*name*$_n$" " | " --)

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 *name*$_1$ is initialized with x_1, identifier *name*$_2$ 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.

See also: E.13.6.2.1795 LOCALS| (page 250).

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:

{: ⟨*arg*⟩* [| ⟨*val*⟩*] [-- ⟨*out*⟩*] :}

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.~~ x:local-interpret

name Interpretation: Interpretation semantics of *name* are undefined. x:local-interpret

TO *name* Run-time: (*x* ––)

Set *name* to the value *x*.

See also: 2.2.5 BNF notation (page 18), 6.2.2295 TO (page 81), 6.2.2405 VALUE (page 82), A.13.6.2.2550 {: (page 221), E.13.6.2.2550 {: (page 251).

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: 3.3.3Data 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 also: 6.1.1650 HERE (page 57), 14.6.1.1605 FREE (page 146), 14.6.1.2145 RESIZE (page 146), F.14.6.1.0707 ALLOCATE (page 332).

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 also: 6.1.1650 HERE (page 57), 14.6.1.0707 ALLOCATE (page 146), 14.6.1.2145 RESIZE (page 146), F.14.6.1.1605 FREE (page 333).

14.6.1.2145 RESIZE MEMORY

($a\text{-}addr_1$ *u* –– $a\text{-}addr_2$ *ior*)

Change the allocation of the contiguous data space starting at the address $a\text{-}addr_1$, 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, $a\text{-}addr_2$ is the aligned starting address of *u* address units of allocated memory and *ior* is zero. $a\text{-}addr_2$ may be, but need not be, the same as $a\text{-}addr_1$. If they are not the same, the values contained in the region at $a\text{-}addr_1$ are copied to $a\text{-}addr_2$, 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 $a\text{-}addr_2$ is not the same as $a\text{-}addr_1$, the region of memory at $a\text{-}addr_1$ is returned to the system according to the operation of FREE.

If the operation fails, $a\text{-}addr_2$ equals $a\text{-}addr_1$, the region of memory at $a\text{-}addr_1$ is unaffected, and *ior* is the implementation-defined I/O result code.

See also: 6.1.1650 HERE (page 57), 14.6.1.0707 ALLOCATE (page 146), 14.6.1.1605 FREE (page 146), F.14.6.1.2145 RESIZE (page 333).

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

name token: An abstract data type identifying a named word. Name tokens can be passed to words (such as NAME>STRING) to obtain information about the named word.

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
quotation-sys	colon definition status	implementation dependent

15.3.2 Colon definition status

The implementation-dependent *quotation-sys* type contains the data that needs to be saved for the enclosing colon definition and restored after the end of the quotation. It is used in combination with *colon-sys*.

15.3.3 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: 3.3The 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 (15.3.3The 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);

- resolving an already resolved *orig* forward reference (15.6.2.1015 CS-PICK, 6.1.1310 ELSE, 6.1.2270 THEN, 6.1.2140 REPEAT); x:cs-drop
- at the end of the current definition unresolved *orig* forward references still exist (15.6.2.0 CS-DROP, 6.1.0460 ;, 6.1.1700 IF, 6.1.1310 ELSE); x:cs-drop
- *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
- The compilation semantics of 15.6.2.0470 ;CODE or 6.1.0460 ; used within a quotation ([: ... ;]).

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 also: 3.3.3.6 Other transient regions (page 30), A.15.6.1.0220 .S (page 222).

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 also: 3.3.3.6 Other transient regions (page 30).

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 also: 3.3.3.6 Other transient regions (page 30).

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 also: 3.3.3.6 Other transient regions (page 30), A.15.6.1.2194 SEE (page 222).

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 also: 3.3.3.6 Other transient regions (page 30), A.15.6.1.2465 WORDS (page 222).

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.

An ambiguous condition exists if the compilation semantics of ;CODE is preformed inside a quotation.

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 also: 6.1.1250 DOES> (page 54), A.15.6.2.0470 ;CODE (page 222).

15.6.2.—— **;]** "semi-bracket" TOOLS EXT
x:quotations

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: *quotation-sys colon-sys* ––)

Ends the current nested definition, and resumes compilation to the previous (containing) current definition. It appends the following run-time action to the (containing) current definition.

Run-time: (–– *xt*)

xt is the execution token of the nested definition.

See also: 15.6.2.0 [: (page 157), A.15.6.2.0 [: (page 226), E.15.6.2.0 ;] (page 251).

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*.

See also: F.15.6.2.0702 AHEAD (page 333).

15.6.2.0740 **ASSEMBLER** TOOLS EXT

(––)

Replace the first word list in the search order with the ASSEMBLER word list.

See also: 16 The optional Search-Order word set (page 160).

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 also: 3.4.1 Parsing (page 31), A.15.6.2.0930 CODE (page 222).

15.6.2.—— CS-DROP “c-s-drop” TOOLS EXT x:cs-drop

x:cs-drop

Interpretation: Interpretation semantics for this word are undefined.

Execution: (C: *dest* | *orig* ––)

Remove the top item (*dest* | *orig*) from the control-flow stack. An ambiguous condition exists if the top control-flow stack item is not a *dest* or an *orig*, or if the control-flow stack is empty.

See also: A.15.6.2.0 CS-DROP (page 222), E.15.6.2.0 CS-DROP (page 252), F.15.6.2.0 CS-DROP (page 333).

15.6.2.1015 CS-PICK “c-s-pick” TOOLS EXT

Interpretation: Interpretation semantics for this word are undefined. x:cs-drop

Execution: (C: ~~$dest_u \ldots orig_0 \mid dest_0$~~ –– ~~$dest_u \ldots orig_0 \mid dest_0\ dest_u$~~)
(C: $dest_u \mid orig_u \ldots dest_0 \mid orig_0$ –– $dest_u \mid orig_u \ldots dest_0 \mid orig_0\ dest_u \mid orig_u$)
(S: *u* ––)

Remove *u*. Copy ~~$dest_u$~~ $orig_u \mid dest_u$ 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 also: A.15.6.2.1015 CS-PICK (page 223), F.15.6.2.1015 CS-PICK (page 333).

15.6.2.1020 CS-ROLL "c-s-roll" TOOLS EXT

Interpretation: Interpretation semantics for this word are undefined.

Execution: (C: $orig_u$ | $dest_u$ $orig_{u-1}$ | $dest_{u-1}$ ... $orig_0$ | $dest_0$ -- $orig_{u-1}$ | $dest_{u-1}$ ... $orig_0$ | $dest_0$ $orig_u$ | $dest_u$) (S: u --)

Remove u. Rotate u+1 elements on top of the control-flow stack so that $orig_u$ | $dest_u$ 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 also: A.15.6.2.1020 CS-ROLL (page 224), F.15.6.2.1020 CS-ROLL (page 334).

15.6.2.1300 EDITOR TOOLS EXT

(--)

Replace the first word list in the search order with the EDITOR word list.

See also: 16 The optional Search-Order word set (page 160).

15.6.2.—— FIND-NAME TOOLS EXT
x:find-name

(*c-addr u* -- *nt* | *0*)

Find the definition identified by the string *c-addr u* in the current search order. Return its name token *nt*, if found, otherwise 0.

See also: 15.6.2.0 FIND-NAME-IN (page 154), A.15.6.2.0 FIND-NAME (page 224), E.15.6.2.0 FIND-NAME (page 252), F.15.6.2.0 FIND-NAME (page 335).

15.6.2.—— FIND-NAME-IN TOOLS EXT
x:find-name

(*c-addr u wid* -- *nt* | *0*)

Find the definition identified by the string *c-addr u* in the wordlist *wid*. Return its name token *nt*, if found, otherwise 0.

See also: 15.6.2.0 FIND-NAME (page 154), A.15.6.2.0 FIND-NAME (page 224), E.15.6.2.0 FIND-NAME-IN (page 253), F.15.6.2.0 FIND-NAME-IN (page 336).

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 also: 3.4.1 Parsing (page 31), A.15.6.2.1580 FORGET (page 225).

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 also: 15.6.2.1940 NR> (page 156), A.15.6.2.1908 N>R (page 225), E.15.6.2.1908 N>R (page 253), F.15.6.2.1908 N>R (page 337).

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 also: 15.6.2.2297 TRAVERSE-WORDLIST (page 157), A.15.6.2.1909.10 NAME>COMPILE (page 225).

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 also: 15.6.2.2297 TRAVERSE-WORDLIST (page 157).

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 also: 15.6.2.2297 TRAVERSE-WORDLIST (page 157).

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 also: 15.6.2.1908 N>R (page 155), A.15.6.2.1908 N>R (page 225), E.15.6.2.1940 NR> (page 254).

15.6.2.2250 STATE TOOLS EXT

(–– *a-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 also: 3.4 The Forth text interpreter (page 31), 6.1.0450 : (page 46), 6.1.0460 ; (page 46), 6.1.0670 ABORT (page 48), 6.1.2050 QUIT (page 62), 6.1.2250 STATE (page 65), 6.1.2500 [(page 69), 6.1.2540] (page 70), 6.2.0455 :NONAME (page 71), 15.6.2.0470 ; CODE (page 151).

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 also: 6.1.1710 IMMEDIATE (page 58), A.15.6.2.2264 SYNONYM (page 225), E.15.6.2.2264 SYNONYM (page 254).

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~~ named word that can be found in the wordlist *wid*, passing the name token *nt* of the word to *xt*, until the wordlist is exhausted or until *xt* returns false. x:traverse-wordlist

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.

Every word that used to be findable in the wordlist but no longer is (because antoher word with the same name has been defined in the wordlist) is returned by TRAVERSE-WORDLIST.

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 also: 15.6.2.1909.20 NAME>INTERPRET (page 155), 15.6.2.1909.10 NAME>COMPILE (page 155), 15.6.2.1909.40 NAME>STRING (page 156), A.15.6.2.2297 TRAVERSE-WORDLIST (page 225).

15.6.2.—— [: “bracket-colon” TOOLS EXT
x:quotations

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: –– *quotation-sys colon-sys*)

Suspends compiling to the current definition, starts a new nested definition with execution token xt, and compilation continues with this nested definition.

Locals may be defined in the nested definition. An ambiguous condition exists if a name is used that satisfies the following constraints:

- It is not the name of a currently visible local of the current quotation;
- It is the name of a local that was visible right before the start of the present quotation or any of the containing quotations.

See also: 15.6.2.0 ;] (page 152), A.15.6.2.0 [: (page 226), E.15.6.2.0 [: (page 254), F.15.6.2.0 [: (page 337).

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. Try to find *name*. Return a true flag if *name* can be found; otherwise return a false flag. [DEFINED] is an immediate word. x:rule-of-find

See also: 6.1.1550 FIND (page 57), 16.6.1.1550 FIND (page 162), E.15.6.2.2530.30 [DEFINED] (page 255).

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 also: 3.4.1 Parsing (page 31), 15.6.2.2532 [IF] (page 158), 15.6.2.2533 [THEN] (page 159), A.15.6.2.2531 [ELSE] (page 227), E.15.6.2.2531 [ELSE] (page 255), F.15.6.2.2533 [THEN] (page 337). ed25 x:simplify-bracket-else

15.6.2.2532 [IF] "bracket-if" TOOLS EXT

Compilation: Perform the execution semantics given below. x:[if]-input

Execution: (~~*flag*~~ *x* | ~~*flag*~~ *x* "⟨*spaces*⟩*name* ..." --)

If ~~*flag* is true~~ any bit of *x* is set, 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 POSTPONEd, or if the end of the input buffer is reached and cannot be refilled before the terminating [ELSE] or [THEN] is parsed.

See also: 3.4.1 Parsing (page 31), 15.6.2.2531 [ELSE] (page 158), 15.6.2.2533 [THEN] (page 159), A.15.6.2.2532 [IF] (page 227), E.15.6.2.2531 [ELSE] (page 255), E.15.6.2.2532 [IF] (page 255), F.15.6.2.2533 [THEN] (page 337). ed25 simplify-bracket-else

15.6.2.2533 [THEN] "bracket-then" TOOLS EXT

Compilation: Perform the execution semantics given below.

Execution: (––)

Does nothing. [THEN] is an immediate word.

ed25

See also: 15.6.2.2532 [IF] (page 158), 15.6.2.2531 [ELSE] (page 158), A.15.6.2.2533 [THEN] (page 227), E.15.6.2.2531 [ELSE] (page 255), E.15.6.2.2533 [THEN] (page 256), F.15.6.2.2533 [THEN] (page 337).

x:simplify-bracket-else

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.

See also: E.15.6.2.2534 [UNDEFINED] (page 256).

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.

find (update the definition in 2.1Definitions of terms): To search the search order or a specified wordlist for a definition name matching a given string.

x:rules-of-find

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: 3.1Data types, 3.4.2Finding definition names, 3.4The Forth text interpreter.

16.3.2 Environmental queries

Append table 16.2 to table 3.5.

See: 3.2.6Environmental 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: 3.4.2Finding definition names.

16.3.4 Contiguous regions

The regions of data space produced by the operations described in 3.3.3.2Contiguous 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 (16.3.3Finding 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 (16.3.3Finding definition names);
- search order empty (16.6.2.2037 PREVIOUS);
- too many word lists in search order (16.6.2.0715 ALSO).
- executing a vocabulary word when the search order is empty (16.6.2.0 VOCABULARY). x:vocabulary

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 (16.3.3Finding 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 also: 16.3.3 Finding definition names (page 160), E.16.6.1.1180 DEFINITIONS (page 256), F.16.6.1.1180 DEFINITIONS (page 338).

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.

x:rule-of-find

See also: ~~3.4.2 Finding definition names (page 32)~~, 6.1.0070 ' (page 41), 6.1.1550 FIND (page 57), 6.1.2033 POSTPONE (page 62), 6.1.2510 ['] (page 69), 15.6.2.2530.30 [DEFINED] (page 158), 16.3.3 Finding definition names (page 160), E.16.6.1.1550 FIND (page 256), F.16.6.1.1550 FIND (page 339).

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.

See also: F.16.6.1.1595 FORTH-WORDLIST (page 339).

16.6.1.1643 GET-CURRENT SEARCH

(-- *wid*)

Return *wid*, the identifier of the compilation word list.

16.6.1.1647 GET-ORDER SEARCH

(–– $wid_n \ldots wid_1\ n$)

Returns the number of word lists n in the search order and the word list identifiers $wid_n \ldots wid_1$ 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.

See also: E.16.6.1.1647 GET-ORDER (page 256).

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 also: A.16.6.1.2192 SEARCH-WORDLIST (page 228), F.16.6.1.2192 SEARCH-WORDLIST (page 339).

16.6.1.2195 SET-CURRENT SEARCH

(*wid* ––)

Set the compilation word list to the word list identified by *wid*.

See also: F.16.6.1.2195 SET-CURRENT (page 339).

16.6.1.2197 SET-ORDER SEARCH

($wid_n \ldots wid_1\ n$ ––)

Set the search order to the word lists identified by $wid_n \ldots wid_1$. 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.

See also: E.16.6.1.2197 SET-ORDER (page 257), F.16.6.1.2197 SET-ORDER (page 340).

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.

See also: E.16.6.2.0715 ALSO (page 257), F.16.6.2.0715 ALSO (page 340).

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, $wid_{FORTH-WORDLIST}$.

See also: E.16.6.2.1590 FORTH (page 257).

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.

See also: E.16.6.2.1965 ONLY (page 257), F.16.6.2.1965 ONLY (page 340).

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 also: 3.3.3.6 Other transient regions (page 30), F.16.6.2.1985 ORDER (page 340).

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.

See also: E.16.6.2.2037 PREVIOUS (page 257).

16.6.2.—— **VOCABULARY** SEARCH EXT x:vocabulary

x:vocabulary

(*“⟨spaces⟩name”* ––)

Skip leading space delimiter. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Create a new empty word list *wid* and associate it with *name*.

name is referred to as a "vocabulary".

name Execution:: (––)

Replace the first word list in the search order by the word list *wid* that is associated with the *name* vocabulary. An ambiguous condition exists if there are no word lists in the search order.

See also: 16.6.1.1647 GET-ORDER (page 163), 16.6.1.2460 WORDLIST (page 163), 16.6.1.2197 SET-ORDER (page 163), 16.6.2.0715 ALSO (page 164), 16.6.2.2037 PREVIOUS (page 164), A.16.6.2.0 VOCABULARY (page 228), E.16.6.2.0 VOCABULARY (page 257).

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-addr* u_1 -- *c-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-addr* u_1. If u_1 is zero or the entire string consists of spaces, u_2 is zero.

See also: F.17.6.1.0170 -TRAILING (page 340).

17.6.1.0245 /STRING "slash-string" STRING

(*c-addr*$_1$ u_1 n -- *c-addr*$_2$ u_2)

Adjust the character string at *c-addr*$_1$ by n characters. The resulting character string, specified by *c-addr*$_2$ u_2, begins at *c-addr*$_1$ plus n characters and is u_1 minus n characters long.

See also: A.17.6.1.0245 /STRING (page 228), F.17.6.1.0245 /STRING (page 341).

17.6.1.0780 BLANK STRING

(*c-addr* u --)

If u is greater than zero, store the character value for space in u consecutive character positions beginning at *c-addr*.

See also: F.17.6.1.0780 BLANK (page 341).

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 also: 17.6.1.0920 CMOVE> (page 168), A.17.6.1.0910 CMOVE (page 228).

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 also: 17.6.1.0910 CMOVE (page 167), A.17.6.1.0920 CMOVE> (page 228).

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.

See also: F.17.6.1.0935 COMPARE (page 341).

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*$_1$.

See also: F.17.6.1.2191 SEARCH (page 341).

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-addr*$_2$ *u*)

Return *c-addr*$_2$ *u* describing a string consisting of the characters specified by *c-addr*$_1$ *u* during compilation. A program shall not alter the returned string.

See also: A.17.6.1.2212 SLITERAL (page 229), F.17.6.1.2212 SLITERAL (page 342).

17.6.2 String extension words

17.6.2.2141 REPLACES STRING EXT

(*c-addr*$_1$ *u*$_1$ *c-addr*$_2$ *u*$_2$ −−)

Set the string *c-addr*$_1$ *u*$_1$ as the text to substitute for the substitution named by *c-addr*$_2$ *u*$_2$. If the substitution does not exist it is created. The program may then reuse the buffer *c-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~~ Therefore REPLACES cannot be performed duing compilation of a colon definition or in the middle of a contiguous region . x:revise-replaces

See also: 3.3.3.2 Contiguous regions (page 29), 3.4.5 Compilation (page 34), 17.6.2.2255 SUBSTITUTE (page 169), E.17.6.2.2141 REPLACES (page 257).

17.6.2.2255 SUBSTITUTE STRING EXT

(*c-addr*$_1$ *u*$_1$ *c-addr*$_2$ *u*$_2$ −− *c-addr*$_2$ *u*$_3$ *n*)

Perform substitution on the string *c-addr*$_1$ *u*$_1$ placing the result at string *c-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-addr*$_2$ *u*$_2$ or if *c-addr*$_2$ is the same as *c-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-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-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 also: 17.6.2.2141 REPLACES (page 169), 17.6.2.2375 UNESCAPE (page 170), A.17.6.2.2255 SUBSTITUTE (page 229), E.17.6.2.2255 SUBSTITUTE (page 259), F.17.6.2.2255 SUBSTITUTE (page 342).

17.6.2.2375 UNESCAPE STRING EXT

(*c-addr*$_1$ *u*$_1$ *c-addr*$_2$ −− *c-addr*$_2$ *u*$_2$)

Replace each '%' character in the input string *c-addr*$_1$ *u*$_1$ by two '%' characters. The output is represented by *c-addr*$_2$ *u*$_2$. The buffer at *c-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-addr*$_2$).

See also: 17.6.2.2255 SUBSTITUTE (page 169), E.17.6.2.2375 UNESCAPE (page 260), F.17.6.2.2375 UNESCAPE (page 343).

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: 2.2.3Parsed-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: 3.1Data 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.

See: 3.2.6Environmental queries.

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

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 3.4.1.3Text 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.

See also: E.18.6.1.2486.50 X-SIZE (page 261).

18.6.1.2487.10 XC!+ "x-c-store-plus" XCHAR

(*xchar xc-addr*$_1$ -- *xc-addr*$_2$)

Stores the *xchar* at *xc-addr*$_1$. *xc-addr*$_2$ points to the first memory location after the stored *xchar*.

See also: E.18.6.1.2487.10 XC!+ (page 261).

18.6.1.2487.15 XC!+? "x-c-store-plus-query" XCHAR

(*xchar xc-addr*$_1$ *u*$_1$ -- *xc-addr*$_2$ *u*$_2$ *flag*)

Stores the *xchar* into the string buffer specified by *xc-addr*$_1$ *u*$_1$. *xc-addr*$_2$ *u*$_2$ is the remaining string buffer. If the *xchar* did fit into the buffer, *flag* is true, otherwise *flag* is false, and *xc-addr*$_2$ *u*$_2$ equal *xc-addr*$_1$ *u*$_1$. XC!+? is safe for buffer overflows.

See also: E.18.6.1.2487.15 XC!+? (page 261), F.18.6.1.2487.15 XC!+? (page 343).

18.6.1.2487.20 XC, "x-c-comma" XCHAR

(*xchar* --)

Append the encoding of *xchar* to the dictionary.

See also: 6.1.0860 C, (page 51), E.18.6.1.2487.20 XC, (page 261).

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.

See also: E.18.6.1.2487.25 XC-SIZE (page 261), F.18.6.1.2487.25 XC-SIZE (page 343).

18.6.1.2487.35 XC@+ "x-c-fetch-plus" XCHAR

(*xc-addr*$_1$ -- *xc-addr*$_2$ *xchar*)

Fetches the *xchar* at *xc-addr*$_1$. *xc-addr*$_2$ points to the first memory location after the retrieved *xchar*.

See also: E.18.6.1.2487.35 XC@+ (page 262).

18.6.1.2487.40 XCHAR+ "x-char-plus" XCHAR

(*xc-addr*$_1$ -- *xc-addr*$_2$)

Adds the size of the xchar stored at *xc-addr*$_1$ to this address, giving *xc-addr*$_2$.

See also: 6.1.0897 CHAR+ (page 52), E.18.6.1.2487.40 XCHAR+ (page 262).

18.6.1.2488.10 XEMIT "x-emit" XCHAR

(*xchar* ––)

Prints an *xchar* on the terminal.

See also: 6.1.1320 EMIT (page 55), E.18.6.1.2488.10 XEMIT (page 262).

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 also: 6.1.1750 KEY (page 59), E.18.6.1.2488.30 XKEY (page 262).

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 also: 10.6.1.1755 KEY? (page 100).

18.6.2 Extended-Character extension words

18.6.2.0145 +X/STRING "plus-x-string" XCHAR EXT

(*xc-addr*$_1$ u_1 –– *xc-addr*$_2$ u_2)

Step forward by one xchar in the buffer defined by *xc-addr*$_1$ u_1. *xc-addr*$_2$ u_2 is the remaining buffer after stepping over the first xchar in the buffer.

See also: E.18.6.2.0145 +X/STRING (page 262).

18.6.2.0175 -TRAILING-GARBAGE "minus-trailing-garbage" XCHAR EXT

(*xc-addr* u_1 –– *xc-addr* u_2)

Examine the last xchar in the string *xc-addr* u_1 — 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.

See also: E.18.6.2.0175 -TRAILING-GARBAGE (page 262).

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 also: 6.1.0895 CHAR (page 52), A.18.6.2.0895 CHAR (page 230), E.18.6.2.0895 CHAR (page 262).

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 also: 10.6.2.1305 EKEY (page 102), 10.6.2.1306 EKEY>CHAR (page 102), 10.6.2.1306.40 EKEY>FKEY (page 102).

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 also: 3.4.1 Parsing (page 31), 6.2.2008 PARSE (page 78), A.6.2.2008 PARSE (page 205).

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.

See also: E.18.6.2.2486.70 X-WIDTH (page 263).

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.

See also: E.18.6.2.2487.30 XC-WIDTH (page 263), F.18.6.2.2487.30 XC-WIDTH (page 343).

18.6.2.2487.45 XCHAR- "x-char-minus" XCHAR EXT

(*xc-addr*$_1$ -- *xc-addr*$_2$)

Goes backward from *xc-addr*$_1$ until it finds an xchar so that the size of this xchar added to *xc-addr*$_2$ gives *xc-addr1*. There is an ambiguous condition when the encoding doesn't permit reliable backward stepping through the text.

See also: E.18.6.2.2487.45 XCHAR- (page 264).

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 also: 6.1.1670 HOLD (page 57), E.18.6.2.2488.20 XHOLD (page 264).

18.6.2.2495 X\STRING- "x-string-minus" XCHAR EXT

(*xc-addr u*$_1$ -- *xc-addr u*$_2$)

Search for the penultimate xchar in the string *xc-addr u*$_1$. The string *xc-addr u*$_2$ contains all xchars of *xc-addr u*$_1$, but the last. Unlike XCHAR-, X\STRING- can be implemented in encodings where xchar boundaries can only reliably detected when scanning in forward direction.

See also: E.18.6.2.2495 X\STRING- (page 264).

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 also: 6.1.2520 [CHAR] (page 69), E.18.6.2.2520 [CHAR] (page 264).

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) 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`.

4) 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 ~). 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.

5) 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.

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

b) **Storage and retrieval**

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

c) **Manipulation on the stack**

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

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

d) **Additional operations**

The following mathematical operators are valid for character data:

```
+ - * / /MOD MOD
```

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

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

A.3.1.3 Single-cell types

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

a) **Storage and retrieval**

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

b) **Manipulation on the stack**

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

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

c) **Comparison operators**

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

```
= <> 0= 0<>
```

A.3.1.3.1 Flags

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

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

```
AND OR XOR INVERT
```

A.3.1.3.2 Integers

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.3.1.3.4Counted 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 range of signed numbers is $-n-1$ to n and for unsigned numbers is 0 to $2n+1$, where n is the largest positive signed number. Signed numbers in the 0 to n range are bitwise identical to the corresponding unsigned number.

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.3.2.3 Stacks

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

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

A.3.2.3.2 Control-flow stack The simplest use of control-flow words is to implement the basic control structures shown in figure **A.1**.

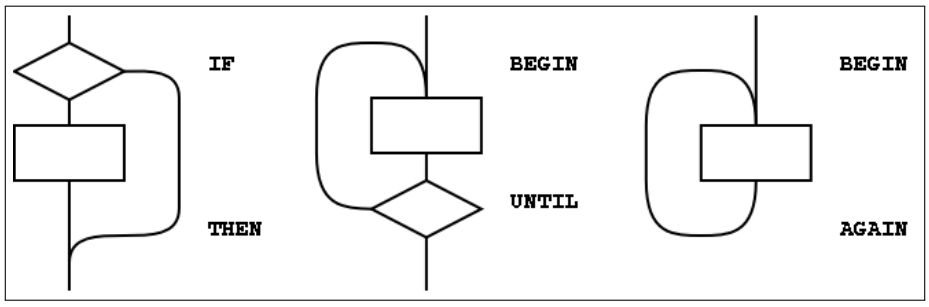

Figure A.1: The basic control-flow patterns

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
```

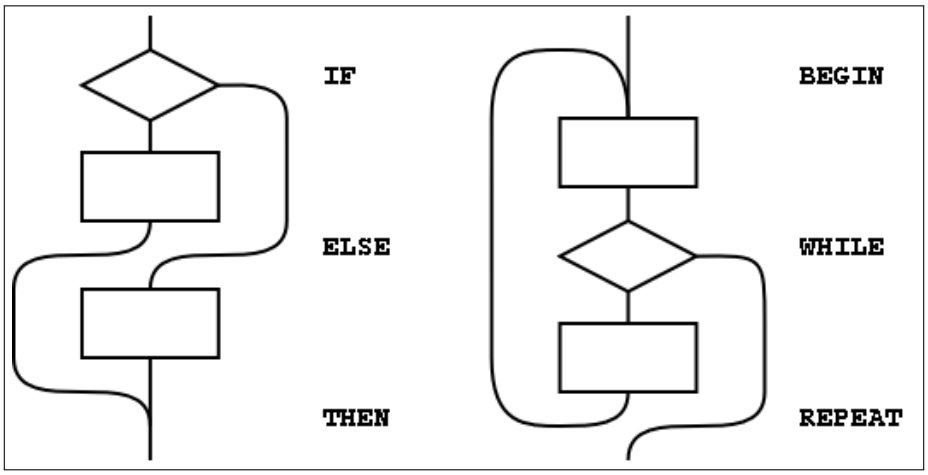

Figure A.2: Additional basic control-flow patterns

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

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

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 -- )
  1+                ( count OFs )
  >R                ( move off the stack in case the control-flow )
                    ( stack is the data stack.  )
```

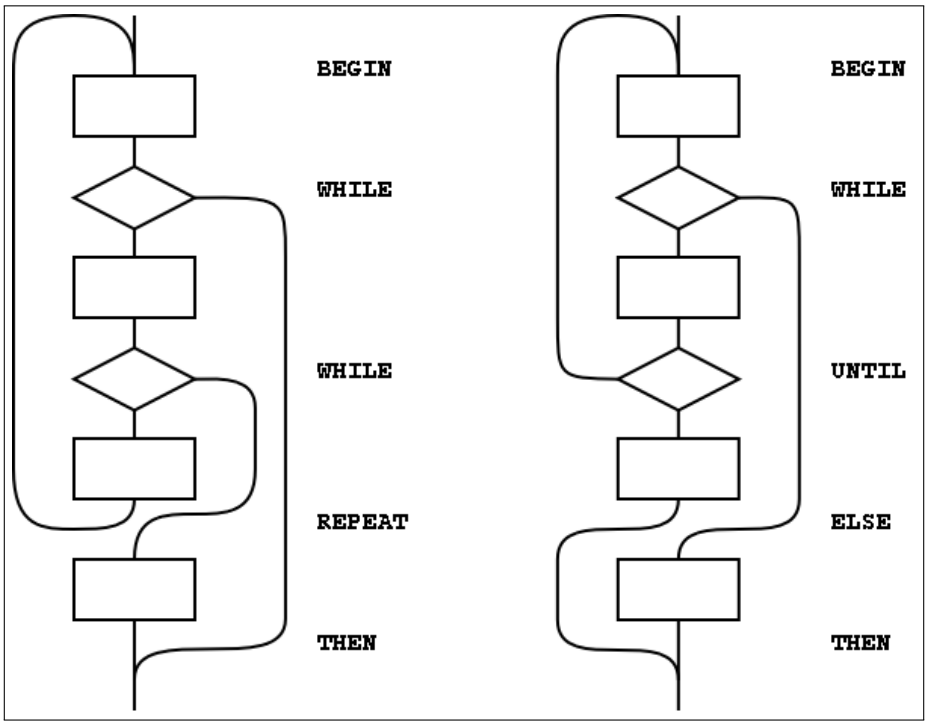

Figure A.3: Extended control-flow patterns

```
  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 3.2.3.3Return 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.3.3.3.4 Text-literal regions

Additional transient buffers are provided for use by C", 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 6.1.2165 S" and 6.2.2266 S\".

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, by acknowledging an environmental dependency on big-endian memory access, a programmer becomes entitled to use the lowest memory address of a cell as its the most significant byte.

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.3.4.3.2Interpretation 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 `:`

Words defined with `:` are called colon definitions.

x:revise-colon-ed

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 (6.1.2165 S" is in an optional word set) and also a means to test the returned flag in interpretation state (e.g. the optional 15.6.2.2532 [IF]).~~ The Core word set lack a means to test the returned flag in interpretation state (e.g. the optional 15.6.2.2532 [IF]). ed21

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:
... S" *ccc*" ...

or

: X ... 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" or S\" in interpretation state. It is intended that no standard words other than S" or 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" or 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.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 LOADing a block containing], the system will be in compilation state.

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

```
: NOP : POSTPONE ; IMMEDIATE ;

NOP ALIGN
NOP ALIGNED
```

Some non-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.3.1.3.4Counted 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:

$$u \text{ ———— } [\ldots\ldots\ldots) \text{ ———— } o$$

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:

$$u \ldots\ldots\ldots) \text{ ———— } [\ldots\ldots\ldots o$$

$n_3 \mid u_3$ 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 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.

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 7.3.2Block 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

An alias for `DROP` that conveys the intent to convert a double-cell to a single-cell integer. The original intention of this word was to support signed-number representations other than two’s complement.

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 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 reTHROWn 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 THROWn.

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 THROWn 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 *x*FIELD: 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 *x*FIELD: 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.11.6 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.11.6.2.1714 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 12.3.7Text 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...LOOPs 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.15.6 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.

x:cs-drop

A.15.6.2.—— CS-DROP

Note that for every CS-DROP of an *orig*, there must be a CS-PICK of that *orig*, because every *orig* must be resolved exactly once (see 15.4.1.2).

Typical use of CS-DROP would be in defining elaborated control structures. For example, a control structure that allows to branch multiple times to an enclosing BEGIN. A corresponding END drops the BEGIN-generated control-flow dest item:

```
: END ( C: dest -- ) \ Compilation
      ( -- )         \ Run-time
```

```
  CS-DROP
; IMMEDIATE

: ?{ ( C: dest -- dest orig dest) \ Compilation
     ( f -- )                     \ Run-time
  POSTPONE IF
  1 CS-PICK
; IMMEDIATE

: }* ( C: orig dest -- )
  POSTPONE AGAIN
  POSTPONE THEN
; IMMEDIATE
```

This can can then be used to define the Collatz function:

```
: even? ( u - f )
  1 AND 0=
;

: collatz ( u - )
  BEGIN
     DUP .
     DUP even? ?{ 2 / }*
     DUP 1 <> ?{ 3 * 1+ }*
  END
  DROP
;

19 collatz ( 19 58 29 88 44 22 11 34 17 52 26 13 40 20 10 5 16
8 4 2 1 ok )
```

A.15.6.2.1015 CS-PICK

~~The intent~~ A common way to use CS-PICK is to copy *dest* ~~on the control-flow stack~~ so that it can be resolved more than once. For example: x:cs-drop

```
\ 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 )
  ;
```

You can also use CS-PICK to copy an *orig*, but then one of the resulting copies must be CS-DROPped because every *orig* must be resolved exactly once (see 15.4.1.2). x:cs-drop

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.—— FIND-NAME

FIND-NAME and FIND-NAME-IN are natural factors of all words that look up words in the dictionary, such as FIND, ', POSTPONE, the text interpreter, and SEARCH-WORDLIST. So implementing them does not cost additional code, only some refactoring effort.

This approach is not compatible with system that use separate word headers for the interpretation and compilation semantics of a word. This problem already exists for the other words that deal with name tokens. However, such systems have been known for a lest two decades, and have seen little to no uptake in standard systems.

Typically used to build a custom text interpreter:

```
: get-name-token ( "name<space>" -- nt )
   PARSE-NAME FIND-NAME DUP 0= -13 AND THROW
;

: ' get-name-token NAME>INTERPRET ;

: POSTPONE
   get-name-token NAME>COMPILE SWAP POSTPONE LITERAL COMPILE,
; IMMEDIATE

\ User-defined text interpreter
: interpret-word
   PARSE-NAME 2DUP FIND-NAME IF
     NIP NIP STATE @ IF
       NAME>COMPILE
     ELSE
       NAME>INTERPRET
     THEN EXECUTE
   ELSE
     \ Process numbers
     0 0 2SWAP >NUMBER 2DROP
```

```
    STATE @ IF 2LITERAL THEN
  THEN
;
```

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,
```

x:synonym

A.15.6.2.2264 SYNONYM

The implementation of 15.6.2.2264 SYNONYM requires detailed knowledge of the host implementation, which is why it is standardized.

