Forth 200*x* Standardisation Committee

Forth 200*x* *Draft 07.1r*

17th August, 2007

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

Notice: ***Inline Word Rationale*** (*r*)
As this is a working draft the rationale for a given glossary entry is shown in the appropriate section (rationale and testing) within the defintion. These sections will appear in the appropriate section of appendix A in the final document.

Contents

x:forward

x:forward

Foreword

On completion of ANS Forth (ANS X3.215-1994 *Information Systems — Programming Languages FORTH*) in 1994, the document was presented to and adopted as an international standard, by the ISO in 1997, being published as ISO/IEC 15145:1997 *Information technology, Programming languages, FORTH*.

The current project to update ANS Forth was launched at the 2004 EuroForth conference. The intention being to allow the Forth community to contribute to a rolling standard. With changes to the document being proposed and discussed in the electronic community, via the `comp.lang.forth` usenet news group, the `forth200x@yahoogroups.com` email list, and the `www.forth200x.org` web site. An open meeting to discuss proposals being held annually, immediately prior to the EuroForth conference.

This document is based on the first draft of the of the standard published by Technical Committee on Forth Programming Systems as part of the first review in 1999. It has been modified in accordance with the directions of the Forth 200*x* Standards Committee which first met on October 21-22, 2005 and subsequently on September 14-15, 2006.

Foreword to ANS Forth

x:forward

(This foreword is not a part of American National Standard X3.215-1994)

Forth is a language for direct communication between human beings and machines. 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 was invented by Mr. Charles Moore to increase programmer productivity without sacrificing machine efficiency. Forth is a layered environment containing the elements of a computer language as well as those of an operating system and a machine monitor. This extensible, layered environment 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 first meeting of the Technical Committee on Forth Programming Systems was convened by the Organizing Committee of the X3J14 Forth Technical Committee on August 3, 1987, and has met subsequently on November 11–12, 1987, February 10–12, 1988, May 25–28, 1988, August 10–13, 1988, October 26–29, 1988, January 25–28, 1989, May 3–6, 1989, July 26–29, 1989, October 25–28, 1989, January 24–27, 1990, May 22–26, 1990, August 21–25, 1990, November 6–10,1990, January 29–February 2, 1991, May 3–4, 1991, June 16–19, 1991, July 30–August 3, 1991, March 17–21, 1992, October 13–17, 1992, January 26–30, 1993, June 28–30, 1993, and June 21, 1994.

This project has operated under joint sponsorship of IEEE as IEEE Project P1141. The TC gratefully acknowledges the support of IEEE in this effort and the participation of the IEEE members who contributed to our work as sponsored members and observers.

Requests for interpretation, suggestions for improvement or addenda, or defect reports are welcome. They should be sent to the X3 Secretariat, Computer and Business Equipment Manufacturers Association, 1250 Eye Street, NW, Suite 200, Washington, DC 20005.

x:forward

x:forward

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@yohoogroups.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.

   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.

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

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.

   If a proposal does not propose extensions or changes to the Forth language, but just a rewording of the current document, there is nothing for the system implementors to implement, and for programmers to use, so a CfV poll does not make sense and is not performed. The proposal will bypass the CfV stage and will simply be frozen and go directly to the committee for consideration.

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 an open standards meeting.

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 will see what it is good for and why they should adopt it (and vote for it).

A proposal RfD should include the following sections.

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 ANS Forth. 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.

x:forward

x:forward

200*x* Membership

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

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

Dr. Peter Knaggs (Editor) ..... Bournemouth University
pknaggs@bournemouth.ac.uk Bournemouth, UK

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

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

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

Dr. Bill Stoddart ..... University of Teesside
bill.stoddart@ntlworld.com Middlesbrough, UK

American National Standard for Information Systems — Programming Language — Forth

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 17, 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.

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 provides an introduction to Forth.

Annex D discusses the compatibility of ANS Forth with earlier Forths, emphasizing the differences from Forth 83.

Annex E presents some techniques for writing portable programs in ANS Forth.

test

Annex F presents a ~~validation~~test suite to test the operation of a system complies with the definitions documented in this standard.

Annex H includes the words from all word sets in a single list, and serves as an index of ANS Forth words.

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 to become obsolescent. Although retained here because of their widespread use, their use in new implementations or new programs is discouraged, because they may be withdrawn from future revisions of the Standard.

This Standard designates the following words as obsolescent:

6.2.0060	**#TIB**	15.6.2.1580	**FORGET**	6.2.2240	**SPAN**
6.2.0970	**CONVERT**	6.2.2040	**QUERY**	6.2.2290	**TIB**
6.2.1390	**EXPECT**				

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 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 for Forth systems and programs.

Ambiguous conditions include such things as the absence of a needed delimiter while parsing, attempted access to a nonexistent file, or attempted use of a nonexistent word. An ambiguous condition also exists when a Standard word is passed values that are improper or out of range.

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

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

definition: A Forth execution procedure compiled into the dictionary.

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

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

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

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

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

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

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

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

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

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 an ANS 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

2.2.4.2 Glossary semantic description

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

2.3 References

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

- ANSI X3.172-1990 *Dictionary for Information Systems*, (**2.1 Definitions of terms**);
- ANSI X3.4-1974 *American Standard Code for Information Interchange* (ASCII), (**3.1.2.1 Graphic characters**);
- ISO 646-1983 *ISO 7-bit coded characterset for information interchange, International Reference Version* (IRV) (**3.1.2.1 Graphic characters**)[1];
- ANSI/IEEE 754-1985 *Floating-point Standard*, (**12.2.1 Definition of terms**).

[1] Available from the American National Standards Institute, 11 West 42nd Street, New York, NY 10036.

3 Usage requirements

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

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

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

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

3.1 Data types

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

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

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

3.1.1 Data-type relationships

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

$+n \Rightarrow u \Rightarrow x$;
$+n \Rightarrow n \Rightarrow x$;
char $\Rightarrow +n$;
a-addr $\Rightarrow$ *c-addr* $\Rightarrow$ *addr* $\Rightarrow u$;
flag $\Rightarrow x$;
xt $\Rightarrow x$;
$+d \Rightarrow d \Rightarrow xd$;
$+d \Rightarrow ud \Rightarrow xd$.

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

3.1.2 Character types

Characters shall be at least one address unit wide, contain at least eight bits, and have a size less than or equal to cell size.

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

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
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**.

3.1.2.1 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.

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.

Table 3.2: Standard graphic characters

Hex	IRV	Hex	IRV	Hex	IRV	Hex	IRV	Hex	IRV	Hex	IRV
ASCII		ASCII		ASCII		ASCII		ASCII		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		

3.1.3 Single-cell types

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

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

3.1.3.1 Flags

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

3.1.3.2 Integers

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

3.1.3.3 Addresses

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

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

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

3.1.3.4 Counted strings

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

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

3.1.3.5 Execution tokens

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

3.1.4 Cell-pair types

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

3.1.4.1 Double-cell integers

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

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

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

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

3.1.4.2 Character strings

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

3.1.5 System types

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

3.1.5.1 System-compilation types

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

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

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

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

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

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

3.1.5.2 System-execution types

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

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

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

3.2 The implementation environment

3.2.1 Numbers

3.2.1.1 Internal number representation

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

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

3.2.1.2 Digit conversion

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

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

3.2.1.3 Free-field number display

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

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

3.2.2 Arithmetic

3.2.2.1 Integer division

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

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

When signed integers are divided, the remainder is not zero, and a and b have the same sign, q is the largest integer less than the true quotient. If only one operand is negative, whether q is rounded toward negative

infinity (floored division) or rounded towards zero (symmetric division) is implementation defined.

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

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

Table 3.3: Floored Division Example

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, both overflow and underflow shall be ignored. The value returned when either overflow or underflow occurs is implementation defined.

3.2.3 Stacks

3.2.3.1 Data stack

Objects on the data stack shall be one cell wide.

3.2.3.2 Control-flow stack

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

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

The control-flow stack may, but need not, physically exist in an implementation. If it does exist, it may be, but need not be, implemented using the data stack. The format of the control-flow stack is implementation defined. Since the control-flow stack may be implemented using the data stack, items placed on the data stack are unavailable to a program after items are placed on the control-flow stack and remain unavailable until the control-flow stack items are removed.

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@** or **2R>**) that it did not place there using **>R** or **2>R**;
- A program shall not access from within a do-loop values placed on the return stack before the loop was entered;

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

3.2.4 Operator terminal

See **1.2.2 Exclusions**.

3.2.4.1 User input device

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

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

3.2.4.2 User output device

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

3.2.5 Mass storage

A system need not provide any standard words for accessing mass storage. If a system provides any standard word for accessing mass storage, it shall also implement the Block word set.

3.2.6 Environmental queries

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

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

3.2.7 Extension queries

X:extension-query

As part of the Forth 200*x* standards procedure, additions to the Standard where labelled as *extensions*. These extensions have been added to the environmental query name space. **ENVIRONMENT?** a *true* if the system has implemented the extension as documented. Table 3.6 contains the valid input strings corresponding to the documented extensions. In order to distinguish such extensions, they start with the string "X:".

The extension to the environment query table (3.2.6) is itself an extension. Known as the X:extension-query extension.

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 3.5: 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
CORE	*flag*	no	true if complete core word set present (i.e., not a subset as defined in 5.1.1)
CORE-EXT	*flag*	no	true if core extensions word set present
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

Table 3.6: Forth 200*x* Extensions

String	Value data type	Constant?	Meaning
X:deferred	–	–	the X:deferred extension is present
X:defined	–	–	the X:defined extension is present
X:ekeys	–	–	the X:ekeys extension is present
X:extension-query	–	–	the X:extension-query extension is present
X:parse-name	–	–	the X:parse-name extension is present
X:required	–	–	the X:required extension is 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.

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 used in ANS Forth are aligned addresses (indicated by *a-addr*) or character-aligned (indicated by *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 **@**, **!**, **,** (comma), **+!**, **2@**, or **2!** is used with an address that is not aligned, or when **C@**, **C!**, or **C,** is used with an address that is not character-aligned.

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

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

3.3.3.2 Contiguous regions

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

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

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

3.3.3.3 Variables

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

```
VARIABLE X 1 CELLS ALLOT
```

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

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

3.3.3.4 Text-literal regions

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

A program shall not store into the text-literal regions created by **S"** and **C"** nor into any read-only system variable or read-only transient regions. An ambiguous condition exists when a program attempts to store into read-only regions.

3.3.3.5 Input buffers

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

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

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

A program may modify the size of the parse area by changing the contents of **>IN** within the limits imposed by this Standard. For example, if the contents of **>IN** are saved before a parsing operation and restored afterwards, the text that was parsed will be available again for subsequent parsing operations. The extent of permissible repositioning using this method depends on the input source (see **7.3.3 Block buffer regions** and **11.3.4 Input source**).

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

3.3.3.6 Other transient regions

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

- a definition is created via a defining word;
- definitions are compiled with **:** or **:NONAME**;
- data space is allocated using **ALLOT**, **,** (comma), **C,** (c-comma), or **ALIGN**.

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

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

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

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

3.4 The Forth text interpreter

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

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

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

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

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

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

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

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

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

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

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

3.4.1 Parsing

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

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

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

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

3.4.1.1 Delimiters

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

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

3.4.1.2 Syntax

Forth has a simple, operator-ordered syntax. The phrase `A B C` returns values as if `A` were executed first, then `B` and finally `C`. Words that cause deviations from this linear flow of control are called control-flow words. Combinations of control-flow words whose stack effects are compatible form control-flow structures. Examples of typical use are given for each control-flow word in A (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 both positive and negative numbers, with a negative number represented by a single minus sign, the character "–", preceding the digits. The value in **BASE** is the radix for number conversion.

3.4.2 Finding definition names

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

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

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

3.4.3 Semantics

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

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

3.4.3.1 Execution semantics

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

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

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

3.4.3.2 Interpretation semantics

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

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

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

3.4.3.3 Compilation semantics

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

3.4.4 Possible actions on an ambiguous condition

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

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

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

3.4.5 Compilation

A program shall not attempt to nest compilation of definitions.

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

4 Documentation requirements

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

4.1 System documentation

4.1.1 Implementation-defined options

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

- aligned address requirements **3.1.3.3 Addresses**;
- behavior of **6.1.1320 `EMIT`** for non-graphic characters;
- character editing of **6.1.0695 `ACCEPT`** and **6.2.1390 `EXPECT`**;
- character set (**3.1.2 Character types**, **6.1.1320 `EMIT`**, **6.1.1750 `KEY`**);
- character-aligned address requirements (**3.1.3.3 Addresses**);
- character-set-extensions matching characteristics (**3.4.2 Finding definition names**);
- conditions under which control characters match a space delimiter (**3.4.1.1 Delimiters**);
- format of the control-flow stack (**3.2.3.2 Control-flow stack**);
- conversion of digits larger than thirty-five (**3.2.1.2 Digit conversion**);
- display after input terminates in **6.1.0695 `ACCEPT`** and **6.2.1390 `EXPECT`**;
- exception abort sequence (as in **6.1.0680 `ABORT"`**);
- input line terminator (**3.2.4.1 User input device**);
- maximum size of a counted string, in characters (**3.1.3.4 Counted strings**, **6.1.2450 `WORD`**);
- maximum size of a parsed string (**3.4.1 Parsing**);
- maximum size of a definition name, in characters (**3.3.1.2 Definition names**);
- maximum string length for **6.1.1345 `ENVIRONMENT?`**, in characters;
- method of selecting **3.2.4.1 User input device**;
- method of selecting **3.2.4.2 User output device**;
- methods of dictionary compilation (**3.3 The Forth dictionary**);
- number of bits in one address unit (**3.1.3.3 Addresses**);
- number representation and arithmetic (**3.2.1.1 Internal number representation**);
- ranges for *n*, *+n*, *u*, *d*, *+d*, and *ud* (**3.1.3 Single-cell types**, **3.1.4 Cell-pair types**);
- read-only data-space regions (**3.3.3 Data space**);

- size of buffer at **6.1.2450 WORD** (**3.3.3.6 Other transient regions**);
- size of one cell in address units (**3.1.3 Single-cell types**);
- size of one character in address units (**3.1.2 Character types**);
- size of the keyboard terminal input buffer (**3.3.3.5 Input buffers**);
- size of the pictured numeric output string buffer (**3.3.3.6 Other transient regions**);
- size of the scratch area whose address is returned by **6.2.2000 PAD** (**3.3.3.6 Other transient regions**);
- system case-sensitivity characteristics (**3.4.2 Finding definition names**);
- system prompt (**3.3 The Forth dictionary**, **6.1.2050 QUIT**);
- type of division rounding (**3.2.2.1 Integer division**, **6.1.0100 */**, **6.1.0110 */MOD**, **6.1.0230 /**, **6.1.0240 /MOD**, **6.1.1890 MOD**);
- values of **6.1.2250 STATE** when true;
- values returned after arithmetic overflow (**3.2.2.2 Other integer operations**);
- whether the current definition can be found after **6.1.1250 DOES>** (**6.1.0450 :**).