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.—— [:

The essence of quotations is to provide nested colon definitions, in which the inner definition(s) are nameless. The expression

```
: foo ...  [: some words ;] ...  ;
```

is equivalent to

```
:NONAME some words ; CONSTANT (temp)
: foo ...  (temp) ...  ;
```

A simple quotation is an anonymous colon definition that is defined inside a colon definition or quotation.

Their advantage is as a syntactic "sugar" that permits a nameless definition in close proximity to its use; and that it avoids generating one-use names only for the purpose of referring to the definition inside another word.

One example use of quotations is to provide a solution to the use of CATCH in a form close to other languages' `try ... catch` blocks.

```
: hex.  ( u -- )
  BASE @ >R
  [: HEX U. ;] CATCH
  R> BASE ! THROW
;
```

The advantage of using CATCH here is that the BASE is restored even if there is an exception (e.g., a user interrupt) during the U..

Moreover, return-address manipulation has often been used as a way to split a definition into several parts, e.g,:

```
: foo bar LIST> bla blub ;
```

where LIST> is a return-address manipulating word and executes the `bla blub` part of the word possibly several times. This demonstrates that introducing a helper definition is unattractive to these programmers; with quotations this code could be written without helper word as

```
: foo bar [: bla blub ;] map-list ;
```

The advantages of this variant are:

- Implementing quotations puts less restrictions on the Forth system than implementing manipulable return-addresses (which would restrict tail-call elimination and inlining).
- It is immediately visible to the reader that there is a separate definition containing `bla blub`.

A quotation may not be able to access the locals of the outer word because it has no knowledge of when it might be executed and hence whether outer locals are still alive. It does permit defining and accessing its own locals. Note that this means that both the quotation and the containing definition may define locals.

A.15.6.2.2531 [ELSE]

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

While it is possible to use [ELSE] without a preceding [IF], it is not recommended.

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.

x:vocabulary

A.16.6.2.—— VOCABULARY

VOCABULARY has been used in traditional Forth systems and it is available with consistent behaviour in many standard systems. First included in Forth-83, it was removed from Forth-94 to promote the use of WORDLIST.

Typical use: VOCABULARY MYVOC

```
ONLY FORTH ALSO MYVOC DEFINITIONS
```

The last line sets the search order to search MYVOC first and FORTH second, and it sets the compilation wordlist to MYVOC.

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-addr*$_2$ lies within the source region (i.e., when *c-addr*$_2$ is not less than *c-addr*$_1$ and *c-addr*$_2$ is less than the quantity *c-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-addr*$_1$ lies within the destination region (i.e., when *c-addr*$_1$ is greater than or equal to *c-addr*$_2$ and *c-addr*$_2$ is less than the quantity *c-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" 09/Jan/2025" ;
: time S" 16:17" ;
date S" date" REPLACES
time S" time" REPLACES
```

The substitution name “date” is defined to be replaced with the string “09/Jan/2025” and “time” to be replaced with “16:17”. Thus SUBSTITUTE would produce the string:

```
Your balance at 16:17 on 09/Jan/2025 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 7) 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 3.4.1.3Text 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 6.2Core extension words:

6.2.0698 ACTION-OF
6.2.0825 BUFFER:
6.2.1173 DEFER
6.2.1175 DEFER!
6.2.1177 DEFER@
6.2.1675 HOLDS
6.2.1725 IS
6.2.2020 PARSE-NAME
6.2.2266 S\"

C.8.8 Double-Number word sets

The following words have been added to 8.6.2Double-Number extension words:

8.6.2.0435 2VALUE

C.8.10 Facility word sets

The following words have been added to 10.6.2Facility extension words:

10.6.2.0135 +FIELD
10.6.2.0763 BEGIN-STRUCTURE
10.6.2.0893 CFIELD:
10.6.2.1306.40 EKEY>FKEY
10.6.2.1336 END-STRUCTURE
10.6.2.1518 FIELD:
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

C.8.11 File-Access word sets

The following words have been added to .6.2File-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 12.6.2Floating-Point extension words:

12.6.2.1207.40 DFFIELD:
12.6.2.1471 F>S
12.6.2.1517 FFIELD:
12.6.2.1627 FTRUNC
12.6.2.1628 FVALUE
12.6.2.2175 S>F
12.6.2.2206.40 SFFIELD:

C.8.13 Locals word sets

The following words have been added to 13.6.2Locals extension words:

13.6.2.2550 {:

C.8.15 Programming-Tools word sets

The following words have been added to the 15.6.2Programming-Tools extension words:

15.6.2.1908 N>R
15.6.2.1909.10 NAME>COMPILE
15.6.2.1909.20 NAME>INTERPRET
15.6.2.1909.40 NAME>STRING
15.6.2.1940 NR>
15.6.2.2264 SYNONYM
15.6.2.2297 TRAVERSE-WORDLIST
15.6.2.2530.30 [DEFINED]
15.6.2.2534 [UNDEFINED]

C.8.17 String word sets

The following words have been added to the 17.6.2String 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 18The optional Extended-Character word set:

18.6.1.2486.50 X-SIZE
18.6.1.2487.10 XC!+
18.6.1.2487.15 XC!+?
18.6.1.2487.20 XC,
18.6.1.2487.25 XC-SIZE
18.6.1.2487.35 XC@+
18.6.1.2487.40 XCHAR+
18.6.1.2488.10 XEMIT
18.6.1.2488.30 XKEY
18.6.1.2488.35 XKEY?
18.6.2.0145 +X/STRING
18.6.2.0175 -TRAILING-GARBAGE

The following words make up 18.6.2Extended-Character extension words:

18.6.2.0895 CHAR
18.6.2.1306.60 EKEY>XCHAR
18.6.2.2008 PARSE
18.6.2.2486.70 X-WIDTH
18.6.2.2487.30 XC-WIDTH
18.6.2.2487.45 XCHAR-
18.6.2.2488.20 XHOLD
18.6.2.2495 X\STRING-
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 3.3.3.1Address 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.

Editor:
This would be a good place to add a discussion of characters and the extended character word set.

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 3.2.3.3Return 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)
Reference Implementations

E.1 Introduction

In the most recent review of this document, proposals were encouraged to include a reference implementation where possible. Where an implementation requires system specific knowledge it was documented.

This appendix contains the reference implementations that have been accepted by the committee. This is not a complete reference implementation nor do the committee recommend these implementations. They are supplied solely for the purpose of providing a detailed understanding of a definitions requirement. System implementors are free to implement any operation in a manner that suits their system~~, but it must exhibit the same behavior as the reference implementation given here~~ .

x:ref-license

We give permission to use these implementations under CC0 1.0 [1]. We encourage you to improve on them as some of them are limited, system-specific and/or may contain bugs. They are distributed in the hope that they will be useful.

[1] https:://creativecommons.org/publicdomain/zero/1.0/

E.6 The Core word set

E.6.1.1910 NEGATE

```
: NEGATE ( n1 -- n2 )
  INVERT 1+
;
```

E.6.1.2050 QUIT

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

This assumes the existence of a system-implementation word INTERPRET that embodies the text interpreter semantics described in 3.4The Forth text interpreter. Further discussion of the interpret loop can be found in A.6.2.0945 COMPILE,.

E.6.2.0698 ACTION-OF

```
: ACTION-OF
  STATE @ IF
     POSTPONE [’] POSTPONE DEFER@
  ELSE
     ’ DEFER@
  THEN ; IMMEDIATE
```

E.6.2.0825 BUFFER:

This implementation depends on children of CREATE returning an aligned address. Other memory location techniques require implementation-specific knowledge of the underlying Forth system.

```
: BUFFER: \ u "<name>" -- ; -- addr
\ Create a buffer of u address units whose address is returned at run time.
  CREATE ALLOT
;
```

E.6.2.1173 DEFER

```
: DEFER ( "name" -- )
  CREATE [’] ABORT ,
DOES> ( ... -- ... )
  @ EXECUTE ;
```

E.6.2.1175 DEFER!

```
: DEFER! ( xt2 xt1 -- )
  >BODY ! ;
```

E.6.2.1177 DEFER@

```
: DEFER@ ( xt1 -- xt2 )
  >BODY @ ;
```

E.6.2.1675 HOLDS

```
: HOLDS ( addr u -- )
  BEGIN DUP WHILE 1- 2DUP + C@ HOLD REPEAT 2DROP ;
```

E.6.2.1725 IS

```
: IS
  STATE @ IF
     POSTPONE [’] POSTPONE DEFER!
  ELSE
     ’ DEFER!
  THEN ; IMMEDIATE
```

E.6.2.2020 PARSE-NAME

```
: isspace? ( c -- f )
  BL 1+ U< ;

: isnotspace? ( c -- f )
  isspace? 0= ;
```

```
: xt-skip ( addr1 n1 xt -- addr2 n2 )
  \ skip all characters satisfying xt ( c -- f )
  >R
  BEGIN
     DUP
  WHILE
     OVER C@ R@ EXECUTE
  WHILE
     1 /STRING
  REPEAT THEN
  R> DROP ;

: parse-name ( "name" -- c-addr u )
  SOURCE >IN @ /STRING
  ['] isspace?  xt-skip OVER >R
  ['] isnotspace?  xt-skip ( end-word restlen r:  start-word )
  2DUP 1 MIN + SOURCE DROP - >IN !
  DROP R> TUCK - ;
```

E.6.2.2440 WITHIN

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

E.8 The optional Double-Number word set

E.8.6.1.1140 D>S

```
: D>S ( d -- n )
  DROP
;
```

E.8.6.2.0435 2VALUE

The implementation of TO to include 2VALUEs requires detailed knowledge of the host implementation of VALUE and TO, which is the main reason why 2VALUE should be standardized. The order in which the two cells are stored in memory is not specified in the definition for 2VALUE but this reference implementation has to assume one ordering — this is not intended to be definitive.

```
: 2VALUE ( x1 x2 -- )
  CREATE , ,
  DOES> 2@ ( -- x1 x2 )
;
```

The corresponding implementation of TO disregards the issue that TO must also work for integer VALUEs and locals.

```
: TO ( x1 x2 "<spaces>name" -- )
  ' >BODY
  STATE @ IF
```

```
      POSTPONE 2LITERAL POSTPONE 2!
    ELSE
      2!
    THEN
  ; IMMEDIATE
```

E.9 The Exception word set

E.9.6.1.0875 CATCH

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

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

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

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

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

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

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

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

This sample implementation does not explicitly handle the case in which CATCH has never been called (i.e., the ABORT behavior). One solution would be to execute a CATCH within QUIT, so that there is always an "exception handler of last resort" present, as shown in E.6.1.2050 QUIT.

E.9.6.1.2275 THROW

This is the counter part to E.9.6.1.0875 CATCH.

```
: THROW ( ???  exception# -- ???  exception# )
  ?DUP IF         ( exc# )      \ 0 THROW is no-op
    HANDLER @ RP!( exc# )       \ restore prev return stack
```

```
    R> HANDLER ! ( exc# )      \ restore prev handler
    R> SWAP >R   ( saved-sp ) \ exc# on return stack
    SP! DROP R>  ( exc# )      \ restore stack
    \ Return to the caller of CATCH because return
    \ stack is restored to the state that existed
    \ when CATCH began execution
  THEN
;
```

E.9.6.2.0670 ABORT

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

E.10 The optional Facility word set

E.10.6.2.0135 +FIELD

Create a new field within a structure definition of size *n* bytes.

```
: +FIELD  \ n <"name"> -- ; Exec:  addr -- 'addr
  CREATE OVER , +
  DOES> @ +
;
```

E.10.6.2.0763 BEGIN-STRUCTURE

Begin definition of a new structure. Use in the form BEGIN-STRUCTURE ⟨*name*⟩. At run time ⟨*name*⟩ returns the size of the structure.

```
: BEGIN-STRUCTURE  \ -- addr 0 ; -- size
  CREATE
    HERE 0 0 ,      \ mark stack, lay dummy
  DOES> @           \ -- rec-len
;
```

E.10.6.2.1306.40 EKEY>FKEY

The implementation is closely tied to the implementation of EKEY and therefore unportable.

E.10.6.2.1336 END-STRUCTURE

Terminate definition of a structure.

```
: END-STRUCTURE  \ addr n --
  SWAP ! ;         \ set len
```

E.11 The optional File-Access word set

E.11.6.2.1714 INCLUDE

```
: INCLUDE ( i*x "name" -- j*x )
  PARSE-NAME INCLUDED ;
```

E.11.6.2.2144.10 REQUIRE

```
: REQUIRE ( i*x "name" -- i*x )
  PARSE-NAME REQUIRED ;
```

E.11.6.2.2144.50 REQUIRED

This implementation does not implement the requirements with regard to MARKER and FORGET (REQUIRED only includes each file once, whether a marker was executed or not), so it is not a correct implementation on systems that support these words. It extends the definition of INCLUDED to record the name of files which have been either included or required previously. The names are recorded in a linked list held in the included-names variable.

```
: save-mem ( addr1 u -- addr2 u ) \ gforth
\ copy a memory block into a newly allocated region in the heap
  SWAP >R
  DUP ALLOCATE THROW
  SWAP 2DUP R> ROT ROT MOVE ;

: name-add ( addr u listp -- )
  >R save-mem ( addr1 u )
  3 CELLS ALLOCATE THROW \ allocate list node
  R@ @ OVER ! \ set next pointer
  DUP R> ! \ store current node in list var
  CELL+ 2! ;

: name-present?  ( addr u list -- f )
  ROT ROT 2>R BEGIN ( list R: addr u )
     DUP
  WHILE
     DUP CELL+ 2@ 2R@ COMPARE 0= IF
       DROP 2R> 2DROP TRUE EXIT
     THEN
     @
  REPEAT
  ( DROP 0 ) 2R> 2DROP ;

: name-join ( addr u list -- )
  >R 2DUP R@ @ name-present?  IF
     R> DROP 2DROP
  ELSE
     R> name-add
  THEN ;

VARIABLE included-names 0 included-names !

: included ( i*x addr u -- j*x )
  2DUP included-names name-join
  INCLUDED ;

: REQUIRED ( i*x addr u -- i*x )
  2DUP included-names @ name-present?  0= IF
     included
  ELSE
```

```
    2DROP
  THEN ;
```

E.12 The optional Floating-Point word set

E.12.6.2.1471 F>S

```
: F>S ( r -- n )
  F>D D>S
;
```

E.12.6.2.1627 FTRUNC

```
: FTRUNC ( r1 -- r2 )
       FDUP F0= 0=
  IF   FDUP F0<
  IF   FNEGATE FLOOR FNEGATE
  ELSE FLOOR
  THEN
  THEN ;
```

E.12.6.2.1628 FVALUE

The implementation of FVALUE requires detailed knowledge of the host implementation of VALUE and TO.

```
VARIABLE %var
: TO 1 %var ! ;

: FVALUE ( F: r -- ) ( "<spaces>name" -- )
  CREATE F,
  DOES> %var @ IF F! ELSE F@ THEN
        0 %var ! ;

: VALUE ( x "<spaces>name" -- )
  CREATE ,
  DOES> %var @ IF  ! ELSE  @ THEN
        0 %var ! ;
```

E.12.6.2.2175 S>F

```
: S>F ( n -- r )
  S>D D>F
;
```

E.13 The optional Locals word set

E.13.6.2.1795 LOCALS|

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

E.13.6.2.2550 {:

```
12345 CONSTANT undefined-value

: match-or-end?  ( c-addr1 u1 c-addr2 u2 -- f )
  2 PICK 0= >R COMPARE 0= R> OR ;

: scan-args
  \ 0 c-addr1 u1 -- c-addr1 u1 ...  c-addrn un n c-addrn+1 un+1
  BEGIN
    2DUP S" |"  match-or-end?  0= WHILE
    2DUP S" --" match-or-end?  0= WHILE
    2DUP S" :}" match-or-end?  0= WHILE
    ROT 1+ PARSE-NAME
  AGAIN THEN THEN THEN ;

: scan-locals
  \ n c-addr1 u1 -- c-addr1 u1 ...  c-addrn un n c-addrn+1 un+1
  2DUP S" |" COMPARE 0= 0= IF
    EXIT
  THEN
  2DROP PARSE-NAME
  BEGIN
    2DUP S" --" match-or-end?  0= WHILE
    2DUP S" :}" match-or-end?  0= WHILE
    ROT 1+ PARSE-NAME
    POSTPONE undefined-value
  AGAIN THEN THEN ;

: scan-end ( c-addr1 u1 -- c-addr2 u2 )
  BEGIN
    2DUP S" :}" match-or-end?  0= WHILE
    2DROP PARSE-NAME
  REPEAT ;

: define-locals ( c-addr1 u1 ...  c-addrn un n -- )
  0 ?DO
    (LOCAL)
  LOOP
  0 0 (LOCAL) ;

: {:  ( -- )
  0 PARSE-NAME
  scan-args scan-locals scan-end
  2DROP define-locals
; IMMEDIATE
```

E.15 The optional Programming-Tools word set

E.15.6.2.—— ;]

See E.15.6.2.0 [:.

x:cs-drop

E.15.6.2.—— CS-DROP

As standard systems are:

- free to choose an appropriate representation for control-flow dest and orig stack items and also
- free to choose the data stack as control-flow stack or a separate stack for this purpose

a standard definition for CS-DROP cannot be provided. However, it can be implemented easily with system specifing knowledge of the control-stack implementation.

For example SwiftForth uses a single cell on the data stack as control-flow items. A SwiftForth definition for CS-DROP, which also takes compiler security into account would be:

```
: CS-DROP ( C: orig | dest -- )
  DROP -BAL
;
```

Win32Forth uses two cells on the data stack as control-flow items including one cell for compiler security, so a defintion for CS-DROP in Win32Forth would be:

```
: CS-DROP ( C: orig | dest -- )
  2DROP
;
```

E.15.6.2.—— FIND-NAME

```
\ FIND-NAME and FIND-NAME-IN
\ mostly by Bernd Paysan
\ http://forth-standard.org/standard/core/FIND#reply-146

[DEFINED] gforth [IF]
  \ Gforth's implementation of {: ... uses a system-specific
  \ wordlist that does not work with the FIND-NAME-IN below.
  \ Therefore we use the reference implementation of {: instead.
  REQUIRE locals.fs
[THEN]

[UNDEFINED] bounds [IF]
   : bounds ( addr len -- addr+len addr )
     OVER + SWAP
   ;
[THEN]

[UNDEFINED] -rot [IF]
   : -rot ( a b c -- c a b )
     ROT ROT
   ;
[THEN]

: >lower ( c1 -- c2 )
  DUP 'A' 'Z' 1+ WITHIN BL AND OR
;
```

```
: istr= ( addr1 u1 addr2 u2 -- flag )
   ROT OVER <> IF 2DROP FALSE EXIT THEN
   bounds ?DO
     DUP C@ >lower I C@ >lower <> IF DROP FALSE UNLOOP EXIT THEN
     1+
   LOOP
   DROP TRUE
;

: find-name-in-helper ( addr u wid -- nt / 0 )
   DUP >R NAME>STRING 2OVER istr= IF
     ROT DROP R> -rot FALSE
   ELSE
     R> DROP TRUE
   THEN
;

: FIND-NAME-IN ( addr u wid -- nt / 0 )
   >R 0 -rot R>
   ['] find-name-in-helper
   SWAP TRAVERSE-WORDLIST 2DROP
;

: FIND-NAME {: c-addr u -- nt | 0 :}
   GET-ORDER
   [DEFINED] gforth [IF]
     [ ' locals >BODY ] LITERAL SWAP 1+
   [ELSE] [DEFINED] vfxforth [IF]
       [ ' localvars >BODY 3 CELLS + ] LITERAL SWAP 1+
     [ELSE]
       CR .( warning: find-name does not find locals )
     [THEN]
   [THEN]
   0 SWAP 0 ?DO ( widn ... widi nt | 0)
     DUP 0= IF
       DROP c-addr u ROT FIND-NAME-IN
     ELSE
       NIP
     THEN
   LOOP
;
```

E.15.6.2.—— FIND-NAME-IN

See E.15.6.2.0 FIND-NAME.

E.15.6.2.1908 N>R

This implementation depends on the return address being on the return stack.

```
: N>R \ xn ..  x1 N -- ; R: -- x1 ..  xn n
\ Transfer N items and count to the return stack.
  DUP                   \ xn ..  x1 N N --
  BEGIN
    DUP
  WHILE
    ROT R> SWAP >R >R   \ xn ..  N N -- ; R: ..  x1 --
    1-                  \ xn ..  N 'N -- ; R: ..  x1 --
  REPEAT
  DROP                  \ N -- ; R: x1 ..  xn --
  R> SWAP >R >R
;
```

E.15.6.2.1940 NR>

This implementation depends on the return address being on the return stack.

```
: NR> \ -- xn ..  x1 N ; R: x1 ..  xn N --
\ Pull N items and count off the return stack.
  R> R> SWAP >R DUP
  BEGIN
    DUP
  WHILE
    R> R> SWAP >R -ROT
    1-
  REPEAT
  DROP
;
```

x:synonym

E.15.6.2.2264 SYNONYM

~~The implementation of SYNONYM requires detailed knowledge of the host implementation, which is one reason why it should be standardized. The implementation below is imperfect and specific to VFX Forth, in particular HIDE, REVEAL and IMMEDIATE? are non-standard words.~~

```
: SYNONYM \ "newname" "oldname" --
\ Create a new definition which redirects to an existing one.
  CREATE IMMEDIATE
     HIDE ' , REVEAL
  DOES>
     @ STATE @ 0= OVER IMMEDIATE? OR
     IF EXECUTE ELSE COMPILE, THEN
;
```

E.15.6.2.—— [:

It in not possible to define quotations in ISO Forth. The following is an outline definition where `save-definition-state` and `restore-definition-state` require carnal knowledge of the system and are left to the implementor.

```
: [: ( C: -- quotation-sys colon-sys )
```

```
  POSTPONE AHEAD save-definition-state :NONAME
; IMMEDIATE

: ;] ( C: quotation-sys colon-sys -- ) ( -- xt )
  POSTPONE ; >R restore-definition-state
  POSTPONE THEN R> POSTPONE LITERAL
; IMMEDIATE
```

E.15.6.2.2530.30 [DEFINED]

```
: [DEFINED] BL WORD FIND NIP 0<> ; IMMEDIATE
```

x:simplify-bracket-else

E.15.6.2.2531 [ELSE]

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

WORDLIST CONSTANT BRACKET-FLOW-WL
BRACKET-FLOW-WL GET-CURRENT SWAP SET-CURRENT
: [IF]   ( level1 -- level2 ) 1+ ;
: [ELSE] ( level1 -- level2 ) DUP 1 = IF 1- THEN ;
: [THEN] ( level1 -- level2 ) 1- ;
SET-CURRENT

: [ELSE] ( -- )
   1 BEGIN BEGIN PARSE-NAME DUP WHILE
     BRACKET-FLOW-WL SEARCH-WORDLIST IF
       EXECUTE DUP 0= IF DROP EXIT THEN
     THEN
   REPEAT 2DROP REFILL 0= UNTIL DROP
; IMMEDIATE
```

E.15.6.2.2532 [IF]

```
: [IF] ( flag -- )
```

```
  0= IF POSTPONE [ELSE] THEN
; IMMEDIATE
```

E.15.6.2.2533 [THEN]

```
: [THEN] ( -- ) ; IMMEDIATE
```

E.15.6.2.2534 [UNDEFINED]

```
: [UNDEFINED] BL WORD FIND NIP 0= ; IMMEDIATE
```

E.16 The optional Search-Order word set

E.16.6.1.1180 DEFINITIONS

```
: discard ( x1 ...  xn u -- ) \ Drop u+1 stack items
  0 ?DO DROP LOOP
;

: DEFINITIONS ( -- )
  GET-ORDER SWAP SET-CURRENT discard
;
```

E.16.6.1.1550 FIND

Assuming `#order` and `context` are defined as per E.16.6.1.1647 GET-ORDER.

```
: FIND ( c-addr -- c-addr 0 | xt 1 | xt -1 )
  0                          ( c-addr 0 )
  #order @ 0 ?DO
    OVER COUNT               ( c-addr 0 c-addr' u )
    I CELLS context + @      ( c-addr 0 c-addr' u wid )
    SEARCH-WORDLIST          ( c-addr 0; 0 | w 1 | q -1 )
    ?DUP IF                  ( c-addr 0; w 1 | w -1 )
       2SWAP 2DROP LEAVE     ( w 1 | w -1 )
    THEN                     ( c-addr 0 )
  LOOP                       ( c-addr 0 | w 1 | w -1 )
;
```

E.16.6.1.1647 GET-ORDER

Here is a very simple search order implementation:

```
VARIABLE #order

CREATE context   16 ( wordlists ) CELLS ALLOT

: GET-ORDER ( -- wid1 ...  widn n )
  #order @ 0 ?DO
     #order @ I - 1- CELLS context + @
  LOOP
  #order @
;
```

E.16.6.1.2197 SET-ORDER

This is the complement of E.16.6.1.1647 GET-ORDER.

```
: SET-ORDER ( wid1 ...  widn n -0 )
  DUP -1 = IF
     DROP ⟨push system default word lists and n⟩
  THEN
  DUP #order !
  0 ?DO   I CELLS context + !   LOOP
;
```

E.16.6.2.0715 ALSO

```
: ALSO ( -- )
  GET-ORDER OVER SWAP 1+ SET-ORDER
;
```

E.16.6.2.1590 FORTH

```
: (wordlist) ( wid "<name>" -- ; )
  CREATE ,
  DOES>
     @ >R
     GET-ORDER NIP
     R> SWAP SET-ORDER
;

FORTH-WORDLIST (wordlist) FORTH
```

E.16.6.2.1965 ONLY

```
: ONLY ( -- ) -1 SET-ORDER ;
```

E.16.6.2.2037 PREVIOUS

```
: PREVIOUS ( -- ) GET-ORDER NIP 1- SET-ORDER ;
```

E.16.6.2.—— VOCABULARY

x:vocabulary

```
: VOCABULARY ( -- )
  WORDLIST CREATE ,
  DOES> ( -- )
     @ >R GET-ORDER SWAP DROP
     R> SWAP SET-ORDER
;
```

E.17 The optional String word set

E.17.6.2.2141 REPLACES

```
DECIMAL

[UNDEFINED] place [IF]
  : place   \  c-addr1 u c-addr2 --
  \ Copy the string described by c-addr1 u as a counted
  \ string at the memory address described by c-addr2.
```

```
    2DUP 2>R
    1 CHARS + SWAP MOVE
    2R> C!
  ;
[THEN]

: "/COUNTED-STRING" S" /COUNTED-STRING" ;
"/COUNTED-STRING" ENVIRONMENT? 0= [IF] 256 [THEN]
CHARS CONSTANT string-max

WORDLIST CONSTANT wid-subst
\ Wordlist ID of the wordlist used to hold substitution names and replacement text.

[DEFINED] VFXforth [IF] \ VFX Forth
  : makeSubst \ c-addr len -- c-addr
  \ Given a name string create a substution and storage space.
  \ Return the address of the buffer for the substitution text.
  \ This word requires system specific knowledge of the host Forth.
  \ Some systems may need to perform case conversion here.
    GET-CURRENT >R wid-subst SET-CURRENT
    ($create)                          \ like CREATE but takes c-addr/len
    R> SET-CURRENT
    HERE string-max ALLOT 0 OVER C! \ create buffer space
  ;
[THEN]

[DEFINED] (WID-CREATE) [IF] \ SwiftForth
  : makeSubst \ c-addr len -- c-addr
    wid-subst (WID-CREATE)             \ like CREATE but takes c-addr/len/wid
    LAST @ >CREATE !
    HERE string-max ALLOT 0 OVER C! \ create buffer space
  ;
[THEN]

: findSubst \ c-addr len -- xt flag | 0
\ Given a name string, find the substitution.
\ Return xt and flag if found, or just zero if not found.
\ Some systems may need to perform case conversion here.
  wid-subst SEARCH-WORDLIST
;

: REPLACES \ text tlen name nlen --
\ Define the string text/tlen as the text to substitute for the substitution named name/nlen.
\ If the substitution does not exist it is created.
  2DUP findSubst IF
    NIP NIP EXECUTE    \ get buffer address
  ELSE
    makeSubst
  THEN
```

```
  place                 \ copy as counted string
;
```

E.17.6.2.2255 SUBSTITUTE

Assuming E.17.6.2.2141 REPLACES has been defined.

```
[UNDEFINED] bounds [IF]
  : bounds   \ addr len -- addr+len addr
     OVER + SWAP
  ;
[THEN]

[UNDEFINED] -rot [IF]
  : -rot   \ a b c -- c a b
     ROT ROT
  ;
[THEN]

CHAR % CONSTANT delim     \ Character used as the substitution name delimiter.
string-max BUFFER: Name   \ Holds substitution name as a counted string.
VARIABLE DestLen          \ Maximum length of the destination buffer.
2VARIABLE Dest            \ Holds destination string current length and address.
VARIABLE SubstErr         \ Holds zero or an error code.

: addDest \ char --
\ Add the character to the destination string.
  Dest @ DestLen @ < IF
     Dest 2@ + C! 1 CHARS Dest +!
  ELSE
     DROP -1 SubstErr !
  THEN
;

: formName \ c-addr len -- c-addr' len'
\ Given a source string pointing at a leading delimiter, place the name string in the name buffer.
  1 /STRING 2DUP delim scan >R DROP \ find length of residue
  2DUP R> - DUP >R Name place       \ save name in buffer
  R> 1 CHARS + /STRING              \ step over name and trailing %
;

: >dest \ c-addr len --
\ Add a string to the output string.
  bounds ?DO
     I C@ addDest
  1 CHARS +LOOP
;

: processName \ -- flag
\ Process the last substitution name. Return true if found, 0 if not found.
```

```
  Name COUNT findSubst DUP >R IF
     EXECUTE COUNT >dest
  ELSE
     delim addDest Name COUNT >dest delim addDest
  THEN
  R>
;

: SUBSTITUTE \ src slen dest dlen -- dest dlen' n
\ Expand the source string using substitutions.
\ Note that this version is simplistic, performs no error checking,
\ and requires a global buffer and global variables.
  Destlen ! 0 Dest 2! 0 -rot \ -- 0 src slen
  0 SubstErr !
  BEGIN
     DUP 0 >
  WHILE
     OVER C@ delim <> IF              \ character not %
       OVER C@ addDest 1 /STRING
     ELSE
       OVER 1 CHARS + C@ delim = IF   \ %% for one output %
         delim addDest 2 /STRING      \ add one % to output
       ELSE
         formName processName IF
           ROT 1+ -rot                \ count substitutions
         THEN
       THEN
     THEN
  REPEAT
  2DROP Dest 2@ ROT SubstErr @ IF
     DROP SubstErr @
  THEN
;
```

E.17.6.2.2375 UNESCAPE

```
: UNESCAPE \ c-addr1 len1 c-addr2 -- c-addr2 len2
\ Replace each '%' character in the input string c-addr1 len1 with two '%' characters.
\ The output is represented by c-addr2 len2.
\ If you pass a string through UNESCAPE and then SUBSTITUTE, you get the original string.
  DUP 2SWAP OVER + SWAP ?DO
     I C@ [CHAR] % = IF
       [CHAR] % OVER C! 1+
     THEN
     I C@ OVER C! 1+
  LOOP
  OVER -
```

```
;
```

E.18 The optional Extended-Character word set

This reference implementation assumes the UTF-8 character encoding is being used.

E.18.6.1.2486.50 X-SIZE

```
: X-SIZE ( xc-addr u1 -- u2 )
  0=          IF DROP 0 EXIT THEN
  \ length of UTF-8 char starting at u8-addr (accesses only u8-addr)
  C@
  DUP $80 U< IF DROP 1 EXIT THEN
  DUP $c0 U< IF -77 THROW   THEN
  DUP $e0 U< IF DROP 2 EXIT THEN
  DUP $f0 U< IF DROP 3 EXIT THEN
  DUP $f8 U< IF DROP 4 EXIT THEN
  DUP $fc U< IF DROP 5 EXIT THEN
  DUP $fe U< IF DROP 6 EXIT THEN
  -77 THROW ;
```

E.18.6.1.2487.10 XC!+

```
: XC!+ ( xchar xc-addr -- xc-addr' )
  OVER $80 U< IF TUCK C! CHAR+ EXIT THEN \ special case ASCII
  >R 0 SWAP $3F
  BEGIN 2DUP U> WHILE
     2/ >R DUP $3F AND $80 OR SWAP 6 RSHIFT R>
  REPEAT $7F XOR 2* OR R>
  BEGIN OVER $80 U< 0= WHILE TUCK C! CHAR+ REPEAT NIP
;
```

E.18.6.1.2487.15 XC!+?

```
: XC!+? ( xchar xc-addr u -- xc-addr' u' flag )
  >R OVER XC-SIZE R@ OVER U< IF ( xchar xc-addr1 len r:  u1 )
     \ not enough space
     DROP NIP R> FALSE
  ELSE
     >R XC!+ R> R> SWAP - TRUE
  THEN ;
```

E.18.6.1.2487.20 XC,

```
: XC, ( xchar -- ) HERE XC!+ DP ! ;
```

E.18.6.1.2487.25 XC-SIZE

```
: XC-SIZE ( xchar -- n )
  DUP $80 U< IF DROP 1 EXIT THEN \ special case ASCII
  $800 2 >R
  BEGIN 2DUP U>= WHILE 5 LSHIFT R> 1+ >R DUP 0= UNTIL THEN
  2DROP R>
;
```

E.18.6.1.2487.35 XC@+

```
: XC@+ ( xc-addr -- xc-addr' u )
  COUNT DUP $80 U< IF EXIT THEN \ special case ASCII
  $7F AND $40 >R
  BEGIN DUP R@ AND WHILE R@ XOR
     6 LSHIFT R> 5 LSHIFT >R >R COUNT
     $3F AND R> OR
  REPEAT R> DROP
;
```

E.18.6.1.2487.40 XCHAR+

```
: XCHAR+ ( xc-addr -- xc-addr' ) XC@+ DROP ;
```

E.18.6.1.2488.10 XEMIT

```
: XEMIT ( xchar -- )
  DUP $80 U< IF EMIT EXIT THEN \ special case ASCII
  0 SWAP $3F
  BEGIN 2DUP U> WHILE
     2/ >R DUP $3F AND $80 OR SWAP 6 RSHIFT R>
  REPEAT $7F XOR 2* OR
  BEGIN DUP $80 U< 0= WHILE EMIT REPEAT DROP
;
```

E.18.6.1.2488.30 XKEY

```
: XKEY ( -- xchar )
  KEY DUP $80 U< IF EXIT THEN \ special case ASCII
  $7F AND $40 >R
  BEGIN DUP R@ AND WHILE R@ XOR
     6 LSHIFT R> 5 LSHIFT >R >R KEY
     $3F AND R> OR
  REPEAT R> DROP ;
```

E.18.6.2.0145 +X/STRING

```
: +X/STRING ( xc-addr1 u1 -- xc-addr2 u2 )
  OVER DUP XCHAR+ SWAP - /STRING ;
```

E.18.6.2.0175 -TRAILING-GARBAGE

```
: -TRAILING-GARBAGE ( xc-addr u1 -- xc-addr u2 )
  2DUP + DUP XCHAR- ( addr u1 end1 end2 )
  2DUP DUP OVER OVER - X-SIZE + = IF \ last xchar ok
     2DROP
  ELSE
     NIP NIP OVER -
  THEN ;
```

E.18.6.2.0895 CHAR

```
: CHAR ( "name" -- xchar ) BL WORD COUNT DROP XC@+ NIP ;
```

E.18.6.2.2486.70 X-WIDTH

```
: X-WIDTH ( xc-addr u -- n )
  0 ROT ROT OVER + SWAP ?DO
     I XC@+ SWAP >R XC-WIDTH +
  R> I - +LOOP ;
```

E.18.6.2.2487.30 XC-WIDTH

```
: wc, ( n low high -- ) 1+ , , , ;

CREATE wc-table \ derived from wcwidth source code, for UCS32
0  0300  0357 wc,     0  035D  036F wc,     0  0483  0486 wc,
0  0488  0489 wc,     0  0591  05A1 wc,     0  05A3  05B9 wc,
0  05BB  05BD wc,     0  05BF  05BF wc,     0  05C1  05C2 wc,
0  05C4  05C4 wc,     0  0600  0603 wc,     0  0610  0615 wc,
0  064B  0658 wc,     0  0670  0670 wc,     0  06D6  06E4 wc,
0  06E7  06E8 wc,     0  06EA  06ED wc,     0  070F  070F wc,
0  0711  0711 wc,     0  0730  074A wc,     0  07A6  07B0 wc,
0  0901  0902 wc,     0  093C  093C wc,     0  0941  0948 wc,
0  094D  094D wc,     0  0951  0954 wc,     0  0962  0963 wc,
0  0981  0981 wc,     0  09BC  09BC wc,     0  09C1  09C4 wc,
0  09CD  09CD wc,     0  09E2  09E3 wc,     0  0A01  0A02 wc,
0  0A3C  0A3C wc,     0  0A41  0A42 wc,     0  0A47  0A48 wc,
0  0A4B  0A4D wc,     0  0A70  0A71 wc,     0  0A81  0A82 wc,
0  0ABC  0ABC wc,     0  0AC1  0AC5 wc,     0  0AC7  0AC8 wc,
0  0ACD  0ACD wc,     0  0AE2  0AE3 wc,     0  0B01  0B01 wc,
0  0B3C  0B3C wc,     0  0B3F  0B3F wc,     0  0B41  0B43 wc,
0  0B4D  0B4D wc,     0  0B56  0B56 wc,     0  0B82  0B82 wc,
0  0BC0  0BC0 wc,     0  0BCD  0BCD wc,     0  0C3E  0C40 wc,
0  0C46  0C48 wc,     0  0C4A  0C4D wc,     0  0C55  0C56 wc,
0  0CBC  0CBC wc,     0  0CBF  0CBF wc,     0  0CC6  0CC6 wc,
0  0CCC  0CCD wc,     0  0D41  0D43 wc,     0  0D4D  0D4D wc,
0  0DCA  0DCA wc,     0  0DD2  0DD4 wc,     0  0DD6  0DD6 wc,
0  0E31  0E31 wc,     0  0E34  0E3A wc,     0  0E47  0E4E wc,
0  0EB1  0EB1 wc,     0  0EB4  0EB9 wc,     0  0EBB  0EBC wc,
0  0EC8  0ECD wc,     0  0F18  0F19 wc,     0  0F35  0F35 wc,
0  0F37  0F37 wc,     0  0F39  0F39 wc,     0  0F71  0F7E wc,
0  0F80  0F84 wc,     0  0F86  0F87 wc,     0  0F90  0F97 wc,
0  0F99  0FBC wc,     0  0FC6  0FC6 wc,     0  102D  1030 wc,
0  1032  1032 wc,     0  1036  1037 wc,     0  1039  1039 wc,
0  1058  1059 wc,     1  0000  1100 wc,     2  1100  115f wc,
0  1160  11FF wc,     0  1712  1714 wc,     0  1732  1734 wc,
0  1752  1753 wc,     0  1772  1773 wc,     0  17B4  17B5 wc,
0  17B7  17BD wc,     0  17C6  17C6 wc,     0  17C9  17D3 wc,
0  17DD  17DD wc,     0  180B  180D wc,     0  18A9  18A9 wc,
0  1920  1922 wc,     0  1927  1928 wc,     0  1932  1932 wc,
0  1939  193B wc,     0  200B  200F wc,     0  202A  202E wc,
```

```
0  2060  2063 wc,     0  206A  206F wc,     0  20D0  20EA wc,
2  2329  232A wc,     0  302A  302F wc,     2  2E80  303E wc,
0  3099  309A wc,     2  3040  A4CF wc,     2  AC00  D7A3 wc,
2  F900  FAFF wc,     0  FB1E  FB1E wc,     0  FE00  FE0F wc,
0  FE20  FE23 wc,     2  FE30  FE6F wc,     0  FEFF  FEFF wc,
2  FF00  FF60 wc,     2  FFE0  FFE6 wc,     0  FFF9  FFFB wc,
0 1D167 1D169 wc,     0 1D173 1D182 wc,     0 1D185 1D18B wc,
0 1D1AA 1D1AD wc,     2 20000 2FFFD wc,     2 30000 3FFFD wc,
0 E0001 E0001 wc,     0 E0020 E007F wc,     0 E0100 E01EF wc,
HERE wc-table - CONSTANT #wc-table
```

\ inefficient table walk:

```
: XC-WIDTH ( xchar -- n )
  wc-table #wc-table OVER + SWAP ?DO
     DUP I 2@ WITHIN IF DROP I 2 CELLS + @ UNLOOP EXIT THEN
  3 CELLS +LOOP DROP 1 ;
```

E.18.6.2.2487.45 XCHAR-

```
: XCHAR- ( xc-addr -- xc-addr' )
  BEGIN 1 CHARS - DUP C@ $C0 AND $80 <> UNTIL ;
```

E.18.6.2.2488.20 XHOLD

```
CREATE xholdbuf 8 ALLOT

: XHOLD ( xchar -- ) xholdbuf TUCK XC!+ OVER - HOLDS ;
```

E.18.6.2.2495 X\STRING-

```
: X\STRING- ( xc-addr u -- xc-addr u' )
  OVER + XCHAR- OVER - ;
```

E.18.6.2.2520 [CHAR]

```
: [CHAR] ( "name" -- rt:xchar )
  CHAR POSTPONE LITERAL ; IMMEDIATE
```

Annex F
(informative)
Test Suite

F.1 Introduction

After the publication of the ANS Forth document (ANSI X3.215-1994), John Hayes developed a test suite, which included both a test harness and a suite of core tests. The harness was extended by Anton Ertl and David N. Williams to allow the testing of floating point operations. The current revision of the test harness is available from the web site:

```
http://www.forth200x.org/tests/ttester.fs
```

The ~~teat~~ test harness can be used to define regression tests for a set of application words. It can also be used to define tests of words in a standard-conforming implementation. ed21

F.2 Test Harness

The tester defines functions that compare the results of a test with a set of expected results. The syntax for each test starts with "`T{`" (T-open brace) followed by a code sequence to test. This is followed by "`->`", the expected results, and "`}T`" (close brace-T). For example, the following:

```
T{ 1 1 + -> 2 }T
```

tests that one plus one indeed equals two.

The "`T{`" records the stack depth prior to the test code so that they can be eliminated from the test. The "`->`" records the stack depth and moves the entire stack contents to an array. In the example test, the recorded stack depth is one and the saved array contains one value, two. The "`}T`" compares the current stack depth to the saved stack depth. If they are equal each value on the stack is removed from the stack and compared to its corresponding value in the array. If the depths are not equal or if the stack comparison fails, an error is reported. For example:

```
T{ 1 2 3 SWAP -> 1 3 2 }T
T{ 1 2 3 SWAP -> 1 2 3 }T INCORRECT RESULT: T{ 1 2 3 SWAP -> 1 2 3 }T
T{ 1 2 SWAP -> 1 }T WRONG NUMBER OF RESULTS: T{ 1 2 SWAP -> 1 }T
```

F.2.1 Floating-Point

Floating point testing can involve further complications. The harness attempts to determine whether floating-point support is present, and if so, whether there is a separate floating-point stack, and behave accordingly. The `CONSTANT`s `HAS-FLOATING` and `HAS-FLOATING-STACK` contain the results of its efforts, so the behavior of the code can be modified by the user if necessary.

Then there are the perennial issues of floating point value comparisons. Exact equality is specified by `SET-EXACT` (the default). If approximate equality tests are desired, execute `SET-NEAR`. Then the `FVARIABLE`s `REL-NEAR` (default 1E-12) and `ABS-NEAR` (default 0E) contain the values to be used in comparisons by the (internal) word `FNEARLY=`.

When there is not a separate floating point stack, and you want to use approximate equality for FP values, it is necessary to identify which stack items are floating point quantities. This can be done by replacing the closing `}T` with a version that specifies this, such as `RRXR}T` which identifies the stack picture

(*r r x r*). The harness provides such words for all combinations of R and X with up to four stack items. They can be used with either an integrated or a separate floating point stacks. Adding more if you need them is straightforward; see the examples in the source. Here is an example which also illustrates controlling the precision of comparisons:

```
SET-NEAR
1E-6 REL-NEAR F!
T{ S" 3.14159E" >FLOAT -> -1E FACOS <TRUE> RX}T
```

F.2.2 Error Processing

The internal word ERROR is vectored, through the ERROR-XT variable, so that its action can be changed by the user (for example, to add a counter for the number of errors). The default action ERROR1 can be used as a factor in the display of error reports.

F.3 Preliminary Testing

The test harness needs a lot of a Forth system to be working before the test programs can be run. This detracts from the progressive nature of the Hayes core tests where words are tested before being used. The tester requires the following to be fully working:

- 32 words from the Core word set
- 4 words from the Core Extension word set
- some rather complex colon definitions
- the word INCLUDED or equivalent (not essential but desirable)

This is perfectly acceptable for a working system being re-tested but not for a new system under development where it is likely that the tester itself has to be debugged before testing can proceed. What is needed for such a system is a bootstrap program that assumes as little as possible of the system is proven and which tests, in a rudimentary way if not exhaustively, all the words used in the test harness.

F.3.1 Minimal requirements

At the very least for manual or automated testing, the Forth outer text interpreter has to be working which implies that the system can:

- accept a line of input from the keyboard/stdin
- display some text on the system monitor
- parse a line of text into space delimited words, this implies that words >IN and WORD work and that SOURCE is available
- search the Forth dictionary for a word, hence FIND or a factor of FIND works
- if the word is found execute it
- otherwise recognize it as a valid integer and convert it using BASE
- otherwise report an “undefined word“ error or fail in some other way
- have working stacks (both data and return)

To avoid tests being typed in it helps if INCLUDED works so that a file can be interpreted. That is not essential as input can be redirected from a file or pasted in to the command window but in the latter cases

it can be difficult to inspect output from the tests due to the interpreted text being displayed between test results. A working INCLUDED is recommended. This file is the documentation for the test program.

F.3.2 Assumptions

1) The outer text interpreter must be functional using at least input from a keyboard/stdin or, preferably a file using INCLUDED or similar. In particular the words SOURCE, TYPE and CR need to work for the initial tests.
2) All core words used in the test harness are available for testing.
3) In the early tests success will be verified by the user inspecting a "Pass" message on the system display until such time as sufficient words have been defined and tested to enable automated testing.
4) If the early tests fail the Forth system reaction is unpredictable and will probably do at least one of:
   - display an error message e.g. "undefined word"
   - crash in some way e.g. enter an infinite loop, access an illegal address, stop running
   - ignore the failure and carry on so that the user can see that the "Pass" message is missing.
5) The system will only display the "Pass" message if the test has actually passed.
6) Tests on standard words will be rudimentary rather than exhaustive, just testing the basic operation.

F.3.2.1 Restrictions

1) Only words from the Core word set and will be used in the test harness, any others will have to be defined unless already defined - this particularly applies to the four Core Extension words used in the harness. This restriction unfortunately rules out the use of ', .(, PARSE and a few other useful words.
2) Only absolutely necessary words will be defined.
3) Check that true and false flags are all 1's or 0's respectively.
4) Make the tests progressive i.e. test before use.

F.3.2.2 The test sequence

1) The first tests simply reproduce the line of Forth code, if they display correctly then the test has passed, there is no "Pass" message.
2) The next few tests are checked by visual inspection of the display which, if passed, will display a line beginning:

   Pass #nn: testing xyz

   where nn is a sequential number used to identify failures. The first few tests have such a message in parentheses.
3) As soon as possible a word is defined that checks the test result and reports errors rather than relying on visual inspection. To avoid having to use IF the check manipulates >IN to implement an interpretive if.
4) The error reporting from point 3 is used to test the rest of the words used in the harness.

5) If the four Core Extension words are not available they are defined.

6) An error report is provided at the end.

F.3.2.3 Notes

If the system under test cannot include files then some way will have to be found to input the test programs into the system. This is system dependent and hence not considered further here.

Initial testing of a result is done by the trick of incrementing `>IN`, a typical example is:

⟨some Forth code returning 0 or 1⟩ `>IN +! xSOURCE TYPE`

where 0 indicates failure and 1 success. If 0 then `>IN` is not incremented and the system tries to find `xSOURCE` and (hopefully) fails with an "undefined word" message. If 1 then `>IN` is incremented and the word `SOURCE` is found and executed and the line displayed. Several variations of this theme are used in the test program. If the Forth code returns some other number then `>IN` may index outside the boundaries of the input buffer and the resulting behavior is unpredictable. In the early tests it is likely that the pass message for the test will not be displayed and the error detected by the user. After `=` is proved to return a well-formed flag and the error reporting word defined and used, this problem no longer exists.

No assumptions have been made about the value held in `BASE` and integers bigger than 1 can't initially be used without possible error. Therefore `BASE` is set to 2 using the tested `1+` and then decimal 10 by using its binary value. The system is then in decimal mode and digits > 1 can be safely used.

An error reporting word is needed, `.(` and `PARSE` would be useful except that they are in the Core Extension word set, only `WORD` is available to parse a message and must be used in a colon definition. Therefore simple colon definitions are tested by simply defining `.SRC` to display the line of source code.

A word called `MSG` is defined to test `WORD` and `COUNT`, note that decimal 41 is the ASCII code for character '`)`'. When tested the address returned by `WORD` is system dependent and cannot be tested, is dropped.

A word `.MSG(` is defined to display parsed text thus removing the need to display the whole line of source code, reducing display noise. This is used until the error reporting word has been defined.

Other notes are included in the relevant section.

The final error report uses some low-level tricks to avoid having to use additional Forth words, this avoids the need to test such words before use.

F.3.2.4 Source

```
CR CR SOURCE TYPE ( Preliminary test ) CR
SOURCE ( These lines test SOURCE, TYPE, CR and parenthetic comments ) TYPE CR
( The next line of output should be blank to test CR ) SOURCE TYPE CR CR
```

It is now assumed that `SOURCE`, `TYPE`, `CR` and comments work. `SOURCE` and `TYPE` will be used to report test passes until something better can be defined to report errors. Until then reporting failures will depend on the system under test and will usually be via reporting an unrecognized word or possibly the system crashing. Tests will be numbered by #n from now on to assist fault finding. Test successes will be indicated by "Pass: #n . . . " and failures by "Error: #n . . . ".

```
( Initial tests of >IN +! and 1+ )
( Check that n >IN +! acts as an interpretive IF, where n >= 0 )
( Pass #1: testing 0 >IN +! ) 0 >IN +! SOURCE TYPE CR
```

```
( Pass #2: testing 1 >IN +! ) 1 >IN +! xSOURCE TYPE CR
( Pass #3: testing 1+ ) 1 1+ >IN +! xxSOURCE TYPE CR
```

Test results can now be reported using the >IN +! trick to skip 1 or more characters.

The value of BASE is unknown so it is not safe to use digits > 1, therefore it will be set it to binary and then decimal, this also tests @ and !.

```
( Pass #4: testing @ ! BASE ) 0 1+ 1+ BASE ! BASE @ >IN +! xxSOURCE TYPE CR
( Set BASE to decimal ) 1010 BASE !
( Pass #5: testing decimal BASE ) BASE @ >IN +! xxxxxxxxxxSOURCE TYPE CR
```

Now in decimal mode and digits >1 can be used.

A better error reporting word is needed, much like .(which can't be used as it is in the Core Extension word set, similarly PARSE can't be used either, only WORD is available to parse a message and must be used in a colon definition. Therefore a simple colon definition is tested next.

```
( Pass #6: testing : ; ) : .SRC SOURCE TYPE CR ; 6 >IN +! xxxxxx.SRC
( Pass #7: testing number input ) 19 >IN +! xxxxxxxxxxxxxxxxxxx.SRC

( VARIABLE is now tested as one will be used instead of DROP e.g. Y ! )

( Pass #8: testing VARIABLE ) VARIABLE Y 2 Y ! Y @ >IN +! xx.SRC

: MSG 41 WORD COUNT ; ( 41 is the ASCII code for right parenthesis )
( Next test MSG leaves 2 items on the data stack )
( Pass #9: testing WORD COUNT ) 5 MSG abcdef) Y ! Y ! >IN +! xxxxx.SRC
( Pass #10: testing WORD COUNT ) MSG ab) >IN +! xxY ! .SRC

( For reporting success .MSG( is now defined )
: .MSG( MSG TYPE ;
.MSG( Pass #11: testing WORD COUNT .MSG ) CR

( To define an error reporting word, = 2* AND will be needed, test them first )
1 1 = 1+ 1+ >IN +! x.MSG( Pass #12: testing = returns all 1's (-1) for true ) CR
1 0 = 1+ >IN +! x.MSG( Pass #13: testing = returns 0 for false ) CR
1 1 = -1 = 1+ 1+ >IN +! x.MSG( Pass #14: testing -1 interpreted correctly ) CR

1 2* >IN +! xx.MSG( Pass #15: testing 2* ) CR
-1 2* 1+ 1+ 1+ >IN +! x.MSG( Pass #16: testing 2* ) CR

-1 -1 AND 1+ 1+ >IN +! x.MSG( Pass #17: testing AND ) CR
-1 0 AND 1+ >IN +! x.MSG( Pass #18: testing AND ) CR
6 -1 AND >IN +! xxxxxx.MSG( Pass #19: testing AND ) CR

( Define ~ to use as a 'to end of line' comment. \ cannot be used as it a Core Extension word )
: ~( -- ) SOURCE >IN ! Y ! ;
```

Rather than relying on a pass message test words can now be defined to report errors in the event of a failure. For convenience words ?T~ and ?F~ are defined together with a helper ?~~ to test for TRUE and FALSE.

Usage is: ⟨test⟩ ?T~ Error #n: ⟨message⟩
Success makes >IN index the ~in ?T~ or ?F~ to skip the error message. Hence it is essential there is only 1 space between ?T~ and Error.

```
: ?~~( -1 | 0 --   )  2* >IN +! ;
: ?F~( f --   )   0 = ?~~ ;
: ?T~( f --   )  -1 = ?~~ ;

( Errors will be counted )
VARIABLE #ERRS 0 #ERRS !
: Error  1 #ERRS +! -6 >IN +! .MSG( CR ;
: Pass  -1 #ERRS +! 1 >IN +! Error ; ~ Pass is defined solely to test Error

-1 ?F~ Pass #20: testing ?F~ ?~~ Pass Error
-1 ?T~ Error #1: testing ?T~ ?~~ ~

 0 0 = 0= ?F~ Error #2: testing 0=
 1 0 = 0= ?T~ Error #3: testing 0=
-1 0 = 0= ?T~ Error #4: testing 0=

 0  0 = ?T~ Error #5: testing =
 0  1 = ?F~ Error #6: testing =
 1  0 = ?F~ Error #7: testing =
-1  1 = ?F~ Error #8: testing =
 1 -1 = ?F~ Error #9: testing =

-1 0< ?T~ Error #10: testing 0<
 0 0< ?F~ Error #11: testing 0<
 1 0< ?F~ Error #12: testing 0<

DEPTH 1+ DEPTH = ?~~ Error #13: testing DEPTH
```

Up to now whether the data stack was empty or not hasn't mattered as long as it didn't overflow. Now it will be emptied - also removing any unreported underflow.

```
DEPTH 0< 0= 1+ >IN +! ~ 0 0 >IN ! Remove any underflow
DEPTH 0= 1+ >IN +! ~ Y ! 0 >IN ! Empty the stack
DEPTH 0= ?T~ Error #14: data stack not emptied

4 -5 SWAP 4 = SWAP -5 = = ?T~ Error #15: testing SWAP
111 222 333 444
DEPTH 4 = ?T~ Error #16: testing DEPTH
444 = SWAP 333 = = DEPTH 3 = = ?T~ Error #17: testing SWAP DEPTH
222 = SWAP 111 = = DEPTH 1 = = ?T~ Error #18: testing SWAP DEPTH
DEPTH 0= ?T~ Error #19: testing DEPTH = 0
```

From now on the stack is expected to be empty after a test so ?~ will be defined to include a check on the stack depth. Note that ?~~ was defined and used earlier instead of ?~ to avoid (irritating) redefinition messages that many systems display had ?~ simply been redefined.

```
: ?~  ( -1 | 0 --   )  DEPTH 1 = AND ?~~ ; ~ -1 test success, 0 test failure
```

```
123 -1 ?~ Pass #21: testing ?~
Y ! ~ equivalent to DROP
```

Testing the remaining Core words used in the harness, with the above definitions these are straightforward.

```
1 DROP DEPTH 0= ?~ Error #20: testing DROP
123 DUP = ?~ Error #21: testing DUP

123 ?DUP  = ?~ Error #22: testing ?DUP
  0 ?DUP 0= ?~ Error #23: testing ?DUP

 123  111 +  234 = ?~ Error #24: testing +
 123 -111 +   12 = ?~ Error #25: testing +
-123  111 +  -12 = ?~ Error #26: testing +
-123 -111 + -234 = ?~ Error #27: testing +

 -1 NEGATE    1 = ?~ Error #28: testing NEGATE
  0 NEGATE     0= ?~ Error #29: testing NEGATE
987 NEGATE -987 = ?~ Error #30: testing NEGATE

HERE DEPTH SWAP DROP 1 = ?~ Error #31: testing HERE
CREATE TST1 HERE TST1 = ?~ Error #32: testing CREATE HERE

 16 ALLOT HERE TST1 NEGATE + 16 = ?~ Error #33: testing ALLOT
-16 ALLOT HERE TST1 = ?~ Error #34: testing ALLOT

0 CELLS 0= ?~ Error #35: testing CELLS
1 CELLS ALLOT HERE TST1 NEGATE + VARIABLE CSZ CSZ !
CSZ @ 0= 0= ?~ Error #36: testing CELLS
3 CELLS CSZ @ DUP 2* + = ?~ Error #37: testing CELLS
-3 CELLS CSZ @ DUP 2* + + 0= ?~ Error #38: testing CELLS

: TST2 ( f -- n ) DUP IF 1+ THEN ;
0 TST2  0= ?~ Error #39: testing IF THEN
1 TST2 2 = ?~ Error #40: testing IF THEN

: TST3 ( n1 -- n2 ) IF 123 ELSE 234 THEN ;
0 TST3 234 = ?~ Error #41: testing IF ELSE THEN
1 TST3 123 = ?~ Error #42: testing IF ELSE THEN

: TST4 ( -- n ) 0 5 0 DO 1+ LOOP ;
TST4 5 = ?~ Error #43: testing DO LOOP

: TST5 ( -- n ) 0 10 0 DO I + LOOP ;
TST5 45 = ?~ Error #44: testing I

: TST6 ( -- n ) 0 10 0 DO DUP 5 = IF LEAVE ELSE 1+ THEN LOOP ;
TST6 5 = ?~ Error #45: testing LEAVE

: TST7 ( -- n1 n2 ) 123 >R 234 R> ;
TST7 NEGATE + 111 = ?~ Error #46: testing >R R>
```

```
: TST8 ( -- ch ) [CHAR] A ;
TST8 65 = ?~ Error #47: testing [CHAR]

: TST9 ( -- ) [CHAR] s [CHAR] s [CHAR] a [CHAR] P 4 0 DO EMIT LOOP ;
TST9 .MSG( #22: testing EMIT ) CR

: TST10 ( -- ) S" Pass #23: testing S" TYPE [CHAR] " EMIT CR ; TST10
```

The core test uses `CONSTANT` before it is tested therefore we test it here

```
1234 CONSTANT CTEST CTEST 1234 = ?~ Error #48: testing CONSTANT
```

The harness uses some words from the Core extension word set. These will be conditionally defined following definition of a word called `?DEFINED` to determine whether these are already defined.

```
VARIABLE TIMM1 0 TIMM1 !
: TIMM2 123 TIMM1 ! ; IMMEDIATE
: TIMM3 TIMM2 ; TIMM1 @ 123 = ?~ Error #49: testing IMMEDIATE

: ?DEFINED ( "name" -- 0 | -1 ) 32 WORD FIND SWAP DROP 0= 0= ;
?DEFINED SWAP ?~ Error #50: testing FIND ?DEFINED
?DEFINED <<no-such-word-hopefully>> 0= ?~ Error #51 testing FIND ?DEFINED

?DEFINED \ ?~ : \ ~ ; IMMEDIATE
\ Error #52: testing \
: TIMM4 \ Error #53: testing \ is IMMEDIATE
;
```

`TRUE` and `FALSE` are defined as colon definitions as they have been used more than `CONSTANT` above.

```
?DEFINED TRUE  ?~ : TRUE 1 NEGATE ;
?DEFINED FALSE ?~ : FALSE 0 ;
?DEFINED HEX   ?~ : HEX 16 BASE ! ;

TRUE -1 =           ?~ Error #54: testing TRUE
FALSE 0=            ?~ Error #55: testing FALSE
10 HEX 0A =         ?~ Error #56: testing HEX
AB 0A BASE ! 171 = ?~ Error #57: testing hex number
```

```
~ Delete the  on the next 2 lines to check the final error report
~ Error #998: testing a deliberate failure
~ Error #999: testing a deliberate failure
```

Describe the messages that should be seen. The previously defined `.MSG(` can be used for text messages.

```
CR .MSG( Results: ) CR
CR .MSG( Pass messages #1 to #23 should be displayed above)
CR .MSG( and no error messages) CR
```

Finally display a message giving the number of tests that failed. This is complicated by the fact that untested words including `.( ."` and `.` cannot be used. Also more colon definitions shouldn't be defined than are needed. To display a number, note that the number of errors will have one or two digits at most and an interpretive loop can be used to display those.

```
CR
0 #ERRS @
~ Loop to calculate the 10's digit (if any)
DUP NEGATE 9 + 0< NEGATE >IN +! ( -10 + SWAP 1+ SWAP 0 >IN ! )
~ Display the error count
SWAP ?DUP 0= 1+ >IN +! ( 48 + EMIT ( ) 48 + EMIT

.MSG( test) #ERRS @ 1 = 1+ >IN +! ~ .MSG( s)
.MSG( failed out of 57 additional tests) CR

CR CR .MSG( --- End of Preliminary Tests --- ) CR
```

F.3.3 Source

The following source code provides the test harness.

```
\ This is the source for the ANS test harness, it is based on the
\ harness originally developed by John Hayes

\ (C) 1995 JOHNS HOPKINS UNIVERSITY / APPLIED PHYSICS LABORATORY
\ MAY BE DISTRIBUTED FREELY AS LONG AS THIS COPYRIGHT NOTICE REMAINS.
\ VERSION 1.1

\ Revision history and possibly newer versions can be found at
\ http://www.forth200x/tests/ttester.fs

BASE @
HEX

VARIABLE ACTUAL-DEPTH          \ stack record
CREATE ACTUAL-RESULTS 20 CELLS ALLOT
VARIABLE START-DEPTH
VARIABLE XCURSOR               \ for ...}T
VARIABLE ERROR-XT

: ERROR ERROR-XT @ EXECUTE ;   \ for vectoring of error reporting

: "FLOATING" S" FLOATING" ;    \ only compiled S" in CORE
: "FLOATING-STACK" S" FLOATING-STACK" ;
"FLOATING" ENVIRONMENT? [IF]
   [IF]
     TRUE
   [ELSE]
     FALSE
   [THEN]
[ELSE]
   FALSE
[THEN] CONSTANT HAS-FLOATING
```

```
"FLOATING-STACK" ENVIRONMENT? [IF]
   [IF]
     TRUE
   [ELSE]
     FALSE
   [THEN]
[ELSE]             \ We don't know whether the FP stack is separate.
   HAS-FLOATING    \ If we have FLOATING, we assume it is.
[THEN] CONSTANT HAS-FLOATING-STACK

HAS-FLOATING [IF]
   \ Set the following to the relative and absolute tolerances you
   \ want for approximate float equality, to be used with F  in
   \ FNEARLY=.  Keep the signs, because F  needs them.
   FVARIABLE REL-NEAR DECIMAL 1E-12 HEX REL-NEAR F!
   FVARIABLE ABS-NEAR DECIMAL    0E HEX ABS-NEAR F!

   \ When EXACT? is TRUE, }F uses FEXACTLY=, otherwise FNEARLY=.