4.1.2 Ambiguous conditions

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

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

- a *name* is neither a valid definition name nor a valid number during text interpretation (**3.4 The Forth text interpreter**);
- a definition name exceeded the maximum length allowed (**3.3.1.2 Definition names**);
- addressing a region not listed in **3.3.3 Data space**;
- argument type incompatible with specified input parameter, e.g., passing a *flag* to a word expecting an *n* (**3.1 Data types**);
- attempting to obtain the execution token, (e.g., with **6.1.0070 '**, **6.1.1550 FIND**, etc. of a definition with undefined interpretation semantics;
- dividing by zero (**6.1.0100 */**, **6.1.0110 */MOD**, **6.1.0230 /**, **6.1.0240 /MOD**, **6.1.1561 FM/MOD**, **6.1.1890 MOD**, **6.1.2214 SM/REM**, **6.1.2370 UM/MOD**, **8.6.1.1820 M*/**);
- insufficient data-stack space or return-stack space (stack overflow);
- insufficient space for loop-control parameters;
- insufficient space in the dictionary;
- interpretating a word with undefined interpretation semantics;
- modifying the contents of the input buffer or a string literal (**3.3.3.4 Text-literal regions**, **3.3.3.5 Input buffers**);
- overflow of a pictured numeric output string;

- parsed string overflow;
- producing a result out of range, e.g., multiplication (using *****) results in a value too big to be represented by a single-cell integer (**6.1.0090 ***, **6.1.0100 */**, **6.1.0110 */MOD**, **6.1.0570 >NUMBER**, **6.1.1561 FM/MOD**, **6.1.2214 SM/REM**, **6.1.2370 UM/MOD**, **6.2.0970 CONVERT**, **8.6.1.1820 M*/**);
- reading from an empty data stack or return stack (stack underflow);
- unexpected end of input buffer, resulting in an attempt to use a zero-length string as a *name*;

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

- **>IN** greater than size of input buffer (**3.4.1 Parsing**);
- **6.1.2120 RECURSE** appears after **6.1.1250 DOES>**;
- argument input source different than current input source for **6.2.2148 RESTORE-INPUT**;
- data space containing definitions is de-allocated (**3.3.3.2 Contiguous regions**);
- data space read/write with incorrect alignment (**3.3.3.1 Address alignment**);
- data-space pointer not properly aligned (**6.1.0150 ,**, **6.1.0860 C,**);
- less than *u*+2 stack items (**6.2.2030 PICK**, **6.2.2150 ROLL**);
- loop-control parameters not available (**6.1.0140 +LOOP**, **6.1.1680 I**, **6.1.1730 J**, **6.1.1760 LEAVE**, **6.1.1800 LOOP**, **6.1.2380 UNLOOP**);
- most recent definition does not have a *name* (**6.1.1710 IMMEDIATE**);
- ~~*name* not defined by **6.2.2405** VALUE used by **6.2.2295** TO;~~ x:to
- **6.2.2295** TO applied to a *name* not defined by a word with a "**TO** *name* run-time" 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** ~~or~~, **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**, **6.1.2210 SIGN**).

The following specific ambiguous conditions are have been introduced as a consequence of specific extensions.

X:deferrd

- access to a deferred word, a word defined by **6.2.0 DEFER**, which has yet to be associated with an *xt*.

- access to a deferred word, a word defined by **6.2.0 DEFER**, which was not defined by **6.2.0 DEFER**.
- **6.1.2033 POSTPONE**, **6.2.2530 [COMPILE]**, **6.1.2510 ['']** or **6.1.0070 '** applied to **6.2.0 ACTION-OF** or **6.2.0 IS**.

x:required

X:required

- a file is required while it is being **REQUIRED** (11.6.2.0) or **INCLUDED** (11.6.1.1718).
- a marker is defined outside and executed inside a file or vice versa, and the file is **REQUIRED** (11.6.2.0) 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 **11.6.2.0 REQUIRED** (by doing some renaming between the invocations of **REQUIRED**).
- the stack effect of including with **11.6.2.0 REQUIRED** the file is not ($i \times x - i \times x$) .

4.1.3 Other system documentation

A system shall provide the following information:

- list of non-standard words using **6.2.2000 PAD** (**3.3.3.6 Other transient regions**);
- operator's terminal facilities available;
- program data space available, in address units;
- return stack space available, in cells;
- stack space available, in cells;
- system dictionary space required, in address units.

4.2 Program documentation

4.2.1 Environmental dependencies

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

- considering the pictured numeric output string buffer a fixed area with unchanging access parameters (**3.3.3.6 Other transient regions**);
- depending on the presence or absence of non-graphic characters in a received string (**6.1.0695 ACCEPT**, **6.2.1390 EXPECT**);
- relying on a particular rounding direction (**3.2.2.1 Integer division**);
- requiring a particular number representation and arithmetic (**3.2.1.1 Internal number representation**);
- requiring non-standard words or techniques (**3 Usage requirements**);
- requiring the ability to send or receive control characters (**3.1.2.2 Control characters**, **6.1.1750 KEY**);
- using control characters to perform specific functions **6.1.1320 EMIT**, **6.1.2310 TYPE**);

- using flags as arithmetic operands (**3.1.3.1 Flags**);
- using lower case for standard definition names or depending on the case sensitivity of a system (**3.3.1.2 Definition names**);
- using the graphic character with a value of hex 24 (**3.1.2.1 Graphic characters**).

4.2.2 Other program documentation

A program shall also document:

- minimum operator's terminal facilities required;
- whether a Standard System exists after the program is loaded.

5 Compliance and labeling

5.1 ANS Forth systems

5.1.1 System compliance

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

5.1.2 System labeling

A Standard System (Subset) shall be labeled an "ANS Forth 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 ANS Forth programs

5.2.1 Program compliance

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

5.2.2 Program labeling

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

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

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

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

6 Glossary

6.1 Core words

6.1.0010 ! "store" CORE

(*x a-addr* –)

Store *x* at *a-addr*.

See: **3.3.3.1 Address alignment**.

Testing: See **A.6.1.0150 ,**

6.1.0030 # "number-sign" CORE

(ud_1 – ud_2)

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

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

Testing:
```
:  GP3 <# 1 0 # # #> S" 01" S= ;
{ GP3 -> <TRUE> }
```

6.1.0040 #> "number-sign-greater" CORE

(*xd* – *c-addr u*)

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

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

Testing: See **A.6.1.0030 #**, **A.6.1.0050 #S**, **A.6.1.1670 HOLD** and **A.6.1.2210 SIGN**.

6.1.0050 #S "number-sign-s" CORE

(ud_1 – ud_2)

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

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

Testing:
```
:  GP4 <# 1 0 #S #> S" 1" S= ;
{ GP4 -> <TRUE> }

:  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 ! ;
{ GP5 -> <TRUE> }

:  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 ;
{ GP6 -> <TRUE> }

:  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 ! ;
{ GP7 -> <TRUE> }
```

6.1.0070 **'** "tick" CORE

("⟨*spaces*⟩*name*" – *xt*)

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

See: **3.4.3.2 Interpretation semantics**, **3.4.1 Parsing**, **A.6.1.2033 POSTPONE**, **A.6.1.2510 ['] **, **D.6.7 Immediacy**.

Rationale: Typical use: ... *' name*.

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

See: **A.3.4.1.2 Interpretation semantics**, **A.6.1.1550 FIND**.

Testing:
```
{ :  GT1 123 ; -> }
{ ' GT1 EXECUTE -> 123 }
```

6.1.0080 ("paren" CORE

Compilation: Perform the execution semantics given below.

Execution: (*"ccc⟨paren⟩"* –)

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

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

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

Rationale: Typical use: ... **(** *ccc*) ...

Testing: None.

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.

Testing:
```
{ 0 0 * -> 0 }          \ TEST IDENTITIE\S
{ 0 1 * -> 0 }
{ 1 0 * -> 0 }
{ 1 2 * -> 2 }
{ 2 1 * -> 2 }
{ 3 3 * -> 9 }
{ -3 3 * -> -9 }
{ 3 -3 * -> -9 }
{ -3 -3 * -> 9 }
{ MID-UINT+1 1 RSHIFT 2 * -> MID-UINT+1 }
{ MID-UINT+1 2 RSHIFT 4 * -> MID-UINT+1 }
{ MID-UINT+1 1 RSHIFT MID-UINT+1 OR 2 * -> MID-UINT+1 }
```

6.1.0100 */ "star-slash" CORE

(n_1 n_2 n_3 – n_4)

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

See: **3.2.2.1 Integer division**.

Testing:
```
IFFLOORED   :  T*/ T*/MOD SWAP DROP ;
IFSYM       :  T*/ T*/MOD SWAP DROP ;
{ 0 2 1 */ -> 0 2 1 T*/ }
{ 1 2 1 */ -> 1 2 1 T*/ }
{ 2 2 1 */ -> 2 2 1 T*/ }
{ -1 2 1 */ -> -1 2 1 T*/ }
{ -2 2 1 */ -> -2 2 1 T*/ }
```

```
{ 0 2 -1 */ -> 0 2 -1 T*/ }
{ 1 2 -1 */ -> 1 2 -1 T*/ }
{ 2 2 -1 */ -> 2 2 -1 T*/ }
{ -1 2 -1 */ -> -1 2 -1 T*/ }
{ -2 2 -1 */ -> -2 2 -1 T*/ }
{ 2 2 2 */ -> 2 2 2 T*/ }
{ -1 2 -1 */ -> -1 2 -1 T*/ }
{ -2 2 -2 */ -> -2 2 -2 T*/ }
{ 7 2 3 */ -> 7 2 3 T*/ }
{ 7 2 -3 */ -> 7 2 -3 T*/ }
{ -7 2 3 */ -> -7 2 3 T*/ }
{ -7 2 -3 */ -> -7 2 -3 T*/ }
{ MAX-INT 2 MAX-INT */ -> MAX-INT 2 MAX-INT T*/ }
{ MIN-INT 2 MIN-INT */ -> MIN-INT 2 MIN-INT T*/ }
```

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

(n_1 n_2 n_3 – n_4 n_5)

Multiply n_1 by n_2 producing the intermediate double-cell result d. Divide d by n_3 producing the single-cell remainder n_4 and the single-cell quotient n_5. An ambiguous condition exists if n_3 is zero, or if the quotient n_5 lies outside the range of a single-cell signed integer. If d and n_3 differ in sign, the implementation-defined result returned will be the same as that returned by either the phrase **`>R M* R> FM/MOD`** or the phrase **`>R M* R> SM/REM`**.

See: **3.2.2.1 Integer division**.

Testing:
```
IFFLOORED   :  T*/MOD >R M* R> FM/MOD ;
IFSYM       :  T*/MOD >R M* R> SM/REM ;
```

```
{ 0 2 1 */MOD -> 0 2 1 T*/MOD }
{ 1 2 1 */MOD -> 1 2 1 T*/MOD }
{ 2 2 1 */MOD -> 2 2 1 T*/MOD }
{ -1 2 1 */MOD -> -1 2 1 T*/MOD }
{ -2 2 1 */MOD -> -2 2 1 T*/MOD }
{ 0 2 -1 */MOD -> 0 2 -1 T*/MOD }
{ 1 2 -1 */MOD -> 1 2 -1 T*/MOD }
{ 2 2 -1 */MOD -> 2 2 -1 T*/MOD }
{ -1 2 -1 */MOD -> -1 2 -1 T*/MOD }
{ -2 2 -1 */MOD -> -2 2 -1 T*/MOD }
{ 2 2 2 */MOD -> 2 2 2 T*/MOD }
{ -1 2 -1 */MOD -> -1 2 -1 T*/MOD }
{ -2 2 -2 */MOD -> -2 2 -2 T*/MOD }
{ 7 2 3 */MOD -> 7 2 3 T*/MOD }
{ 7 2 -3 */MOD -> 7 2 -3 T*/MOD }
{ -7 2 3 */MOD -> -7 2 3 T*/MOD }
{ -7 2 -3 */MOD -> -7 2 -3 T*/MOD }
{ MAX-INT 2 MAX-INT */MOD -> MAX-INT 2 MAX-INT T*/MOD }
{ MIN-INT 2 MIN-INT */MOD -> MIN-INT 2 MIN-INT T*/MOD }
```

6.1.0120 + "plus" CORE

(n_1 | u_1 n_2 | u_2 – n_3 | u_3)

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

See: **3.3.3.1 Address alignment**.

Testing:
```
{ 0 5 + -> 5 }
{ 5 0 + -> 5 }
{ 0 -5 + -> -5 }
{ -5 0 + -> -5 }
{ 1 2 + -> 3 }
{ 1 -2 + -> -1 }
{ -1 2 + -> 1 }
{ -1 -2 + -> -3 }
{ -1 1 + -> 0 }
{ MID-UINT 1 + -> MID-UINT+1 }
```

6.1.0130 +! "plus-store" CORE

(*n* | *u a-addr* –)

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

See: **3.3.3.1 Address alignment**.

Testing:
```
{ 0 1ST ! -> }
{ 1 1ST +!  -> }
{ 1ST @ -> 1 }
{ -1 1ST +!  1ST @ -> 0 }
```

6.1.0140 +LOOP "plus-loop" CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: *do-sys* –)

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

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

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

See: **6.1.1240 DO**, **6.1.1680 I**, **6.1.1760 LEAVE**.

Rationale: Typical use: **:** X … limit first **DO** … step **+LOOP ;**

Testing:
```
{ :  GD2 DO I -1 +LOOP ; -> }
{ 1 4 GD2 -> 4 3 2 1 }
{ -1 2 GD2 -> 2 1 0 -1 }
{ MID-UINT MID-UINT+1 GD2 -> MID-UINT+1 MID-UINT }
```

6.1.0150 **,** "comma" CORE

(*x* –)

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

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

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

See: **6.2.0945 COMPILE,**.

Testing:
```
HERE 1 ,
HERE 2 ,
CONSTANT 2ND
CONSTANT 1ST
{ 1ST 2ND U< -> <TRUE> }   \ HERE MUST GROW WITH ALLOT
{ 1ST CELL+ -> 2ND }       \ ...  BY ONE CELL
{ 1ST 1 CELLS + -> 2ND }
{ 1ST @ 2ND @ -> 1 2 }
{ 5 1ST ! -> }
{ 1ST @ 2ND @ -> 5 2 }
{ 6 2ND ! -> }
{ 1ST @ 2ND @ -> 5 6 }
{ 1ST 2@ -> 6 5 }
{ 2 1 1ST 2!  -> }
{ 1ST 2@ -> 2 1 }
{ 1S 1ST ! 1ST @ -> 1S }   \ CAN STORE CELL-WIDE VALUE
```

6.1.0160 **–** "minus" CORE

(n_1 | u_1 n_2 | u_2 – n_3 | u_3)

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

See: **3.3.3.1 Address alignment**.

Testing:
```
{ 0 5 - -> -5 }
{ 5 0 - -> 5 }
{ 0 -5 - -> 5 }
{ -5 0 - -> -5 }
{ 1 2 - -> -1 }
{ 1 -2 - -> 3 }
{ -1 2 - -> -3 }
{ -1 -2 - -> 1 }
{ 0 1 - -> -1 }
{ MID-UINT+1 1 - -> MID-UINT }
```

6.1.0180 **.** "dot" CORE

(*n* –)

Display *n* in free field format.

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

Testing:

See: **A.6.1.1320 EMIT**.

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

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (*"ccc⟨quote⟩"* –)

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

Run-time: (–)

Display *ccc*.

See: **3.4.1 Parsing**, **6.2.0200 .(**.

Rationale: 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.

Testing: See **A.6.1.1320 EMIT**.

6.1.0230 **/** "slash" CORE

(n_1 n_2 – n_3)

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

See: **3.2.2.1 Integer division**.

Testing:
```
IFFLOORED   :  T/ T/MOD SWAP DROP ;
IFSYM       :  T/ T/MOD SWAP DROP ;
{ 0 1 / -> 0 1 T/ }
{ 1 1 / -> 1 1 T/ }
{ 2 1 / -> 2 1 T/ }
{ -1 1 / -> -1 1 T/ }
{ -2 1 / -> -2 1 T/ }
{ 0 -1 / -> 0 -1 T/ }
{ 1 -1 / -> 1 -1 T/ }
{ 2 -1 / -> 2 -1 T/ }
{ -1 -1 / -> -1 -1 T/ }
{ -2 -1 / -> -2 -1 T/ }
{ 2 2 / -> 2 2 T/ }
```

```
{ -1 -1 / -> -1 -1 T/ }
{ -2 -2 / -> -2 -2 T/ }
{ 7 3 / -> 7 3 T/ }
{ 7 -3 / -> 7 -3 T/ }
{ -7 3 / -> -7 3 T/ }
{ -7 -3 / -> -7 -3 T/ }
{ MAX-INT 1 / -> MAX-INT 1 T/ }
{ MIN-INT 1 / -> MIN-INT 1 T/ }
{ MAX-INT MAX-INT / -> MAX-INT MAX-INT T/ }
{ MIN-INT MIN-INT / -> MIN-INT MIN-INT T/ }
```

6.1.0240 `/MOD` "slash-mod" CORE

(n_1 n_2 – n_3 n_4)

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

See: **3.2.2.1 Integer division**.

Testing:
```
IFFLOORED   :  T/MOD >R S>D R> FM/MOD ;
IFSYM       :  T/MOD >R S>D R> SM/REM ;
```

```
{ 0 1 /MOD -> 0 1 T/MOD }
{ 1 1 /MOD -> 1 1 T/MOD }
{ 2 1 /MOD -> 2 1 T/MOD }
{ -1 1 /MOD -> -1 1 T/MOD }
{ -2 1 /MOD -> -2 1 T/MOD }
{ 0 -1 /MOD -> 0 -1 T/MOD }
{ 1 -1 /MOD -> 1 -1 T/MOD }
{ 2 -1 /MOD -> 2 -1 T/MOD }
{ -1 -1 /MOD -> -1 -1 T/MOD }
{ -2 -1 /MOD -> -2 -1 T/MOD }
{ 2 2 /MOD -> 2 2 T/MOD }
{ -1 -1 /MOD -> -1 -1 T/MOD }
{ -2 -2 /MOD -> -2 -2 T/MOD }
{ 7 3 /MOD -> 7 3 T/MOD }
{ 7 -3 /MOD -> 7 -3 T/MOD }
{ -7 3 /MOD -> -7 3 T/MOD }
{ -7 -3 /MOD -> -7 -3 T/MOD }
{ MAX-INT 1 /MOD -> MAX-INT 1 T/MOD }
{ MIN-INT 1 /MOD -> MIN-INT 1 T/MOD }
{ MAX-INT MAX-INT /MOD -> MAX-INT MAX-INT T/MOD }
{ MIN-INT MIN-INT /MOD -> MIN-INT MIN-INT T/MOD }
```

6.1.0250 `0<` "zero-less" CORE

(*n* – *flag*)

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

Testing:
```
{ 0 0< -> <FALSE> }
{ -1 0< -> <TRUE> }
```

```
{ MIN-INT 0< -> <TRUE> }
{ 1 0< -> <FALSE> }
{ MAX-INT 0< -> <FALSE> }
```

6.1.0270 `0=` "zero-equals" CORE

(*x* – *flag*)

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

Testing:
```
{ 0 0= -> <TRUE> }
{ 1 0= -> <FALSE> }
{ 2 0= -> <FALSE> }
{ -1 0= -> <FALSE> }
{ MAX-UINT 0= -> <FALSE> }
{ MIN-INT 0= -> <FALSE> }
{ MAX-INT 0= -> <FALSE> }
```

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.

Testing:
```
{ 0 1+ -> 1 }
{ -1 1+ -> 0 }
{ 1 1+ -> 2 }
{ MID-UINT 1+ -> MID-UINT+1 }
```

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.

Testing:
```
{ 2 1- -> 1 }
{ 1 1- -> 0 }
{ 0 1- -> -1 }
{ MID-UINT+1 1- -> MID-UINT }
```

6.1.0310 `2!` "two-store" CORE

(x_1 x_2 *a-addr* –)

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

See: **3.3.3.1 Address alignment**.

Testing: See **A.6.1.0150 ,**

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.

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

Testing:
```
{ 0S 2* -> 0S }
{ 1 2* -> 2 }
{ 4000 2* -> 8000 }
{ 1S 2* 1 XOR -> 1S }
{ MSB 2* -> 0S }
```

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.

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

Testing:
```
{ 0S 2/ -> 0S }
{ 1 2/ -> 0 }
{ 4000 2/ -> 2000 }
{ 1S 2/ -> 1S }          \ MSB PROPOGATED
{ 1S 1 XOR 2/ -> 1S }
{ MSB 2/ MSB AND -> MSB }
```

6.1.0350 `2@` "two-fetch" CORE

(*a-addr* – x_1 x_2)

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

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

Rationale: With **2@** the storage order is specified by the Standard.

Testing: See **A.6.1.0150 ,**

6.1.0370 `2DROP` "two-drop" CORE

(x_1 x_2 –)

Drop cell pair x_1 x_2 from the stack.

Testing:
```
{ 1 2 2DROP -> }
```

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.

Testing: `{ 1 2 2DUP -> 1 2 1 2 }`

6.1.0400 `2OVER` "two-over" CORE

(x_1 x_2 x_3 x_4 – x_1 x_2 x_3 x_4 x_1 x_2)

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

Testing: `{ 1 2 3 4 2OVER -> 1 2 3 4 1 2 }`

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.

Testing: `{ 1 2 3 4 2SWAP -> 3 4 1 2 }`

6.1.0450 `:` "colon" CORE

(C: "⟨*spaces*⟩*name*" – *colon-sys*)

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

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

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

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

name Execution: ($i \times x$ – $j \times x$)

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

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

Rationale: 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.

Testing:
```
{ :  NOP : POSTPONE ; ; -> }
{ NOP NOP1 NOP NOP2 -> }
{ NOP1 -> }
{ NOP2 -> }
```

The following tests the dictionary search order:

```
{ :  GDX 123 ;  :  GDX GDX 234 ; -> }
{ GDX -> 123 234 }
```

6.1.0460 ; "semicolon" CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: *colon-sys* –)

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

Run-time: (–) (R: *nest-sys* –)

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

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

Rationale: 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.

Testing: See **A.6.1.0450 :**.

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: **6.1.2340 U<**.

Testing:
```
{ 0 1 < -> <TRUE> }
{ 1 2 < -> <TRUE> }
{ -1 0 < -> <TRUE> }
{ -1 1 < -> <TRUE> }
{ MIN-INT 0 < -> <TRUE> }
{ MIN-INT MAX-INT < -> <TRUE> }
{ 0 MAX-INT < -> <TRUE> }
{ 0 0 < -> <FALSE> }
{ 1 1 < -> <FALSE> }
{ 1 0 < -> <FALSE> }
{ 2 1 < -> <FALSE> }
{ 0 -1 < -> <FALSE> }
{ 1 -1 < -> <FALSE> }
```

```
{ 0 MIN-INT < -> <FALSE> }
{ MAX-INT MIN-INT < -> <FALSE> }
{ MAX-INT 0 < -> <FALSE> }
```

6.1.0490 <# "less-number-sign" CORE

(–)

Initialize the pictured numeric output conversion process.

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

Testing: See **A.6.1.0030 #**, **A.6.1.0050 #S**, **A.6.1.1670 HOLD** and **A.6.1.2210 SIGN**.

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.

Testing:
```
{ 0 0 = -> <TRUE> }
{ 1 1 = -> <TRUE> }
{ -1 -1 = -> <TRUE> }
{ 1 0 = -> <FALSE> }
{ -1 0 = -> <FALSE> }
{ 0 1 = -> <FALSE> }
{ 0 -1 = -> <FALSE> }
```

6.1.0540 > "greater-than" CORE

(n_1 n_2 – *flag*)

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

See: **6.2.2350 U>**.

Testing:
```
{ 0 1 > -> <FALSE> }
{ 1 2 > -> <FALSE> }
{ -1 0 > -> <FALSE> }
{ -1 1 > -> <FALSE> }
{ MIN-INT 0 > -> <FALSE> }
{ MIN-INT MAX-INT > -> <FALSE> }
{ 0 MAX-INT > -> <FALSE> }
{ 0 0 > -> <FALSE> }
{ 1 1 > -> <FALSE> }
{ 1 0 > -> <TRUE> }
{ 2 1 > -> <TRUE> }
{ 0 -1 > -> <TRUE> }
{ 1 -1 > -> <TRUE> }
{ 0 MIN-INT > -> <TRUE> }
{ MAX-INT MIN-INT > -> <TRUE> }
{ MAX-INT 0 > -> <TRUE> }
```

6.1.0550 `>BODY` "to-body" CORE

(*xt* – *a-addr*)

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

See: **3.3.3 Data space**.

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

Testing:
```
{ CREATE CR0 -> }
{ ' CR0 >BODY -> HERE }
```

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.

Testing:
```
VARIABLE SCANS
: RESCAN? -1 SCANS +!  SCANS @ IF 0 >IN ! THEN ;
{ 2 SCANS !
345 RESCAN?
-> 345 345 }
:  GS2 5 SCANS ! S" 123 RESCAN?" EVALUATE ;
{ GS2 -> 123 123 123 123 123 }
```

6.1.0570 `>NUMBER` "to-number" CORE

(ud_1 $c\text{-}addr_1$ u_1 – ud_2 $c\text{-}addr_2$ u_2)

ud_2 is the unsigned result of converting the characters within the string specified by $c\text{-}addr_1$ 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. $c\text{-}addr_2$ 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: **3.2.1.2 Digit conversion**.

Testing:
```
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 ;
{ 0 0 GN' 0' >NUMBER -> 0 0 GN-CONSUMED }
{ 0 0 GN' 1' >NUMBER -> 1 0 GN-CONSUMED }
{ 1 0 GN' 1' >NUMBER -> BASE @ 1+ 0 GN-CONSUMED }
{ 0 0 GN' -' >NUMBER -> 0 0 GN-STRING }   \ SHOULD FAIL TO CONVERT
{ 0 0 GN' +' >NUMBER -> 0 0 GN-STRING }
{ 0 0 GN' .' >NUMBER -> 0 0 GN-STRING }
```

```
:  >NUMBER-BASED
  BASE @ >R BASE ! >NUMBER R> BASE ! ;

{ 0 0 GN' 2' 10 >NUMBER-BASED -> 2 0 GN-CONSUMED }
{ 0 0 GN' 2' 2 >NUMBER-BASED -> 0 0 GN-STRING }
{ 0 0 GN' F' 10 >NUMBER-BASED -> F 0 GN-CONSUMED }
{ 0 0 GN' G' 10 >NUMBER-BASED -> 0 0 GN-STRING }
{ 0 0 GN' G' MAX-BASE >NUMBER-BASED -> 10 0 GN-CONSUMED }
{ 0 0 GN' Z' MAX-BASE >NUMBER-BASED -> 23 0 GN-CONSUMED }

:  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 ! ;

{ 0 0 2 GN1 -> 0 0 0 }
{ MAX-UINT 0 2 GN1 -> MAX-UINT 0 0 }
{ MAX-UINT DUP 2 GN1 -> MAX-UINT DUP 0 }
{ 0 0 MAX-BASE GN1 -> 0 0 0 }
{ MAX-UINT 0 MAX-BASE GN1 -> MAX-UINT 0 0 }
{ MAX-UINT DUP MAX-BASE GN1 -> MAX-UINT DUP 0 }
```

6.1.0580 `>R` "to-r" CORE

Interpretation: Interpretation semantics for this word are undefined.

Execution: (*x* –) (R: – *x*)

Move *x* to the return stack.

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

Testing: `TESTING >R R> R@`

```
{ :  GR1 >R R> ; -> }
{ :  GR2 >R R@ R> DROP ; -> }
{ 123 GR1 -> 123 }
{ 123 GR2 -> 123 }
{ 1S GR1 -> 1S }      ( RETURN STACK HOLDS CELLS )
```

6.1.0630 `?DUP` "question-dupe" CORE

(*x* – *0* | *x x*)

Duplicate *x* if it is non-zero.

Testing:

```
{ -1 ?DUP -> -1 -1 }
{ 0 ?DUP -> 0 }
{ 1 ?DUP -> 1 1 }
```

6.1.0650 `@` "fetch" CORE

(*a-addr* – *x*)

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

See: **3.3.3.1 Address alignment**.

Testing: See **A.6.1.0150 ,**

6.1.0670 `ABORT` CORE

(*i*×*x* –) (R: *j*×*x* –)

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

See: **9.6.2.0670 `ABORT`**.

Testing: None.

6.1.0680 `ABORT"` "abort-quote" CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: ("*ccc*⟨*quote*⟩" –)

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

Run-time: (*i*×*x* x_1 – | *i*×*x*) (R: *j*×*x* – | *j*×*x*)

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

See: **3.4.1 Parsing**, **9.6.2.0680 `ABORT"`**.

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

Testing: None.

6.1.0690 `ABS` "abs" CORE

(*n* – *u*)

u is the absolute value of *n*.

Testing:
```
{ 0 ABS -> 0 }
{ 1 ABS -> 1 }
{ -1 ABS -> 1 }
{ MIN-INT ABS -> MID-UINT+1 }
```

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*.

Rationale: Previous standards specified that collection of the input string terminates when either a "return" is received or when $+n_1$ characters have been received. Terminating when $+n_1$ characters have been received is difficult, expensive, or impossible to implement in some system environments. Consequently, a number of existing implementations do not comply with this requirement. Since line-editing and collection functions are often implemented by system components beyond the control of the Forth implementation, this Standard imposes no such requirement. A Standard Program may only assume that it can receive an input string with **ACCEPT** or **EXPECT**. The detailed sequence of user actions necessary to prepare and transmit that line are beyond the scope of this Standard.

Specification of a non-zero, positive integer count ($+n_1$) for **ACCEPT** allows some implementors to continue their practice of using a zero or negative value as a flag to trigger special behavior. Insofar as such behavior is outside the Standard, Standard Programs cannot depend upon it, but the Technical Committee doesn't wish to preclude it unnecessarily. Since actual values are almost always small integers, no functionality is impaired by this restriction.

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

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

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

As in previous standards, **EXPECT** returns the input string immediately after the requested number of characters are entered, as well as when a line terminator is received. The "automatic termination after specified count of characters have been entered" behavior is widely considered undesirable because the user "loses control" of the input editing process at a potentially unknown time (the user does not necessarily know how many characters were requested from **EXPECT**). Thus **EXPECT** and **SPAN** have been made obsolescent and exist in the Standard only as a concession to existing implementations. If **EXPECT** exists in a Standard System it must have the "automatic termination" behavior.

ACCEPT does not have the "automatic termination" behavior of **EXPECT**. However, because external system hardware and software may perform the **ACCEPT** function, when a line terminator is received the action of the cursor, and therefore the display,

is implementation-defined. It is recommended that the cursor remain immediately following the entered text after a line terminator is received.

Testing:
```
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
;
{ ACCEPT-TEST -> }
```

6.1.0705 ALIGN CORE

(–)

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

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

Rationale: 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.

Testing:
```
ALIGN 1 ALLOT HERE ALIGN HERE 3 CELLS ALLOT
CONSTANT A-ADDR CONSTANT UA-ADDR { UA-ADDR ALIGNED -> A-ADDR
}
{ 1 A-ADDR C! A-ADDR C@ -> 1 }
{ 1234 A-ADDR ! A-ADDR @ -> 1234 }
{ 123 456 A-ADDR 2!  A-ADDR 2@ -> 123 456 }
{ 2 A-ADDR CHAR+ C! A-ADDR CHAR+ C@ -> 2 }
{ 3 A-ADDR CELL+ C! A-ADDR CELL+ C@ -> 3 }
{ 1234 A-ADDR CELL+ !  A-ADDR CELL+ @ -> 1234 }
{ 123 456 A-ADDR CELL+ 2!  A-ADDR CELL+ 2@ -> 123 456 }
```

6.1.0706 ALIGNED CORE

(*addr* – *a-addr*)

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

See: **3.3.3.1 Address alignment**.

Rationale: See: **A.6.1.0705 ALIGN**.

Testing: See **A.6.1.0705 ALIGN**.

6.1.0710 ALLOT CORE

(*n* –)

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

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

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

See: **3.3.3 Data space**.

Testing:
```
HERE 1 ALLOT
HERE
CONSTANT 2NDA
CONSTANT 1STA
{ 1STA 2NDA U< -> <TRUE> }   \ HERE MUST GROW WITH ALLOT
{ 1STA 1+ -> 2NDA }          \ ...  BY ONE ADDRESS UNIT
( MISSING TEST: NEGATIVE ALLOT )
```

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.