   TRUE VALUE EXACT?
   : SET-EXACT   ( -- )   TRUE TO EXACT? ;
   : SET-NEAR    ( -- )  FALSE TO EXACT? ;

   DECIMAL
   : FEXACTLY=   ( F: X Y -- S: FLAG )
     (
     Leave TRUE if the two floats are identical.
     )
     0E F~ ;
   HEX

   : FABS=   ( F: X Y -- S: FLAG )
     (
     Leave TRUE if the two floats are equal within the tolerance
     stored in ABS-NEAR.
     )
     ABS-NEAR F@ F~ ;

   : FREL=   ( F: X Y -- S: FLAG )
     (
     Leave TRUE if the two floats are relatively equal based on the
     tolerance stored in ABS-NEAR.
     )
     REL-NEAR F@ FNEGATE F~ ;

   : F2DUP  FOVER FOVER ;
```

```
   : F2DROP FDROP FDROP ;

   : FNEARLY=  ( F: X Y -- S: FLAG )
     (
     Leave TRUE if the two floats are nearly equal.  This is a
     refinement of Dirk Zoller's FEQ to also allow X = Y, including
     both zero, or to allow approximately equality when X and Y are too
     small to satisfy the relative approximation mode in the F~
     specification.
     )
     F2DUP FEXACTLY= IF F2DROP TRUE EXIT THEN
     F2DUP FREL=     IF F2DROP TRUE EXIT THEN
     FABS= ;

   : FCONF= ( R1 R2 -- F )
     EXACT? IF
        FEXACTLY=
     ELSE
        FNEARLY=
     THEN ;
[THEN]

HAS-FLOATING-STACK [IF]
   VARIABLE ACTUAL-FDEPTH
   CREATE ACTUAL-FRESULTS 20 FLOATS ALLOT
   VARIABLE START-FDEPTH
   VARIABLE FCURSOR

   : EMPTY-FSTACK ( ... -- ... )
     FDEPTH START-FDEPTH @ < IF
         FDEPTH START-FDEPTH @ SWAP DO 0E LOOP
     THEN
     FDEPTH START-FDEPTH @ > IF
         FDEPTH START-FDEPTH @ DO FDROP LOOP
     THEN ;

   : F{ ( -- )
     FDEPTH START-FDEPTH ! 0 FCURSOR ! ;

   : F-> ( ... -- ... )
     FDEPTH DUP ACTUAL-FDEPTH !
     START-FDEPTH @ > IF
        FDEPTH START-FDEPTH @ - 0 DO ACTUAL-FRESULTS I FLOATS + F! LOOP
     THEN ;

   : F} ( ... -- ... )
```

```
    FDEPTH ACTUAL-FDEPTH @ = IF
       FDEPTH START-FDEPTH @ > IF
          FDEPTH START-FDEPTH @ - 0 DO
             ACTUAL-FRESULTS I FLOATS + F@ FCONF= INVERT IF
                S" INCORRECT FP RESULT: " ERROR LEAVE
             THEN
          LOOP
       THEN
    ELSE
       S" WRONG NUMBER OF FP RESULTS: " ERROR
    THEN ;

  : F...}T ( -- )
    FCURSOR @ START-FDEPTH @ + ACTUAL-FDEPTH @ <> IF
      S" NUMBER OF FLOAT RESULTS BEFORE '->' DOES NOT MATCH ...}T "
      S" SPECIFICATION: " ERROR
    ELSE FDEPTH START-FDEPTH @ = 0= IF
      S" NUMBER OF FLOAT RESULTS BEFORE AND AFTER '->' DOES NOT MATCH: "
      ERROR
    THEN THEN ;

  : FTESTER ( R -- )
    FDEPTH 0= ACTUAL-FDEPTH @ FCURSOR @ START-FDEPTH @ + 1+ < OR IF
      S" NUMBER OF FLOAT RESULTS AFTER '->' BELOW ...}T SPECIFICATION: "
      ERROR
    ELSE ACTUAL-FRESULTS FCURSOR @ FLOATS + F@ FCONF= 0= IF
      S" INCORRECT FP RESULT: " ERROR
    THEN THEN
    1 FCURSOR +! ;

[ELSE]
  : EMPTY-FSTACK ;
  : F{ ;
  : F-> ;
  : F} ;
  : F...}T ;

  HAS-FLOATING [IF]
    DECIMAL
    : COMPUTE-CELLS-PER-FP ( -- U )
        DEPTH 0E DEPTH 1- >R FDROP R> SWAP - ;
    HEX

    COMPUTE-CELLS-PER-FP CONSTANT CELLS-PER-FP

    : FTESTER ( R -- )
```

```
        DEPTH CELLS-PER-FP <
        ACTUAL-DEPTH @ XCURSOR @ START-DEPTH @ + CELLS-PER-FP + <
        OR IF
           S" NUMBER OF RESULTS AFTER '->' BELOW ...}T SPECIFICATION: "
           ERROR EXIT
        ELSE ACTUAL-RESULTS XCURSOR @ CELLS + F@ FCONF= 0= IF
           S" INCORRECT FP RESULT: " ERROR
        THEN THEN
        CELLS-PER-FP XCURSOR +! ;
   [THEN]
[THEN]

: EMPTY-STACK \ ( ... -- ) empty stack; handles underflowed stack too.
   DEPTH START-DEPTH @ < IF
     DEPTH START-DEPTH @ SWAP DO 0 LOOP
   THEN
   DEPTH START-DEPTH @ > IF
     DEPTH START-DEPTH @ DO DROP LOOP
   THEN
   EMPTY-FSTACK ;

: ERROR1 \ ( C-ADDR U -- ) display an error message
         \ followed by the line that had the error.
   TYPE SOURCE TYPE CR \ display line corresponding to error
   EMPTY-STACK         \ throw away everything else
;

' ERROR1 ERROR-XT !

: T{ \ ( -- ) record the pre-test depth.
   DEPTH START-DEPTH ! 0 XCURSOR ! F{ ;

: -> \ ( ... -- ) record depth and contents of stack.
   DEPTH DUP ACTUAL-DEPTH ! \ record depth
   START-DEPTH @ > IF \ if there is something on the stack
     DEPTH START-DEPTH @ - 0 DO \ save them
        ACTUAL-RESULTS I CELLS + !
     LOOP
   THEN
   F-> ;

: }T \ ( ... -- ) comapre stack (expected) contents with saved
     \ (actual) contents.
   DEPTH ACTUAL-DEPTH @ = IF         \ if depths match
     DEPTH START-DEPTH @ > IF        \ if something on the stack
       DEPTH START-DEPTH @ - 0 DO    \ for each stack item
```

```
        ACTUAL-RESULTS I CELLS + @   \ compare actual with expected
        <> IF S" INCORRECT RESULT: " ERROR LEAVE THEN
      LOOP
    THEN
  ELSE                               \ depth mismatch
    S" WRONG NUMBER OF RESULTS: " ERROR
  THEN
  F} ;

: ...}T ( -- )
  XCURSOR @ START-DEPTH @ + ACTUAL-DEPTH @ <> IF
    S" NUMBER OF CELL RESULTS BEFORE '->' DOES NOT MATCH ...}T "
    S" SPECIFICATION: " ERROR
  ELSE DEPTH START-DEPTH @ = 0= IF
    S" NUMBER OF CELL RESULTS BEFORE AND AFTER '->' DOES NOT MATCH: "
    ERROR
  THEN THEN
  F...}T ;

: XTESTER ( X -- )
  DEPTH 0= ACTUAL-DEPTH @ XCURSOR @ START-DEPTH @ + 1+ < OR IF
    S" NUMBER OF CELL RESULTS AFTER '->' BELOW ...}T SPECIFICATION: "
    ERROR EXIT
  ELSE ACTUAL-RESULTS XCURSOR @ CELLS + @ <> IF
    S" INCORRECT CELL RESULT: " ERROR
  THEN THEN
  1 XCURSOR +! ;

: X}T    XTESTER                          ...}T ;
: XX}T   XTESTER XTESTER                  ...}T ;
: XXX}T  XTESTER XTESTER XTESTER          ...}T ;
: XXXX}T XTESTER XTESTER XTESTER XTESTER  ...}T ;

HAS-FLOATING [IF]
  : R}T    FTESTER                          ...}T ;
  : XR}T   FTESTER XTESTER                  ...}T ;
  : RX}T   XTESTER FTESTER                  ...}T ;
  : RR}T   FTESTER FTESTER                  ...}T ;
  : XXR}T  FTESTER XTESTER XTESTER          ...}T ;
  : XRX}T  XTESTER FTESTER XTESTER          ...}T ;
  : XRR}T  FTESTER FTESTER XTESTER          ...}T ;
  : RXX}T  XTESTER XTESTER FTESTER          ...}T ;
  : RXR}T  FTESTER XTESTER FTESTER          ...}T ;
  : RRX}T  XTESTER FTESTER FTESTER          ...}T ;
  : RRR}T  FTESTER FTESTER FTESTER          ...}T ;
  : XXXR}T FTESTER XTESTER XTESTER XTESTER  ...}T ;
```

```
   : XXRX}T XTESTER FTESTER XTESTER XTESTER ...}T ;
   : XXRR}T FTESTER FTESTER XTESTER XTESTER ...}T ;
   : XRXX}T XTESTER XTESTER FTESTER XTESTER ...}T ;
   : XRXR}T FTESTER XTESTER FTESTER XTESTER ...}T ;
   : XRRX}T XTESTER FTESTER FTESTER XTESTER ...}T ;
   : XRRR}T FTESTER FTESTER FTESTER XTESTER ...}T ;
   : RXXX}T XTESTER XTESTER XTESTER FTESTER ...}T ;
   : RXXR}T FTESTER XTESTER XTESTER FTESTER ...}T ;
   : RXRX}T XTESTER FTESTER XTESTER FTESTER ...}T ;
   : RXRR}T FTESTER FTESTER XTESTER FTESTER ...}T ;
   : RRXX}T XTESTER XTESTER FTESTER FTESTER ...}T ;
   : RRXR}T FTESTER XTESTER FTESTER FTESTER ...}T ;
   : RRRX}T XTESTER FTESTER FTESTER FTESTER ...}T ;
   : RRRR}T FTESTER FTESTER FTESTER FTESTER ...}T ;
[THEN]

\ Set the following flag to TRUE for more verbose output; this may
\ allow you to tell which test caused your system to hang.
VARIABLE VERBOSE
   FALSE VERBOSE !

: TESTING \ ( -- ) TALKING COMMENT.
   SOURCE VERBOSE @
   IF DUP >R TYPE CR R> >IN !
   ELSE >IN ! DROP
   THEN ;

BASE !
```

F.4 Core Tests

The test cases in John Hayes' original test suite were designed to test features before they were used in later tests. Due to the structure of this annex the progressive testing has been lost. This section attempts to retain the integrity of the original test suite by laying out the test progression for the core word set.

While this suite does test many aspects of the core word set, it is not comprehensive. A standard system *should* pass all of the tests within this suite. A system cannot claim to be standard simply because it passes this test suite.

The test starts by verifying basic assumptions about number representation. It then builds on this with tests of boolean logic, shifting, and comparisons. It then tests the basic stack manipulations and arithmetic. Ultimately, it tests the Forth interpreter and compiler.

Note that all of the tests in this suite assume the current base is *hexadecimal*.

F.4.1 Basic Assumptions

These test assume a two's complement implementation where the range of signed numbers is $-2^{n-1} \cdots 2^{n-1}-1$ and the range of unsinged numbers is $0 \cdots 2^n-1$.

A method for testing KEY, QUIT, ABORT, ABORT", ENVIRONMENT?, etc has yet to be proposed.

```
T{  ->  }T                  ( Start with a clean slate )
( Test if any bits are set; Answer in base 1 )
T{ : BITSSET? IF 0 0 ELSE 0 THEN ; ->  }T
T{  0 BITSSET? -> 0 }T      ( Zero is all bits clear )
T{  1 BITSSET? -> 0 0 }T    ( Other numbers have at least one bit )
T{ -1 BITSSET? -> 0 0 }T
```

F.4.2 Booleans

To test the booleans it is first neccessary to test F.6.1.0720 AND, and F.6.1.1720 INVERT. Before moving on to the test F.6.1.0950 CONSTANT. The latter defines two constants (0S and 1S) which will be used in the further test.

It is now possible to complete the testing of F.6.1.0720 AND, F.6.1.1980 OR, and F.6.1.2490 XOR.

F.4.3 Shifts

To test the shift operators it is necessary to calculate the most significant bit of a cell:

```
1S 1 RSHIFT INVERT CONSTANT MSB
```

RSHIFT is tested later. MSB must have at least one bit set:

```
T{ MSB BITSSET? -> 0 0 }T
```

The test F.6.1.0320 2*, F.6.1.0330 2/, F.6.1.1805 LSHIFT, and F.6.1.2162 RSHIFT can now be performed.

F.4.4 Numeric notation

The numeric representation can be tested with the following test cases:

```
DECIMAL
T{ #1289        -> 1289         }T
T{ #12346789.   -> 12346789.    }T
T{ #-1289       -> -1289        }T
T{ #-12346789.  -> -12346789.   }T
T{ $12eF        -> 4847         }T
T{ $12aBcDeF.   -> 313249263.   }T
T{ $-12eF       -> -4847        }T
T{ $-12AbCdEf.  -> -313249263.  }T
T{ %10010110    -> 150          }T
T{ %10010110.   -> 150.         }T
T{ %-10010110   -> -150         }T
T{ %-10010110.  -> -150.        }T
T{ 'z'          -> 122          }T
```

F.4.5 Comparisons

Before testing the comparison operators it is necessary to define a few constants to allow the testing of the upper and lower bounds.

```
0 INVERT                 CONSTANT MAX-UINT
0 INVERT 1 RSHIFT        CONSTANT MAX-INT
0 INVERT 1 RSHIFT INVERT CONSTANT MIN-INT
0 INVERT 1 RSHIFT        CONSTANT MID-UINT
```

```
0 INVERT 1 RSHIFT INVERT CONSTANT MID-UINT+1

0S CONSTANT <FALSE>
1S CONSTANT <TRUE>
```

With these constants defined, it is now possible to perform the F.6.1.0270 0=, F.6.1.0530 =, F.6.1.0250 0<, F.6.1.0480 <, F.6.1.0540 >, F.6.1.2340 U<, F.6.1.1880 MIN, and F.6.1.1870 MAX test.

F.4.6 Stack Operators

The stack operators can be tested without any prepatory work. The "normal" operators (F.6.1.1260 DROP, F.6.1.1290 DUP, F.6.1.1990 OVER, F.6.1.2160 ROT, and F.6.1.2260 SWAP) should be tested first, followed by the two-cell variants (F.6.1.0370 2DROP, F.6.1.0380 2DUP, F.6.1.0400 2OVER and F.6.1.0430 2SWAP) with F.6.1.0630 ?DUP and F.6.1.1200 DEPTH being performed last.

F.4.7 Return Stack Operators

The test F.6.1.0580 >R will test all three basic return stack operators (>R, R>, and R@).

F.4.8 Addition and Subtraction

Basic addition and subtraction should be tested in the order: F.6.1.0120 +, F.6.1.0160 −, F.6.1.0290 1+, F.6.1.0300 1−, F.6.1.0690 ABS and F.6.1.1910 NEGATE.

F.4.9 Multiplication

The multiplication operators should be tested in the order: F.6.1.2170 S>D, F.6.1.0090 *, F.6.1.1810 M*, and F.6.1.2360 UM*.

F.4.10 Division

Due to the complexity of the division operators they are tested separately from the multiplication operators. The basic division operators are tested first: F.6.1.1561 FM/MOD, F.6.1.2214 SM/REM, and F.6.1.2370 UM/MOD.

As the standard allows a system to provide either floored or symmetric division, the remaining operators have to be tested depending on the system behaviour. Two words are defined that provide a form of conditional compilation.

```
: IFFLOORED [ -3 2 / -2 = INVERT ] LITERAL IF POSTPONE \ THEN ;
: IFSYM     [ -3 2 / -1 = INVERT ] LITERAL IF POSTPONE \ THEN ;
```

IFSYM will ignore the rest of the line when it is performed on a system with floored division and perform the line on a system with symmetric division. IFFLOORED is the direct inverse, ignoring the rest of the line on systems with symmetric division and processing it on systems with floored division.

The remaining division operators are tested by defining a version of the operator using words which have already been tested (S>D, M*, FM/MOD and SM/REM). The test definition handles the special case of differing signes. As the test definitions use the words which have just been tested, the tests must be performed in the order: F.6.1.0240 /MOD, F.6.1.0230 /, F.6.1.1890 MOD, F.6.1.0100 */, and F.6.1.0110 */MOD.

F.4.11 Memory

As with the other sections, the tests for the memory access words build on previously tested words and thus require an order to the testing.

The first test (F.6.1.0150 , (comma)) tests HERE, the signle cell memory access words @, ! and CELL+ as well as the double cell access words 2@ and 2!. The tests F.6.1.0130 +! and F.6.1.0890 CELLS should

then be performed.

The test (F.6.1.0860 C,) also tests the single character memory words C@, C!, and CHAR+, leaving the test F.6.1.0898 CHARS to be performed seperatly.

Finally, the memory access alignment test F.6.1.0705 ALIGN includes a test of ALIGNED, leaving F.6.1.0710 ALLOT as the final test in this group.

F.4.12 Characters

Basic character handling: F.6.1.0770 BL, F.6.1.0895 CHAR, F.6.1.2520 [CHAR], F.6.1.2500 [which also tests], and F.6.1.2165 S".

F.4.13 Dictionary

The dictionary tests define a number of words as part of the test, these are included in the approperate test: F.6.1.0070 ', F.6.1.2510 ['] both of which also test EXECUTE, F.6.1.1550 FIND, F.6.1.1780 LITERAL, F.6.1.0980 COUNT, F.6.1.2033 POSTPONE, F.6.1.2250 STATE

F.4.14 Flow Control

The flow control words have to be tested in matching groups. First test F.6.1.1700 IF, ELSE, THEN group. Followed by the BEGIN, F.6.1.2430 WHILE, REPEAT group, and the BEGIN, F.6.1.2390 UNTIL pairing. Finally the F.6.1.2120 RECURSE function should be tested.

F.4.15 Counted Loops

Counted loops have a set of special condition that require testing. As with the flow control words, these words have to be tested as a group. First the basic counted loop: DO; I; F.6.1.1800 LOOP, followed by loops with a non regular increment: F.6.1.0140 +LOOP, loops within loops: F.6.1.1730 J, and aborted loops: F.6.1.1760 LEAVE; F.6.1.2380 UNLOOP which includes a test for EXIT.

F.4.16 Defining Words

Although most of the defining words have already been used within the test suite, they still need to be tested fully. The tests include F.6.1.0450 : which also tests ;, F.6.1.0950 CONSTANT, F.6.1.2410 VARIABLE, F.6.1.1250 DOES> which includes tests CREATE, and F.6.1.0550 >BODY which also tests CREATE.

F.4.17 Evaluate

As with the defining words, F.6.1.1360 EVALUATE has already been used, but it must still be tested fully.

F.4.18 Parser Input Source Control

Testing of the input source can be quit dificult. The tests require line breaks within the test: F.6.1.2216 SOURCE, F.6.1.0560 >IN, and F.6.1.2450 WORD.

F.4.19 Number Patterns

The number formatting words produce a string, a word that compares two strings is required. This test suite assumes that the optional String word set is unavailable. Thus a string comparison word is defined, using only trusted words:

```
: S= \ ( ADDR1 C1 ADDR2 C2 -- T/F ) Compare two strings.
  >R SWAP R@ = IF           \ Make sure strings have same length
     R> ?DUP IF             \ If non-empty strings
       0 DO
         OVER C@ OVER C@ - IF 2DROP <FALSE> UNLOOP EXIT THEN
         SWAP CHAR+ SWAP CHAR+
       LOOP
```

```
    THEN
    2DROP <TRUE>           \ If we get here, strings match
  ELSE
    R> DROP 2DROP <FALSE>  \ Lengths mismatch
  THEN ;
```

The number formatting words have to be tested as a group with F.6.1.1670 HOLD, F.6.1.2210 SIGN, and F.6.1.0030 # all including tests for <# and #>.

Before the F.6.1.0050 #S test can be performed it is necessary to calculate the number of bits required to store the largest double value.

```
24 CONSTANT MAX-BASE                \ BASE 2 ...  36
: COUNT-BITS
  0 0 INVERT BEGIN DUP WHILE >R 1+ R> 2* REPEAT DROP ;
COUNT-BITS 2* CONSTANT #BITS-UD   \ NUMBER OF BITS IN UD
```

The F.6.1.0570 >NUMBER test can now be performed. Finally, the F.6.1.0750 BASE test, which includes tests for HEX and DECIMAL, can be performed.

F.4.20 Memory Movement

Frist two memory buffers are defined:

```
CREATE FBUF 00 C, 00 C, 00 C,
CREATE SBUF 12 C, 34 C, 56 C,
: SEEBUF FBUF C@ FBUF CHAR+ C@ FBUF CHAR+ CHAR+ C@ ;
```

As the content of FBUF is changed by the F.6.1.1540 FILL test, this must be executed before the F.6.1.1900 MOVE test.

F.4.21 Output

As there is no provision for capturing the output stream so that it can be compared to an expected result there is not automatic method of testing the output generation words. The user is required to validate the output for the F.6.1.1320 EMIT test. This tests the selection of output words ., ."", CR, SPACE, SPACES, TYPE, and U..

F.4.22 Input

To test the input word (F.6.1.0695 ACCEPT) the user is required to type up to 80 characters. The system will buffer the input sequence and output it to the user for inspection.

F.4.23 Dictionary Search Rules

The final test in this suite is included with F.6.1.0450 : and tests the search order of the dictionary. It asserts that a definition that uses its own name in the definition is not recursive but rather refers to the previous definition of the word.

```
T{ : GDX     123 ; ->  }T  \ First defintion
T{ : GDX GDX 234 ; ->  }T  \ Second defintion
T{ GDX -> 123 234 }T
```

F.6 The Core word set

F.6.1.0010 !

See F.6.1.0150 , .

F.6.1.0030

```
: GP3 <# 1 0 # # #> S" 01" S= ;
T{ GP3 -> <TRUE> }T
```

F.6.1.0040 #>

See F.6.1.0030 #, F.6.1.0050 #S, F.6.1.1670 HOLD and F.6.1.2210 SIGN.

F.6.1.0050 #S

```
: GP4 <# 1 0 #S #> S" 1" S= ;
T{ GP4 -> <TRUE> }T

: GP5
  BASE @ <TRUE>
  MAX-BASE 1+ 2 DO      \ FOR EACH POSSIBLE BASE
     I BASE !            \ TBD: ASSUMES BASE WORKS
       I 0 <# #S #> S" 10" S= AND
  LOOP
  SWAP BASE ! ;
T{ GP5 -> <TRUE> }T

: GP6
  BASE @ >R 2 BASE !
  MAX-UINT MAX-UINT <# #S #>   \ MAXIMUM UD TO BINARY
  R> BASE !                    \ S: C-ADDR U
  DUP #BITS-UD = SWAP
  0 DO                         \ S: C-ADDR FLAG
     OVER C@ [CHAR] 1 = AND    \ ALL ONES
     >R CHAR+ R>
  LOOP SWAP DROP ;
T{ GP6 -> <TRUE> }T

: GP7
  BASE @ >R MAX-BASE BASE !
  <TRUE>
  A 0 DO
     I 0 <# #S #>
     1 = SWAP C@ I 30 + = AND AND
  LOOP
  MAX-BASE A DO
     I 0 <# #S #>
     1 = SWAP C@ 41 I A - + = AND AND
  LOOP
  R> BASE ! ;
T{ GP7 -> <TRUE> }T
```

F.6.1.0070 '

```
T{ : GT1 123 ;    ->      }T
T{ ' GT1 EXECUTE -> 123 }T
```

F.6.1.0080 (

\ There is no space either side of the).

```
T{ ( A comment)1234 -> 1234    }T
T{ : pc1 ( A comment)1234 ; pc1 -> 1234 }T
```

x:contribution-132

F.6.1.0090 *

```
T{  0  0 * ->  0 }T        \ TEST IDENTITIE\S
T{  0  1 * ->  0 }T
T{  1  0 * ->  0 }T
T{  1  2 * ->  2 }T
T{  2  1 * ->  2 }T
T{  3  3 * ->  9 }T
T{ -3  3 * -> -9 }T
T{  3 -3 * -> -9 }T
T{ -3 -3 * ->  9 }T

T{ MID-UINT+1 1 RSHIFT 2 *               -> MID-UINT+1 }T
T{ MID-UINT+1 2 RSHIFT 4 *               -> MID-UINT+1 }T
T{ MID-UINT+1 1 RSHIFT MID-UINT+1 OR 2 * -> MID-UINT+1 }T
```

F.6.1.0100 */

```
IFFLOORED   : T*/ T*/MOD SWAP DROP ;
IFSYM       : T*/ T*/MOD SWAP DROP ;

T{        0 2        1 */ ->        0 2        1 T*/ }T
T{        1 2        1 */ ->        1 2        1 T*/ }T
T{        2 2        1 */ ->        2 2        1 T*/ }T
T{       -1 2        1 */ ->       -1 2        1 T*/ }T
T{       -2 2        1 */ ->       -2 2        1 T*/ }T
T{        0 2       -1 */ ->        0 2       -1 T*/ }T
T{        1 2       -1 */ ->        1 2       -1 T*/ }T
T{        2 2       -1 */ ->        2 2       -1 T*/ }T
T{       -1 2       -1 */ ->       -1 2       -1 T*/ }T
T{       -2 2       -1 */ ->       -2 2       -1 T*/ }T
T{        2 2        2 */ ->        2 2        2 T*/ }T
T{       -1 2       -1 */ ->       -1 2       -1 T*/ }T
T{       -2 2       -2 */ ->       -2 2       -2 T*/ }T
T{        7 2        3 */ ->        7 2        3 T*/ }T
T{        7 2       -3 */ ->        7 2       -3 T*/ }T
T{       -7 2        3 */ ->       -7 2        3 T*/ }T
T{       -7 2       -3 */ ->       -7 2       -3 T*/ }T
T{ MAX-INT 2 MAX-INT */ -> MAX-INT 2 MAX-INT T*/ }T
T{ MIN-INT 2 MIN-INT */ -> MIN-INT 2 MIN-INT T*/ }T
```

F.6.1.0110 */MOD

```
IFFLOORED   : T*/MOD >R M* R> FM/MOD ;
IFSYM       : T*/MOD >R M* R> SM/REM ;

T{        0 2        1 */MOD ->        0 2        1 T*/MOD }T
T{        1 2        1 */MOD ->        1 2        1 T*/MOD }T
T{        2 2        1 */MOD ->        2 2        1 T*/MOD }T
T{       -1 2        1 */MOD ->       -1 2        1 T*/MOD }T
T{       -2 2        1 */MOD ->       -2 2        1 T*/MOD }T
T{        0 2       -1 */MOD ->        0 2       -1 T*/MOD }T
T{        1 2       -1 */MOD ->        1 2       -1 T*/MOD }T
T{        2 2       -1 */MOD ->        2 2       -1 T*/MOD }T
T{       -1 2       -1 */MOD ->       -1 2       -1 T*/MOD }T
T{       -2 2       -1 */MOD ->       -2 2       -1 T*/MOD }T
T{        2 2        2 */MOD ->        2 2        2 T*/MOD }T
T{       -1 2       -1 */MOD ->       -1 2       -1 T*/MOD }T
T{       -2 2       -2 */MOD ->       -2 2       -2 T*/MOD }T
T{        7 2        3 */MOD ->        7 2        3 T*/MOD }T
T{        7 2       -3 */MOD ->        7 2       -3 T*/MOD }T
T{       -7 2        3 */MOD ->       -7 2        3 T*/MOD }T
T{       -7 2       -3 */MOD ->       -7 2       -3 T*/MOD }T
T{ MAX-INT 2 MAX-INT */MOD -> MAX-INT 2 MAX-INT T*/MOD }T
T{ MIN-INT 2 MIN-INT */MOD -> MIN-INT 2 MIN-INT T*/MOD }T
```

F.6.1.0120 +

```
T{         0  5 + ->          5 }T
T{         5  0 + ->          5 }T
T{         0 -5 + ->         -5 }T
T{        -5  0 + ->         -5 }T
T{         1  2 + ->          3 }T
T{         1 -2 + ->         -1 }T
T{        -1  2 + ->          1 }T
T{        -1 -2 + ->         -3 }T
T{        -1  1 + ->          0 }T
T{ MID-UINT  1 + -> MID-UINT+1 }T
```

F.6.1.0130 +!

```
T{  0 1ST !        ->   }T
T{  1 1ST +!       ->   }T
T{    1ST @        -> 1 }T
T{ -1 1ST +! 1ST @ -> 0 }T
```

F.6.1.0140 +LOOP

```
T{ : GD2 DO I -1 +LOOP ; ->  }T
T{         1          4 GD2 -> 4 3 2  1 }T
T{        -1          2 GD2 -> 2 1 0 -1 }T
T{ MID-UINT MID-UINT+1 GD2 -> MID-UINT+1 MID-UINT }T
```

```
VARIABLE gditerations
VARIABLE gdincrement

: gd7 ( limit start increment -- )
  gdincrement !
  0 gditerations !
  DO
     1 gditerations +!
     I
     gditerations @ 6 = IF LEAVE THEN
     gdincrement @
  +LOOP gditerations @
;

T{     4  4  -1 gd7 ->  4                    1  }T
T{     1  4  -1 gd7 ->  4  3  2  1           4  }T
T{     4  1  -1 gd7 ->  1  0 -1 -2  -3  -4 6  }T
T{     4  1   0 gd7 ->  1  1  1  1   1   1 6  }T
T{     0  0   0 gd7 ->  0  0  0  0   0   0 6  }T
T{     1  4   0 gd7 ->  4  4  4  4   4   4 6  }T
T{     1  4   1 gd7 ->  4  5  6  7   8   9 6  }T
T{     4  1   1 gd7 ->  1  2  3              3  }T
T{     4  4   1 gd7 ->  4  5  6  7   8   9 6  }T
T{     2 -1  -1 gd7 -> -1 -2 -3 -4  -5  -6 6  }T
T{    -1  2  -1 gd7 ->  2  1  0 -1           4  }T
T{     2 -1   0 gd7 -> -1 -1 -1 -1  -1  -1 6  }T
T{    -1  2   0 gd7 ->  2  2  2  2   2   2 6  }T
T{    -1  2   1 gd7 ->  2  3  4  5   6   7 6  }T
T{     2 -1   1 gd7 -> -1 0 1                3  }T
T{   -20 30 -10 gd7 -> 30 20 10  0 -10 -20 6  }T
T{   -20 31 -10 gd7 -> 31 21 11  1  -9 -19 6  }T
T{   -20 29 -10 gd7 -> 29 19  9 -1 -11     5  }T
```

\ With large and small increments

```
MAX-UINT 8 RSHIFT 1+ CONSTANT ustep
ustep NEGATE CONSTANT -ustep
MAX-INT 7 RSHIFT 1+ CONSTANT step
step NEGATE CONSTANT -step

VARIABLE bump

T{  : gd8 bump ! DO 1+ bump @ +LOOP ; ->  }T

T{  0 MAX-UINT 0 ustep gd8 -> 256 }T
T{  0 0 MAX-UINT -ustep gd8 -> 256 }T


T{  0 MAX-INT MIN-INT step gd8 -> 256 }T
T{  0 MIN-INT MAX-INT -step gd8 -> 256 }T
```

F.6.1.0150 ,

```
HERE 1 ,
HERE 2 ,
CONSTANT 2ND
CONSTANT 1ST

T{       1ST 2ND U< -> <TRUE> }T\ HERE MUST GROW WITH ALLOT
T{       1ST CELL+  -> 2ND }T \ ...  BY ONE CELL
T{   1ST 1 CELLS +  -> 2ND }T
T{     1ST @ 2ND @  -> 1 2 }T
T{          5 1ST !  ->     }T
T{     1ST @ 2ND @  -> 5 2 }T
T{          6 2ND !  ->     }T
T{     1ST @ 2ND @  -> 5 6 }T
T{           1ST 2@ -> 6 5 }T
T{        2 1 1ST 2! ->     }T
T{           1ST 2@ -> 2 1 }T
T{ 1S 1ST !  1ST @  -> 1S  }T  \ CAN STORE CELL-WIDE VALUE
```

F.6.1.0160 -

```
T{           0  5 - ->       -5 }T
T{           5  0 - ->        5 }T
T{           0 -5 - ->        5 }T
T{          -5  0 - ->       -5 }T
T{           1  2 - ->       -1 }T
T{           1 -2 - ->        3 }T
T{          -1  2 - ->       -3 }T
T{          -1 -2 - ->        1 }T
T{           0  1 - ->       -1 }T
T{ MID-UINT+1  1 - -> MID-UINT }T
```

F.6.1.0180 .

See F.6.1.1320 EMIT.

F.6.1.0190 ."