Testing:
```
{ 0 0 AND -> 0 }
{ 0 1 AND -> 0 }
{ 1 0 AND -> 0 }
{ 1 1 AND -> 1 }

{ 0 INVERT 1 AND -> 1 }
{ 1 INVERT 1 AND -> 0 }

{ 0S 0S AND -> 0S }
{ 0S 1S AND -> 0S }
{ 1S 0S AND -> 0S }
{ 1S 1S AND -> 1S }
```

6.1.0750 BASE CORE

(– *a-addr*)

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

Testing:
```
:  GN2 \ ( – 16 10 )
  BASE @ >R HEX BASE @ DECIMAL BASE @ R> BASE ! ;
{ GN2 -> 10 A }
```

6.1.0760 `BEGIN` CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: – *dest*)

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

Run-time: (–)

Continue execution.

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

Rationale: Typical use:

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

or

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

Testing: See **A.6.1.2430 `WHILE`** and **A.6.1.2390 `UNTIL`**.

6.1.0770 `BL` "b-l" CORE

(– *char*)

char is the character value for a space.

Rationale: 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.

Testing: `{` **`BL`** `->` `20` `}`

6.1.0850 `C!` "c-store" CORE

(*char c-addr* –)

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

See: **3.3.3.1 Address alignment**.

Testing: See **A.6.1.0860 `C,`**

6.1.0860 `C,` "c-comma" CORE

(*char* –)

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

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

Testing:
```
HERE 1 C,
HERE 2 C,
CONSTANT 2NDC
CONSTANT 1STC
{ 1STC 2NDC U< -> <TRUE> }   \ HERE MUST GROW WITH ALLOT
{ 1STC CHAR+ -> 2NDC }       \ ...  BY ONE CHAR
{ 1STC 1 CHARS + -> 2NDC }
{ 1STC C@ 2NDC C@ -> 1 2 }
{ 3 1STC C! -> }
{ 1STC C@ 2NDC C@ -> 3 2 }
{ 4 2NDC C! -> }
{ 1STC C@ 2NDC C@ -> 3 4 }
```

6.1.0870 C@ "c-fetch" CORE

(*c-addr* – *char*)

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

See: **3.3.3.1 Address alignment**.

Testing: See **A.6.1.0860 C,**

6.1.0880 CELL+ "cell-plus" CORE

(*a-addr*$_1$ – *a-addr*$_2$)

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

See: **3.3.3.1 Address alignment**.

Rationale: 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
```

Testing: See **A.6.1.0150 ,**

6.1.0890 CELLS CORE

(n_1 – n_2)

n_2 is the size in address units of n_1 cells.

Rationale: See: **A.6.1.0880 CELL+**.

Example: **CREATE** NUMBERS 100 **CELLS ALLOT**
(Allots space in the array NUMBERS for 100 cells of data.)

Testing:
```
: 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
)
{ 1 CELLS 1 < -> <FALSE> }
{ 1 CELLS 1 CHARS MOD -> 0 }
{ 1S BITS 10 < -> <FALSE> }
```

6.1.0895 CHAR "char" CORE

("⟨*spaces*⟩*name*" - *char*)

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

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

Rationale: Typical use: ... **CHAR** A **CONSTANT** "A" ...

Testing:
```
{ CHAR X -> 58 }
{ CHAR HELLO -> 48 }
```

6.1.0897 CHAR+ "char-plus" CORE

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

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

See: **3.3.3.1 Address alignment**.

Testing: See **A.6.1.0860 C,**

6.1.0898 CHARS "chars" CORE

(n_1 - n_2)

n_2 is the size in address units of n_1 characters.

Testing:
```
( CHARACTERS >= 1 AU, <= SIZE OF CELL, >= 8 BITS )
{ 1 CHARS 1 < -> <FALSE> }
{ 1 CHARS 1 CELLS > -> <FALSE> }
( TBD: HOW TO FIND NUMBER OF BITS? )
```

6.1.0950 CONSTANT CORE

(*x* "⟨*spaces*⟩*name*" -)

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

name is referred to as a "constant".

name Execution: (- *x*)

Place *x* on the stack.

See: **3.4.1 Parsing**.

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

Testing:
```
{ 123 CONSTANT X123 -> }
{ X123 -> 123 }
{ :  EQU CONSTANT ; -> }
{ X123 EQU Y123 -> }
{ Y123 -> 123 }
```

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*$_2$.

Testing: `{ GT1STRING COUNT -> GT1STRING CHAR+ 3 }`

6.1.0990 `CR` "c-r" CORE

(–)

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

Testing: See **A.6.1.1320 EMIT**.

6.1.1000 `CREATE` CORE

("⟨*spaces*⟩*name*" –)

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

name Execution: (– *a-addr*)

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

See: **3.3.3 Data space**, **6.1.1250 DOES>**.

Rationale: 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` ...

Testing: See **A.6.1.0550 >BODY** and **A.6.1.1250 DOES>**.

6.1.1170 `DECIMAL` CORE

(–)

Set the numeric conversion radix to ten (decimal).

Testing: See **A.6.1.0750 BASE**.

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.

Testing:
```
{ 0 1 DEPTH -> 0 1 2 }
{ 0 DEPTH -> 0 1 }
{ DEPTH -> 0 }
```

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: **3.2.3.2 Control-flow stack**, **6.1.0140 +LOOP**, **6.1.1800 LOOP**.

Rationale: Typical use:

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

or

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

Testing: See **A.6.1.1800 LOOP**, **A.6.1.0140 +LOOP**, **A.6.1.1730 J**, **A.6.1.1760 LEAVE** and **A.6.1.2380 UNLOOP**.

6.1.1250 `DOES>` “does” CORE

Interpretation: Interpretation semantics for this word are undefined.

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

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

Run-time: (–) (R: *nest-sys*$_1$ –)

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

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

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

name Execution: (*i*×*x* – *j*×*x*)

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

See: **6.1.1000 CREATE**.

Rationale: 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.

Testing:
```
{ :  DOES1 DOES> @ 1 + ; -> }
{ :  DOES2 DOES> @ 2 + ; -> }
{ CREATE CR1 -> }
{ CR1 -> HERE }
{ 1 , -> }
{ CR1 @ -> 1 }
{ DOES1 -> }
{ CR1 -> 2 }
{ DOES2 -> }
{ CR1 -> 3 }
{ :  WEIRD: CREATE DOES> 1 + DOES> 2 + ; -> }
{ WEIRD: W1 -> }
{ ' W1 >BODY -> HERE }
{ W1 -> HERE 1 + }
{ W1 -> HERE 2 + }
```

6.1.1260 DROP CORE

(*x* –)

Remove *x* from the stack.

Testing:
```
{ 1 2 DROP -> 1 }
{ 0 DROP -> }
```

6.1.1290 DUP "dupe" CORE

(*x* – *x x*)

Duplicate *x*.

Testing: `{ 1 DUP -> 1 1 }`

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: **6.1.1700 IF**, **6.1.2270 THEN**.

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

Testing: See **A.6.1.1700 IF**

6.1.1320 EMIT CORE

(*x* –)

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

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

See: **6.1.2310 TYPE**.

Testing: `TESTING OUTPUT: . ." CR EMIT SPACE SPACES TYPE U.`

```
: 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
;
{ OUTPUT-TEST -> }
```

6.1.1345 ENVIRONMENT? "environment-query" CORE

(*c-addr u* – *false* | *i*×*x true*)

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

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

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

Testing:
```
\ should be the same for any query starting with X:
{ S" X:deferred" ENVIRONMENT? DUP 0= XOR INVERT -> <TRUE> }
{ S" X:notfound" ENVIRONMENT? DUP 0= XOR INVERT -> <FALSE> }
```

6.1.1360 EVALUATE CORE

(*i*×*x c-addr u* – *j*×*x*)

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

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

Testing:
```
: GE1 S" 123" ; IMMEDIATE
: GE2 S" 123 1+" ; IMMEDIATE
: GE3 S" : GE4 345 ;" ;
: GE5 EVALUATE ; IMMEDIATE
{ GE1 EVALUATE -> 123 }       ( TEST EVALUATE IN INTERP. STATE
)
{ GE2 EVALUATE -> 124 }
{ GE3 EVALUATE -> }
{ GE4 -> 345 }
{ : GE6 GE1 GE5 ; -> }        ( TEST EVALUATE IN COMPILE STATE
)
{ GE6 -> 123 }
{ : GE7 GE2 GE5 ; -> }
{ GE7 -> 124 }
```

6.1.1370 EXECUTE CORE

(*i*×*x xt* – *j*×*x*)

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

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

Testing: See **A.6.1.0070 '** and **A.6.1.2510 ['']**.

6.1.1380 EXIT CORE

Interpretation: Interpretation semantics for this word are undefined.

Execution: (–) (R: *nest-sys* –)

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

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

Rationale: Typical use: **:** X ... *test* **IF** ... **EXIT THEN** ... **;**

Testing: See **A.6.1.2380 UNLOOP**.

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*.

Testing:
```
{ FBUF 0 20 FILL -> }
{ SEEBUF -> 00 00 00 }
{ FBUF 1 20 FILL -> }
{ SEEBUF -> 20 00 00 }
{ FBUF 3 20 FILL -> }
{ SEEBUF -> 20 20 20 }
```

6.1.1550 FIND CORE

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

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

See: **3.4.2 Finding definition names**, **A.6.1.0070 '** , **A.6.1.2510 ['] **, **A.6.1.2033 POSTPONE**, **D.6.7 Immediacy**.

Rationale: 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.

Testing:
```
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
{ GT1STRING FIND -> ' GT1 -1 }
{ GT2STRING FIND -> ' GT2 1 }
( HOW TO SEARCH FOR NON-EXISTENT WORD? )
```

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

(d_1 n_1 – n_2 n_3)

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

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

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

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

```
: /_MOD ( n1 n2 – n3 n4) >R S>D R> FM/MOD ;
: /_ ( n1 n2 – n3) /_MOD SWAP DROP ;
: _MOD ( n1 n2 – n3) /_MOD DROP ;
: */_MOD ( n1 n2 n3 – n4 n5) >R M* R> FM/MOD ;
: */_ ( n1 n2 n3 – n4 ) */_MOD SWAP DROP ;
```

Testing:

```
{ 0 S>D 1 FM/MOD -> 0 0 }
{ 1 S>D 1 FM/MOD -> 0 1 }
{ 2 S>D 1 FM/MOD -> 0 2 }
{ -1 S>D 1 FM/MOD -> 0 -1 }
{ -2 S>D 1 FM/MOD -> 0 -2 }
{ 0 S>D -1 FM/MOD -> 0 0 }
{ 1 S>D -1 FM/MOD -> 0 -1 }
{ 2 S>D -1 FM/MOD -> 0 -2 }
{ -1 S>D -1 FM/MOD -> 0 1 }
{ -2 S>D -1 FM/MOD -> 0 2 }
{ 2 S>D 2 FM/MOD -> 0 1 }
{ -1 S>D -1 FM/MOD -> 0 1 }
{ -2 S>D -2 FM/MOD -> 0 1 }
{ 7 S>D 3 FM/MOD -> 1 2 }
{ 7 S>D -3 FM/MOD -> -2 -3 }
{ -7 S>D 3 FM/MOD -> 2 -3 }
{ -7 S>D -3 FM/MOD -> -1 2 }
{ MAX-INT S>D 1 FM/MOD -> 0 MAX-INT }
{ MIN-INT S>D 1 FM/MOD -> 0 MIN-INT }
{ MAX-INT S>D MAX-INT FM/MOD -> 0 1 }
{ MIN-INT S>D MIN-INT FM/MOD -> 0 1 }
{ 1S 1 4 FM/MOD -> 3 MAX-INT }
{ 1 MIN-INT M* 1 FM/MOD -> 0 MIN-INT }
{ 1 MIN-INT M* MIN-INT FM/MOD -> 0 1 }
{ 2 MIN-INT M* 2 FM/MOD -> 0 MIN-INT }
{ 2 MIN-INT M* MIN-INT FM/MOD -> 0 2 }
```

```
{ 1 MAX-INT M* 1 FM/MOD -> 0 MAX-INT }
{ 1 MAX-INT M* MAX-INT FM/MOD -> 0 1 }
{ 2 MAX-INT M* 2 FM/MOD -> 0 MAX-INT }
{ 2 MAX-INT M* MAX-INT FM/MOD -> 0 2 }
{ MIN-INT MIN-INT M* MIN-INT FM/MOD -> 0 MIN-INT }
{ MIN-INT MAX-INT M* MIN-INT FM/MOD -> 0 MAX-INT }
{ MIN-INT MAX-INT M* MAX-INT FM/MOD -> 0 MIN-INT }
{ MAX-INT MAX-INT M* MAX-INT FM/MOD -> 0 MAX-INT }
```

6.1.1650 HERE CORE

(– *addr*)

addr is the data-space pointer.

See: **3.3.3.2 Contiguous regions**.

Testing: See **A.6.1.0150 ,** , **A.6.1.0710 ALLOT** and **A.6.1.0860 C,** .

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.

Testing:
```
: GP1 <# 41 HOLD 42 HOLD 0 0 #> S" BA" S= ;
{ GP1 -> <TRUE> }
```

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.

Testing: See **A.6.1.1800 LOOP**, **A.6.1.0140 +LOOP**, **A.6.1.1730 J**, **A.6.1.1760 LEAVE** and **A.6.1.2380 UNLOOP**.

6.1.1700 IF CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: – *orig*)

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

Run-time: (*x* –)

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

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

Rationale: Typical use:

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

or

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

Testing:
```
{ :  GI1 IF 123 THEN ; -> }
{ :  GI2 IF 123 ELSE 234 THEN ; -> }
{ 0 GI1 -> }
{ 1 GI1 -> 123 }
{ -1 GI1 -> 123 }
{ 0 GI2 -> 234 }
{ 1 GI2 -> 123 }
{ -1 GI1 -> 123 }
```

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.

See: **D.6.7 Immediacy**.

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

Testing: See **A.6.1.2510 ['']**, **A.6.1.2033 POSTPONE**, **A.6.1.2250 STATE**, and **A.6.1.2165 S"**.

6.1.1720 INVERT CORE

(x_1 – x_2)

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

See: **6.1.1910 NEGATE**, **6.1.0270 0=**.

Rationale: 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.

Testing:
```
{ 0S INVERT -> 1S }
{ 1S INVERT -> 0S }
```

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.

Rationale: **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** ... **;**

Testing:
```
{ : GD3 DO 1 0 DO J LOOP LOOP ; -> }
{ 4 1 GD3 -> 1 2 3 }
{ 2 -1 GD3 -> -1 0 1 }
{ MID-UINT+1 MID-UINT GD3 -> MID-UINT }
{ : GD4 DO 1 0 DO J LOOP -1 +LOOP ; -> }
{ 1 4 GD4 -> 4 3 2 1 }
{ -1 2 GD4 -> 2 1 0 -1 }
{ MID-UINT MID-UINT+1 GD4 -> MID-UINT+1 MID-UINT }
```

6.1.1750 KEY CORE

(– *char*)

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

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

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

See: **10.6.2.1305 EKEY**, **10.6.2.1307 EKEY?**.

Testing: None.

6.1.1760 LEAVE CORE

Interpretation: Interpretation semantics for this word are undefined.

Execution: (–) (R: *loop-sys* –)

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

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

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

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

Testing:
```
{ :  GD5 123 SWAP 0 DO I 4 > IF DROP 234 LEAVE THEN LOOP ; ->
}
{ 1 GD5 -> 123 }
{ 5 GD5 -> 123 }
{ 6 GD5 -> 234 }
```

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.

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

Testing:
```
{ :  GT3 GT2 LITERAL ; -> }
{ GT3 -> ' GT1 }
```

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: **6.1.1240 DO**, **6.1.1680 I**, **6.1.1760 LEAVE**.

Rationale: Typical use:

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

or

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

Testing:
```
{ :  GD1 DO I LOOP ; -> }
{ 4 1 GD1 -> 1 2 3 }
{ 2 -1 GD1 -> -1 0 1 }
{ MID-UINT+1 MID-UINT GD1 -> MID-UINT }
```

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.

Testing:
```
{ 1 0 LSHIFT -> 1 }
{ 1 1 LSHIFT -> 2 }
{ 1 2 LSHIFT -> 4 }
{ 1 F LSHIFT -> 8000 }          \ BIGGEST GUARANTEED SHIFT
{ 1S 1 LSHIFT 1 XOR -> 1S }
{ MSB 1 LSHIFT -> 0 }
```

6.1.1810 M* "m-star" CORE

(n_1 n_2 – d)

d is the signed product of n_1 times n_2.

Rationale: 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.

Testing:
```
{ 0 0 M* -> 0 S>D }
{ 0 1 M* -> 0 S>D }
{ 1 0 M* -> 0 S>D }
{ 1 2 M* -> 2 S>D }
{ 2 1 M* -> 2 S>D }
{ 3 3 M* -> 9 S>D }
{ -3 3 M* -> -9 S>D }
{ 3 -3 M* -> -9 S>D }
{ -3 -3 M* -> 9 S>D }
{ 0 MIN-INT M* -> 0 S>D }
{ 1 MIN-INT M* -> MIN-INT S>D }
{ 2 MIN-INT M* -> 0 1S }
{ 0 MAX-INT M* -> 0 S>D }
{ 1 MAX-INT M* -> MAX-INT S>D }
{ 2 MAX-INT M* -> MAX-INT 1 LSHIFT 0 }
{ MIN-INT MIN-INT M* -> 0 MSB 1 RSHIFT }
{ MAX-INT MIN-INT M* -> MSB MSB 2/ }
{ MAX-INT MAX-INT M* -> 1 MSB 2/ INVERT }
```

6.1.1870 MAX CORE

(n_1 n_2 – n_3)

n_3 is the greater of n_1 and n_2.

Testing:
```
{ 0 1 MAX -> 1 }
{ 1 2 MAX -> 2 }
{ -1 0 MAX -> 0 }
{ -1 1 MAX -> 1 }
{ MIN-INT 0 MAX -> 0 }
{ MIN-INT MAX-INT MAX -> MAX-INT }
```

```
{ 0 MAX-INT MAX -> MAX-INT }
{ 0 0 MAX -> 0 }
{ 1 1 MAX -> 1 }
{ 1 0 MAX -> 1 }
{ 2 1 MAX -> 2 }
{ 0 -1 MAX -> 0 }
{ 1 -1 MAX -> 1 }
{ 0 MIN-INT MAX -> 0 }
{ MAX-INT MIN-INT MAX -> MAX-INT }
{ MAX-INT 0 MAX -> MAX-INT }
```

6.1.1880 MIN CORE

(n_1 n_2 – n_3)

n_3 is the lesser of n_1 and n_2.

Testing:

```
{ 0 1 MIN -> 0 }
{ 1 2 MIN -> 1 }
{ -1 0 MIN -> -1 }
{ -1 1 MIN -> -1 }
{ MIN-INT 0 MIN -> MIN-INT }
{ MIN-INT MAX-INT MIN -> MIN-INT }
{ 0 MAX-INT MIN -> 0 }
{ 0 0 MIN -> 0 }
{ 1 1 MIN -> 1 }
{ 1 0 MIN -> 0 }
{ 2 1 MIN -> 1 }
{ 0 -1 MIN -> -1 }
{ 1 -1 MIN -> -1 }
{ 0 MIN-INT MIN -> MIN-INT }
{ MAX-INT MIN-INT MIN -> MIN-INT }
{ MAX-INT 0 MIN -> 0 }
```

6.1.1890 MOD CORE

(n_1 n_2 – n_3)

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

See: **3.2.2.1 Integer division**.

Testing:

```
IFFLOORED   :  TMOD T/MOD DROP ;
IFSYM       :  TMOD T/MOD DROP ;
{ 0 1 MOD -> 0 1 TMOD }
{ 1 1 MOD -> 1 1 TMOD }
{ 2 1 MOD -> 2 1 TMOD }
{ -1 1 MOD -> -1 1 TMOD }
```

```
{ -2 1 MOD -> -2 1 TMOD }
{ 0 -1 MOD -> 0 -1 TMOD }
{ 1 -1 MOD -> 1 -1 TMOD }
{ 2 -1 MOD -> 2 -1 TMOD }
{ -1 -1 MOD -> -1 -1 TMOD }
{ -2 -1 MOD -> -2 -1 TMOD }
{ 2 2 MOD -> 2 2 TMOD }
{ -1 -1 MOD -> -1 -1 TMOD }
{ -2 -2 MOD -> -2 -2 TMOD }
{ 7 3 MOD -> 7 3 TMOD }
{ 7 -3 MOD -> 7 -3 TMOD }
{ -7 3 MOD -> -7 3 TMOD }
{ -7 -3 MOD -> -7 -3 TMOD }
{ MAX-INT 1 MOD -> MAX-INT 1 TMOD }
{ MIN-INT 1 MOD -> MIN-INT 1 TMOD }
{ MAX-INT MAX-INT MOD -> MAX-INT MAX-INT TMOD }
{ MIN-INT MIN-INT MOD -> MIN-INT MIN-INT TMOD }
```

6.1.1900 MOVE CORE

($addr_1$ $addr_2$ u –)

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

See: **17.6.1.0910 CMOVE**, **17.6.1.0920 CMOVE>**.

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

Testing:

```
{ FBUF FBUF 3 CHARS MOVE -> }       \ BIZARRE SPECIAL CASE
{ SEEBUF -> 20 20 20 }

{ SBUF FBUF 0 CHARS MOVE -> }
{ SEEBUF -> 20 20 20 }

{ SBUF FBUF 1 CHARS MOVE -> }
{ SEEBUF -> 12 20 20 }

{ SBUF FBUF 3 CHARS MOVE -> }
{ SEEBUF -> 12 34 56 }

{ FBUF FBUF CHAR+ 2 CHARS MOVE -> }
{ SEEBUF -> 12 12 34 }

{ FBUF CHAR+ FBUF 2 CHARS MOVE -> }
{ SEEBUF -> 12 34 34 }
```

6.1.1910 NEGATE CORE

(n_1 – n_2)

Negate n_1, giving its arithmetic inverse n_2.

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

Testing:
```
{ 0 NEGATE -> 0 }
{ 1 NEGATE -> -1 }
{ -1 NEGATE -> 1 }
{ 2 NEGATE -> -2 }
{ -2 NEGATE -> 2 }
```

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.

Testing:
```
{ 0S 0S OR -> 0S }
{ 0S 1S OR -> 1S }
{ 1S 0S OR -> 1S }
{ 1S 1S OR -> 1S }
```

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.

Testing:
```
{ 1 2 OVER -> 1 2 1 }
```

6.1.2033 POSTPONE CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (“⟨*spaces*⟩*name*” –)

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

See: **3.4.1 Parsing**.

Rationale: 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*.

Testing:
```
{ :  GT4 POSTPONE GT1 ; IMMEDIATE -> }
{ :  GT5 GT4 ; -> }
{ GT5 -> 123 }

{ :  GT6 345 ; IMMEDIATE -> }
{ :  GT7 POSTPONE GT6 ; -> }
{ GT7 -> 345 }
```

6.1.2050 QUIT CORE

(–) (R: *i*×*x* –)

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

- Accept a line from the input source into the input buffer, set **>IN** to zero, and interpret.
- Display the implementation-defined system prompt if in interpretation state, all processing has been completed, and no ambiguous condition exists.

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

Testing: None.

6.1.2060 R> "r-from" CORE

Interpretation: Interpretation semantics for this word are undefined.

Execution: (– *x*) (R: *x* –)

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

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

Testing: See **A.6.1.0580 >R**.

6.1.2070 `R@` "r-fetch" CORE

Interpretation: Interpretation semantics for this word are undefined.

Execution: (– *x*) (R: *x* – *x*)

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

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

Testing: See **A.6.1.0580 >R**.

6.1.2120 `RECURSE` CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (–)

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

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

Rationale: 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**.

Testing:
```
{ :  GI6 ( N – 0,1,..N ) DUP IF DUP >R 1- RECURSE R> THEN ;
-> }
{ 0 GI6 -> 0 }
{ 1 GI6 -> 0 1 }
{ 2 GI6 -> 0 1 2 }
{ 3 GI6 -> 0 1 2 3 }
{ 4 GI6 -> 0 1 2 3 4 }
```

6.1.2140 `REPEAT` CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: *orig dest* –)

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

Run-time: (–)

Continue execution at the location given by *dest*.

See: **6.1.0760 BEGIN**, **6.1.2430 WHILE**.

Rationale: Typical use:

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

Testing: See **A.6.1.2430 WHILE**

6.1.2160 ROT "rote" CORE

(x_1 x_2 x_3 – x_2 x_3 x_1)

Rotate the top three stack entries.

Testing:
```
{ 1 2 3 ROT -> 2 3 1 }
```

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.

Testing:
```
{ 1 0 RSHIFT -> 1 }
{ 1 1 RSHIFT -> 0 }
{ 2 1 RSHIFT -> 1 }
{ 4 2 RSHIFT -> 1 }
{ 8000 F RSHIFT -> 1 }              \ BIGGEST
{ MSB 1 RSHIFT MSB AND -> 0 }       \ RSHIFT ZERO FILLS MSBS
{ MSB 1 RSHIFT 2* -> MSB }
```

6.1.2165 S" "s-quote" CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: ("*ccc*⟨*quote*⟩" –)

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

Run-time: (– *c-addr u*)

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

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

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

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

Testing:
```
{ :  GC4 S" XY" ; -> }
{ GC4 SWAP DROP -> 2 }
{ GC4 DROP DUP C@ SWAP CHAR+ C@ -> 58 59 }
```

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.

Testing:
```
{ 0 S>D -> 0 0 }
{ 1 S>D -> 1 0 }
{ 2 S>D -> 2 0 }
{ -1 S>D -> -1 -1 }
{ -2 S>D -> -2 -1 }
{ MIN-INT S>D -> MIN-INT -1 }
{ MAX-INT S>D -> MAX-INT 0 }
```

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.

Testing:
```
:  GP2 <# -1 SIGN 0 SIGN -1 SIGN 0 0 #> S" -" S= ;
{ GP2 -> <TRUE> }
```

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: **3.2.2.1 Integer division**, **6.1.1561 FM/MOD**, **6.1.2370 UM/MOD**.

Rationale: 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 ;
```

Testing:
```
{ 0 S>D 1 SM/REM -> 0 0 }
{ 1 S>D 1 SM/REM -> 0 1 }
{ 2 S>D 1 SM/REM -> 0 2 }
{ -1 S>D 1 SM/REM -> 0 -1 }
{ -2 S>D 1 SM/REM -> 0 -2 }
{ 0 S>D -1 SM/REM -> 0 0 }
{ 1 S>D -1 SM/REM -> 0 -1 }
{ 2 S>D -1 SM/REM -> 0 -2 }
{ -1 S>D -1 SM/REM -> 0 1 }
{ -2 S>D -1 SM/REM -> 0 2 }
{ 2 S>D 2 SM/REM -> 0 1 }
{ -1 S>D -1 SM/REM -> 0 1 }
{ -2 S>D -2 SM/REM -> 0 1 }
{ 7 S>D 3 SM/REM -> 1 2 }
{ 7 S>D -3 SM/REM -> 1 -2 }
{ -7 S>D 3 SM/REM -> -1 -2 }
{ -7 S>D -3 SM/REM -> -1 2 }
{ MAX-INT S>D 1 SM/REM -> 0 MAX-INT }
{ MIN-INT S>D 1 SM/REM -> 0 MIN-INT }
{ MAX-INT S>D MAX-INT SM/REM -> 0 1 }
{ MIN-INT S>D MIN-INT SM/REM -> 0 1 }
{ 1S 1 4 SM/REM -> 3 MAX-INT }
{ 2 MIN-INT M* 2 SM/REM -> 0 MIN-INT }
{ 2 MIN-INT M* MIN-INT SM/REM -> 0 2 }
{ 2 MAX-INT M* 2 SM/REM -> 0 MAX-INT }
{ 2 MAX-INT M* MAX-INT SM/REM -> 0 2 }
{ MIN-INT MIN-INT M* MIN-INT SM/REM -> 0 MIN-INT }
{ MIN-INT MAX-INT M* MIN-INT SM/REM -> 0 MAX-INT }
{ MIN-INT MAX-INT M* MAX-INT SM/REM -> 0 MIN-INT }
{ MAX-INT MAX-INT M* MAX-INT SM/REM -> 0 MAX-INT }
```

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.

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

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

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

Testing:
```
: GS1 S" SOURCE" 2DUP EVALUATE >R SWAP >R = R> R> = ;
{ GS1 -> <TRUE> <TRUE> }

: GS4 SOURCE >IN ! DROP ;
{ GS4 123 456
-> }
```

6.1.2220 SPACE CORE

(–)

Display one space.

Testing: See **A.6.1.1320 EMIT**.

6.1.2230 SPACES CORE

(*n* –)

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

Testing: See **A.6.1.1320 EMIT**.

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), and **]** (right-bracket).

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

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

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

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

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

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

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

```
: NOP : POSTPONE ; IMMEDIATE ;

NOP ALIGN
NOP ALIGNED
```

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

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

Testing:
```
{ :  GT8 STATE @ ; IMMEDIATE -> }
{ GT8 -> 0 }
{ :  GT9 GT8 LITERAL ; -> }
{ GT9 0= -> <FALSE> }
```

6.1.2260 SWAP CORE

(x_1 x_2 – x_2 x_1)

Exchange the top two stack items.

Testing:
```
{ 1 2 SWAP -> 2 1 }
```

6.1.2270 THEN CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: *orig* –)

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

Run-time: (–)

Continue execution.

See: **6.1.1310 ELSE**, **6.1.1700 IF**.

Rationale: Typical use:

: X ... *test* **IF** ... **THEN** ... ;

or

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

Testing: See **A.6.1.1700 IF**.

6.1.2310 TYPE CORE

(*c-addr u* –)

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

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

See: **6.1.1320 EMIT**.

Testing: See **A.6.1.1320 EMIT**.

6.1.2320 `U.` "u-dot" CORE

(*u* –)

Display *u* in free field format.

Testing: See **A.6.1.1320 EMIT**.

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: **6.1.0480 <**.

Testing:
```
{ 0 1 U< -> <TRUE> }
{ 1 2 U< -> <TRUE> }
{ 0 MID-UINT U< -> <TRUE> }
{ 0 MAX-UINT U< -> <TRUE> }
{ MID-UINT MAX-UINT U< -> <TRUE> }
{ 0 0 U< -> <FALSE> }
{ 1 1 U< -> <FALSE> }
{ 1 0 U< -> <FALSE> }
{ 2 1 U< -> <FALSE> }
{ MID-UINT 0 U< -> <FALSE> }
{ MAX-UINT 0 U< -> <FALSE> }
{ MAX-UINT MID-UINT U< -> <FALSE> }
```

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.