```
T{ : pb1 CR ." You should see 2345: "." 2345"; pb1 ->  }T
```

See F.6.1.1320 EMIT.

F.6.1.0230 /

```
IFFLOORED   : T/ T/MOD SWAP DROP ;
IFSYM       : T/ T/MOD SWAP DROP ;

T{        0        1 / ->        0       1 T/ }T
T{        1        1 / ->        1       1 T/ }T
T{        2        1 / ->        2       1 T/ }T
T{       -1        1 / ->       -1       1 T/ }T
T{       -2        1 / ->       -2       1 T/ }T
```

```
T{        0       -1 / ->        0       -1 T/ }T
T{        1       -1 / ->        1       -1 T/ }T
T{        2       -1 / ->        2       -1 T/ }T
T{       -1       -1 / ->       -1       -1 T/ }T
T{       -2       -1 / ->       -2       -1 T/ }T
T{        2        2 / ->        2        2 T/ }T
T{       -1       -1 / ->       -1       -1 T/ }T
T{       -2       -2 / ->       -2       -2 T/ }T
T{        7        3 / ->        7        3 T/ }T
T{        7       -3 / ->        7       -3 T/ }T
T{       -7        3 / ->       -7        3 T/ }T
T{       -7       -3 / ->       -7       -3 T/ }T
T{  MAX-INT        1 / ->  MAX-INT        1 T/ }T
T{  MIN-INT        1 / ->  MIN-INT        1 T/ }T
T{  MAX-INT  MAX-INT / ->  MAX-INT  MAX-INT T/ }T
T{  MIN-INT  MIN-INT / ->  MIN-INT  MIN-INT T/ }T
```

F.6.1.0240 /MOD

```
IFFLOORED   : T/MOD >R S>D R> FM/MOD ;
IFSYM       : T/MOD >R S>D R> SM/REM ;
```

```
T{        0        1 /MOD ->        0        1 T/MOD }T
T{        1        1 /MOD ->        1        1 T/MOD }T
T{        2        1 /MOD ->        2        1 T/MOD }T
T{       -1        1 /MOD ->       -1        1 T/MOD }T
T{       -2        1 /MOD ->       -2        1 T/MOD }T
T{        0       -1 /MOD ->        0       -1 T/MOD }T
T{        1       -1 /MOD ->        1       -1 T/MOD }T
T{        2       -1 /MOD ->        2       -1 T/MOD }T
T{       -1       -1 /MOD ->       -1       -1 T/MOD }T
T{       -2       -1 /MOD ->       -2       -1 T/MOD }T
T{        2        2 /MOD ->        2        2 T/MOD }T
T{       -1       -1 /MOD ->       -1       -1 T/MOD }T
T{       -2       -2 /MOD ->       -2       -2 T/MOD }T
T{        7        3 /MOD ->        7        3 T/MOD }T
T{        7       -3 /MOD ->        7       -3 T/MOD }T
T{       -7        3 /MOD ->       -7        3 T/MOD }T
T{       -7       -3 /MOD ->       -7       -3 T/MOD }T
T{  MAX-INT        1 /MOD ->  MAX-INT        1 T/MOD }T
T{  MIN-INT        1 /MOD ->  MIN-INT        1 T/MOD }T
T{  MAX-INT  MAX-INT /MOD ->  MAX-INT  MAX-INT T/MOD }T
T{  MIN-INT  MIN-INT /MOD ->  MIN-INT  MIN-INT T/MOD }T
```

F.6.1.0250 0<

```
T{        0 0< -> <FALSE> }T
T{       -1 0< -> <TRUE>  }T
T{  MIN-INT 0< -> <TRUE>  }T
```

```
T{       1 0< -> <FALSE> }T
T{ MAX-INT 0< -> <FALSE> }T
```

F.6.1.0270 0=

```
T{         0 0= -> <TRUE>  }T
T{         1 0= -> <FALSE> }T
T{         2 0= -> <FALSE> }T
T{        -1 0= -> <FALSE> }T
T{ MAX-UINT 0= -> <FALSE> }T
T{ MIN-INT  0= -> <FALSE> }T
T{ MAX-INT  0= -> <FALSE> }T
```

F.6.1.0290 1+

```
T{         0 1+ ->          1 }T
T{        -1 1+ ->          0 }T
T{         1 1+ ->          2 }T
T{ MID-UINT 1+ -> MID-UINT+1 }T
T{ MAX-INT  1+ ->    MIN-INT }T
```

F.6.1.0300 1-

```
T{           2 1- ->         1 }T
T{           1 1- ->         0 }T
T{           0 1- ->        -1 }T
T{ MID-UINT+1 1- -> MID-UINT }T
```

F.6.1.0310 2!

See F.6.1.0150 ,.

F.6.1.0320 2*

```
T{   0S 2*        ->   0S }T
T{    1 2*        ->    2 }T
T{ 4000 2*        -> 8000 }T
T{   1S 2* 1 XOR ->   1S }T
T{  MSB 2*        ->   0S }T
```

F.6.1.0330 2/

```
T{           0S 2/ ->   0S }T
T{            1 2/ ->    0 }T
T{         4000 2/ -> 2000 }T
T{           1S 2/ ->   1S }T\ MSB PROPOGATED
T{    1S 1 XOR 2/ ->   1S }T
T{ MSB 2/ MSB AND ->  MSB }T
```

F.6.1.0350 2@

See F.6.1.0150 ,.

F.6.1.0370 2DROP

```
T{ 1 2 2DROP ->  }T
```

F.6.1.0380 2DUP

```
T{ 1 2 2DUP -> 1 2 1 2 }T
```

F.6.1.0400 2OVER

```
T{ 1 2 3 4 2OVER -> 1 2 3 4 1 2 }T
```

F.6.1.0430 2SWAP

```
T{ 1 2 3 4 2SWAP -> 3 4 1 2 }T
```

F.6.1.0450 :

```
T{ : NOP : POSTPONE ; ; ->  }T
T{ NOP NOP1 NOP NOP2 ->  }T
T{ NOP1 ->  }T
T{ NOP2 ->  }T
```

The following tests the dictionary search order:

```
T{ : GDX   123 ;    : GDX   GDX 234 ; ->  }T
T{ GDX -> 123 234 }T
```

F.6.1.0460 ;

See F.6.1.0450 :.

F.6.1.0480 <

```
T{       0       1 < -> <TRUE>  }T
T{       1       2 < -> <TRUE>  }T
T{      -1       0 < -> <TRUE>  }T
T{      -1       1 < -> <TRUE>  }T
T{ MIN-INT       0 < -> <TRUE>  }T
T{ MIN-INT MAX-INT < -> <TRUE>  }T
T{       0 MAX-INT < -> <TRUE>  }T
T{       0       0 < -> <FALSE> }T
T{       1       1 < -> <FALSE> }T
T{       1       0 < -> <FALSE> }T
T{       2       1 < -> <FALSE> }T
T{       0      -1 < -> <FALSE> }T
T{       1      -1 < -> <FALSE> }T
T{       0 MIN-INT < -> <FALSE> }T
T{ MAX-INT MIN-INT < -> <FALSE> }T
T{ MAX-INT       0 < -> <FALSE> }T
```

F.6.1.0490 <#

See F.6.1.0030 #, F.6.1.0050 #S, F.6.1.1670 HOLD, F.6.1.2210 SIGN.

F.6.1.0530 =

```
T{  0  0 = -> <TRUE>  }T
T{  1  1 = -> <TRUE>  }T
T{ -1 -1 = -> <TRUE>  }T
T{  1  0 = -> <FALSE> }T
```

```
T{ -1  0 = -> <FALSE> }T
T{  0  1 = -> <FALSE> }T
T{  0 -1 = -> <FALSE> }T
```

F.6.1.0540 >

```
T{       0       1 > -> <FALSE> }T
T{       1       2 > -> <FALSE> }T
T{      -1       0 > -> <FALSE> }T
T{      -1       1 > -> <FALSE> }T
T{ MIN-INT       0 > -> <FALSE> }T
T{ MIN-INT MAX-INT > -> <FALSE> }T
T{       0 MAX-INT > -> <FALSE> }T
T{       0       0 > -> <FALSE> }T
T{       1       1 > -> <FALSE> }T
T{       1       0 > -> <TRUE>  }T
T{       2       1 > -> <TRUE>  }T
T{       0      -1 > -> <TRUE>  }T
T{       1      -1 > -> <TRUE>  }T
T{       0 MIN-INT > -> <TRUE>  }T
T{ MAX-INT MIN-INT > -> <TRUE>  }T
T{ MAX-INT       0 > -> <TRUE>  }T
```

F.6.1.0550 >BODY

```
T{  CREATE CR0 ->      }T
T{ ' CR0 >BODY -> HERE }T
```

F.6.1.0560 >IN

```
VARIABLE SCANS
: RESCAN? -1 SCANS +! SCANS @ IF 0 >IN ! THEN ;

T{   2 SCANS !
   345 RESCAN?
-> 345 345 }T

: GS2 5 SCANS ! S" 123 RESCAN?" EVALUATE ;
T{ GS2 -> 123 123 123 123 123 }T

\ These tests must start on a new line
DECIMAL
T{ 123456 DEPTH OVER 9 < 35 AND + 3 + >IN !
-> 123456 23456 3456 456 56 6 }T
T{ 14145 8115 ?DUP 0= 34 AND >IN +! TUCK MOD 14 >IN ! GCD calculation
-> 15 }T
```

F.6.1.0570 >NUMBER

```
CREATE GN-BUF 0 C,
: GN-STRING GN-BUF 1 ;
```

```
: GN-CONSUMED GN-BUF CHAR+ 0 ;
: GN' [CHAR] ' WORD CHAR+ C@ GN-BUF C! GN-STRING ;

T{ 0 0 GN' 0' >NUMBER ->          0 0 GN-CONSUMED }T
T{ 0 0 GN' 1' >NUMBER ->          1 0 GN-CONSUMED }T
T{ 1 0 GN' 1' >NUMBER -> BASE @ 1+ 0 GN-CONSUMED }T
\ FOLLOWING SHOULD FAIL TO CONVERT
T{ 0 0 GN' -' >NUMBER ->          0 0 GN-STRING   }T
T{ 0 0 GN' +' >NUMBER ->          0 0 GN-STRING   }T
T{ 0 0 GN' .' >NUMBER ->          0 0 GN-STRING   }T

: >NUMBER-BASED
  BASE @ >R BASE ! >NUMBER R> BASE ! ;

T{ 0 0 GN' 2'        10 >NUMBER-BASED ->  2 0 GN-CONSUMED }T
T{ 0 0 GN' 2'         2 >NUMBER-BASED ->  0 0 GN-STRING   }T
T{ 0 0 GN' F'        10 >NUMBER-BASED ->  F 0 GN-CONSUMED }T
T{ 0 0 GN' G'        10 >NUMBER-BASED ->  0 0 GN-STRING   }T
T{ 0 0 GN' G' MAX-BASE >NUMBER-BASED -> 10 0 GN-CONSUMED }T
T{ 0 0 GN' Z' MAX-BASE >NUMBER-BASED -> 23 0 GN-CONSUMED }T

: GN1 ( UD BASE -- UD' LEN )
  \ UD SHOULD EQUAL UD' AND LEN SHOULD BE ZERO.
  BASE @ >R BASE !
  <# #S #>
  0 0 2SWAP >NUMBER SWAP DROP   \ RETURN LENGTH ONLY
  R> BASE ! ;

T{         0   0         2 GN1 ->         0   0 0 }T
T{ MAX-UINT    0         2 GN1 -> MAX-UINT    0 0 }T
T{ MAX-UINT DUP          2 GN1 -> MAX-UINT DUP 0 }T
T{         0   0 MAX-BASE GN1 ->         0   0 0 }T
T{ MAX-UINT    0 MAX-BASE GN1 -> MAX-UINT    0 0 }T
T{ MAX-UINT DUP MAX-BASE GN1 -> MAX-UINT DUP 0 }T
```

F.6.1.0580 >R

```
T{ : GR1 >R R> ; ->  }T
T{ : GR2 >R R@ R> DROP ; ->  }T
T{ 123 GR1 -> 123 }T
T{ 123 GR2 -> 123 }T
T{  1S GR1 ->  1S }T     ( Return stack holds cells )
```

F.6.1.0630 ?DUP

```
T{ -1 ?DUP -> -1 -1 }T
T{  0 ?DUP ->  0    }T
T{  1 ?DUP ->  1  1 }T
```

F.6.1.0650 @

See F.6.1.0150 , .

F.6.1.0690 ABS

```
T{       0 ABS ->          0 }T
T{       1 ABS ->          1 }T
T{      -1 ABS ->          1 }T
T{ MIN-INT ABS -> MID-UINT+1 }T
```

F.6.1.0695 ACCEPT

```
CREATE ABUF 80 CHARS ALLOT

: ACCEPT-TEST
    CR ." PLEASE TYPE UP TO 80 CHARACTERS:" CR
    ABUF 80 ACCEPT
    CR ." RECEIVED: " [CHAR] " EMIT
    ABUF SWAP TYPE [CHAR] " EMIT CR
;

T{ ACCEPT-TEST ->  }T
```

F.6.1.0705 ALIGN

```
ALIGN 1 ALLOT HERE ALIGN HERE 3 CELLS ALLOT
CONSTANT A-ADDR CONSTANT UA-ADDR
T{ UA-ADDR ALIGNED -> A-ADDR }T
T{       1 A-ADDR C!          A-ADDR       C@ ->       1 }T
T{    1234 A-ADDR !           A-ADDR       @  ->    1234 }T
T{ 123 456 A-ADDR 2!          A-ADDR       2@ -> 123 456 }T
T{       2 A-ADDR CHAR+ C!    A-ADDR CHAR+ C@ ->       2 }T
T{       3 A-ADDR CELL+ C!    A-ADDR CELL+ C@ ->       3 }T
T{    1234 A-ADDR CELL+ !     A-ADDR CELL+ @  ->    1234 }T
T{ 123 456 A-ADDR CELL+ 2!    A-ADDR CELL+ 2@ -> 123 456 }T
```

F.6.1.0710 ALLOT

```
HERE 1 ALLOT
HERE
CONSTANT 2NDA
CONSTANT 1STA
T{ 1STA 2NDA U< -> <TRUE> }T  \ HERE MUST GROW WITH ALLOT
T{      1STA 1+ ->   2NDA }T  \ ...  BY ONE ADDRESS UNIT
( MISSING TEST: NEGATIVE ALLOT )
```

F.6.1.0720 AND

```
T{ 0 0 AND -> 0 }T
T{ 0 1 AND -> 0 }T
T{ 1 0 AND -> 0 }T
T{ 1 1 AND -> 1 }T

T{ 0 INVERT 1 AND -> 1 }T
T{ 1 INVERT 1 AND -> 0 }T
```

```
T{ 0S 0S AND -> 0S }T
T{ 0S 1S AND -> 0S }T
T{ 1S 0S AND -> 0S }T
T{ 1S 1S AND -> 1S }T
```

F.6.1.0750 BASE

```
: GN2 \ ( -- 16 10 )
  BASE @ >R HEX BASE @ DECIMAL BASE @ R> BASE ! ;
T{ GN2 -> 10 A }T
```

F.6.1.0760 BEGIN

See F.6.1.2430 WHILE, F.6.1.2390 UNTIL.

F.6.1.0770 BL

```
T{ BL -> $20 }T
```

F.6.1.0850 C!

See F.6.1.0860 C,.

F.6.1.0860 C,

```
HERE 1 C,
HERE 2 C,
CONSTANT 2NDC
CONSTANT 1STC

T{    1STC 2NDC U< -> <TRUE> }T\ HERE MUST GROW WITH ALLOT
T{      1STC CHAR+ ->  2NDC  }T\ ...  BY ONE CHAR
T{  1STC 1 CHARS + ->  2NDC  }T
T{ 1STC C@ 2NDC C@ ->   1 2  }T
T{       3 1STC C! ->        }T
T{ 1STC C@ 2NDC C@ ->   3 2  }T
T{       4 2NDC C! ->        }T
T{ 1STC C@ 2NDC C@ ->   3 4  }T
```

F.6.1.0870 C@

See F.6.1.0860 C,.

F.6.1.0880 CELL+

See F.6.1.0150 ,.

F.6.1.0890 CELLS

```
: BITS ( X -- U )
  0 SWAP BEGIN DUP WHILE
     DUP MSB AND IF >R 1+ R> THEN 2*
  REPEAT DROP ;

( CELLS >= 1 AU, INTEGRAL MULTIPLE OF CHAR SIZE, >= 16 BITS )
T{ 1 CELLS 1 <          -> <FALSE> }T
T{ 1 CELLS 1 CHARS MOD ->    0    }T
T{ 1S BITS 10 <         -> <FALSE> }T
```

F.6.1.0895 CHAR

```
T{ CHAR X     -> 58 }T
T{ CHAR HELLO -> 48 }T
```

F.6.1.0897 CHAR+

```
T{ 0 CHAR+ 1 = -> <TRUE> }T
```

See F.6.1.0860 C, for additional test.

F.6.1.0898 CHARS

```
( CHARACTERS >= 1 AU, <= SIZE OF CELL, >= 8 BITS )
T{ 1 CHARS 1 <       -> <FALSE> }T
T{ 1 CHARS 1 CELLS > -> <FALSE> }T
T{ 1 CHARS 1 =       -> <TRUE>  }T
( TBD: HOW TO FIND NUMBER OF BITS? )
```

F.6.1.0950 CONSTANT

```
T{ 123 CONSTANT X123 ->  }T
T{ X123 -> 123 }T

T{ : EQU CONSTANT ; ->  }T
T{ X123 EQU Y123 ->  }T
T{ Y123 -> 123 }T
```

F.6.1.0980 COUNT

```
T{ GT1STRING COUNT -> GT1STRING CHAR+ 3 }T
```

F.6.1.0990 CR

See F.6.1.1320 EMIT.

F.6.1.1000 CREATE

See F.6.1.0550 >BODY and F.6.1.1250 DOES>.

F.6.1.1170 DECIMAL

See F.6.1.0750 BASE.

F.6.1.1200 DEPTH

```
T{ 0 1 DEPTH -> 0 1 2 }T
T{   0 DEPTH -> 0 1   }T
T{     DEPTH -> 0     }T
```

F.6.1.1240 DO

See F.6.1.1800 LOOP, F.6.1.0140 +LOOP, F.6.1.1730 J, F.6.1.1760 LEAVE, F.6.1.2380 UNLOOP.

F.6.1.1250 DOES>

```
T{ : DOES1 DOES> @ 1 + ; ->  }T
T{ : DOES2 DOES> @ 2 + ; ->  }T
T{ CREATE CR1 ->  }T
T{ CR1   -> HERE }T
T{ 1 ,   ->   }T
```

```
T{ CR1 @ -> 1 }T
T{ DOES1 ->   }T
T{ CR1   -> 2 }T
T{ DOES2 ->   }T
T{ CR1   -> 3 }T

T{ : WEIRD: CREATE DOES> 1 + DOES> 2 + ; ->  }T
T{ WEIRD: W1 ->  }T
T{ ' W1 >BODY -> HERE }T
T{ W1 -> HERE 1 + }T
T{ W1 -> HERE 2 + }T
```

F.6.1.1260 DROP

```
T{ 1 2 DROP -> 1 }T
T{ 0   DROP ->   }T
```

F.6.1.1290 DUP

```
T{ 1 DUP -> 1 1 }T
```

F.6.1.1310 ELSE

See F.6.1.1700 IF.

F.6.1.1320 EMIT

```
: OUTPUT-TEST
  ." YOU SHOULD SEE THE STANDARD GRAPHIC CHARACTERS:" CR
  41 BL DO I EMIT LOOP CR
  61 41 DO I EMIT LOOP CR
  7F 61 DO I EMIT LOOP CR

  ." YOU SHOULD SEE 0-9 SEPARATED BY A SPACE:" CR
  9 1+ 0 DO I . LOOP CR

  ." YOU SHOULD SEE 0-9 (WITH NO SPACES):" CR
  [CHAR] 9 1+ [CHAR] 0 DO I 0 SPACES EMIT LOOP CR

  ." YOU SHOULD SEE A-G SEPARATED BY A SPACE:" CR
  [CHAR] G 1+ [CHAR] A DO I EMIT SPACE LOOP CR

  ." YOU SHOULD SEE 0-5 SEPARATED BY TWO SPACES:" CR
  5 1+ 0 DO I [CHAR] 0 + EMIT 2 SPACES LOOP CR

  ." YOU SHOULD SEE TWO SEPARATE LINES:" CR
  S" LINE 1" TYPE CR S" LINE 2" TYPE CR

  ." YOU SHOULD SEE THE NUMBER RANGES OF SIGNED AND UNSIGNED NUMBERS:" CR
  ."   SIGNED: " MIN-INT . MAX-INT . CR
  ." UNSIGNED: " 0 U. MAX-UINT U. CR
;

T{ OUTPUT-TEST ->  }T
```

F.6.1.1345 ENVIRONMENT?

\ should be the same for any query starting with X:

```
T{ S" X:deferred" ENVIRONMENT? DUP 0= XOR INVERT -> <TRUE>  }T
T{ S" X:notfound" ENVIRONMENT? DUP 0= XOR INVERT -> <FALSE> }T
```

F.6.1.1360 EVALUATE

```
: GE1 S" 123" ; IMMEDIATE
: GE2 S" 123 1+" ; IMMEDIATE
: GE3 S" : GE4 345 ;" ;
: GE5 EVALUATE ; IMMEDIATE

T{ GE1 EVALUATE -> 123 }T( TEST EVALUATE IN INTERP. STATE )
T{ GE2 EVALUATE -> 124 }T
T{ GE3 EVALUATE ->     }T
T{ GE4          -> 345 }T

T{ : GE6 GE1 GE5 ; ->  }T( TEST EVALUATE IN COMPILE STATE )
T{ GE6 -> 123 }T
T{ : GE7 GE2 GE5 ; ->  }T
T{ GE7 -> 124 }T
```

See F.9.3.6 for additional test.

F.6.1.1370 EXECUTE

See F.6.1.0070 ' and F.6.1.2510 ['].

F.6.1.1380 EXIT

See F.6.1.2380 UNLOOP.

F.6.1.1540 FILL

```
T{ FBUF 0 20 FILL ->  }T
T{ SEEBUF -> 00 00 00 }T

T{ FBUF 1 20 FILL ->  }T
T{ SEEBUF -> 20 00 00 }T

T{ FBUF 3 20 FILL ->  }T
T{ SEEBUF -> 20 20 20 }T
```

F.6.1.1550 FIND

```
HERE 3 C, CHAR G C, CHAR T C, CHAR 1 C, CONSTANT GT1STRING
HERE 3 C, CHAR G C, CHAR T C, CHAR 2 C, CONSTANT GT2STRING
T{ GT1STRING FIND -> ' GT1 -1 }T
T{ GT2STRING FIND -> ' GT2 1  }T
( HOW TO SEARCH FOR NON-EXISTENT WORD? )
```

F.6.1.1561 FM/MOD

```
T{       0 S>D              1 FM/MOD ->  0       0 }T
T{       1 S>D              1 FM/MOD ->  0       1 }T
T{       2 S>D              1 FM/MOD ->  0       2 }T
```

```
T{       -1 S>D              1 FM/MOD ->  0       -1 }T
T{       -2 S>D              1 FM/MOD ->  0       -2 }T
T{        0 S>D             -1 FM/MOD ->  0        0 }T
T{        1 S>D             -1 FM/MOD ->  0       -1 }T
T{        2 S>D             -1 FM/MOD ->  0       -2 }T
T{       -1 S>D             -1 FM/MOD ->  0        1 }T
T{       -2 S>D             -1 FM/MOD ->  0        2 }T
T{        2 S>D              2 FM/MOD ->  0        1 }T
T{       -1 S>D             -1 FM/MOD ->  0        1 }T
T{       -2 S>D             -2 FM/MOD ->  0        1 }T
T{        7 S>D              3 FM/MOD ->  1        2 }T
T{        7 S>D             -3 FM/MOD -> -2       -3 }T
T{       -7 S>D              3 FM/MOD ->  2       -3 }T
T{       -7 S>D             -3 FM/MOD -> -1        2 }T
T{ MAX-INT S>D              1 FM/MOD ->  0 MAX-INT }T
T{ MIN-INT S>D              1 FM/MOD ->  0 MIN-INT }T
T{ MAX-INT S>D        MAX-INT FM/MOD ->  0        1 }T
T{ MIN-INT S>D        MIN-INT FM/MOD ->  0        1 }T
T{    1S 1                  4 FM/MOD ->  3 MAX-INT }T
T{       1 MIN-INT M*       1 FM/MOD ->  0 MIN-INT }T
T{       1 MIN-INT M* MIN-INT FM/MOD ->  0        1 }T
T{       2 MIN-INT M*       2 FM/MOD ->  0 MIN-INT }T
T{       2 MIN-INT M* MIN-INT FM/MOD ->  0        2 }T
T{       1 MAX-INT M*       1 FM/MOD ->  0 MAX-INT }T
T{       1 MAX-INT M* MAX-INT FM/MOD ->  0        1 }T
T{       2 MAX-INT M*       2 FM/MOD ->  0 MAX-INT }T
T{       2 MAX-INT M* MAX-INT FM/MOD ->  0        2 }T
T{ MIN-INT MIN-INT M* MIN-INT FM/MOD ->  0 MIN-INT }T
T{ MIN-INT MAX-INT M* MIN-INT FM/MOD ->  0 MAX-INT }T
T{ MIN-INT MAX-INT M* MAX-INT FM/MOD ->  0 MIN-INT }T
T{ MAX-INT MAX-INT M* MAX-INT FM/MOD ->  0 MAX-INT }T
```

F.6.1.1650 HERE

See F.6.1.0150 `,`, F.6.1.0710 `ALLOT`, F.6.1.0860 `C,`.

F.6.1.1670 HOLD

```
: GP1 <# 41 HOLD 42 HOLD 0 0 #> S" BA" S= ;
T{ GP1 -> <TRUE> }T
```

F.6.1.1680 I

See F.6.1.1800 `LOOP`, F.6.1.0140 `+LOOP`, F.6.1.1730 `J`, F.6.1.1760 `LEAVE`, F.6.1.2380 `UNLOOP`.

F.6.1.1700 IF

```
T{ : GI1 IF 123 THEN ; ->  }T
T{ : GI2 IF 123 ELSE 234 THEN ; ->  }T
T{  0 GI1 ->     }T
T{  1 GI1 -> 123 }T
```

```
T{ -1 GI1 -> 123 }T
T{  0 GI2 -> 234 }T
T{  1 GI2 -> 123 }T
T{ -1 GI1 -> 123 }T
```

\ Multiple ELSEs in an IF statement
```
: melse IF 1 ELSE 2 ELSE 3 ELSE 4 ELSE 5 THEN ;
T{ <FALSE> melse -> 2 4 }T
T{ <TRUE>  melse -> 1 3 5 }T
```

F.6.1.1710 IMMEDIATE

```
T{ 123 CONSTANT iw1 IMMEDIATE iw1 -> 123 }T
T{ : iw2 iw1 LITERAL ; iw2 -> 123 }T

T{ VARIABLE iw3 IMMEDIATE 234 iw3 ! iw3 @ -> 234 }T
T{ : iw4 iw3 [ @ ] LITERAL ; iw4 -> 234 }T

T{ :NONAME [ 345 ] iw3 [ ! ] ; DROP iw3 @ -> 345 }T
T{ CREATE iw5 456 , IMMEDIATE ->  }T
T{ :NONAME iw5 [ @ iw3 ! ] ; DROP iw3 @ -> 456 }T

T{ : iw6 CREATE , IMMEDIATE DOES> @ 1+ ; ->  }T
T{ 111 iw6 iw7 iw7 -> 112 }T
T{ : iw8 iw7 LITERAL 1+ ; iw8 -> 113 }T

T{ : iw9 CREATE , DOES> @ 2 + IMMEDIATE ; ->  }T
: find-iw BL WORD FIND NIP ;
T{ 222 iw9 iw10 find-iw iw10 -> -1 }T  \ iw10 is not immediate
T{ iw10 find-iw iw10 -> 224 1 }T       \ iw10 becomes immediate
```

See: F.6.1.2510 ['], F.6.1.2033 POSTPONE, F.6.1.2250 STATE, F.6.1.2165 S".

F.6.1.1720 INVERT

```
T{ 0S INVERT -> 1S }T
T{ 1S INVERT -> 0S }T
T{  0 INVERT -> -1 }T
```

F.6.1.1730 J

```
T{ : GD3 DO 1 0 DO J LOOP LOOP ; ->  }T
T{          4        1 GD3 ->  1 2 3   }T
T{          2       -1 GD3 -> -1 0 1   }T
T{ MID-UINT+1 MID-UINT GD3 -> MID-UINT }T

T{ : GD4 DO 1 0 DO J LOOP -1 +LOOP ; ->  }T
T{        1          4 GD4 -> 4 3 2 1             }T
T{       -1          2 GD4 -> 2 1 0 -1            }T
T{ MID-UINT MID-UINT+1 GD4 -> MID-UINT+1 MID-UINT }T
```

F.6.1.1760 LEAVE

```
T{ : GD5 123 SWAP 0 DO
     I 4 > IF DROP 234 LEAVE THEN
```

```
   LOOP ; ->  }T
T{ 1 GD5 -> 123 }T
T{ 5 GD5 -> 123 }T
T{ 6 GD5 -> 234 }T
```

F.6.1.1780 LITERAL

```
T{ : GT3 GT2 LITERAL ; ->  }T
T{ GT3 -> ' GT1 }T
```

F.6.1.1800 LOOP

```
T{ : GD1 DO I LOOP ; ->  }T
T{          4         1 GD1 ->  1 2 3   }T
T{          2        -1 GD1 -> -1 0 1   }T
T{ MID-UINT+1 MID-UINT GD1 -> MID-UINT }T
```

F.6.1.1805 LSHIFT

```
T{   1 0 LSHIFT ->    1 }T
T{   1 1 LSHIFT ->    2 }T
T{   1 2 LSHIFT ->    4 }T
T{   1 F LSHIFT -> 8000 }T    \ BIGGEST GUARANTEED SHIFT
T{  1S 1 LSHIFT 1 XOR -> 1S }T
T{ MSB 1 LSHIFT ->    0 }T
```

F.6.1.1810 M*

```
T{       0       0 M* ->       0 S>D }T
T{       0       1 M* ->       0 S>D }T
T{       1       0 M* ->       0 S>D }T
T{       1       2 M* ->       2 S>D }T
T{       2       1 M* ->       2 S>D }T
T{       3       3 M* ->       9 S>D }T
T{      -3       3 M* ->      -9 S>D }T
T{       3      -3 M* ->      -9 S>D }T
T{      -3      -3 M* ->       9 S>D }T
T{       0 MIN-INT M* ->       0 S>D }T
T{       1 MIN-INT M* -> MIN-INT S>D }T
T{       2 MIN-INT M* ->       0 1S  }T
T{       0 MAX-INT M* ->       0 S>D }T
T{       1 MAX-INT M* -> MAX-INT S>D }T
T{       2 MAX-INT M* -> MAX-INT     1 LSHIFT 0 }T
T{ MIN-INT MIN-INT M* ->       0 MSB 1 RSHIFT   }T
T{ MAX-INT MIN-INT M* ->     MSB MSB 2/         }T
T{ MAX-INT MAX-INT M* ->       1 MSB 2/ INVERT  }T
```

F.6.1.1870 MAX

```
T{       0       1 MAX ->       1 }T
T{       1       2 MAX ->       2 }T
T{      -1       0 MAX ->       0 }T
```

```
T{       -1        1 MAX ->        1 }T
T{ MIN-INT         0 MAX ->        0 }T
T{ MIN-INT MAX-INT MAX -> MAX-INT }T
T{        0 MAX-INT MAX -> MAX-INT }T
T{        0        0 MAX ->        0 }T
T{        1        1 MAX ->        1 }T
T{        1        0 MAX ->        1 }T
T{        2        1 MAX ->        2 }T
T{        0       -1 MAX ->        0 }T
T{        1       -1 MAX ->        1 }T
T{        0 MIN-INT MAX ->        0 }T
T{ MAX-INT MIN-INT MAX -> MAX-INT }T
T{ MAX-INT         0 MAX -> MAX-INT }T
```

F.6.1.1880 MIN

```
T{        0        1 MIN ->        0 }T
T{        1        2 MIN ->        1 }T
T{       -1        0 MIN ->       -1 }T
T{       -1        1 MIN ->       -1 }T
T{ MIN-INT         0 MIN -> MIN-INT }T
T{ MIN-INT MAX-INT MIN -> MIN-INT }T
T{        0 MAX-INT MIN ->        0 }T
T{        0        0 MIN ->        0 }T
T{        1        1 MIN ->        1 }T
T{        1        0 MIN ->        0 }T
T{        2        1 MIN ->        1 }T
T{        0       -1 MIN ->       -1 }T
T{        1       -1 MIN ->       -1 }T
T{        0 MIN-INT MIN -> MIN-INT }T
T{ MAX-INT MIN-INT MIN -> MIN-INT }T
T{ MAX-INT         0 MIN ->        0 }T
```

F.6.1.1890 MOD

```
IFFLOORED   : TMOD T/MOD DROP ;
IFSYM       : TMOD T/MOD DROP ;

T{        0        1 MOD ->        0        1 TMOD }T
T{        1        1 MOD ->        1        1 TMOD }T
T{        2        1 MOD ->        2        1 TMOD }T
T{       -1        1 MOD ->       -1        1 TMOD }T
T{       -2        1 MOD ->       -2        1 TMOD }T
T{        0       -1 MOD ->        0       -1 TMOD }T
T{        1       -1 MOD ->        1       -1 TMOD }T
T{        2       -1 MOD ->        2       -1 TMOD }T
T{       -1       -1 MOD ->       -1       -1 TMOD }T
T{       -2       -1 MOD ->       -2       -1 TMOD }T
```

```
T{        2        2 MOD ->        2        2 TMOD }T
T{       -1       -1 MOD ->       -1       -1 TMOD }T
T{       -2       -2 MOD ->       -2       -2 TMOD }T
T{        7        3 MOD ->        7        3 TMOD }T
T{        7       -3 MOD ->        7       -3 TMOD }T
T{       -7        3 MOD ->       -7        3 TMOD }T
T{       -7       -3 MOD ->       -7       -3 TMOD }T
T{ MAX-INT        1 MOD -> MAX-INT        1 TMOD }T
T{ MIN-INT        1 MOD -> MIN-INT        1 TMOD }T
T{ MAX-INT MAX-INT MOD -> MAX-INT MAX-INT TMOD }T
T{ MIN-INT MIN-INT MOD -> MIN-INT MIN-INT TMOD }T
```

F.6.1.1900 MOVE

```
T{ FBUF FBUF 3 CHARS MOVE ->  }T                \ BIZARRE SPECIAL CASE
T{ SEEBUF -> 20 20 20 }T

T{ SBUF FBUF 0 CHARS MOVE ->  }T
T{ SEEBUF -> 20 20 20 }T

T{ SBUF FBUF 1 CHARS MOVE ->  }T
T{ SEEBUF -> 12 20 20 }T

T{ SBUF FBUF 3 CHARS MOVE ->  }T
T{ SEEBUF -> 12 34 56 }T

T{ FBUF FBUF CHAR+ 2 CHARS MOVE ->  }T
T{ SEEBUF -> 12 12 34 }T

T{ FBUF CHAR+ FBUF 2 CHARS MOVE ->  }T
T{ SEEBUF -> 12 34 34 }T
```

F.6.1.1910 NEGATE

```
T{  0 NEGATE ->  0 }T
T{  1 NEGATE -> -1 }T
T{ -1 NEGATE ->  1 }T
T{  2 NEGATE -> -2 }T
T{ -2 NEGATE ->  2 }T
```

F.6.1.1980 OR

```
T{ 0S 0S OR -> 0S }T
T{ 0S 1S OR -> 1S }T
T{ 1S 0S OR -> 1S }T
T{ 1S 1S OR -> 1S }T
```

F.6.1.1990 OVER

```
T{ 1 2 OVER -> 1 2 1 }T
```

F.6.1.2033 POSTPONE

```
T{ : GT4 POSTPONE GT1 ; IMMEDIATE ->  }T
T{ : GT5 GT4 ; ->  }T
T{ GT5 -> 123 }T

T{ : GT6 345 ; IMMEDIATE ->  }T
T{ : GT7 POSTPONE GT6 ; ->  }T
T{ GT7 -> 345 }T
```

F.6.1.2060 R>

See F.6.1.0580 >R.

F.6.1.2070 R@

See F.6.1.0580 >R.

F.6.1.2120 RECURSE

```
T{ : GI6 ( N -- 0,1,..N )
     DUP IF DUP >R 1- RECURSE R> THEN ; ->  }T
T{ 0 GI6 -> 0 }T
T{ 1 GI6 -> 0 1 }T
T{ 2 GI6 -> 0 1 2 }T
T{ 3 GI6 -> 0 1 2 3 }T
T{ 4 GI6 -> 0 1 2 3 4 }T

DECIMAL
T{ :NONAME ( n -- 0, 1, .., n )
    DUP IF DUP >R 1- RECURSE R> THEN
  ;
  CONSTANT rn1 ->  }T
T{ 0 rn1 EXECUTE -> 0 }T
T{ 4 rn1 EXECUTE -> 0 1 2 3 4 }T