Testing:
```
{ 0 0 UM* -> 0 0 }
{ 0 1 UM* -> 0 0 }
{ 1 0 UM* -> 0 0 }
{ 1 2 UM* -> 2 0 }
{ 2 1 UM* -> 2 0 }
{ 3 3 UM* -> 9 0 }
{ MID-UINT+1 1 RSHIFT 2 UM* -> MID-UINT+1 0 }
{ MID-UINT+1 2 UM* -> 0 1 }
{ MID-UINT+1 4 UM* -> 0 2 }
{ 1S 2 UM* -> 1S 1 LSHIFT 1 }
{ MAX-UINT MAX-UINT UM* -> 1 1 INVERT }
```

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

Testing:
```
{ 0 0 1 UM/MOD -> 0 0 }
{ 1 0 1 UM/MOD -> 0 1 }
{ 1 0 2 UM/MOD -> 1 0 }
{ 3 0 2 UM/MOD -> 1 1 }
{ MAX-UINT 2 UM* 2 UM/MOD -> 0 MAX-UINT }
{ MAX-UINT 2 UM* MAX-UINT UM/MOD -> 0 2 }
{ MAX-UINT MAX-UINT UM* MAX-UINT UM/MOD -> 0 MAX-UINT }
```

6.1.2380 UNLOOP CORE

Interpretation: Interpretation semantics for this word are undefined.

Execution: (–) (R: *loop-sys* –)

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

See: **3.2.3.3 Return stack**.

Rationale: 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.

Testing:
```
{ : 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 ; -> }
{ 1 GD6 -> 1 }
{ 2 GD6 -> 3 }
{ 3 GD6 -> 4 1 2 }
```

6.1.2390 UNTIL CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: *dest* –)

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

Run-time: (*x* –)

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

See: **6.1.0760 BEGIN**.

Rationale: Typical use: **:** X ... **BEGIN** ... *test* **UNTIL** ... **;**

Testing:
```
{ :  GI4 BEGIN DUP 1+ DUP 5 > UNTIL ; -> }
{ 3 GI4 -> 3 4 5 6 }
{ 5 GI4 -> 5 6 }
{ 6 GI4 -> 6 7 }
```

6.1.2410 VARIABLE CORE

(“⟨*spaces*⟩*name*” –)

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

name is referred to as a “variable”.

name Execution: (– *a-addr*)

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

See: **3.4.1 Parsing**.

Rationale: Typical use: ... **VARIABLE** XYZ ...

Testing:
```
{ VARIABLE V1 -> }
{ 123 V1 !  -> }
{ V1 @ -> 123 }
```

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*.

Rationale: Typical use: **:** X ... **BEGIN** ... *test* **WHILE** ... **REPEAT** ... **;**

Testing:
```
{ :  GI3 BEGIN DUP 5 < WHILE DUP 1+ REPEAT ; -> }
{ 0 GI3 -> 0 1 2 3 4 5 }
{ 4 GI3 -> 4 5 }
{ 5 GI3 -> 5 }
{ 6 GI3 -> 6 }
{ :  GI5 BEGIN DUP 2 > WHILE
    DUP 5 < WHILE DUP 1+ REPEAT
  123 ELSE 345 THEN ; -> }
{ 1 GI5 -> 1 345 }
```

```
{ 2 GI5 -> 2 345 }
{ 3 GI5 -> 3 4 5 123 }
{ 4 GI5 -> 4 5 123 }
{ 5 GI5 -> 5 123 }
```

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 space, not included in the length, follows the string. A program may replace characters within the string.

Note: The requirement to follow the string with a space is obsolescent and is included as a concession to existing programs that use **CONVERT**. A program shall not depend on the existence of the space.

See: **3.3.3.6 Other transient regions**, **3.4.1 Parsing**.

Rationale: Typical use: *char* **WORD** *ccc*⟨*char*⟩

Testing:
```
: GS3 WORD COUNT SWAP C@ ;
{ BL GS3 HELLO -> 5 CHAR H }
{ CHAR " GS3 GOODBYE" -> 7 CHAR G }
{ BL GS3
DROP -> 0 }          \ BLANK LINE RETURN ZERO-LENGTH STRING
```

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.

Testing:
```
{ 0S 0S XOR -> 0S }
{ 0S 1S XOR -> 1S }
{ 1S 0S XOR -> 1S }
{ 1S 1S XOR -> 0S }
```

6.1.2500 ["left-bracket" CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: Perform the execution semantics given below.

Execution: (–)

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

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

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

Testing:
```
{ :  GC3 [ GC1 ] LITERAL ; -> }
{ GC3 -> 58 }
```

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: **3.4.1 Parsing**, **A.6.1.0070 ' A.6.1.2033 POSTPONE**, **D.6.7 Immediacy**.

Rationale: Typical use: **:** X ... **[']** *name* ... **;**

See: **A.6.1.1550 FIND**.

Testing:
```
{ :  GT2 ['] GT1 ; IMMEDIATE -> }
{ GT2 EXECUTE -> 123 }
```

6.1.2520 [CHAR] "bracket-char" CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (*"⟨spaces⟩name"* –)

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

Run-time: (– *char*)

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

See: **3.4.1 Parsing**, **6.1.0895 CHAR**.

Rationale: Typical use: **:** X ... **[CHAR]** *c* ... **;**

Testing:
```
{ :  GC1 [CHAR] X ; -> }
{ :  GC2 [CHAR] HELLO ; -> }
{ GC1 -> 58 }
{ GC2 -> 48 }
```

6.1.2540] "right-bracket" CORE

(–)

Enter compilation state.

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

Rationale: Typical use: **:** X ... **[** 4321 **]** **LITERAL** ... **;**

Testing: See **A.6.1.2500 [**

6.2 Core extension words

6.2.0060 `#TIB` "number-t-i-b" CORE EXT

(– *a-addr*)

a-addr is the address of a cell containing the number of characters in the terminal input buffer.

Note: This word is obsolescent and is included as a concession to existing implementations.

Rationale: The function of **#TIB** has been superseded by **SOURCE**.

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

Compilation: Perform the execution semantics given below.

Execution: ("*ccc*⟨*paren*⟩" –)

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

See: **3.4.1 Parsing**, **6.1.0190 ."**.

Rationale: Typical use: **.(** *ccc*)

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.

Rationale: In **.R**, "R" is short for RIGHT.

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

(*x* – *flag*)

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

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

(*n* – *flag*)

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

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

Interpretation: Interpretation semantics for this word are undefined.

Execution: (x_1 x_2 –) (R: – x_1 x_2)

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

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

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

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

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>** wordSWAP.

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

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

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

Interpretation: Interpretation semantics for this word are undefined.

Execution: (– x_1 x_2) (R: x_1 x_2 – x_1 x_2)

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

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

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

(C: – *colon-sys*) (S: – *xt*)

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

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

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

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

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

xt Execution: ($i \times x$ – $j \times x$)

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

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

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

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

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

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

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

```
:NONAME ...  code for command 0 ...  ; 0 CMD !
:NONAME ...  code for command 1 ...  ; 1 CMD !
  ...
:NONAME ...  code for command 99 ...  ; 99 CMD !
...  5 CMD @ EXECUTE ...
```

The definitions of the table building words are:

```
CREATE CMD-TABLE \ table for command execution tokens
  100 CELLS ALLOT
:  CMD ( n - a-addr ) \ nth element address in table
  CELLS CMD-TABLE + ;
```

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

```
:  DOCOLON ( - )
\ Modify CREATEd word to execute like a colon def
  DOES> ( i*x a-addr - j*x )
    1 OVER +!              \ count executions
    CELL+ @ EXECUTE        \ execute :NONAME definition
;
:  OLD: :  ;               \ just an alias
OLD: :  ( "name" - a-addr xt colon-sys )
\ begins an execution-counting colon definition
  CREATE HERE 0 ,          \ storage for execution counter
  0 ,                      \ storage for execution token
  DOCOLON                  \ set run time for CREATEd word
  :NONAME                  \ begin unnamed colon definition
;
```

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

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

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

... **:** xxx ... **;** ... xxx ...

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

... **'** xxx **>BODY ?** ...

6.2.0500 <> "not-equals" CORE EXT

(x_1 x_2 – *flag*)

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

6.2.0620 ?DO "question-do" CORE EXT

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: – *do-sys*)

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

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

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

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

Rationale: Typical use:

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

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

6.2.—— ACTION-OF CORE EXT
X:deferred

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

Skip leading spaces and parse name delimited by a space. *xt* is the *xt* associated with *name*. An ambiguous condition exists if name was not defined by **DEFER**, or if the name has not been associated with an *xt* yet.

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 associated with *name* when the run-time semantics is performed. An ambiguous condition exists if *name* has not been associated with an *xt* yet.

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

See: **6.2.0 DEFER**, **6.2.0 DEFER!**, **6.2.0 DEFER@** and **6.2.0 IS**.

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

Testing:
```
{ DEFER defer1 -> }
{ : action-defer1 ACTION-OF defer1 ; -> }
{ ' * ' defer1 DEFER! -> }
{ 2 3 defer1 -> 6 }
{ ACTION-OF defer1 -> ' * }
{ action-defer1 -> ' * }
{ ' + IS defer1 -> }
{ 1 2 defer1 -> 3 }
{ ACTION-OF defer1 -> ' + }
{ action-defer1 -> ' + }
```

6.2.0700 AGAIN CORE EXT

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: *dest* –)

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

Run-time: (–)

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

See: **6.1.0760 BEGIN**.

Rationale: Typical use: **:** X ... **BEGIN** ... **AGAIN** ... **;**

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

6.2.0855 C" "c-quote" CORE EXT

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (*"ccc⟨quote⟩"* –)

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

Run-time: (– *c-addr*)

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

See: **3.4.1 Parsing**, **6.1.2165 S"**, **11.6.1.2165 S"**.

Rationale: Typical use: **:** X ... **C"** *ccc*" ... **;**

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

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

See: **A.3.1.3.4 Counted strings**.

6.2.0873 CASE CORE EXT

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: – *case-sys*)

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

Run-time: (–)

Continue execution.

See: **6.2.1342 ENDCASE**, **6.2.1343 ENDOF**, **6.2.1950 OF**.

Rationale: Typical use:

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

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.

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

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

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

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

6.2.0970 CONVERT CORE EXT

(ud_1 *c-addr*$_1$ – ud_2 *c-addr*$_2$)

ud_2 is the result of converting the characters within the text beginning at the first character after *c-addr*$_1$ into digits, using the number in **BASE**, and adding each digit to ud_1 after multiplying ud_1 by the number in **BASE**. Conversion continues until a character that is not convertible is encountered. *c-addr*$_2$ is the location of the first unconverted character. An ambiguous condition exists if ud_2 overflows.

Note: This word is obsolescent and is included as a concession to existing implementations. Its function is superseded by **6.1.0570 >NUMBER**.

See: **3.2.1.2 Digit conversion**.

Rationale: **CONVERT** may be defined as follows:

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

6.2.—— DEFER CORE EXT
X:deferred

("⟨*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 *xt* associated with name. An ambiguous condition exists if *name* has not been associated with an *xt* yet.

See: **6.2.0 ACTION-OF**, **6.2.0 DEFER!**, **6.2.0 DEFER@**, and **6.2.0 IS**.

Implementation:
```
:  DEFER ( "name" – )
  CREATE ['] ABORT ,
DOES> ( ...  – ...  )
  @ EXECUTE ;
```

Testing:
```
{ DEFER defer2 -> }
{ ' * ' defer2 DEFER! -> }
{ 2 3 defer2 -> 6 }
{ ' + IS defer2 -> }
{ 1 2 defer2 -> 3 }
```

6.2.—— DEFER! "defer-store" CORE EXT X:deferred

(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 via **DEFER**.

See: **6.2.0 ACTION-OF**, **6.2.0 DEFER**, **6.2.0 DEFER@**, and **6.2.0 IS**.

Implementation:
```
:  DEFER! ( xt2 xt1 – )
  >BODY ! ;
```

Testing:
```
{ DEFER defer3 -> }
{ ' * ' defer3 DEFER! -> }
{ 2 3 defer3 -> 6 }
{ ' + ' defer3 DEFER! -> }
{ 1 2 defer3 -> 3 }
```

6.2.—— DEFER@ "defer-fetch" CORE EXT X:deferred

(xt_1 – xt_2)

xt_2 is the *xt* associated with the deferred word corresponding to xt_1. An ambiguous condition exists if xt_1 is not for a word defined via **DEFER**, or if the deferred word has not been associated with an *xt* yet.

See: **6.2.0 ACTION-OF**, **6.2.0 DEFER**, **6.2.0 DEFER!**, and **6.2.0 IS**.

Implementation:
```
:  DEFER@ ( xt1 – xt2 )
  >BODY @ ;
```

Testing:
```
{ DEFER defer4 -> }
{ ' * ' defer4 DEFER! -> }
{ 2 3 defer4 -> 6 }
{ ' defer4 DEFER@ -> ' * }
{ ' + IS defer4 -> }
{ 1 2 defer4 -> 3 }
{ ' defer4 DEFER@ -> ' + }
```

6.2.1342 ENDCASE "end-case" CORE EXT

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: *case-sys* –)

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

Run-time: (*x* –)

Discard the case selector *x* and continue execution.

See: **6.2.0873 CASE**, **6.2.1343 ENDOF**, **6.2.1950 OF**.

Rationale: Typical use:

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

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: **6.2.0873 CASE**, **6.2.1342 ENDCASE**, **6.2.1950 OF**.

Rationale: Typical use:

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

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*.

6.2.1390 EXPECT CORE EXT

(*c-addr +n* –)

Receive a string of at most *+n* characters. 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 of characters are implementation-defined.

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

Store the string at *c-addr* and its length in **SPAN**.

Note: This word is obsolescent and is included as a concession to existing implementations. Its function is superseded by **6.1.0695 ACCEPT**.

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

6.2.1485 FALSE CORE EXT

(– *false*)

Return a *false* flag.

See: **3.1.3.1 Flags**

6.2.1660 HEX CORE EXT

(–)

Set contents of **BASE** to sixteen.

Testing: See **A.6.1.0750 BASE**.

6.2.—— IS CORE EXT
X:deferred

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

Skip leading spaces and parse name delimited by a space. Set *name* to execute *xt*. An ambiguous condition exists if *name* was not defined by **DEFER**.

Compilation: ("⟨*spaces*⟩*name*" –)

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

Run-time: (*xt* –)

Set *name* to execute *xt*.

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

See: **6.2.0 ACTION-OF**, **6.2.0 DEFER**, **6.2.0 DEFER!**, and **6.2.0 DEFER@**.

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

Testing:
```
{ DEFER defer5 -> }
{ :  is-defer5 IS defer5 ; -> }
{ ' * IS defer5 -> }
{ 2 3 defer5 -> 6 }
{ ' + is-defer5 -> }
{ 1 2 defer5 -> 3 }
```

6.2.1850 MARKER CORE EXT

("⟨*spaces*⟩*name*" –)

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

name Execution: (–)

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

See: **3.4.1 Parsing**, **15.6.2.1580 FORGET**.

Rationale: As dictionary implementations have ~~gotten~~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.

6.2.1930 NIP CORE EXT

(x_1 x_2 – x_2)

Drop the first item below the top of stack.

6.2.1950 OF CORE EXT

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: – *of-sys*)

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

Run-time: (x_1 x_2 – x_1)

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

See: **6.2.0873 CASE**, **6.2.1342 ENDCASE**, **6.2.1343 ENDOF**.

Rationale: Typical use:

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

6.2.2000 PAD CORE EXT

(– *c-addr*)

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

See: **3.3.3.6 Other transient regions**.

Rationale: **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.

6.2.2008 PARSE CORE EXT

(*char* “*ccc*⟨*char*⟩” – *c-addr u*)

Parse *ccc* delimited by the delimiter *char*.

c-addr is the address (within the input buffer) and *u* is the length of the parsed string. If the parse area was empty, the resulting string has a zero length.

See: **3.4.1 Parsing**.

Rationale: Typical use: *char* **PARSE** *ccc*⟨*char*⟩

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

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

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

This works fine when used in a line like:

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

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

```
.( )  5 .
```

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

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

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

This definition avoids the "empty string" anomaly.

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

1) The characters accepted by **WORD** must be copied from the input buffer into a temporary buffer, in order to make room for the count character that must be at the beginning of the counted string. The copy step is inefficient, compared to **PARSE**, which leaves the string in the input buffer and doesn't need to copy it anywhere.
2) **WORD** must be careful not to store too many characters into the temporary buffer, thus overwriting something beyond the end of the buffer. This adds to the overhead of the copy step. (**WORD** may have to scan a lot of characters before finding the trailing delimiter.)
3) The count character limits the length of the string returned by **WORD** to 255 characters (longer strings can easily be stored in blocks!). This limitation does not exist for **PARSE**.
4) The temporary buffer is typically overwritten by the next use of **WORD**. This introduces a temporal dependency; the value returned by **WORD** is only valid for a limited duration. **PARSE** has a temporal dependency, too, related to the lifetime of the input buffer, but that is less severe in most cases than **WORD**'s temporal dependency.

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

PARSE-WORD ("⟨*spaces*⟩*name*" – *c-addr u*)

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

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

6.2.—— PARSE-NAME CORE EXT X:parse-name

("⟨*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.

Implementation:

```
: 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 - ;
```

Testing:

```
{ PARSE-NAME abcd S" abcd" S= -> <TRUE> }
{ PARSE-NAME ␣␣abcde␣␣ S" abcde" S= -> <TRUE> }
\ test empty parse area
{ PARSE-NAME
  NIP -> 0 } \ empty line
{ PARSE-NAME ␣␣
  NIP -> 0 } \ line with white space
{ : parse-name-test ( "name1" "name2" - n )
  PARSE-NAME PARSE-NAME S= ; -> }
{ parse-name-test abcd abcd -> <TRUE> }
{ parse-name-test ␣abcd ␣␣abcd␣␣ -> <TRUE> }
{ parse-name-test abcde abcdf -> <FALSE> }
{ parse-name-test abcdf abcde -> <FALSE> }
{ parse-name-test abcde abcde
  -> <TRUE> }
{ parse-name-test abcde abcde␣␣
  -> <TRUE> }
```

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.

Rationale: `0` **PICK** is equivalent to **DUP** and `1` **PICK** is equivalent to **OVER**.

6.2.2040 QUERY CORE EXT

(-)

Make the user input device the input source. Receive input into the terminal input buffer, replacing any previous contents. Make the result, whose address is returned by **TIB**, the input buffer. Set **>IN** to zero.

Note: This word is obsolescent and is included as a concession to existing implementations.

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

6.2.2125 REFILL CORE EXT

(– *flag*)

Attempt to fill the input buffer from the input source, returning a true flag if successful.

When the input source is the user input device, attempt to receive input into the terminal input buffer. If successful, make the result the input buffer, set **>IN** to zero, and return *true*. Receipt of a line containing no characters is considered successful. If there is no input available from the current input source, return *false*.

When the input source is a string from **EVALUATE**, return *false* and perform no other action.

See: **7.6.2.2125 REFILL**, **11.6.2.2125 REFILL**.

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

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: **A.6.2.2182 SAVE-INPUT**.

6.2.2150 ROLL CORE EXT

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

Remove u. Rotate u+1 items on the top of the stack. An ambiguous condition exists if there are less than u+2 items on the stack before **ROLL** is executed.

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

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**.

Rationale: **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**.

6.2.2218 SOURCE-ID "source-i-d" CORE EXT

(- *0* | *-1*)

Identifies the input source as follows:

SOURCE-ID	Input source
-1	String (via **EVALUATE**)
0	User input device

See: **11.6.1.2218 SOURCE-ID**.

6.2.2240 SPAN CORE EXT

(- *a-addr*)

a-addr is the address of a cell containing the count of characters stored by the last execution of **EXPECT**.

Note: This word is obsolescent and is included as a concession to existing implementations.

6.2.2290 TIB "t-i-b" CORE EXT

(- *c-addr*)

c-addr is the address of the terminal input buffer.

Note: This word is obsolescent and is included as a concession to existing implementations.

Rationale: The function of **TIB** has been superseded by **SOURCE**.

6.2.2295 TO CORE EXT

x:to

Interpretation: (*i*× *x* "⟨*spaces*⟩*name*" –)

x:to

Skip leading spaces and parse *name* delimited by a space. ~~Store *x* in *name*.~~ 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 **VALUE**~~ by a word with "TO *name* run-time" semantics .

Compilation: ("⟨*spaces*⟩*name*" –)

Skip leading spaces and parse *name* delimited by a space. Append the run-time semantics given ~~below~~in the definition for the defining word of *name* to the current definition. An ambiguous condition exists if *name* was not defined ~~by **VALUE**~~by a word with "TO *name* run-time" semantics .

Run-time: (~~*x*~~ –)

~~Store *x* in *name*.~~

Note: An ambiguous condition exists if ~~either~~any of **POSTPONE** ~~or~~, **[COMPILE]** ~~is~~, **'** , or **[']** are applied to **TO**.

See: **6.2.2405 VALUE**, ~~**13.6.1.2295** TO~~ and **13.6.1.0086** (LOCAL) .

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

Typical use: `x` **TO** *name*

Testing: See **6.2.2405** VALUE.

6.2.2298 TRUE CORE EXT

(– *true*)

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

See: **3.1.3.1 Flags**.

Rationale: **TRUE** is equivalent to the phrase `0` **0=**.

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: **6.1.0540 >**.

6.2.2395 UNUSED CORE EXT

(– *u*)

u is the amount of space remaining in the region addressed by **HERE**, in address units.

6.2.2405 VALUE CORE EXT

(*x* "⟨*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 associated with *name*.

TO *name* Run-time: (*x* –) x:to

Associate the value *x* with *name*. x:to

See: **3.4.1 Parsing** and **6.2.2295** TO .

Rationale: 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. x:to

Testing:

```
{ 0 VALUE Tval -> }
{ Tval -> 0 }

{ 1 TO Tval -> }
{ Tval -> 1 }

:  SETTval Tval SWAP TO Tval ;
{ 2 SETTval Tval -> 1 2 }
```

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.

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

u————————————————————*o*

There are two cases to consider: either the n_2 | u_2…n_3 | u_3 range straddles the overflow/underflow points or it does not. Lets examine the non-straddle case first:

u————————[. . . . .)————————*o*

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

$$n_2 \mid u_2 \leq n_1 \mid u_1 \text{ and } n_1 \mid u_1 < n_3 \mid u_3.$$

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

u . . . . .)————————[. . . . . *o*

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

$$n_2 \mid u_2 \leq n_1 \mid u_1 \text{ or } n_1 \mid u_1 < n_3 \mid u_3.$$

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

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

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

```
33000   32000   34000   WITHIN
```

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

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

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

```
1   -5   5   WITHIN
```

would give an incorrect answer.

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

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

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.

See: **3.4.1 Parsing**.

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

6.2.2535 `\` "backslash" CORE EXT

Compilation: Perform the execution semantics given below.

Execution: ("*ccc*⟨*eol*⟩" -)

Parse and discard the remainder of the parse area. **** is an immediate word.

See: **7.6.2.2535 **.

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

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 Environmental queries

Append table 7.1 to table 3.5.

See: **3.2.6 Environmental queries**.

Table 7.1: Environmental Query Strings

String	Value data type	Constant?	Meaning
BLOCK	*flag*	no	block word set present
BLOCK-EXT	*flag*	no	block extensions word set present

7.3.2 Data space

A program may access memory within a valid block buffer.

See: **3.3.3 Data space**.

7.3.3 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.

7.3.4 Parsing

The Block word set implements an alternative input source for the text interpreter. When the input source is a block, **BLK** shall contain the non-zero block number and the input buffer is the 1024-character buffer containing that block.

A block is conventionally displayed as 16 lines of 64 characters.

A program may switch the input source to a block by using **LOAD** or **THRU**. Input sources may be nested using **LOAD** and **EVALUATE** in any order.

A program may reposition the parse area within a block by manipulating **>IN**. More extensive repositioning can be accomplished using **SAVE-INPUT** and **RESTORE-INPUT**.

See: **3.4.1 Parsing**.

7.3.5 Possible action on an ambiguous condition

See: **3.4.4 Possible actions on an ambiguous condition**.

- A system with the Block word set may set interpretation state and interpret a block.

7.4 Additional documentation requirements

7.4.1 System documentation

7.4.1.1 Implementation-defined options

- the format used for display by **7.6.2.1770 LIST** (if implemented);
- the length of a line affected by **7.6.2.2535 ** (if implemented).

7.4.1.2 Ambiguous conditions

- Correct block read was not possible;
- I/O exception in block transfer;
- Invalid block number (**7.6.1.0800 BLOCK**, **7.6.1.0820 BUFFER**, **7.6.1.1790 LOAD**);
- A program directly alters the contents of **7.6.1.0790 BLK**;
- No current block buffer for **7.6.1.2400 UPDATE**.

7.4.1.3 Other system documentation

- any restrictions a multiprogramming system places on the use of buffer addresses;
- the number of blocks available for source text and data.

7.4.2 Program documentation

- the number of blocks required by the program.

7.5 Compliance and labeling

7.5.1 ANS Forth 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 ANS Forth programs

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

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

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

7.6 Glossary

7.6.1 Block words

7.6.1.0790 `BLK` "b-l-k" BLOCK

(– *a-addr*)

a-addr is the address of a cell containing zero or the number of the mass-storage block being interpreted. If **BLK** contains zero, the input source is not a block and can be identified by **SOURCE-ID**, if **SOURCE-ID** is available. An ambiguous condition exists if a program directly alters the contents of **BLK**.

See: **7.3.3 Block buffer regions**.

7.6.1.0800 `BLOCK` BLOCK

(*u* – *a-addr*)

a-addr is the address of the first character of the block buffer assigned to mass-storage block *u*. An ambiguous condition exists if *u* is not an available block number.

If block *u* is already in a block buffer, *a-addr* is the address of that block buffer.

If block *u* is not already in memory and there is an unassigned block buffer, transfer block *u* from mass storage to an unassigned block buffer. *a-addr* is the address of that block buffer.

If block *u* is not already in memory and there are no unassigned block buffers, unassign a block buffer. If the block in that buffer has been **UPDATE**d, transfer the block to mass storage and transfer block *u* from mass storage into that buffer. *a-addr* is the address of that block buffer.

At the conclusion of the operation, the block buffer pointed to by *a-addr* is the current block buffer and is assigned to *u*.

7.6.1.0820 BUFFER BLOCK

(*u* – *a-addr*)

a-addr is the address of the first character of the block buffer assigned to block *u*. The contents of the block are unspecified. An ambiguous condition exists if *u* is not an available block number.

If block *u* is already in a block buffer, *a-addr* is the address of that block buffer.

If block *u* is not already in memory and there is an unassigned buffer, *a-addr* is the address of that block buffer.

If block *u* is not already in memory and there are no unassigned block buffers, unassign a block buffer. If the block in that buffer has been **UPDATE**d, transfer the block to mass storage. *a-addr* is the address of that block buffer.

At the conclusion of the operation, the block buffer pointed to by *a-addr* is the current block buffer and is assigned to *u*.

See: **7.6.1.0800 BLOCK**.

7.6.1.1360 EVALUATE BLOCK

Extend the semantics of **6.1.1360 EVALUATE** to include: Store zero in **BLK**.

7.6.1.1559 FLUSH BLOCK

(–)

Perform the function of **SAVE-BUFFERS**, then unassign all block buffers.

7.6.1.1790 LOAD BLOCK

(*i*×*x u* – *j*×*x*)

Save the current input-source specification. Store *u* in **BLK** (thus making block *u* the input source and setting the input buffer to encompass its contents), set **>IN** to zero, and interpret. When the parse area is exhausted, restore the prior input source specification. Other stack effects are due to the words **LOAD**ed.

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

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

7.6.1.2180 SAVE-BUFFERS BLOCK

(–)

Transfer the contents of each **UPDATE**d block buffer to mass storage. Mark all buffers as unmodified.

7.6.1.2400 UPDATE BLOCK

(–)

Mark the current block buffer as modified. An ambiguous condition exists if there is no current block buffer.

UPDATE does not immediately cause I/O.

See: **7.6.1.0800 BLOCK**, **7.6.1.0820 BUFFER**, **7.6.1.1559 FLUSH**, **7.6.1.2180 SAVE-BUFFERS**.

7.6.2 Block extension words

7.6.2.1330 EMPTY-BUFFERS BLOCK EXT

(–)

Unassign all block buffers. Do not transfer the contents of any **UPDATE**d block buffer to mass storage.

See: **7.6.1.0800 BLOCK**.

7.6.2.1770 LIST BLOCK EXT

(*u* –)

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

See: **7.6.1.0800 BLOCK**.

7.6.2.2125 REFILL BLOCK EXT

(– *flag*)

Extend the execution semantics of **6.2.2125 REFILL** with the following:

When the input source is a block, make the next block the input source and current input buffer by adding one to the value of **BLK** and setting **>IN** to zero. Return *true* if the new value of BLK is a valid block number, otherwise *false*.

See: **6.2.2125 REFILL**, **11.6.2.2125 REFILL**.

7.6.2.2190 SCR "s-c-r" BLOCK EXT

(– *a-addr*)

a-addr is the address of a cell containing the block number of the block most recently **LIST**ed.

Rationale: **SCR** is short for screen.

7.6.2.2280 THRU BLOCK EXT

(*i*×*x* u_1 u_2 – *j*×*x*)

LOAD the mass storage blocks numbered u_1 through u_2 in sequence. Other stack effects are due to the words **LOAD**ed.

7.6.2.2535 \ "backslash" BLOCK EXT

Extend the semantics of **6.2.2535 ** to be:

Compilation: Perform the execution semantics given below.

Execution: (*"ccc⟨eol⟩"* –)

If **BLK** contains zero, parse and discard the remainder of the parse area; otherwise parse and discard the portion of the parse area corresponding to the remainder of the current line. **** is an immediate word.

8 The optional Double-Number word set

8.1 Introduction

Sixteen-bit Forth systems often use double-length numbers. However, many Forths on small embedded systems do not, and many users of Forth on systems with a cell size of 32 bits or more find that the use of double-length numbers is much diminished. Therefore, the words that manipulate double-length entities have been placed in this optional word set.

8.2 Additional terms and notation

None.

8.3 Additional usage requirements

8.3.1 Environmental queries

Append table 8.1 to table 3.5.

See: **3.2.6 Environmental queries**.

Table 8.1: Environmental Query Strings

String	Value data type	Constant?	Meaning
`DOUBLE`	*flag*	no	double-number word set present
`DOUBLE-EXT`	*flag*	no	double-number extensions word set present

8.3.2 Text interpreter input number conversion

When the text interpreter processes a number that is immediately followed by a decimal point and is not found as a definition name, the text interpreter shall convert it to a double-cell number.

For example, entering **`DECIMAL`** `1234` leaves the single-cell number `1234` on the stack, and entering **`DECIMAL`** `1234.` leaves the double-cell number `1234 0` on the stack.

See: **3.4.1.3 Text interpreter input number conversion**.

8.4 Additional documentation requirements

8.4.1 System documentation

8.4.1.1 Implementation-defined options

– no additional requirements.

8.4.1.2 Ambiguous conditions

– *d* outside range of *n* in **8.6.1.1140 `D>S`**.

8.4.1.3 Other system documentation

– no additional requirements.

8.4.2 Program documentation

– no additional requirements.

8.5 Compliance and labeling

8.5.1 ANS Forth 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 ANS Forth 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: **3.4.1 Parsing**.

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

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.

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

8.6.1.0440 `2VARIABLE` "two-variable" DOUBLE

("⟨spaces⟩*name*" –)

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Reserve two consecutive cells of data space.

name is referred to as a "two-variable".

name Execution: (– *a-addr*)

a-addr is the address of the first (lowest address) cell of two consecutive cells in data space reserved by **2VARIABLE** when it defined *name*. A program is responsible for initializing the contents.

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

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

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.

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.

8.6.1.1060 `D.` "d-dot" DOUBLE

(*d* –)

Display *d* in free field format.