:NONAME ( n -- n1 )
  1- DUP
  CASE 0 OF EXIT ENDOF
    1 OF 11 SWAP RECURSE ENDOF
    2 OF 22 SWAP RECURSE ENDOF
    3 OF 33 SWAP RECURSE ENDOF
    DROP ABS RECURSE EXIT
  ENDCASE
; CONSTANT rn2

T{  1 rn2 EXECUTE -> 0 }T
T{  2 rn2 EXECUTE -> 11 0 }T
T{  4 rn2 EXECUTE -> 33 22 11 0 }T
T{ 25 rn2 EXECUTE -> 33 22 11 0 }T
```

F.6.1.2140 REPEAT

See F.6.1.2430 WHILE.

F.6.1.2160 ROT

```
T{ 1 2 3 ROT -> 2 3 1 }T
```

F.6.1.2162 RSHIFT

```
T{    1 0 RSHIFT -> 1 }T
T{    1 1 RSHIFT -> 0 }T
T{    2 1 RSHIFT -> 1 }T
T{    4 2 RSHIFT -> 1 }T
T{ 8000 F RSHIFT -> 1 }T            \ Biggest
T{  MSB 1 RSHIFT MSB AND ->   0 }T  \ RSHIFT zero fills MSBs
T{  MSB 1 RSHIFT     2*  -> MSB }T
```

F.6.1.2165 S"

```
T{ : GC4 S" XY" ; ->   }T
T{ GC4 SWAP DROP  -> 2 }T
T{ GC4 DROP DUP C@ SWAP CHAR+ C@ -> 58 59 }T
```

```
: GC5 S" A String"2DROP ; \ There is no space between the " and 2DROP
T{ GC5 ->  }T
```

```
T{ S" A String"2DROP ->  }T  \ There is no space between the " and 2DROP
```

F.6.1.2170 S>D

```
T{       0 S>D ->       0  0 }T
T{       1 S>D ->       1  0 }T
T{       2 S>D ->       2  0 }T
T{      -1 S>D ->      -1 -1 }T
T{      -2 S>D ->      -2 -1 }T
T{ MIN-INT S>D -> MIN-INT -1 }T
T{ MAX-INT S>D -> MAX-INT  0 }T
```

F.6.1.2210 SIGN

```
: GP2 <# -1 SIGN 0 SIGN -1 SIGN 0 0 #> S" --" S= ;
T{ GP2 -> <TRUE> }T
```

F.6.1.2214 SM/REM

```
T{       0 S>D              1 SM/REM ->  0       0 }T
T{       1 S>D              1 SM/REM ->  0       1 }T
T{       2 S>D              1 SM/REM ->  0       2 }T
T{      -1 S>D              1 SM/REM ->  0      -1 }T
T{      -2 S>D              1 SM/REM ->  0      -2 }T
T{       0 S>D             -1 SM/REM ->  0       0 }T
T{       1 S>D             -1 SM/REM ->  0      -1 }T
T{       2 S>D             -1 SM/REM ->  0      -2 }T
T{      -1 S>D             -1 SM/REM ->  0       1 }T
T{      -2 S>D             -1 SM/REM ->  0       2 }T
T{       2 S>D              2 SM/REM ->  0       1 }T
T{      -1 S>D             -1 SM/REM ->  0       1 }T
```

```
T{       -2 S>D              -2 SM/REM ->  0        1 }T
T{        7 S>D               3 SM/REM ->  1        2 }T
T{        7 S>D              -3 SM/REM ->  1       -2 }T
T{       -7 S>D               3 SM/REM -> -1       -2 }T
T{       -7 S>D              -3 SM/REM -> -1        2 }T
T{  MAX-INT S>D               1 SM/REM ->  0  MAX-INT }T
T{  MIN-INT S>D               1 SM/REM ->  0  MIN-INT }T
T{  MAX-INT S>D         MAX-INT SM/REM ->  0        1 }T
T{  MIN-INT S>D         MIN-INT SM/REM ->  0        1 }T
T{       1S 1                 4 SM/REM ->  3  MAX-INT }T
T{        2 MIN-INT M*        2 SM/REM ->  0  MIN-INT }T
T{        2 MIN-INT M*  MIN-INT SM/REM ->  0        2 }T
T{        2 MAX-INT M*        2 SM/REM ->  0  MAX-INT }T
T{        2 MAX-INT M*  MAX-INT SM/REM ->  0        2 }T
T{  MIN-INT MIN-INT M*  MIN-INT SM/REM ->  0  MIN-INT }T
T{  MIN-INT MAX-INT M*  MIN-INT SM/REM ->  0  MAX-INT }T
T{  MIN-INT MAX-INT M*  MAX-INT SM/REM ->  0  MIN-INT }T
T{  MAX-INT MAX-INT M*  MAX-INT SM/REM ->  0  MAX-INT }T
```

x:contribution-138

F.6.1.2216 SOURCE

```
: GS1 S" SOURCE" 2DUP EVALUATE >R SWAP >R = R> R> = ;
T{ GS1 -> <TRUE> <TRUE> }T

: GS4 SOURCE >IN ! DROP ;
T{ GS4 123 456
   ->  }T
```

F.6.1.2220 SPACE

See F.6.1.1320 EMIT.

F.6.1.2230 SPACES

See F.6.1.1320 EMIT.

F.6.1.2250 STATE

```
T{ : GT8 STATE @ ; IMMEDIATE ->  }T
T{ GT8 -> 0 }T
T{ : GT9 GT8 LITERAL ; ->  }T
T{ GT9 0= -> <FALSE> }T
```

F.6.1.2260 SWAP

```
T{ 1 2 SWAP -> 2 1 }T
```

F.6.1.2270 THEN

See F.6.1.1700 IF.

F.6.1.2310 TYPE

See F.6.1.1320 EMIT.

F.6.1.2320 U.

See F.6.1.1320 EMIT.

F.6.1.2340 U<

```
T{         0         1 U< -> <TRUE>  }T
T{         1         2 U< -> <TRUE>  }T
T{         0 MID-UINT U< -> <TRUE>  }T
T{         0 MAX-UINT U< -> <TRUE>  }T
T{ MID-UINT MAX-UINT U< -> <TRUE>  }T
T{         0         0 U< -> <FALSE> }T
T{         1         1 U< -> <FALSE> }T
T{         1         0 U< -> <FALSE> }T
T{         2         1 U< -> <FALSE> }T
T{ MID-UINT         0 U< -> <FALSE> }T
T{ MAX-UINT         0 U< -> <FALSE> }T
T{ MAX-UINT MID-UINT U< -> <FALSE> }T
```

F.6.1.2360 UM*

```
T{ 0 0 UM* -> 0 0 }T
T{ 0 1 UM* -> 0 0 }T
T{ 1 0 UM* -> 0 0 }T
T{ 1 2 UM* -> 2 0 }T
T{ 2 1 UM* -> 2 0 }T
T{ 3 3 UM* -> 9 0 }T

T{ MID-UINT+1 1 RSHIFT 2 UM* ->  MID-UINT+1 0 }T
T{ MID-UINT+1          2 UM* ->           0 1 }T
T{ MID-UINT+1          4 UM* ->           0 2 }T
T{         1S          2 UM* -> 1S 1 LSHIFT 1 }T
T{   MAX-UINT   MAX-UINT UM* ->    1 1 INVERT }T
```

F.6.1.2370 UM/MOD

```
T{         0             0        1 UM/MOD -> 0         0 }T
T{         1             0        1 UM/MOD -> 0         1 }T
T{         1             0        2 UM/MOD -> 1         0 }T
T{         3             0        2 UM/MOD -> 1         1 }T
T{ MAX-UINT         2 UM*        2 UM/MOD -> 0 MAX-UINT }T
T{ MAX-UINT         2 UM* MAX-UINT UM/MOD -> 0         2 }T
T{ MAX-UINT MAX-UINT UM* MAX-UINT UM/MOD -> 0 MAX-UINT }T
```

F.6.1.2380 UNLOOP

```
T{ : GD6 ( PAT: {0 0},{0 0}{1 0}{1 1},{0 0}{1 0}{1 1}{2 0}{2 1}{2 2} )
     0 SWAP 0 DO
       I 1+ 0 DO
         I J + 3 = IF I UNLOOP I UNLOOP EXIT THEN 1+
       LOOP
     LOOP ; ->  }T
```

```
T{ 1 GD6 -> 1 }T
T{ 2 GD6 -> 3 }T
T{ 3 GD6 -> 4 1 2 }T
```

F.6.1.2390 UNTIL

```
T{ : GI4 BEGIN DUP 1+ DUP 5 > UNTIL ; ->  }T
T{ 3 GI4 -> 3 4 5 6 }T
T{ 5 GI4 -> 5 6 }T
T{ 6 GI4 -> 6 7 }T
```

F.6.1.2410 VARIABLE

```
T{ VARIABLE V1 ->     }T
T{    123 V1 ! ->     }T
T{        V1 @ -> 123 }T
```

F.6.1.2430 WHILE

```
T{ : GI3 BEGIN DUP 5 < WHILE DUP 1+ REPEAT ; ->  }T
T{ 0 GI3 -> 0 1 2 3 4 5 }T
T{ 4 GI3 -> 4 5 }T
T{ 5 GI3 -> 5 }T
T{ 6 GI3 -> 6 }T

T{ : GI5 BEGIN DUP 2 > WHILE
     DUP 5 < WHILE DUP 1+ REPEAT
     123 ELSE 345 THEN ; ->  }T
T{ 1 GI5 -> 1 345 }T
T{ 2 GI5 -> 2 345 }T
T{ 3 GI5 -> 3 4 5 123 }T
T{ 4 GI5 -> 4 5 123 }T
T{ 5 GI5 -> 5 123 }T
```

F.6.1.2450 WORD

```
: GS3 WORD COUNT SWAP C@ ;
T{ BL GS3 HELLO -> 5 CHAR H }T
T{ CHAR " GS3 GOODBYE" -> 7 CHAR G }T
T{ BL GS3
   DROP -> 0 }T                          \ Blank lines return zero-length strings
```

F.6.1.2490 XOR

```
T{ 0S 0S XOR -> 0S }T
T{ 0S 1S XOR -> 1S }T
T{ 1S 0S XOR -> 1S }T
T{ 1S 1S XOR -> 0S }T
```

F.6.1.2500 [

```
T{ : GC3 [ GC1 ] LITERAL ; ->  }T
T{ GC3 -> 58 }T
```

F.6.1.2510 [']

```
T{ : GT2 ['] GT1 ; IMMEDIATE ->  }T
T{ GT2 EXECUTE -> 123 }T
```

F.6.1.2520 [CHAR]

```
T{ : GC1 [CHAR] X     ; ->  }T
T{ : GC2 [CHAR] HELLO ; ->  }T
T{ GC1 -> 58 }T
T{ GC2 -> 48 }T
```

F.6.1.2540]

See F.6.1.2500 [.

F.6.2.0455 :NONAME

```
VARIABLE nn1
VARIABLE nn2
T{ :NONAME 1234 ; nn1 ! ->  }T
T{ :NONAME 9876 ; nn2 ! ->  }T
T{ nn1 @ EXECUTE -> 1234 }T
T{ nn2 @ EXECUTE -> 9876 }T
```

F.6.2.0620 ?DO

```
DECIMAL

: qd ?DO I LOOP ;
T{   789   789 qd ->  }T
T{ -9876 -9876 qd ->  }T
T{     5     0 qd -> 0 1 2 3 4 }T

: qd1 ?DO I 10 +LOOP ;
T{ 50 1 qd1 -> 1 11 21 31 41 }T
T{ 50 0 qd1 -> 0 10 20 30 40 }T

: qd2 ?DO I 3 > IF LEAVE ELSE I THEN LOOP ;
T{ 5 -1 qd2 -> -1 0 1 2 3 }T

: qd3 ?DO I 1 +LOOP ;
T{ 4  4 qd3 ->  }T
T{ 4  1 qd3 ->  1 2 3 }T
T{ 2 -1 qd3 -> -1 0 1 }T

: qd4 ?DO I -1 +LOOP ;
T{  4 4 qd4 ->  }T
T{  1 4 qd4 -> 4 3 2  1 }T
T{ -1 2 qd4 -> 2 1 0 -1 }T

: qd5 ?DO I -10 +LOOP ;
T{   1 50 qd5 -> 50 40 30 20 10   }T
T{   0 50 qd5 -> 50 40 30 20 10 0 }T
T{ -25 10 qd5 -> 10 0 -10 -20     }T
```

```
VARIABLE qditerations
VARIABLE qdincrement

: qd6 ( limit start increment -- )   qdincrement !
  0 qditerations !
  ?DO
     1 qditerations +!
     I
     qditerations @ 6 = IF LEAVE THEN
     qdincrement @
  +LOOP qditerations @
;

T{  4  4 -1 qd6 ->                    0  }T
T{  1  4 -1 qd6 ->  4  3  2  1        4  }T
T{  4  1 -1 qd6 ->  1  0 -1 -2 -3 -4  6  }T
T{  4  1  0 qd6 ->  1  1  1  1  1  1  6  }T
T{  0  0  0 qd6 ->                    0  }T
T{  1  4  0 qd6 ->  4  4  4  4  4  4  6  }T
T{  1  4  1 qd6 ->  4  5  6  7  8  9  6  }T
T{  4  1  1 qd6 ->  1  2  3           3  }T
T{  4  4  1 qd6 ->                    0  }T
T{  2 -1 -1 qd6 -> -1 -2 -3 -4 -5 -6  6  }T
T{ -1  2 -1 qd6 ->  2  1  0 -1        4  }T
T{  2 -1  0 qd6 -> -1 -1 -1 -1 -1 -1  6  }T
T{ -1  2  0 qd6 ->  2  2  2  2  2  2  6  }T
T{ -1  2  1 qd6 ->  2  3  4  5  6  7  6  }T
T{  2 -1  1 qd6 -> -1  0  1           3  }T
```

F.6.2.0698 ACTION-OF

```
T{ DEFER defer1 ->  }T
T{ : action-defer1 ACTION-OF defer1 ; ->  }T

T{ ' * ' defer1 DEFER! ->   }T
T{          2 3 defer1 -> 6 }T
T{ ACTION-OF defer1 -> ' * }T
T{    action-defer1 -> ' * }T

T{ ' + IS defer1 ->   }T
T{    1 2 defer1 -> 3 }T
T{ ACTION-OF defer1 -> ' + }T
T{    action-defer1 -> ' + }T
```

F.6.2.0825 BUFFER:

```
DECIMAL
T{ 127 CHARS BUFFER: TBUF1 ->  }T
T{ 127 CHARS BUFFER: TBUF2 ->  }T
```

```
\ Buffer is aligned
T{ TBUF1 ALIGNED -> TBUF1 }T

\ Buffers do not overlap
T{ TBUF2 TBUF1 - ABS 127 CHARS < -> <FALSE> }T

\ Buffer can be written to
1 CHARS CONSTANT /CHAR
: TFULL? ( c-addr n char -- flag )
  TRUE 2SWAP CHARS OVER + SWAP ?DO
    OVER I C@ = AND
  /CHAR +LOOP NIP
;

T{ TBUF1 127 CHAR * FILL   ->        }T
T{ TBUF1 127 CHAR * TFULL? -> <TRUE> }T

T{ TBUF1 127 0 FILL   ->        }T
T{ TBUF1 127 0 TFULL? -> <TRUE> }T
```

F.6.2.0855 C"

```
T{ : cq1 C" 123" ; ->  }T
T{ : cq2 C" " ;    ->  }T
T{ cq1 COUNT EVALUATE -> 123 }T
T{ cq2 COUNT EVALUATE ->     }T
```

F.6.2.0873 CASE

```
: cs1 CASE 1 OF 111 ENDOF
  2 OF 222 ENDOF
  3 OF 333 ENDOF
  >R 999 R>
  ENDCASE
;

T{ 1 cs1 -> 111 }T
T{ 2 cs1 -> 222 }T
T{ 3 cs1 -> 333 }T
T{ 4 cs1 -> 999 }T


: cs2 >R CASE
  -1 OF CASE R@ 1 OF 100 ENDOF
              2 OF 200 ENDOF
              >R -300 R>
        ENDCASE
     ENDOF
  -2 OF CASE R@ 1 OF -99 ENDOF
              >R -199 R>
        ENDCASE
```

```
    ENDOF
    >R 299 R>
  ENDCASE R> DROP ;

T{ -1 1 cs2 ->  100 }T
T{ -1 2 cs2 ->  200 }T
T{ -1 3 cs2 -> -300 }T
T{ -2 1 cs2 ->  -99 }T
T{ -2 2 cs2 -> -199 }T
T{  0 2 cs2 ->  299 }T
```

F.6.2.0945 COMPILE,

```
:NONAME DUP + ; CONSTANT dup+
T{ : q dup+ COMPILE, ; ->  }T
T{ : as [ q ] ; ->  }T
T{ 123 as -> 246 }T
```

F.6.2.1173 DEFER

```
T{ DEFER defer2 ->   }T
T{ ' * ' defer2 DEFER! ->  }T
T{   2 3 defer2 -> 6 }T

T{ ' + IS defer2 ->   }T
T{    1 2 defer2 -> 3 }T
```

F.6.2.1175 DEFER!

```
T{ DEFER defer3 ->  }T

T{ ' * ' defer3 DEFER! ->  }T
T{ 2 3 defer3 -> 6 }T

T{ ' + ' defer3 DEFER! ->  }T
T{ 1 2 defer3 -> 3 }T
```

F.6.2.1177 DEFER@

```
T{ DEFER defer4 ->  }T

T{ ' * ' defer4 DEFER! ->  }T
T{ 2 3 defer4 -> 6 }T
T{ ' defer4 DEFER@ -> ' * }T

T{ ' + IS defer4 ->  }T
T{ 1 2 defer4 -> 3 }T
T{ ' defer4 DEFER@ -> ' + }T
```

F.6.2.1342 ENDCASE

See F.6.2.0873 CASE.

F.6.2.1343 ENDOF

See F.6.2.0873 CASE.

F.6.2.1485 FALSE

```
T{ FALSE -> 0 }T
T{ FALSE -> <FALSE> }T
```

F.6.2.1660 HEX

See F.6.1.0750 BASE.

F.6.2.1675 HOLDS

```
T{ 0. <# S" Test" HOLDS #> S" Test" COMPARE -> 0 }T
```

F.6.2.1725 IS

```
T{ DEFER defer5 ->  }T
T{ : is-defer5 IS defer5 ; ->  }T

T{ ' * IS defer5 ->  }T
T{ 2 3 defer5 -> 6 }T

T{ ' + is-defer5 ->  }T
T{ 1 2 defer5 -> 3 }T
```

F.6.2.1950 OF

See F.6.2.0873 CASE.

F.6.2.2020 PARSE-NAME

```
T{ PARSE-NAME abcd S" abcd" S= -> <TRUE> }T
T{ PARSE-NAME   abcde   S" abcde" S= -> <TRUE> }T

\ test empty parse area
T{ PARSE-NAME
   NIP -> 0 }T  \ empty line
T{ PARSE-NAME
   NIP -> 0 }T  \ line with white space

T{ : parse-name-test ( "name1" "name2" -- n )
  PARSE-NAME PARSE-NAME S= ; ->  }T

T{ parse-name-test abcd abcd -> <TRUE> }T
T{ parse-name-test  abcd   abcd   -> <TRUE> }T
T{ parse-name-test abcde abcdf -> <FALSE> }T
T{ parse-name-test abcdf abcde -> <FALSE> }T
T{ parse-name-test abcde abcde
   -> <TRUE> }T
T{ parse-name-test abcde abcde
   -> <TRUE> }T  \ line with white space
```

F.6.2.2182 SAVE-INPUT

Testing with a file source

```
VARIABLE siv -1 siv !
```

```
: NeverExecuted
  ." This should never be executed" ABORT
;

11111 SAVE-INPUT

siv @

[IF]
  0 siv !
  RESTORE-INPUT
  NeverExecuted
[ELSE]
  \ Testing the ELSE part is executed
  22222
[THEN]

T{  -> 11111 0 22222 }T  \ 0 comes from RESTORE-INPUT
```

Testing with a string source

```
VARIABLE si_inc 0 si_inc !

: si1
  si_inc @ >IN +!
  15 si_inc !
;

: s$ S" SAVE-INPUT si1 RESTORE-INPUT 12345" ;

T{ s$ EVALUATE si_inc @ -> 0 2345 15 }T
```

Testing nesting

```
: read_a_line
  REFILL 0=
  ABORT" REFILL failed"
;

0 si_inc !
2VARIABLE 2res -1. 2res 2!

: si2
  read_a_line
  read_a_line
  SAVE-INPUT
  read_a_line
  read_a_line
  s$ EVALUATE 2res 2!
  RESTORE-INPUT
;
```

WARNING: do not delete or insert lines of text after si2 is called otherwise the next test will fail

```
si2
33333                   \ This line should be ignored
2res 2@ 44444           \ RESTORE-INPUT should return to this line

55555

T{  -> 0 0 2345 44444 55555 }T
```

F.6.2.2295 TO

See F.6.2.2405 VALUE.

F.6.2.2298 TRUE

```
T{ TRUE -> <TRUE> }T
T{ TRUE -> 0 INVERT }T
```

F.6.2.2405 VALUE

```
T{  111 VALUE v1 ->  }T
T{ -999 VALUE v2 ->  }T
T{ v1 ->  111 }T
T{ v2 -> -999 }T
T{ 222 TO v1 ->  }T
T{ v1 -> 222 }T

T{ : vd1 v1 ; ->  }T
T{ vd1 -> 222 }T

T{ : vd2 TO v2 ; ->  }T
T{ v2 -> -999 }T
T{ -333 vd2 ->  }T
T{ v2 -> -333 }T
T{ v1 ->  222 }T
```

F.6.2.2530 [COMPILE]

With default compilation semantics

```
T{ : [c1] [COMPILE] DUP ; IMMEDIATE ->  }T
T{ 123 [c1] -> 123 123 }T
```

With an immediate word

```
T{ : [c2] [COMPILE] [c1] ; ->  }T
T{ 234 [c2] -> 234 234 }T
```

With special compilation semantics

```
T{ : [cif] [COMPILE] IF ; IMMEDIATE ->  }T
T{ : [c3]  [cif] 111 ELSE 222 THEN ; ->  }T
T{ -1 [c3] -> 111 }T
T{  0 [c3] -> 222 }T
```

F.8 The optional Double-Number word set

Two additional constants are defined to assist tests in this word set:

```
MAX-INT 2/  CONSTANT HI-INT   \ 001...1
MIN-INT 2/  CONSTANT LO-INT   \ 110...1
```

Before anything can be tested, the text interpreter must be tested (F.8.3.2). Once the F.8.6.1.0360 2CONSTANT test has been preformed we can also define a number of double constants:

```
      1S MAX-INT 2CONSTANT MAX-2INT   \ 01...1
       0 MIN-INT 2CONSTANT MIN-2INT   \ 10...0
MAX-2INT      2/ 2CONSTANT HI-2INT    \ 001...1
MIN-2INT      2/ 2CONSTANT LO-2INT    \ 110...0
```

The rest of the word set can be tesed: F.8.6.1.1230 DNEGATE, F.8.6.1.1040 D+, F.8.6.1.1050 D-, F.8.6.1.1075 D0<, F.8.6.1.1080 D0=, F.8.6.1.1090 D2*, F.8.6.1.1100 D2/, F.8.6.1.1110 D<, F.8.6.1.1120 D=, F.8.6.1.0390 2LITERAL, F.8.6.1.0440 2VARIABLE, F.8.6.1.1210 DMAX, F.8.6.1.1220 DMIN, F.8.6.1.1140 D>S, F.8.6.1.1160 DABS, F.8.6.1.1830 M+, F.8.6.1.1820 M*/ and F.8.6.1.1070 D.R which also tests D. before moving on to the existion words with the F.8.6.2.0420 2ROT and F.8.6.2.1270 DU< tests.

F.8.3.2 Text interpreter input number conversion

```
T{  1. ->  1  0 }T
T{ -2. -> -2 -1 }T
T{ : rdl1  3. ; rdl1 ->  3  0 }T
T{ : rdl2 -4. ; rdl2 -> -4 -1 }T
```

F.8.6.1.0360 2CONSTANT

```
T{ 1 2 2CONSTANT 2c1 ->  }T
T{ 2c1 -> 1 2 }T

T{ : cd1 2c1 ; ->  }T
T{ cd1 -> 1 2 }T

T{ : cd2 2CONSTANT ; ->  }T
T{ -1 -2 cd2 2c2 ->  }T
T{ 2c2 -> -1 -2 }T

T{ 4 5 2CONSTANT 2c3 IMMEDIATE 2c3 -> 4 5 }T
T{ : cd6 2c3 2LITERAL ; cd6 -> 4 5 }T
```

F.8.6.1.0390 2LITERAL

```
T{ : cd1 [ MAX-2INT ] 2LITERAL ; ->  }T
T{ cd1 -> MAX-2INT }T

T{ 2VARIABLE 2v4 IMMEDIATE 5 6 2v4 2! ->  }T
T{ : cd7 2v4 [ 2@ ] 2LITERAL ; cd7 -> 5 6 }T
T{ : cd8 [ 6 7 ] 2v4 [ 2! ] ; 2v4 2@ -> 6 7 }T
```

F.8.6.1.0440 2VARIABLE

```
T{ 2VARIABLE 2v1 ->  }T
T{ 0. 2v1 2! ->    }T
T{    2v1 2@ -> 0. }T
T{ -1 -2 2v1 2! ->       }T
T{       2v1 2@ -> -1 -2 }T

T{ : cd2 2VARIABLE ; ->  }T
T{ cd2 2v2 ->  }T
T{ : cd3 2v2 2! ; ->  }T
T{ -2 -1 cd3 ->  }T
T{ 2v2 2@ -> -2 -1 }T

T{ 2VARIABLE 2v3 IMMEDIATE 5 6 2v3 2! ->  }T
T{ 2v3 2@ -> 5 6 }T
```

F.8.6.1.1040 D+

```
T{  0.  5. D+ ->  5. }T                      \ small integers
T{ -5.  0. D+ -> -5. }T
T{  1.  2. D+ ->  3. }T
T{  1. -2. D+ -> -1. }T
T{ -1.  2. D+ ->  1. }T
T{ -1. -2. D+ -> -3. }T
T{ -1.  1. D+ ->  0. }T

T{  0  0  0  5 D+ ->  0  5 }T                \ mid range integers
T{ -1  5  0  0 D+ -> -1  5 }T
T{  0  0  0 -5 D+ ->  0 -5 }T
T{  0 -5 -1  0 D+ -> -1 -5 }T
T{  0  1  0  2 D+ ->  0  3 }T
T{ -1  1  0 -2 D+ -> -1 -1 }T
T{  0 -1  0  2 D+ ->  0  1 }T
T{  0 -1 -1 -2 D+ -> -1 -3 }T
T{ -1 -1  0  1 D+ -> -1  0 }T

T{ MIN-INT 0 2DUP D+ -> 0 1 }T
T{ MIN-INT S>D MIN-INT 0 D+ -> 0 0 }T

T{  HI-2INT       1. D+ -> 0 HI-INT 1+ }T  \ large double integers
T{  HI-2INT     2DUP D+ -> 1S 1- MAX-INT }T
T{ MAX-2INT MIN-2INT D+ -> -1. }T
T{ MAX-2INT  LO-2INT D+ -> HI-2INT }T
T{  LO-2INT     2DUP D+ -> MIN-2INT }T
T{  HI-2INT MIN-2INT D+ 1. D+ -> LO-2INT }T
```

F.8.6.1.1050 D-

```
T{  0.  5. D- -> -5. }T              \ small integers
T{  5.  0. D- ->  5. }T
T{  0. -5. D- ->  5. }T
```

```
T{  1.  2. D- -> -1. }T
T{  1. -2. D- ->  3. }T
T{ -1.  2. D- -> -3. }T
T{ -1. -2. D- ->  1. }T
T{ -1. -1. D- ->  0. }T


T{  0  0  0  5 D- ->  0 -5 }T      \ mid-range integers
T{ -1  5  0  0 D- -> -1  5 }T
T{  0  0 -1 -5 D- ->  1  4 }T
T{  0 -5  0  0 D- ->  0 -5 }T
T{ -1  1  0  2 D- -> -1 -1 }T
T{  0  1 -1 -2 D- ->  1  2 }T
T{  0 -1  0  2 D- ->  0 -3 }T
T{  0 -1  0 -2 D- ->  0  1 }T
T{  0  0  0  1 D- ->  0 -1 }T

T{ MIN-INT 0 2DUP D- -> 0. }T
T{ MIN-INT S>D MAX-INT 0D- -> 1 1s }T


T{ MAX-2INT max-2INT D- -> 0. }T  \ large integers
T{ MIN-2INT min-2INT D- -> 0. }T
T{ MAX-2INT  hi-2INT D- -> lo-2INT DNEGATE }T
T{  HI-2INT  lo-2INT D- -> max-2INT }T
T{  LO-2INT  hi-2INT D- -> min-2INT 1. D+ }T
T{ MIN-2INT min-2INT D- -> 0. }T
T{ MIN-2INT  lo-2INT D- -> lo-2INT }T
```

F.8.6.1.1060 D.

See F.8.6.1.1070 D.R.

F.8.6.1.1070 D.R

```
MAX-2INT 71 73 M*/ 2CONSTANT dbl1
MIN-2INT 73 79 M*/ 2CONSTANT dbl2

: d>ascii  ( d -- caddr u )
   DUP >R <# DABS #S R> SIGN #>    ( -- caddr1 u )
   HERE SWAP 2DUP 2>R CHARS DUP ALLOT MOVE 2R>
;

dbl1 d>ascii 2CONSTANT "dbl1"
dbl2 d>ascii 2CONSTANT "dbl2"

: DoubleOutput
   CR ." You should see lines duplicated:" CR
    5 SPACES "dbl1" TYPE CR
    5 SPACES  dbl1  D. CR
    8 SPACES "dbl1" DUP >R TYPE CR
```

```
     5 SPACES  dbl1  R> 3 + D.R CR
     5 SPACES "dbl2" TYPE CR
     5 SPACES  dbl2  D. CR
    10 SPACES "dbl2" DUP >R TYPE CR
     5 SPACES  dbl2  R> 5 + D.R CR
;

T{ DoubleOutput ->  }T
```

F.8.6.1.1075 D0<

```
T{                0. D0< -> <FALSE> }T
T{                1. D0< -> <FALSE> }T
T{  MIN-INT        0 D0< -> <FALSE> }T
T{         0  MAX-INT D0< -> <FALSE> }T
T{           MAX-2INT D0< -> <FALSE> }T
T{               -1. D0< -> <TRUE>  }T
T{           MIN-2INT D0< -> <TRUE>  }T
```

F.8.6.1.1080 D0=

```
T{               1. D0= -> <FALSE> }T
T{ MIN-INT        0 D0= -> <FALSE> }T
T{          MAX-2INT D0= -> <FALSE> }T
T{       -1  MAX-INT D0= -> <FALSE> }T
T{               0. D0= -> <TRUE>  }T
T{              -1. D0= -> <FALSE> }T
T{        0  MIN-INT D0= -> <FALSE> }T
```

F.8.6.1.1090 D2*

```
T{              0. D2* -> 0. D2* }T
T{ MIN-INT       0 D2* -> 0 1 }T
T{          HI-2INT D2* -> MAX-2INT 1. D- }T
T{          LO-2INT D2* -> MIN-2INT }T
```

F.8.6.1.1100 D2/

```
T{       0. D2/ -> 0.        }T
T{       1. D2/ -> 0.        }T
T{      0 1 D2/ -> MIN-INT 0 }T
T{ MAX-2INT D2/ -> HI-2INT   }T
T{      -1. D2/ -> -1.       }T
T{ MIN-2INT D2/ -> LO-2INT   }T
```

F.8.6.1.1110 D<

```
T{       0.       1. D< -> <TRUE>  }T
T{       0.       0. D< -> <FALSE> }T
T{       1.       0. D< -> <FALSE> }T
T{      -1.       1. D< -> <TRUE>  }T
T{      -1.       0. D< -> <TRUE>  }T
T{      -2.      -1. D< -> <TRUE>  }T
```

```
T{       -1.       -2. D< -> <FALSE> }T
T{       -1. MAX-2INT D< -> <TRUE>  }T
T{ MIN-2INT MAX-2INT D< -> <TRUE>  }T
T{ MAX-2INT       -1. D< -> <FALSE> }T
T{ MAX-2INT MIN-2INT D< -> <FALSE> }T

T{ MAX-2INT 2DUP -1. D+ D< -> <FALSE> }T
T{ MIN-2INT 2DUP  1. D+ D< -> <TRUE>  }T
```

F.8.6.1.1120 D=

```
T{       -1.       -1. D= -> <TRUE>  }T
T{       -1.        0. D= -> <FALSE> }T
T{       -1.        1. D= -> <FALSE> }T
T{        0.       -1. D= -> <FALSE> }T
T{        0.        0. D= -> <TRUE>  }T
T{        0.        1. D= -> <FALSE> }T
T{        1.       -1. D= -> <FALSE> }T
T{        1.        0. D= -> <FALSE> }T
T{        1.        1. D= -> <TRUE>  }T

T{    0   -1    0  -1 D= -> <TRUE>  }T
T{    0   -1    0   0 D= -> <FALSE> }T
T{    0   -1    0   1 D= -> <FALSE> }T
T{    0    0    0  -1 D= -> <FALSE> }T
T{    0    0    0   0 D= -> <TRUE>  }T
T{    0    0    0   1 D= -> <FALSE> }T
T{    0    1    0  -1 D= -> <FALSE> }T
T{    0    1    0   0 D= -> <FALSE> }T
T{    0    1    0   1 D= -> <TRUE>  }T

T{ MAX-2INT MIN-2INT D= -> <FALSE> }T
T{ MAX-2INT       0. D= -> <FALSE> }T
T{ MAX-2INT MAX-2INT D= -> <TRUE>  }T
T{ MAX-2INT HI-2INT  D= -> <FALSE> }T
T{ MAX-2INT MIN-2INT D= -> <FALSE> }T
T{ MIN-2INT MIN-2INT D= -> <TRUE>  }T
T{ MIN-2INT LO-2INT  D= -> <FALSE> }T
T{ MIN-2INT MAX-2INT D= -> <FALSE> }T
```

F.8.6.1.1140 D>S

```
T{    1234  0 D>S ->  1234   }T
T{   -1234 -1 D>S -> -1234   }T
T{ MAX-INT  0 D>S -> MAX-INT }T
T{ MIN-INT -1 D>S -> MIN-INT }T
```

F.8.6.1.1160 DABS

```
T{       1. DABS -> 1.       }T
T{      -1. DABS -> 1.       }T
```

```
T{ MAX-2INT DABS -> MAX-2INT }T
T{ MIN-2INT 1. D+ DABS -> MAX-2INT }T
```

F.8.6.1.1210 DMAX

```
T{       1.       2. DMAX ->  2.      }T
T{       1.       0. DMAX ->  1.      }T
T{       1.      -1. DMAX ->  1.      }T
T{       1.       1. DMAX ->  1.      }T
T{       0.       1. DMAX ->  1.      }T
T{       0.      -1. DMAX ->  0.      }T
T{      -1.       1. DMAX ->  1.      }T
T{      -1.      -2. DMAX -> -1.      }T

T{ MAX-2INT  HI-2INT DMAX -> MAX-2INT }T
T{ MAX-2INT MIN-2INT DMAX -> MAX-2INT }T
T{ MIN-2INT MAX-2INT DMAX -> MAX-2INT }T
T{ MIN-2INT  LO-2INT DMAX -> LO-2INT  }T

T{ MAX-2INT       1. DMAX -> MAX-2INT }T
T{ MAX-2INT      -1. DMAX -> MAX-2INT }T
T{ MIN-2INT       1. DMAX ->  1.      }T
T{ MIN-2INT      -1. DMAX -> -1.      }T
```

F.8.6.1.1220 DMIN

```
T{       1.       2. DMIN ->  1.      }T
T{       1.       0. DMIN ->  0.      }T
T{       1.      -1. DMIN -> -1.      }T
T{       1.       1. DMIN ->  1.      }T
T{       0.       1. DMIN ->  0.      }T
T{       0.      -1. DMIN -> -1.      }T
T{      -1.       1. DMIN -> -1.      }T
T{      -1.      -2. DMIN -> -2.      }T

T{ MAX-2INT  HI-2INT DMIN -> HI-2INT  }T
T{ MAX-2INT MIN-2INT DMIN -> MIN-2INT }T
T{ MIN-2INT MAX-2INT DMIN -> MIN-2INT }T
T{ MIN-2INT  LO-2INT DMIN -> MIN-2INT }T

T{ MAX-2INT       1. DMIN ->  1.      }T
T{ MAX-2INT      -1. DMIN -> -1.      }T
T{ MIN-2INT       1. DMIN -> MIN-2INT }T
T{ MIN-2INT      -1. DMIN -> MIN-2INT }T
```

F.8.6.1.1230 DNEGATE

```
T{   0. DNEGATE ->  0. }T
T{   1. DNEGATE -> -1. }T
T{  -1. DNEGATE ->  1. }T
T{ max-2int DNEGATE -> min-2int SWAP 1+ SWAP }T
T{ min-2int SWAP 1+ SWAP DNEGATE -> max-2int }T
```

F.8.6.1.1820 M*/

To correct the result if the division is floored, only used when necessary, i.e., negative quotient and remainder $\neq 0$.

```
: ?floored [ -3 2 / -2 = ] LITERAL IF 1. D- THEN ;

T{       5.       7            11 M*/ ->  3. }T
T{       5.      -7            11 M*/ -> -3. ?floored }T
T{      -5.       7            11 M*/ -> -3. ?floored }T
T{      -5.      -7            11 M*/ ->  3. }T
T{ MAX-2INT       8            16 M*/ -> HI-2INT }T
T{ MAX-2INT      -8            16 M*/ -> HI-2INT DNEGATE ?floored }T
T{ MIN-2INT       8            16 M*/ -> LO-2INT }T
T{ MIN-2INT      -8            16 M*/ -> LO-2INT DNEGATE }T

T{ MAX-2INT MAX-INT        MAX-INT M*/ -> MAX-2INT }T
T{ MAX-2INT MAX-INT 2/     MAX-INT M*/ -> MAX-INT 1- HI-2INT NIP }T
T{ MIN-2INT LO-2INT NIP DUP NEGATE M*/ -> MIN-2INT }T
T{ MIN-2INT LO-2INT NIP 1- MAX-INT M*/ -> MIN-INT 3 + HI-2INT NIP 2 + }T
T{ MAX-2INT LO-2INT NIP DUP NEGATE M*/ -> MAX-2INT DNEGATE }T
T{ MIN-2INT MAX-INT            DUP M*/ -> MIN-2INT }T
```

F.8.6.1.1830 M+

```
T{ HI-2INT   1 M+ -> HI-2INT   1. D+ }T
T{ MAX-2INT -1 M+ -> MAX-2INT -1. D+ }T
T{ MIN-2INT  1 M+ -> MIN-2INT  1. D+ }T
T{ LO-2INT  -1 M+ -> LO-2INT  -1. D+ }T
```

F.8.6.2.0420 2ROT

```
T{       1.       2. 3. 2ROT ->       2. 3.       1. }T
T{ MAX-2INT MIN-2INT 1. 2ROT -> MIN-2INT 1. MAX-2INT }T
```

F.8.6.2.0435 2VALUE

```
T{ 1 2 2VALUE t2val ->  }T
T{ t2val -> 1 2 }T

T{ 3 4 TO t2val ->  }T
T{ t2val -> 3 4 }T

: sett2val t2val 2SWAP TO t2val ;
T{ 5 6 sett2val t2val -> 3 4 5 6 }T
```

F.8.6.2.1270 DU<

```
T{       1.       1. DU< -> <FALSE> }T
T{       1.      -1. DU< -> <TRUE>  }T
T{      -1.       1. DU< -> <FALSE> }T
T{      -1.      -2. DU< -> <FALSE> }T

T{ MAX-2INT  HI-2INT DU< -> <FALSE> }T
T{  HI-2INT MAX-2INT DU< -> <TRUE>  }T
T{ MAX-2INT MIN-2INT DU< -> <TRUE>  }T
```

```
T{ MIN-2INT MAX-2INT DU< -> <FALSE> }T
T{ MIN-2INT  LO-2INT DU< -> <TRUE>  }T
```

F.9 The Exception word set

The test F.9.6.1.0875 CATCH also test THROW. This should be followed by the test F.9.6.2.0680 ABORT" which also test ABORT. Finally, the general exception handling is tested in F.9.3.6.

F.9.3.6 Exception handling

Ideally all of the throw codes should be tested. Here only the thow code for an "Undefined Word" exception is tested, assuming that the word $$UndefedWord$$ is undefined.

```
DECIMAL
: t7 S" 333 $$UndefedWord$$ 334" EVALUATE 335 ;
: t8 S" 222 t7 223" EVALUATE 224 ;
: t9 S" 111 112 t8 113" EVALUATE 114 ;

T{ 6 7 ' t9 c6 3 -> 6 7 13 3 }T
```

F.9.6.1.0875 CATCH

See F.9.6.1.2275 THROW.

F.9.6.1.2275 THROW

```
DECIMAL

: t1 9 ;
: c1 1 2 3 ['] t1 CATCH ;
T{ c1 -> 1 2 3 9 0 }T  \ No THROW executed

: t2 8 0 THROW ;
: c2 1 2 ['] t2 CATCH ;
T{ c2 -> 1 2 8 0 }T  \ 0 THROW does nothing

: t3 7 8 9 99 THROW ;
: c3 1 2 ['] t3 CATCH ;
T{ c3 -> 1 2 99 }T  \ Restores stack to CATCH depth

: t4 1- DUP 0> IF RECURSE ELSE 999 THROW -222 THEN ;
: c4 3 4 5 10 ['] t4 CATCH -111 ;
T{ c4 -> 3 4 5 0 999 -111 }T      \ Test return stack unwinding

: t5 2DROP 2DROP 9999 THROW ;
: c5 1 2 3 4 ['] t5 CATCH          \ Test depth restored correctly
  DEPTH >R DROP 2DROP 2DROP R> ;   \ after stack has been emptied
T{ c5 -> 5 }T
```

F.9.6.2.0670 ABORT

See F.9.6.2.0680 ABORT".

F.9.6.2.0680 ABORT"

```
DECIMAL
 -1 CONSTANT exc_abort
 -2 CONSTANT exc_abort"
```

```
-13 CONSTANT exc_undef
: t6 ABORT ;
```

The 77 in `t10` is necessary for the second `ABORT"` test as the data stack is restored to a depth of 2 when `THROW` is executed. The 77 ensures the top of stack value is known for the results check.

```
: t10 77 SWAP ABORT" This should not be displayed" ;
: c6 CATCH
  CASE exc_abort  OF 11 ENDOF
       exc_abort" OF 12 ENDOF
       exc_undef  OF 13 ENDOF
  ENDCASE
;

T{ 1 2 '  t6 c6 -> 1 2 11  }T  \ Test that ABORT is caught
T{ 3 0 ' t10 c6 -> 3 77    }T  \ ABORT" does nothing
T{ 4 5 ' t10 c6 -> 4 77 12 }T  \ ABORT" caught, no message
```

F.10 The optional Facility word set

F.10.6.2.1306.40 EKEY>FKEY

```
: TFKEY" ( "ccc<quote>" -- u flag )
   CR ." Please press " POSTPONE ." EKEY EKEY>FKEY ;

T{ TFKEY" <left>"  -> K-LEFT  <TRUE> }T
T{ TFKEY" <right>" -> K-RIGHT <TRUE> }T
T{ TFKEY" <up>"    -> K-UP    <TRUE> }T
T{ TFKEY" <down>"  -> K-DOWN  <TRUE> }T
T{ TFKEY" <home>"  -> K-HOME  <TRUE> }T
T{ TFKEY" <end>"   -> K-END   <TRUE> }T
T{ TFKEY" <prior>" -> K-PRIOR <TRUE> }T
T{ TFKEY" <next>"  -> K-NEXT  <TRUE> }T

T{ TFKEY" <F1>"  -> K-F1  <TRUE> }T
T{ TFKEY" <F2>"  -> K-F2  <TRUE> }T
T{ TFKEY" <F3>"  -> K-F3  <TRUE> }T
T{ TFKEY" <F4>"  -> K-F4  <TRUE> }T
T{ TFKEY" <F5>"  -> K-F5  <TRUE> }T
T{ TFKEY" <F6>"  -> K-F6  <TRUE> }T
T{ TFKEY" <F7>"  -> K-F7  <TRUE> }T
T{ TFKEY" <F8>"  -> K-F8  <TRUE> }T
T{ TFKEY" <F9>"  -> K-F9  <TRUE> }T
T{ TFKEY" <F10>" -> K-F10 <TRUE> }T
T{ TFKEY" <F11>" -> K-F11 <TRUE> }T
T{ TFKEY" <F11>" -> K-F12 <TRUE> }T

T{ TFKEY" <shift-left>" -> K-LEFT K-SHIFT-MASK OR <TRUE> }T
T{ TFKEY" <ctrl-left>"  -> K-LEFT K-CTRL-MASK  OR <TRUE> }T
T{ TFKEY" <alt-left>"   -> K-LEFT K-ALT-MASK   OR <TRUE> }T
```

```
T{ TFKEY" <a>" SWAP EKEY>CHAR -> <FALSE> CHAR a <TRUE> }T
```

F.11 The optional File-Access word set

These tests create files in the current directory, if all goes well these will be deleted. If something fails they may not be deleted. If this is a problem ensure you set a suitable directory before running this test. Currently, there is no ANS standard way of doing this. the file names used in these test are: "`fatest1.txt`", "`fatest2.txt`" and "`fatest3.txt`".

The test F.11.6.1.1010 `CREATE-FILE` also tests `CLOSE-FILE`, F.11.6.1.2485 `WRITE-LINE` also tests `W/O` and `OPEN-FILE`, F.11.6.1.2090 `READ-LINE` includes a test for `R/O`, F.11.6.1.2142 `REPOSITION-FILE` includes tests for `R/W`, `WRITE-FILE`, `READ-FILE`, `FILE-POSITION`, and `S"`. The F.11.6.1.1522 `FILE-SIZE` test includes a test for `BIN`. The test F.11.6.1.2147 `RESIZE-FILE` should then be run followed by the F.11.6.1.1190 `DELETE-FILE` test.

The F.11.6.1.0080 `(` test should be next, followed by F.11.6.1.2218 `SOURCE-ID` the test which test the extended versions of `(` and `SOURCE-ID` respectively.

Finally F.11.6.2.2130 `RENAME-FILE` tests the extended words `RENAME-FILE`, `FILE-STATUS`, and `FLUSH-FILE`.

F.11.6.1.0080 `(`

```
T{ ( 1 2 3
     4 5 6
     7 8 9 ) 11 22 33 -> 11 22 33 }T
```

F.11.6.1.1010 `CREATE-FILE`

```
: fn1 S" fatest1.txt" ;
VARIABLE fid1

T{ fn1 R/W CREATE-FILE SWAP fid1 ! -> 0 }T
T{ fid1 @ CLOSE-FILE -> 0 }T
```

F.11.6.1.1190 `DELETE-FILE`

```
T{ fn2 DELETE-FILE -> 0 }T
T{ fn2 R/W BIN OPEN-FILE SWAP DROP -> 0 }T
T{ fn2 DELETE-FILE -> 0 }T
```

F.11.6.1.1522 `FILE-SIZE`

```
: cbuf buf bsize 0 FILL ;
: fn2 S" fatest2.txt" ;
VARIABLE fid2
: setpad PAD 50 0 DO I OVER C! CHAR+ LOOP DROP ;

setpad
```

Note: If anything else is defined `setpad` must be called again as the pad may move

```
T{ fn2 R/W BIN CREATE-FILE SWAP fid2 ! -> 0 }T
T{ PAD 50 fid2 @ WRITE-FILE fid2 @ FLUSH-FILE -> 0 0 }T
T{ fid2 @ FILE-SIZE -> 50. 0 }T
```

```
T{ 0. fid2 @ REPOSITION-FILE -> 0 }T
T{ cbuf buf 29 fid2 @ READ-FILE -> 29 0 }T
T{ PAD 29 buf 29 COMPARE -> 0 }T
T{ PAD 30 buf 30 COMPARE -> 1 }T
T{ cbuf buf 29 fid2 @ READ-FILE -> 21 0 }T
T{ PAD 29 + 21 buf 21 COMPARE -> 0 }T
T{ fid2 @ FILE-SIZE DROP fid2 @ FILE-POSITION DROP D= -> <TRUE> }T
T{ buf 10 fid2 @ READ-FILE -> 0 0 }T
T{ fid2 @ CLOSE-FILE -> 0 }T
```

F.11.6.1.1718 INCLUDED

See F.11.6.2.2144.50 REQUIRED.

F.11.6.1.2090 READ-LINE

```
200 CONSTANT bsize
CREATE buf bsize ALLOT
VARIABLE #chars

T{ fn1 R/O OPEN-FILE SWAP fid1 ! -> 0 }T
T{ fid1 @ FILE-POSITION -> 0. 0 }T
T{ buf 100 fid1 @ READ-LINE ROT DUP #chars ! ->
   <TRUE> 0 line1 SWAP DROP }T
T{ buf #chars @ line1 COMPARE -> 0 }T
T{ fid1 @ CLOSE-FILE -> 0 }T
```

F.11.6.1.2142 REPOSITION-FILE

```
: line2 S" Line 2 blah blah blah" ;
: rl1 buf 100 fid1 @ READ-LINE ;
2VARIABLE fp

T{ fn1 R/W OPEN-FILE SWAP fid1 ! -> 0 }T
T{ fid1 @ FILE-SIZE DROP fid1 @ REPOSITION-FILE -> 0 }T
T{ fid1 @ FILE-SIZE -> fid1 @ FILE-POSITION }T

T{ line2 fid1 @ WRITE-FILE -> 0 }T
T{ 10. fid1 @ REPOSITION-FILE -> 0 }T
T{ fid1 @ FILE-POSITION -> 10. 0 }T

T{ 0. fid1 @ REPOSITION-FILE -> 0 }T
T{ rl1 -> line1 SWAP DROP <TRUE> 0 }T
T{ rl1 -> ROT DUP #chars ! }T<TRUE> 0 line2 SWAP DROP
T{ buf #chars @ line2 COMPARE -> 0 }T
T{ rl1 -> 0 <FALSE> 0 }T

T{ fid1 @ FILE-POSITION ROT ROT fp 2! -> 0 }T
T{ fp 2@ fid1 @ FILE-SIZE DROP D= -> <TRUE> }T
T{ S" " fid1 @ WRITE-LINE -> 0 }T
T{ S" " fid1 @ WRITE-LINE -> 0 }T
T{ fp 2@ fid1 @ REPOSITION-FILE -> 0 }T
```

```
T{ rl1 -> 0 <TRUE>  0 }T
T{ rl1 -> 0 <TRUE>  0 }T
T{ rl1 -> 0 <FALSE> 0 }T
T{ fid1 @ CLOSE-FILE -> 0 }T
```

F.11.6.1.2147 RESIZE-FILE

```
setpad
T{ fn2 R/W BIN OPEN-FILE SWAP fid2 ! -> 0 }T
T{ 37. fid2 @ RESIZE-FILE -> 0 }T
T{ fid2 @ FILE-SIZE -> 37. 0 }T
T{ 0. fid2 @ REPOSITION-FILE -> 0 }T
T{ cbuf buf 100 fid2 @ READ-FILE -> 37 0 }T
T{ PAD 37 buf 37 COMPARE -> 0 }T
T{ PAD 38 buf 38 COMPARE -> 1 }T
T{ 500. fid2 @ RESIZE-FILE -> 0 }T
T{ fid2 @ FILE-SIZE -> 500. 0 }T
T{ 0. fid2 @ REPOSITION-FILE -> 0 }T
T{ cbuf buf 100 fid2 @ READ-FILE -> 100 0 }T
T{ PAD 37 buf 37 COMPARE -> 0 }T
T{ fid2 @ CLOSE-FILE -> 0 }T
```

F.11.6.1.2218 SOURCE-ID

```
T{ SOURCE-ID DUP -1 = SWAP 0= OR -> <FALSE> }T
```

F.11.6.1.2485 WRITE-LINE

```
: line1 S" Line 1" ;

T{ fn1 W/O OPEN-FILE SWAP fid1 ! -> 0 }T
T{ line1 fid1 @ WRITE-LINE -> 0 }T
T{ fid1 @ CLOSE-FILE -> 0 }T
```

F.11.6.2.1714 INCLUDE

See F.11.6.2.2144.50 REQUIRED.

F.11.6.2.2130 RENAME-FILE

```
: fn3 S" fatest3.txt" ;
: >end fid1 @ FILE-SIZE DROP fid1 @ REPOSITION-FILE ;

T{ fn3 DELETE-FILE DROP ->  }T
T{ fn1 fn3 RENAME-FILE -> 0 }T
\ Return value is undefined
T{ fn1 FILE-STATUS SWAP DROP 0= -> <FALSE> }T
T{ fn3 FILE-STATUS SWAP DROP 0= -> <TRUE>  }T
T{ fn3 R/W OPEN-FILE SWAP fid1 ! -> 0 }T
T{ >end -> 0 }T
T{ S" Final line" fid1 @ WRITE-LINE -> 0 }T
T{ fid1 @ FLUSH-FILE -> 0 }T  \ Can only test FLUSH-FILE doesn't fail
T{ fid1 @ CLOSE-FILE -> 0 }T
```

```
\ Tidy the test folder
T{ fn3 DELETE-FILE DROP ->  }T
```

F.11.6.2.2144.10 REQUIRE

See F.11.6.2.2144.50 REQUIRED.

F.11.6.2.2144.50 REQUIRED

This test requires two additional files: `required-helper1.fs` and `required-helper2.fs`. Both of which hold the text:

```
    1+
```

As for the test themselves:

```
T{ 0
   S" required-helper1.fs" REQUIRED \ Increment TOS
   REQUIRE required-helper1.fs \ Ignore - already loaded
   INCLUDE required-helper1.fs \ Increment TOS
-> 2 }T


T{ 0
   INCLUDE required-helper2.fs \ Increment TOS
   S" required-helper2.fs" REQUIRED \ Ignored - already loaded
   REQUIRE required-helper2.fs \ Ignored - already loaded
   S" required-helper2.fs" INCLUDED \ Increment TOS
-> 2 }T
```

F.12 The optional Floating-Point word set

F.12.6.2.1489 FATAN2

```
[UNDEFINED]  NaN [IF]  0e 0e F/ FCONSTANT  NaN [THEN]
[UNDEFINED] +Inf [IF]  1e 0e F/ FCONSTANT +Inf [THEN]
[UNDEFINED] -Inf [IF] -1e 0e F/ FCONSTANT -Inf [THEN]

TRUE verbose !
DECIMAL
```

The test harness default for `EXACT?` is TRUE. Uncomment the following line if your system needs it to be FALSE

```
\ SET-NEAR

VARIABLE #errors 0 #errors !

:NONAME ( c-addr u -- )
  ( Display an error message followed by the line that had the error.  )
  1 #errors +! error1 ; error-xt !

[UNDEFINED] pi [IF]
  0.31415926535897932384626433E1 FCONSTANT pi
[THEN]
```

```
[UNDEFINED] -pi [IF]
  pi FNEGATE FCONSTANT -pi
[THEN]

FALSE [IF]
   0.78539816339744830961566E0 FCONSTANT   pi/4
  -0.78539816339744830961566E0 FCONSTANT  -pi/4
   0.15707963267948966192313E1 FCONSTANT   pi/2
  -0.15707963267948966192313E1 FCONSTANT  -pi/2
   0.47123889803846898576939E1 FCONSTANT  3pi/2
   0.23561944901923449288469E1 FCONSTANT  3pi/4
  -0.23561944901923449288469E1 FCONSTANT -3pi/4
[ELSE]
   pi   4e F/ FCONSTANT   pi/4
  -pi   4e F/ FCONSTANT  -pi/4
   pi   2e F/ FCONSTANT   pi/2
  -pi   2e F/ FCONSTANT  -pi/2
   pi/2 3e F* FCONSTANT  3pi/2
   pi/4 3e F* FCONSTANT  3pi/4
  -pi/4 3e F* FCONSTANT -3pi/4
[THEN]

verbose @ [IF]
  :NONAME ( -- fp.separate?  )
     DEPTH >R 1e DEPTH R> FDROP 2R> = ; EXECUTE
  CR .( floating-point and data stacks )
  [IF] .( *separate* ) [ELSE] .( *not separate* ) [THEN]
  CR
[THEN]

TESTING normal values

\   y   x              rad         deg
T{  0e  1e FATAN2 ->   0e   R}T  \   0
T{  1e  1e FATAN2 ->   pi/4 R}T  \  45
T{  1e  0e FATAN2 ->   pi/2 R}T  \  90
T{ -1e -1e FATAN2 -> -3pi/4 R}T  \ 135
T{  0e -1e FATAN2 ->   pi   R}T  \ 180
T{ -1e  1e FATAN2 ->  -pi/4 R}T  \ 225
T{ -1e  0e FATAN2 ->  -pi/2 R}T  \ 270
T{ -1e  1e FATAN2 ->  -pi/4 R}T  \ 315

TESTING Single UNIX 3 special values spec

\ ISO C / Single UNIX Specification Version 3:
\   http://www.unix.org/single_unix_specification/
\ Select "Topic", then "Math Interfaces", then "atan2()":
\   http://www.opengroup.org/onlinepubs/009695399/
\   functions/atan2f.html
```

```
\ If y is +/-0 and x is < 0, +/-pi shall be returned.
T{  0e -1e FATAN2 ->  pi R}T
T{ -0e -1e FATAN2 -> -pi R}T

\ If y is +/-0 and x is > 0, +/-0 shall be returned.
T{  0e  1e FATAN2 ->  0e R}T
T{ -0e  1e FATAN2 -> -0e R}T

\ If y is < 0 and x is +/-0, -pi/2 shall be returned.
T{ -1e  0e FATAN2 -> -pi/2 R}T
T{ -1e -0e FATAN2 -> -pi/2 R}T

\ If y is > 0 and x is +/-0, pi/2 shall be returned.
T{  1e  0e FATAN2 -> pi/2 R}T
T{  1e -0e FATAN2 -> pi/2 R}T

TESTING Single UNIX 3 special values optional spec

\ Optional ISO C / single UNIX specs:

\ If either x or y is NaN, a NaN shall be returned.
T{ NaN  1e FATAN2 -> NaN R}T
T{  1e NaN FATAN2 -> NaN R}T
T{ NaN NaN FATAN2 -> NaN R}T

\ If y is +/-0 and x is -0, +/-pi shall be returned.
T{  0e -0e FATAN2 ->  pi R}T
T{ -0e -0e FATAN2 -> -pi R}T

\ If y is +/-0 and x is +0, +/-0 shall be returned.
T{  0e  0e FATAN2 -> +0e R}T
T{ -0e  0e FATAN2 -> -0e R}T

\ For finite values of +/-y > 0, if x is -Inf, +/-pi shall be returned.
T{  1e -Inf FATAN2 ->  pi R}T
T{ -1e -Inf FATAN2 -> -pi R}T

\ For finite values of +/-y > 0, if x is +Inf, +/-0 shall be returned.
T{  1e +Inf FATAN2 -> +0e R}T
T{ -1e +Inf FATAN2 -> -0e R}T

\ For finite values of x, if y is +/-Inf, +/-pi/2 shall be returned.
T{ +Inf  1e FATAN2 ->  pi/2 R}T
T{ +Inf -1e FATAN2 ->  pi/2 R}T
T{ +Inf  0e FATAN2 ->  pi/2 R}T
T{ +Inf -0e FATAN2 ->  pi/2 R}T
T{ -Inf  1e FATAN2 -> -pi/2 R}T
T{ -Inf -1e FATAN2 -> -pi/2 R}T
```

```
T{ -Inf  0e FATAN2 -> -pi/2 R}T
T{ -Inf -0e FATAN2 -> -pi/2 R}T

\ If y is +/-Inf and x is -Inf, +/-3pi/4 shall be returned.
T{ +Inf -Inf FATAN2 ->  3pi/4 R}T
T{ -Inf -Inf FATAN2 -> -3pi/4 R}T

\ If y is +/-Inf and x is +Inf, +/-pi/4 shall be returned.
T{ +Inf +Inf FATAN2 ->  pi/4 R}T
T{ -Inf +Inf FATAN2 -> -pi/4 R}T

verbose @ [IF]
  CR .( #ERRORS: ) #errors @ . CR
[THEN]
```

F.12.6.2.1627 FTRUNC

```
SET-EXACT

T{ -0E          FTRUNC F0= -> <TRUE> }T
T{ -1E-9        FTRUNC F0= -> <TRUE> }T
T{ -0.9E        FTRUNC F0= -> <TRUE> }T
T{ -1E  1E-5 F+ FTRUNC F0= -> <TRUE> }T
T{ 0E           FTRUNC     ->  0E   R}T
T{ 1E-9         FTRUNC     ->  0E   R}T
T{ -1E -1E-5 F+ FTRUNC     -> -1E   R}T
T{ 3.14E        FTRUNC     ->  3E   R}T
T{ 3.99E        FTRUNC     ->  3E   R}T
T{ 4E           FTRUNC     ->  4E   R}T
T{ -4E          FTRUNC     -> -4E   R}T
T{ -4.1E        FTRUNC     -> -4E   R}T
```

F.12.6.2.1628 FVALUE

```
T{ 0e0 FVALUE Tval ->  }T
T{ Tval -> 0e0 R}T
T{ 1e0 TO Tval ->  }T
T{ Tval -> 1e0 R}T

: setTval Tval FSWAP TO Tval ;
T{ 2e0 setTval Tval -> 1e0 2e0 RR}T

T{ 5e0 TO Tval ->  }T
: [execute] EXECUTE ; IMMEDIATE
T{ ' Tval ] [execute] [ -> 2e0 R}T
```

F.14 The optional Memory-Allocation word set

These test require a new variable to hold the address of the allocated memory. Two helper words are defined to populate the allocated memory and to check the memory:

```
VARIABLE addr

: write-cell-mem ( addr n -- )
   1+ 1 DO I OVER ! CELL+ LOOP DROP
;

: check-cell-mem ( addr n -- )
   1+ 1 DO
     I SWAP >R >R
     T{ R> ( I ) -> R@ ( addr ) @ }T
     R> CELL+
  LOOP DROP
;

: write-char-mem ( addr n -- )
  1+ 1 DO I OVER C! CHAR+ LOOP DROP
;

: check-char-mem ( addr n -- )
  1+ 1 DO
     I SWAP >R >R
     T{ R> ( I ) -> R@ ( addr ) C@ }T
     R> CHAR+
  LOOP DROP
;
```

The test F.14.6.1.0707 ALLOCATE includes a test for FREE.

F.14.6.1.0707 ALLOCATE

```
    VARIABLE datsp
    HERE datsp !

    T{ 50 CELLS ALLOCATE SWAP addr ! -> 0 }T
    T{ addr @ ALIGNED -> addr @ }T  \ Test address is aligned
    T{ HERE -> datsp @ }T           \ Check data space pointer is unaffected
    addr @ 50 write-cell-mem
    addr @ 50 check-cell-mem        \ Check we can access the heap
    T{ addr @ FREE -> 0 }T

    T{ 99 ALLOCATE SWAP addr ! -> 0 }T
    T{ addr @ ALIGNED -> addr @ }T  \ Test address is aligned
    T{ addr @ FREE -> 0 }T
    T{ HERE -> datsp @ }T           \ Data space pointer unaffected by FREE


    T{ -1 ALLOCATE SWAP DROP 0= -> <FALSE> }T  \ Memory allocate failed
```

F.14.6.1.1605 FREE

See F.14.6.1.0707 ALLOCATE

F.14.6.1.2145 RESIZE

```
T{ 50 CHARS ALLOCATE SWAP addr ! -> 0 }T
addr @ 50 write-char-mem addr @ 50 check-char-mem
```

\ Resize smaller does not change content.

```
T{ addr @ 28 CHARS RESIZE SWAP addr ! -> 0 }T
addr @ 28 check-char-mem
```

\ Resize larger does not change original content.

```
T{ addr @ 100 CHARS RESIZE SWAP addr ! -> 0 }T
addr @ 28 check-char-mem
```

\ Resize error does not change *addr*

```
T{ addr @ -1 RESIZE 0= -> addr @ <FALSE> }T
```

```
T{ addr @ FREE -> 0 }T
T{ HERE -> datsp @ }T  \ Data space pointer is unaffected
```

F.15 The optional Programming-Tools word set

F.15.6.2.0702 AHEAD

```
T{ : pt1 AHEAD 1111 2222 THEN 3333 ; ->  }T
T{ pt1 -> 3333 }T
```

x:cs-drop

F.15.6.2.—— CS-DROP

The following test assures that CS-DROP actually removes the top most dest item from the control-flow stack:

```
T{ 99 :NONAME BEGIN [ CS-DROP ] ; DROP -> 99 }T
```

The following test assures that CS-PICK can copy *orig* items and CS-DROP can discard them:

```
T{ 99 :NONAME IF [ 1 CS-PICK CS-DROP ] THEN ; DROP -> 99 }T
```

F.15.6.2.1015 CS-PICK

```
: ?repeat
  0 CS-PICK POSTPONE UNTIL
; IMMEDIATE

VARIABLE pt4

: <= > 0= ;

T{ : pt5  ( n1 -- )
     pt4 !
     BEGIN
       -1 pt4 +!
       pt4 @ 4 <= ?repeat     \ Back to BEGIN if false
       111
       pt4 @ 3 <= ?repeat
```

```
        222
        pt4 @ 2 <= ?repeat
        333
        pt4 @ 1 =
      UNTIL
    ; ->  }T

T{ 6 pt5 -> 111 111 222 111 222 333 111 222 333 }T
```

F.15.6.2.1020 CS-ROLL

```
T{ : ?DONE ( dest -- orig dest )      \ Same as WHILE
     POSTPONE IF 1 CS-ROLL
   ; IMMEDIATE ->  }T
T{ : pt6
     >R
     BEGIN
       R@
     ?DONE
       R@
       R> 1- >R
     REPEAT
     R> DROP
   ; ->  }T

T{ 5 pt6 -> 5 4 3 2 1 }T

: mix_up 2 CS-ROLL ; IMMEDIATE   \ cs-rot

: pt7 ( f3 f2 f1 -- ?  )
  IF 1111 ROT ROT   ( -- 1111 f3 f2 )      ( cs: -- o1 )
     IF 2222 SWAP   ( -- 1111 2222 f3 )    ( cs: -- o1 o2 )
       IF                                  ( cs: -- o1 o2 o3 )
         3333 mix_up ( -- 1111 2222 3333 ) ( cs: -- o2 o3 o1 )
       THEN                                ( cs: -- o2 o3 )
       4444     \ Hence failure of first IF comes here and falls through
     THEN                                  ( cs: -- o2 )
     5555       \ Failure of 3rd IF comes here
   THEN                                    ( cs: -- )
   6666         \ Failure of 2nd IF comes here
  ;

T{ -1 -1 -1 pt7 -> 1111 2222 3333 4444 5555 6666 }T
T{  0 -1 -1 pt7 -> 1111 2222 5555 6666           }T
T{  0  0 -1 pt7 -> 1111 0    6666                }T
T{  0  0  0 pt7 -> 0    0    4444 5555 6666      }T

: [1cs-roll] 1 CS-ROLL ; IMMEDIATE

T{ : pt8
```

```
      >R
      AHEAD 111
      BEGIN 222
          [1cs-roll]
          THEN
          333
          R> 1- >R
          R@ 0<
      UNTIL
      R> DROP
    ; ->  }T

T{ 1 pt8 -> 333 222 333 }T
```

F.15.6.2. —— FIND-NAME

```
: >lower ( c1 -- c2 )
   DUP 'A' 'Z' 1+ WITHIN BL AND OR
;

: istr= ( addr1 u1 addr2 u2 -- flag )
   ROT OVER <> IF 2DROP DROP FALSE EXIT THEN
   bounds ?DO
     DUP C@ >lower I C@ >lower <> IF DROP FALSE UNLOOP EXIT THEN
     1+
   LOOP
   DROP TRUE
;

WORDLIST CONSTANT fntwl
GET-CURRENT fntwl SET-CURRENT
: fnt1 25 ;
: fnt2 34 ; IMMEDIATE
SET-CURRENT


T{ S" fnt1" fntwl FIND-NAME-IN NAME>INTERPRET EXECUTE -> 25 }T

T{ : fnt3
   [ S" fnt1" fntwl FIND-NAME-IN NAME>COMPILE EXECUTE ]
; fnt3 -> 25 }T
T{ S" fnt1" fntwl FIND-NAME-IN NAME>STRING S" fnt1" istr= -> TRUE }T

T{ S" fnt2" fntwl FIND-NAME-IN NAME>INTERPRET EXECUTE -> 34 }T
T{ S" fnt2" fntwl FIND-NAME-IN NAME>COMPILE   EXECUTE -> 34 }T

: fnt4 fntwl FIND-NAME-IN NAME>COMPILE EXECUTE ; IMMEDIATE

T{ S" fnt2" ] fnt4 [ -> 34 }T

T{ S" fnt0" fntwl FIND-NAME-IN -> 0 }T
```

```
: fnt5 42 ;
: fnt6 51 ; IMMEDIATE

T{ S" fnt5" FIND-NAME NAME>INTERPRET EXECUTE -> 42 }T

T{ : fnt7
   [ S" fnt5" FIND-NAME NAME>COMPILE EXECUTE ]
; fnt7 -> 42 }T

T{ S" fnt5" FIND-NAME NAME>STRING S" fnt5" istr= -> TRUE }T

T{ S" fnt6" FIND-NAME NAME>INTERPRET EXECUTE -> 51 }T
T{ S" fnt6" FIND-NAME NAME>COMPILE   EXECUTE -> 51 }T

: fnt8 FIND-NAME NAME>COMPILE EXECUTE ; IMMEDIATE

T{ S" fnt6" ] fnt8 [ -> 51 }T
T{ S" fnt0hfshkshdfskl" FIND-NAME -> 0 }T
T{ S\" s\"" FIND-NAME NAME>INTERPRET EXECUTE bla" S" bla" COMPARE ->
0 }T

T{ : fnt9 [ S\" s\"" FIND-NAME NAME>COMPILE EXECUTE ble" ] ; ->  }T
T{ fnt9 S" ble" COMPARE -> 0 }T

: fnta FIND-NAME NAME>INTERPRET EXECUTE ; IMMEDIATE
T{ : fntb [ S\" s\"" ] fnta bli" 2LITERAL ; ->  }T
T{ fntb S" bli" COMPARE -> 0 }T

: fnt-interpret-words ( ...  "rest-of-line" -- ...  )
   BEGIN
     PARSE-NAME DUP
   WHILE
     2DUP FIND-NAME DUP 0= -13 AND THROW
     NIP NIP STATE @ IF
       NAME>COMPILE
     ELSE
       NAME>INTERPRET
     THEN EXECUTE
   REPEAT 2DROP
;

T{ fnt-interpret-words fnt5 VALUE fntd fnt6 TO fntd
fntd -> fnt6 }T
T{ fnt-interpret-words S" yyy" : fnte S" yyy" ; fnte COMPARE -> 0 }T
T{ fnt-interpret-words : fntc {: xa xb :}
   xa xb TO xa TO xb xa xb S" xxx"
; S" xxx" SWAP fntc COMPARE -> 0 }T
```

F.15.6.2. —— FIND-NAME-IN

See E.15.6.2.0 FIND-NAME.

F.15.6.2.1908 N>R

```
: TNR1 N>R SWAP NR> ;
T{ 1 2 10 20 30 3 TNR1 -> 2 1 10 20 30 3 }T

: TNR2 N>R N>R SWAP NR> NR> ;
T{ 1 2 10 20 30 3 40 50 2 TNR2 -> 2 1 10 20 30 3 40 50 2 }T
```

F.15.6.2.—— [:

```
T{ : q1 [: 1 ;] ;                                  q1 EXECUTE ->   1 }T
T{ : q2 [: [: 2 ;] ;] ;                      q2 EXECUTE EXECUTE ->   2 }T
T{ : q3 {: a :} [: {: a b :} b a ;] ;      1 2 3 q3 EXECUTE -> 2 1 }T
T{ : q4 [: DUP IF DUP 1- RECURSE THEN ;] ; 3 q4 EXECUTE .S ->
                                                          3 2 1 0 }T
T{ : q5 [: DOES> DROP 4 ;] 5 SWAP ;  CREATE x q5 EXECUTE x -> 5 4 }T
T{ : q6 {: a :} [: {: a b :} b a ;] a 1+ ; 1 2 q6 SWAP EXECUTE
                                                        -> 3 1 }T
T{ 1 2 q6 q6 SWAP EXECUTE EXECUTE                       -> 4 1 }T
T{ 1 2 3 q3 SWAP q6 SWAP EXECUTE EXECUTE                -> 3 1 }T
```

F.15.6.2.2533 [THEN]

```
T{ <TRUE>  [IF] 111 [ELSE] 222 [THEN] -> 111 }T
T{ <FALSE> [IF] 111 [ELSE] 222 [THEN] -> 222 }T

\ Check words are immediate
: tfind BL WORD FIND ;
T{ tfind [IF]     NIP -> 1 }T
T{ tfind [ELSE] NIP -> 1 }T
T{ tfind [THEN] NIP -> 1 }T

T{ : pt2 [  0 ] [IF] 1111 [ELSE] 2222 [THEN] ; pt2 -> 2222 }T
T{ : pt3 [ -1 ] [IF] 3333 [ELSE] 4444 [THEN] ; pt3 -> 3333 }T

\ Code spread over more than 1 line
T{ <TRUE>  [IF] 1
              2
           [ELSE]
              3
              4
           [THEN] -> 1 2 }T
T{ <FALSE> [IF]
              1 2
           [ELSE]
              3 4
           [THEN] -> 3 4 }T

\ Nested
: <T> <TRUE> ;
: <F> <FALSE> :
T{ <T> [IF] 1 <T> [IF] 2 [ELSE] 3 [THEN] [ELSE] 4 [THEN] -> 1 2 }T
```

```
T{ <F> [IF] 1 <T> [IF] 2 [ELSE] 3 [THEN] [ELSE] 4 [THEN] -> 4 }T
T{ <T> [IF] 1 <F> [IF] 2 [ELSE] 3 [THEN] [ELSE] 4 [THEN] -> 1 3 }T
T{ <F> [IF] 1 <F> [IF] 2 [ELSE] 3 [THEN] [ELSE] 4 [THEN] -> 4 }T
```

F.16 The optional Search-Order word set

The search order is reset to a known state before the tests can be run.

```
ONLY FORTH DEFINITIONS
```

Define two word list (wid) variables used by the tests.

```
VARIABLE wid1
VARIABLE wid2
```

In order to test the search order it in necessary to remember the existing search order before modifying it. The existing search order is saved and the `get-orderlist` defined to access it.

```
: save-orderlist ( widn ...  wid1 n -- )
  DUP , 0 ?DO , LOOP
;

CREATE order-list
T{ GET-ORDER save-orderlist ->  }T

: get-orderlist ( -- widn ...  wid1 n )
  order-list DUP @ CELLS    ( -- ad n )
  OVER +                    ( -- AD AD' )
  ?DO I @ -1 CELLS +LOOP    ( -- )
;
```

Having obtained a copy of the current wordlist, the testing of the wordlist can begin with test F.16.6.1.1595 `FORTH-WORDLIST` followed by F.16.6.1.2197 `SET-ORDER` which also test `GET-ORDER`, then F.16.6.2.0715 `ALSO` and F.16.6.2.1965 `ONLY` before moving on to F.16.6.1.2195 `SET-CURRENT` which also test `GET-CURRENT` and `WORDLIST`. This should be followed by the test F.16.6.1.1180 `DEFINITIONS` which also tests `PREVIOUS` and the F.16.6.1.2192 `SEARCH-WORDLIST` and F.16.6.1.1550 `FIND` tests. Finally the F.16.6.2.1985 `ORDER` test can be performed.

F.16.6.1.1180 DEFINITIONS

```
T{ ONLY FORTH DEFINITIONS ->  }T
T{ GET-CURRENT -> FORTH-WORDLIST }T

T{ GET-ORDER wid2 @ SWAP 1+ SET-ORDER DEFINITIONS GET-CURRENT
-> wid2 @ }T
T{ GET-ORDER -> get-orderlist wid2 @ SWAP 1+ }T
T{ PREVIOUS GET-ORDER -> get-orderlist }T
T{ DEFINITIONS GET-CURRENT -> FORTH-WORDLIST }T

: alsowid2 ALSO GET-ORDER wid2 @ ROT DROP SWAP SET-ORDER ;
alsowid2
: w1 1234 ;
DEFINITIONS : w1 -9876 ; IMMEDIATE
```

```
ONLY FORTH
T{ w1 -> 1234 }T
DEFINITIONS
T{ w1 -> 1234 }T
alsowid2
T{ w1 -> -9876 }T
DEFINITIONS T{ w1 -> -9876 }T

ONLY FORTH DEFINITIONS
: so5 DUP IF SWAP EXECUTE THEN ;

T{ S" w1" wid1 @ SEARCH-WORDLIST so5 -> -1  1234 }T
T{ S" w1" wid2 @ SEARCH-WORDLIST so5 ->  1 -9876 }T

: c"w1" C" w1" ;
T{ alsowid2 c"w1" FIND so5 ->  1 -9876 }T
T{ PREVIOUS c"w1" FIND so5 -> -1  1234 }T
```

F.16.6.1.1550 FIND

```
: c"dup" C" DUP" ;
: c".("  C" .("  ;
: c"x"   C" unknown word" ;

T{ c"dup" FIND -> xt  @ -1 }T
T{ c".("  FIND -> xti @  1 }T
T{ c"x"   FIND -> c"x"   0 }T
```

F.16.6.1.1595 FORTH-WORDLIST

```
T{ FORTH-WORDLIST wid1 ! ->  }T
```

F.16.6.1.2192 SEARCH-WORDLIST

```
ONLY FORTH DEFINITIONS
VARIABLE xt  ' DUP xt  !
VARIABLE xti ' .(  xti ! \ Immediate word

T{ S" DUP" wid1 @ SEARCH-WORDLIST -> xt  @ -1 }T
T{ S" .("  wid1 @ SEARCH-WORDLIST -> xti @  1 }T
T{ S" DUP" wid2 @ SEARCH-WORDLIST ->        0 }T
```

F.16.6.1.2195 SET-CURRENT

```
T{ GET-CURRENT -> wid1 @ }T

T{ WORDLIST wid2 ! ->  }T
T{ wid2 @ SET-CURRENT ->  }T
T{ GET-CURRENT -> wid2 @ }T

T{ wid1 @ SET-CURRENT ->  }T
```

F.16.6.1.2197 SET-ORDER

```
T{ GET-ORDER OVER      -> GET-ORDER wid1 @ }T
T{ GET-ORDER SET-ORDER ->  }T
T{ GET-ORDER           -> get-orderlist }T

T{ get-orderlist DROP get-orderList 2* SET-ORDER ->  }T
T{ GET-ORDER -> get-orderlist DROP get-orderList 2* }T
T{ get-orderlist SET-ORDER GET-ORDER -> get-orderlist }T

: so2a GET-ORDER get-orderlist SET-ORDER ;
: so2 0 SET-ORDER so2a ;

T{ so2 -> 0 }T    \ 0 SET-ORDER leaves an empty search order

: so3 -1 SET-ORDER so2a ;
: so4 ONLY so2a ;

T{ so3 -> so4 }T  \ -1 SET-ORDER is the same as ONLY
```

F.16.6.2.0715 ALSO

```
T{ ALSO GET-ORDER ONLY -> get-orderlist OVER SWAP 1+ }T
```

F.16.6.2.1965 ONLY

```
T{ ONLY FORTH GET-ORDER -> get-orderlist }T

: so1 SET-ORDER ; \ In case it is unavailable in the forth wordlist

T{ ONLY FORTH-WORDLIST 1 SET-ORDER get-orderlist so1 ->  }T
T{ GET-ORDER -> get-orderlist }T
```

F.16.6.2.1985 ORDER

```
CR .( ONLY FORTH DEFINITIONS search order and compilation list) CR
T{ ONLY FORTH DEFINITIONS ORDER ->  }T

CR .( Plus another unnamed wordlist at head of search order) CR
T{ alsowid2 DEFINITIONS ORDER ->  }T
```

F.17 The optional String word set

Most of the tests in this wordlist require a known string which is defined as:

```
T{ : s1 S" abcdefghijklmnopqrstuvwxyz" ; ->  }T
```

The tests should be carried out in the order: F.17.6.1.0245 /STRING, F.17.6.1.2191 SEARCH, F.17.6.1.0170 -TRAILING, F.17.6.1.0935 COMPARE, F.17.6.1.0780 BLANK and F.17.6.1.2212 SLITERAL.

F.17.6.1.0170 -TRAILING

```
T{ :  s8 S" abc  " ; ->  }T
T{ :  s9 S"      " ; ->  }T
T{ : s10 S"    a " ; ->  }T

T{  s1 -TRAILING -> s1 }T  \ “abcdefghijklmnopqrstuvwxyz”
T{  s8 -TRAILING -> s8 2 - }T     \ “abc  ”
```

```
T{  s7 -TRAILING -> s7 }T          \ " "
T{  s9 -TRAILING -> s9 DROP 0 }T   \ "     "
T{ s10 -TRAILING -> s10 1- }T      \ "   a "
```

F.17.6.1.0245 /STRING

```
T{ s1  5 /STRING -> s1 SWAP 5 + SWAP 5 - }T
T{ s1 10 /STRING -4 /STRING -> s1 6 /STRING }T
T{ s1  0 /STRING -> s1 }T
```

F.17.6.1.0780 BLANK

```
: s13 S" aaaaa      a" ;           \ Six spaces

T{ PAD 25 CHAR a FILL ->  }T       \ Fill PAD with 25 'a's
T{ PAD 5 CHARS + 6 BLANK ->  }T    \ Put 6 spaced from character 5
T{ PAD 12 s13 COMPARE -> 0 }T      \ PAD Should now be same as s13
```

F.17.6.1.0935 COMPARE

```
T{ s1        s1 COMPARE ->  0  }T
T{ s1  PAD SWAP CMOVE   ->     }T  \ Copy s1 to PAD
T{ s1  PAD OVER COMPARE ->  0  }T
T{ s1     PAD 6 COMPARE ->  1  }T
T{ PAD 10    s1 COMPARE -> -1  }T
T{ s1     PAD 0 COMPARE ->  1  }T
T{ PAD  0    s1 COMPARE -> -1  }T
T{ s1        s6 COMPARE ->  1  }T
T{ s6        s1 COMPARE -> -1  }T

: "abdde"  S" abdde"  ;
: "abbde"  S" abbde"  ;
: "abcdf"  S" abcdf"  ;
: "abcdee" S" abcdee" ;

T{ s1 "abdde"  COMPARE -> -1 }T
T{ s1 "abbde"  COMPARE ->  1 }T
T{ s1 "abcdf"  COMPARE -> -1 }T
T{ s1 "abcdee" COMPARE ->  1 }T

: s11 S" 0abc" ;
: s12 S" 0aBc" ;

T{ s11 s12 COMPARE ->  1 }T
T{ s12 s11 COMPARE -> -1 }T
```

F.17.6.1.2191 SEARCH

```
T{ : s2 S" abc"   ; ->  }T
T{ : s3 S" jklmn" ; ->  }T
T{ : s4 S" z"     ; ->  }T
T{ : s5 S" mnoq"  ; ->  }T
T{ : s6 S" 12345" ; ->  }T
T{ : s7 S" "      ; ->  }T
```

```
T{ s1 s2 SEARCH -> s1 <TRUE>  }T
T{ s1 s3 SEARCH -> s1  9 /STRING <TRUE>  }T
T{ s1 s4 SEARCH -> s1 25 /STRING <TRUE>  }T
T{ s1 s5 SEARCH -> s1 <FALSE> }T
T{ s1 s6 SEARCH -> s1 <FALSE> }T
T{ s1 s7 SEARCH -> s1 <TRUE>  }T
```

F.17.6.1.2212 SLITERAL

```
T{ : s14 [ s1 ] SLITERAL ; ->  }T
T{ s1 s14 COMPARE -> 0 }T
T{ s1 s14 ROT = ROT ROT = -> <TRUE> <FALSE> }T
```

F.17.6.2.2255 SUBSTITUTE

```
30 CHARS BUFFER: subbuff \ Destination buffer

\ Define a few string constants
: "hi" S" hi" ;
: "wld" S" wld" ;
: "hello" S" hello" ;
: "world" S" world" ;

\ Define a few test strings
: sub1 S" Start: %hi%,%wld%! :End" ;    \ Original string
: sub2 S" Start: hello,world! :End" ;   \ First target string
: sub3 S" Start: world,hello! :End" ;   \ Second target string

\ Define the hi and wld substitutions
T{ "hello" "hi"  REPLACES ->  }T        \ Replace "%hi%" with "hello"
T{ "world" "wld" REPLACES ->  }T        \ Replace "%wld%" with "world"

\ "%hi%,%wld%" changed to "hello,world"
T{ sub1 subbuff 30 SUBSTITUTE ROT ROT sub2 COMPARE -> 2 0 }T

\ Change the hi and wld substitutions
T{ "world" "hi"  REPLACES ->  }T
T{ "hello" "wld" REPLACES ->  }T

\ Now "%hi%,%wld%" should be changed to "world,hello"
T{ sub1 subbuff 30 SUBSTITUTE ROT ROT sub3 COMPARE -> 2 0 }T

\ Where the subsitution name is not defined
: sub4 S" aaa%bbb%ccc" ;
T{ sub4 subbuff 30 SUBSTITUTE ROT ROT sub4 COMPARE -> 0 0 }T

\ Finally the % character itself
: sub5 S" aaa%%bbb" ;
: sub6 S" aaa%bbb" ;
T{ sub5 subbuff 30 SUBSTITUTE ROT ROT sub6 COMPARE -> 0 0 }T
```

F.17.6.2.2375 UNESCAPE

Using subbuff, sub5 and sub6 from F.17.6.2.2255 SUBSTITUTE.

```
T{ sub6 subbuff UNESCAPE sub5 COMPARE -> 0 }T
```

F.18 The optional Extended Character word set

These test assume the UTF-8 character encoding is being used.

F.18.6.1.2487.15 XC!+?

```
T{ $ffff PAD 4 XC!+? -> PAD 3 + 1 <TRUE> }T
```

F.18.6.1.2487.25 XC-SIZE

This test assumes UTF-8 encoding is being used.

```
HEX
T{       0 XC-SIZE -> 1 }T
T{      7f XC-SIZE -> 1 }T
T{      80 XC-SIZE -> 2 }T
T{     7ff XC-SIZE -> 2 }T
T{     800 XC-SIZE -> 3 }T
T{    ffff XC-SIZE -> 3 }T
T{   10000 XC-SIZE -> 4 }T
T{ 1fffff XC-SIZE -> 4 }T
```

F.18.6.2.2487.30 XC-WIDTH

```
T{ $606D XC-WIDTH -> 2 }T
T{   $41 XC-WIDTH -> 1 }T
T{ $2060 XC-WIDTH -> 0 }T
```

Annex G
(informative)
Change Log

16.1 Bath Meeting (30 September – 2 October, 2015)

– 200*x* Membership:

(1) x:membership: Replaced membership rules.

(2) ed16: Moved Andrew Haley and Willi Stricker from current to past members.

3 Usage requirements:

(1) x:2-complement: Removed "Programs that use flags as arithmetic operands have an environmental dependency." from 3.1.3.1Flags.

(2) x:2-complement: Replaced -32767 with -32768 in 3.1.3.2Integers.

(3) x:2-complement: Removed reference to alternate arithmetic architectures from 3.2.1.1Internal number representation.

(4) x:2-complement: Replaced implementation defined response to overflow/underflow of signed numbers with specific text in 3.2.2.2Other integer operations.

4 Documentation requirements:

(1) x:2-complement: Removed "values returned after arithmetic overflow" from 4.1.1Implementation-defined options.

12 Floating-Point Word Set:

(1) ed16: Replaced `0E` with `-0E` in 12.6.2.1489 `FATAN2`.

A Rationale (Annex A):

(1) x:2-complement: Removed number representations from A.3.1.2Character types.

(2) ed16: Removed reference to EBCDIC from A.3.1.2Character types.

(3) x:2-complement: Remove discussion of alternative arithmetic architectures from A.3.2.1Numbers.

(4) x:2-complement: Revised example in A.3.2.2.2 Other integer operations.

(5) x:2-complement: Replaced environmental dependency example in A.5.2.2Program labeling.

(6) x:2-complement: Removed reference to two's-complement from A.6.2.2440 `WITHIN`.

(7) x:2-complement: Replaced rational for A.8.6.1.1140 `D>S`.

D Portability guide (Annex D):

(1) x:2-complement: Removed D.3.2 ALU organization.

E Reference Implementations (Annex E):

(1) x:2-complement: Added E.6.2.2440 `WITHIN`.

(2) x:2-complement: Added E.8.6.1.1140 `D>S`.

F Test Suite (Annex F):

(1) x:2-complement: Added test case to F.6.1.0290 `1+`.

(2) x:2-complement: Added test case to F.6.1.1720 `INVERT`.

17.1 Konstanz Meeting (7–9 September, 2016)

– Title page:

(1) x:reset: Replaced “ANS X3.215–1994” with “the Forth 2012 standard” in document status;

(2) x:reset: Renamed “The Standardisation Committee” to “The Forth 200*x* Standardisation Committee” in document status.

– Foreword:

(1) x:reset: Revised to refer to Forth 2012 as the starting point for the document;

(2) x:reset: Removed meetings prior to 2015 (Bath) meeting;

(3) ed17: Added 2016 (Konstanz) meeting.

– 200*x* Membership:

(1) ed17: Added Sergey Baranov;

(2) ed17: Moved Simon Kaphahn from members list to old members list;

(3) ed17: Reinstated Andrew Haley, moving him from the old members list back into the members list;

(4) ed17: Added officer title against officer holders.

1 Introduction:

(1) ed17: Replaced “2012” with “200*x*” in title on page 12.

3 Usage requirements:

(1) x:1 chars = 1: Replace “at least one” with “exactly one” in 3.1.2Character types;

(2) x:2-complement: Replaced “two’s complement” with “two’s-complement” in 3.2.1.1Internal number representation;

(3) x:2-complement: Replaced “divisions” with “division” in 3.2.2.2Other integer operations;

(4) x:2-complement: Removed “in these case” from 3.2.2.2Other integer operations;

(5) x:2-complement: Added “for operations other than division” to 3.2.2.2Other integer operations;

(6) x:2-complement: Replaced “2^{n-1}” with “-2^{n-1}” in 3.2.2.2Other integer operations;

(7) x:reset: Removed “X:wordset-query” from 3.2.7Obsolescent Environmental Queries;

(8) x:reset: Removed 3.2.8 Extension queries and table 3.7 Forth 200x Extensions.

6–18 All word sets:
x:reset: Removed proposal labels;

12 Floating-Point Word Set:

(1) x:to-f-round: Removed D>F and S>F conditions from 12.4.1.2Ambiguous conditions;

(2) x:to-f-round: Revised definition of 12.6.1.1130 D>F;

(3) x:to-f-round: Revised definition of 12.6.2.2175 S>F.

A Rationale (Annex A):

(1) x:2-complement: Made definition of *n* a clause of previous sentence in A.3.2.1Numbers;

(2) x:2-complement: Removed "likely" from second paragraph of A.3.2.1Numbers;

(3) x:2-complement: Removed final paragraph of A.3.2.1Numbers;

(4) x:2-complement: Remove A.3.2.2.2 Other integer operations;

(5) x:2-complement: Replaced "entitle" with "entitled" in A.5.2.2Program labeling;

(6) x:2-complement: Replaced "lower" with "lowest" in A.5.2.2Program labeling;

(7) x:2-complement: Replaced "twos-complement" with "two's complement" in A.8.6.1.1140 D>S.

E Reference Implementations (Annex E):

(1) x:2-complement: Added E.6.1.1910 NEGATE;

(2) x:2-complement: Corrected "S>D" to "D>S" in E.8.6.1.1140 D>S.

F Test Suite (Annex F):

(1) x:1 chars = 1: Added test case to F.6.1.0897 CHAR+;

(2) x:1 chars = 1: Added test case to F.6.1.0898 CHARS.

H Alphabetic list of words (Annex H):
x:reset: Removed proposal names.

18.1 Bad Vöslau Meeting (6–8 September, 2017)

– Foreword:

(1) ed18: Added 2017 (Bad Vöslau) meeting.

– 200*x* Membership:

(1) ed18: Added Paul E. Bennet as member;

(2) ed18: Moved Howerd Oakford from contributor to member.

3 Usage requirements:

(1) ed18: The Exception word set is no longer optional (3Usage requirements);

(2) ed18: Moved A.11.3.4 Other transient regions to 3.3.3.4Text-literal regions;

(3) ed18: Replaced ambiguous condition In 3.4 (d) with exception;

(4) ed18: Replaced ambiguous condition in last sentence of the second paragraph in 3.4.1 with exception.

4 Documentation requirements:

(1) ed18: Moved .4.1.1Implementation-defined options documentation requirements to 4.1.1Implementation-defined options.

6 Core Word Set:

(1) x:quotation: Added quotation ambiguous condition to 6.1.0460 `;`;

(2) x:quotation: Added quotation ambiguous condition to 6.1.1250 `DOES>`;

(3) x:quotation: Added quotation requirement to 6.1.2120 `RECURSE`;

(4) ed18: Merged 11.6.1.2165 `S"` with 6.1.2165 `S"`;

(5) ed18 Added cross reference to 6.2.0855 `C"`;

(6) ed18: Merged 11.6.2.2266 `S\"` with 6.2.2266 `S\"`.

9 Exception Word Set:

(1) ed18: The Exception word set is not longer optional (9The Exception word set);

(2) ed18: Removed 9.5 Compliance and labeling.

11 File-Access Word Set:

(1) ed18: Merged .3.4Other transient regions with 3.3.3.4Text-literal regions;

(2) ed18: Removed .3.4Other transient regions related system documentation requirement;

(3) ed18: Removed 11.6.1.2165 `S"`.

(4) ed18: Removed 11.6.2.2266 `S\"`.

15 Programming-Tools Word Set:

(1) x:quotations: Added 15.3.2Colon definition status;

(2) x:quotations: Extended 15.4.1.1Implementation-defined options;

(3) x:quotations: Added ambiguous condition to 15.6.2.0470 `;CODE`;

(4) x:quotations: Added 15.6.2.0 `;]`;

(5) x:quotations: Added 15.6.2.0 `[:`.

A Rationale (Annex A):

(1) ed18: Added A.3.3.3.4Text-literal regions;

(2) ed18: Merged A.11.6.1.2165 `S"` with A.6.1.2165 `S"`;

(3) ed18: Removed A.11.3.4 Other transient regions;

(4) ed18: Removed A.11.6.1.2165 `S"`;

(5) x:quotations: Added A.15.6.2.0 `[:`.

E Reference Implementations (Annex E):

(1) ed18: Added E.15.6.2.0 `[:` and E.15.6.2.0 `;]`.

F Test Suite (Annex F):

(1) ed18: Moved F.11.6.1.2165 `S"` to F.6.1.2165 `S"`;

(2) ed18: Added F.15.6.2.0 `[:`.

H Alphabetic list of words (Annex H):
ed18 Added reference to Rational, Testing and Implementation where available.

19.1 Queensferry Meeting (12–14 September 2018)

– Foreword:

(1) ed19: Added 2018 (Queensferry) meeting.

– 200*x* Membership:

(1) ed19: Added Jermain Davies as member;

3 Usage requirements:

(1) ed19: Add reference to exception -19 (definition name too long) to 3.3.1.2Definition names.

4.1.2 Ambiguous Conditions:

(1) ed19: Removed undefined word ambiguous condition;

(2) ed19: Removed maximum length ambiguous condition;

(3) ed19: Removed example from argument type ambiguous condition;

(4) ed19: Removed insufficient space in the dictionary ambiguous condition;

15 Programming-Tools Word Set:

(1) x:find-name: Added 15.6.2.0 `FIND-NAME` and 15.6.2.0 `FIND-NAME-IN`;

A Rationale (Annex A):

(1) x:find-name: Added A.15.6.2.0 `FIND-NAME`;

E Reference Implementations (Annex E):

(1) x:find-name: Added E.15.6.2.0 `FIND-NAME` and E.15.6.2.0 `FIND-NAME-IN`.

F Test Suite (Annex F):

(1) x:find-name: Added F.15.6.2.0 `FIND-NAME` and E.15.6.2.0 `FIND-NAME-IN`.

20.1 Hamburg Meeting (11–13 September 2019)

– Foreword:

(1) ed19: Added 2019 (Hamburg) meeting.

– 200*x* Membership:

(1) ed20: Moved Sergey Baranov to list of ex-committee members;

2 Terms, notation, and references:

2.1 Definition of terms

(1) x:defined-find: Added reference to 16.2Additional terms and notation to “find”.

15 Programming-Tools Word Set:

(1) x:defined-find: Removed reference to `FIND` from 15.6.2.2530.30 `[DEFINED]`;

(2) x:name-token: Added name token to 15.2Additional terms and notation.

16 Search-Order Word Set:

(1) x:defined-find: Added “find” to 16.2Additional terms and notation.

A Rationale (Annex A):

(1) x:else-then: Add `[ELSE]` without proceeding `[IF]` is not recommended to A.15.6.2.2531 `[ELSE]`;

(2) x:name-token: Removed A.15.3.1 Name tokens.

21.1 Online (1–3 September 2020)

– Foreword:

(1) ed21: Added 2020 (Online) meeting.

– 200*x* Membership:

(1) ed21: Moved Paul E. Bennet to list of ex-committee members;

(2) ed21: Added Krishna Myneni as a member;

6 Core Word Set:

(1) revise-colon: Remove ", called a "colon definition" from first paragraph of 6.1.0450 `:` (colon);

(2) x:rules-of-find: Added cross references for 15.6.2.2530.30 `[DEFINED]` and 16.6.1.1550 `FIND` to 6.1.1550 `FIND`.

(3) ed21: Moved escape sequences in 6.2.2266 `S\"` to a table figure with minor reformatting of the table.

13 Locals Word Set:

(1) Add *local* interpretation semantics to 13.6.1.0086 `(LOCAL)`;

(2) Remove ambiguous condition from *name* execution semantics of 13.6.2.2550 `{:`;

(3) Add *name* interpretation semantics to 13.6.2.2550 `{:`;

15 Programming-Tools Word Set:

(1) x:cs-drop: Added 15.6.2.0 `CS-DROP`;

(2) x:cs-drop: Changed stack description of 15.6.2.1015 `CS-PICK`;

(3) x:traverse-wordlist: Wording changes and new paragraph added to 15.6.2.2297TRAVERSE-WORDLIST;

(4) x:[if]-input: Revised input of 15.6.2.2532 `[IF]` to match 6.1.1700 `IF`;

(5) ed21: Add reference to `[IF]` and `[THEN]` to 15.6.2.2531 `[ELSE]`.;

(6) ed21: Add reference to `[ELSE]` and `[THEN]` to 15.6.2.2532 `[IF]`;

(7) ed21: Add reference to `[IF]` and `[ELSE]` to 15.6.2.2533 `[THEN]`;

(8) x:simplify-bracket-else: Add reference to the implementation of `[ELSE]` to: 15.6.2.2532 `[IF]`, 15.6.2.2531 `[ELSE]` and 15.6.2.2533 `[THEN]`;

(9) x:simplify-bracket-else: Add reference to the testing of `[THEN]` to: 15.6.2.2532 `[IF]`, 15.6.2.2531 `[ELSE]` and 15.6.2.2533 `[THEN]`;

16 Search-Order Word Set:

(1) x:vote-9: Remove ambiguous condition from 16.3.3Finding definition names;

(2) x:rules-of-find: Add cross reference for 15.6.2.2530.30 `[DEFINED]` to 16.6.1.1550 `FIND`;

(3) x:vocabulary: Added 16.6.2.0 `VOCABULARY`;

17 String Word Set:

(1) x:revise-replaces: Revise last paragraph of 17.6.2.2141 `REPLACES`.

A Rationale (Annex A):

(1) ed21: Remove reference to interpretation version of S" from A.6.1.1345 `ENVIRONMENT?`

(2) x:cs-drop: Added A.15.6.2.0 `CS-DROP`;

(3) x:rule-of-find: Added cross references for 6.1.1550 `FIND` and 16.6.1.1550 `FIND` to 15.6.2.2530.30 `[DEFINED]`;

(4) x:vocabulary: Added A.16.6.2.0 `VOCABULARY`;

E Reference Implementations (Annex E):

(1) ref-license: Add a license to use the reference implementations to E.1Introduction.

(2) x:cs-drop: Added E.15.6.2.0 `CS-DROP`;

(3) x:synonym Remove E.15.6.2.2264 `SYNONYM`;

(4) simplify-bracket-else: Replace reference implementation for E.15.6.2.2531 `[ELSE]` and remove reference implementations for E.15.6.2.2532 `[IF]` and E.15.6.2.2533 `[THEN]`;

(5) x:vocabulary Added E.16.6.2.0 VOCABULARY;

F Test Suite (Annex F):

(1) ed21: Fix typo in final paragraph of F.1Introduction.

(2) x:cs-drop: Added F.15.6.2.0 CS-DROP;

Annex H
(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.

At the end of the line, after the page number of the word defintion, are three letters indicating if the word has a presence in the R (rational), T (testing) and I (implementation) appenices (annex A, F and E respectivly).

x:quotations