8.6.1.1070 `D.R` "d-dot-r" DOUBLE

(*d n* –)

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

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

8.6.1.1075 D0< "d-zero-less" DOUBLE

(*d* – *flag*)

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

8.6.1.1080 D0= "d-zero-equals" DOUBLE

(*xd* – *flag*)

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

8.6.1.1090 D2* "d-two-star" DOUBLE

(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.

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

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.

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

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.

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.

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.

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

8.6.1.1160 DABS "d-abs" DOUBLE

(*d* – *ud*)
ud is the absolute value of *d*.

8.6.1.1210 DMAX "d-max" DOUBLE

(d_1 d_2 – d_3)
d_3 is the greater of d_1 and d_2.

8.6.1.1220 DMIN "d-min" DOUBLE

(d_1 d_2 – d_3)
d_3 is the lesser of d_1 and d_2.

8.6.1.1230 DNEGATE "d-negate" DOUBLE

(d_1 – d_2)
d_2 is the negation of d_1.

8.6.1.1820 M*/ "m-star-slash" DOUBLE

(d_1 n_1 $+n_2$ – d_2)
Multiply d_1 by n_1 producing the triple-cell intermediate result t. Divide t by $+n_2$ giving the double-cell quotient d_2. An ambiguous condition exists if $+n_2$ is zero or negative, or the quotient lies outside of the range of a double-precision signed integer.

Rationale: **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.

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.

Rationale: **M+** is the classical method for integrating.

8.6.2 Double-Number extension words

8.6.2.0420 2ROT "two-rote" DOUBLE EXT

(x_1 x_2 x_3 x_4 x_5 x_6 – x_3 x_4 x_5 x_6 x_1 x_2)
Rotate the top three cell pairs on the stack bringing cell pair x_1 x_2 to the top of the stack.

8.6.2.1270 DU< "d-u-less" DOUBLE EXT

(ud_1 ud_2 − *flag*)

flag is true if and only if ud_1 is less than ud_2.

9 The optional Exception word set

9.1 Introduction

9.2 Additional terms and notation

None.

9.3 Additional usage requirements

9.3.1 THROW values

The **THROW** values {-255...-1} shall be used only as assigned by this Standard. The values {-4095...-256} shall be used only as assigned by a system.

If the File-Access or Memory-Allocation word sets are implemented, it is recommended that the non-zero values of *ior* lie within the range of system **THROW** values, as defined above. In an operating-system environment, this can sometimes be accomplished by "biasing" the range of operating-system exception codes to fall within the **THROW** range.

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 Environmental queries

Append table 9.1 to table 3.5.

See: **3.2.6 Environmental queries**.

Table 9.1: Environmental Query Strings

String	Value data type	Constant?	Meaning
EXCEPTION	*flag*	no	Exception word set present
EXCEPTION-EXT	*flag*	no	Exception extensions word set present

9.3.5 Possible actions on an ambiguous condition

A system choosing to execute **THROW** when detecting one of the ambiguous conditions listed in table 9.2 shall use the throw code listed there.

See: **3.4.4 Possible actions on an ambiguous condition**.

Table 9.2: **THROW** code assignments

Code	Reserved for	Code	Reserved for
-1	**ABORT**	-30	obsolescent feature
-2	**ABORT"**	-31	**>BODY** used on non-**CREATE**d definition
-3	stack overflow	-32	invalid name argument (e.g., **TO** xxx)
-4	stack underflow	-33	block read exception
-5	return stack overflow	-34	block write exception
-6	return stack underflow	-35	invalid block number
-7	do-loops nested too deeply during execution	-36	invalid file position
-8	dictionary overflow	-37	file I/O exception
-9	invalid memory address	-38	non-existent file
-10	division by zero	-39	unexpected end of file
-11	result out of range	-40	invalid **BASE** for floating point conversion
-12	argument type mismatch	-41	loss of precision
-13	undefined word	-42	floating-point divide by zero
-14	interpreting a compile-only word	-43	floating-point result out of range
-15	invalid **FORGET**	-44	floating-point stack overflow
-16	attempt to use zero-length string as a name	-45	floating-point stack underflow
-17	pictured numeric output string overflow	-46	floating-point invalid argument
-18	parsed string overflow	-47	compilation word list deleted
-19	definition name too long	-48	invalid **POSTPONE**
-20	write to a read-only location	-49	search-order overflow
-21	unsupported operation	-50	search-order underflow
	(e.g., **AT-XY** on a too-dumb terminal)	-51	compilation word list changed
-22	control structure mismatch	-52	control-flow stack overflow
-23	address alignment exception	-53	exception stack overflow
-24	invalid numeric argument	-54	floating-point underflow
-25	return stack imbalance	-55	floating-point unidentified fault
-26	loop parameters unavailable	-56	**QUIT**
-27	invalid recursion	-57	exception in sending or receiving a character
-28	user interrupt	-58	**[IF]**, **[ELSE]**, or **[THEN]** exception
-29	compiler nesting		

9.3.6 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** (**9.3.1 THROW values**, **9.3.5 Possible actions on an ambiguous condition**).

9.4.1.2 Ambiguous conditions

– no additional requirements.

9.4.1.3 Other system documentation

– no additional requirements.

9.4.2 Program documentation

– no additional requirements.

9.5 Compliance and labeling

9.5.1 ANS Forth systems

The phrase “Providing the Exception word set” shall be appended to the label of any Standard System that provides all of the Exception word set.

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

The phrase “Providing the Exception Extensions word set” shall be appended to the label of any Standard System that provides all of the Exception and Exception Extensions word sets.

9.5.2 ANS Forth programs

The phrase “Requiring the Exception word set” shall be appended to the label of Standard Programs that require the system to provide the Exception word set.

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

The phrase “Requiring the Exception Extensions word set” shall be appended to the label of Standard Programs that require the system to provide all of the Exception and Exception Extensions word sets.

9.6 Glossary

9.6.1 Exception words

9.6.1.0875 CATCH EXCEPTION

($i \times x\ xt$ – $j \times x\ 0$ | $i \times x\ n$)

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**.

9.6.1.2275 THROW EXCEPTION

($k \times x\ n$ – $k \times x$ | $i \times x\ n$)

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).

Rationale: 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 $i \times x$ 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
;
```

9.6.2 Exception extension words

9.6.2.0670 ABORT EXCEPTION EXT

Extend the semantics of **6.1.0670 ABORT** to be:

($i \times x$ -) (R: $j \times x$ -)

Perform the function of -1 **THROW**.

See: **6.1.0670 ABORT**.

9.6.2.0680 `ABORT"` “abort-quote” EXCEPTION EXT

Extend the semantics of **6.1.0680 `ABORT"`** to be:

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (*“ccc⟨quote⟩”* –)

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

Run-time: (*i*×*x* *x*$_1$ – | *i*×*x*) (R: *j*×*x* – | *j*×*x*)

Remove *x*$_1$ from the stack. If any bit of *x*$_1$ is not zero, perform the function of `-2` **`THROW`**, displaying *ccc* if there is no exception frame on the exception stack.

See: **3.4.1 Parsing**, **6.1.0680 `ABORT"`**.

10 The optional Facility word set

10.1 Introduction

10.2 Additional terms and notation

None.

10.3 Additional usage requirements

10.3.1 Character types

Programs that use more than seven bits of a character by **EKEY** have an environmental dependency.

See: **3.1.2 Character types**.

10.3.2 Environmental queries

Append table 10.1 to table 3.5.

See: **3.2.6 Environmental queries**.

Table 10.1: Environmental Query Strings

String	Value data type	Constant?	Meaning
FACILITY	*flag*	no	facility word set present
FACILITY-EXT	*flag*	no	facility extensions word set present

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.

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 ANS Forth 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 ANS Forth 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.

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

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.

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

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

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

10.6.2.1305 EKEY "e-key" FACILITY EXT

(– *u*)

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

See: **10.6.1.1755 KEY?**, **6.1.1750 KEY**.

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

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

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

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

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

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

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

> The MAX-CHAR environmental query would return 255.
>
> Assume the existence of a word DOS-KEY (– *char*) which executes the MS-DOS "Direct STDIN Input" system call (Interrupt 21h, Function 07h) and a word DOS-KEY? (– *flag*) which executes the MS-DOS "Check STDIN Status" system call (Interrupt 21h, Function 0Bh).

```
:  EKEY?  ( – flag )  DOS-KEY?  0<> ;
:  EKEY  ( – u )  DOS-KEY  ?DUP 0= IF  DOS-KEY 256 +  THEN ;
:  EKEY>CHAR ( u – u false | char true )
  DUP 255 > IF      ( u )
    DUP 259 = IF    \ 259 is Ctrl-@ (ASCII NUL)
      DROP 0 TRUE EXIT so replace with character
    THEN FALSE EXIT \ otherwise extended character
  THEN TRUE         \ normal extended ASCII char.
;
VARIABLE PENDING-CHAR   -1 PENDING-CHAR !
:  KEY?  ( – flag )
  PENDING-CHAR @ 0< IF
    BEGIN  EKEY? WHILE
      EKEY EKEY>CHAR IF
        PENDING-CHAR !  TRUE EXIT
      THEN DROP
    REPEAT  FALSE EXIT
  THEN  TRUE
;
:  KEY  ( – char )
  PENDING-CHAR @ 0< IF
    BEGIN  EKEY  EKEY>CHAR 0= WHILE
      DROP
    REPEAT  EXIT
  THEN  PENDING-CHAR @  -1 PENDING-CHAR !
;
```

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

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

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

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

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

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

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

The disadvantages of the simpler version are:

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

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

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

10.6.2.1306 EKEY>CHAR "e-key-to-char" FACILITY EXT

(*u* – *u false* | *char true*)

If the keyboard event *u* corresponds to a character in the implementation-defined character set, return that character and *true*. Otherwise return *u* and *false*.

Rationale: **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.

10.6.2.—— EKEY>FKEY "e-key-to-f-key" FACILITY EXT
X:ekeys

(u_1 – u_2 *f*)

If the keyboard event u_1 corresponds to a keypress in the implementation-defined special key set, return that key's id u_2 and *true*. Otherwise return u_1 and *false*.

See: **10.6.2.1305 EKEY**, **10.6.2.0 K-DELETE**, **10.6.2.0 K-DOWN**, **10.6.2.0 K-END**, **10.6.2.0 K-HOME**, **10.6.2.0 K-INSERT**, **10.6.2.0 K-LEFT**, **10.6.2.0 K-NEXT**, **10.6.2.0 K-PRIOR**, **10.6.2.0 K-RIGHT**, **10.6.2.0 K-UP**, **10.6.2.0 K-ALT-MASK**, **10.6.2.0 K-CTRL-MASK**, **10.6.2.0 K-SHIFT-MASK**, **10.6.2.0 K-F1**, **10.6.2.0 K-F2**, **10.6.2.0 K-F3**, **10.6.2.0 K-F4**, **10.6.2.0 K-F5**, **10.6.2.0 K-F6**, **10.6.2.0 K-F7**, **10.6.2.0 K-F8**, **10.6.2.0 K-F9**, **10.6.2.0 K-F10**, **10.6.2.0 K-F11**, and **10.6.2.0 K-F12**.

Rationale: This words has been provided to allow for systems which may not be able to interpret the value returned from **EKEY** directly. A simple implementation for a system which is capable of interpreting the value returned from **EKEY** would be:

```
:  EKEY>FKEY ( u1 -- u2 flag )
  DUP EKEY>CHAR NIP 0= ;
```

The value returned from **EKEY>FKEY** can be interpreted via a set of pre-defined constants, and masks. This provides the application programmer with the ability to process the non-ASCII keys in a standard way. Note however, that not all keyboards provide these keys. Indeed, some devices will not have a keyboard. A standard program should be written in such a way that they will continue to work without these additional keys. Albeit with a limited or reduced functionality.

The following words produce the same values that **EKEY EKEY>FKEY** may produce when the user presses the corresponding key. Note that, even if this extension is present, the keyboard may lack some of the keys, or the system the capability to report them, so programs should be written such that they also work (although less conveniently or with less functionality) if these key numbers cannot be produced.

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
```

EKEY return values:

The intention behind **EKEY** was that it would return the implementation defined keyboard scan code. Unfortunately there are some systems where the scan code is larger than a single cell. The implementation defined nature of the value means that a standard program may not use the value is any way, other than passing it on to **EKEY>CHAR** to convert it into an ASCII character.

EKEY>FKEY has been designed to be used in a similar manner. It provides a implementation defined value which corresponds to a combination of keys. A standard program can check which key combination has been used via a set of pre-defined constants.

A system where the scan code is larger than a single cell may approach this in one of two ways. It may manipulate the scan code to produce a single cell value. Alternatively it may return the address of a buffer which contains the actual scan code.

Note that many environments do not have or do not report all these keys and key combinations, so it is a good idea to write a program such that it is still useful even if these keys and key combinations cannot be pressed or are not recognised.

Shift keys:

Note that, as defined, the shift key masks (**10.6.2.0 K-SHIFT-MASK**, **10.6.2.0 K-CTRL-MASK** and **10.6.2.0 K-ALT-MASK**) are only useful for recognising shifted cursor and function keys, because these are the only keys that **EKEY** return values are defined for. E.g., a standard program cannot recognise ALT-T, because no **EKEY** return value for `T` has been defined.

Implementation: The implementation is closely tied to the implementation of **EKEY**, and therefore unportable.

Testing:

```
{ K-LEFT TRUE -> CR ." Please press <left>" EKEY EKEY>FKEY }
{ K-RIGHT TRUE -> CR ." Please press <right>" EKEY EKEY>FKEY
}
{ K-UP TRUE -> CR ." Please press <up>" EKEY EKEY>FKEY }
{ K-DOWN TRUE -> CR ." Please press <down>" EKEY EKEY>FKEY }
{ K-HOME TRUE -> CR ." Please press <home>" EKEY EKEY>FKEY }
{ K-END TRUE -> CR ." Please press <end>" EKEY EKEY>FKEY }
{ K-PRIOR TRUE -> CR ." Please press <prior>" EKEY EKEY>FKEY
}
{ K-NEXT TRUE -> CR ." Please press <next>" EKEY EKEY>FKEY }

{ K-F1 TRUE -> CR ." Please press <F1>" EKEY EKEY>FKEY }
{ K-F2 TRUE -> CR ." Please press <F2>" EKEY EKEY>FKEY }
{ K-F3 TRUE -> CR ." Please press <F3>" EKEY EKEY>FKEY }
{ K-F4 TRUE -> CR ." Please press <F4>" EKEY EKEY>FKEY }
{ K-F5 TRUE -> CR ." Please press <F5>" EKEY EKEY>FKEY }
{ K-F6 TRUE -> CR ." Please press <F6>" EKEY EKEY>FKEY }
{ K-F7 TRUE -> CR ." Please press <F7>" EKEY EKEY>FKEY }
{ K-F8 TRUE -> CR ." Please press <F8>" EKEY EKEY>FKEY }
{ K-F9 TRUE -> CR ." Please press <F9>" EKEY EKEY>FKEY }
{ K-F10 TRUE -> CR ." Please press <F10>" EKEY EKEY>FKEY }
{ K-F11 TRUE -> CR ." Please press <F11>" EKEY EKEY>FKEY }
{ K-F12 TRUE -> CR ." Please press <F12>" EKEY EKEY>FKEY }

{ K-LEFT K-SHIFT-MASK OR TRUE ->
  CR ." Please press <shift-left>" EKEY EKEY>FKEY }
{ K-LEFT K-CTRL-MASK OR TRUE ->
```

```
  CR ." Please press <ctrl-left>" EKEY EKEY>FKEY }
{ K-LEFT K-ALT-MASK OR TRUE ->
  CR ." Please press <alt-left>" EKEY EKEY>FKEY }

{ CR ." Please press <a>" EKEY EKEY>FKEY NIP SWAP EKEY>CHAR
  -> FALSE CHAR a TRUE }
```

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.

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

10.6.2.—— K-ALT-MASK FACILITY EXT
X:ekeys

(– *u*)

Mask for the ALT key, that can be **OR**ed with the key value to produce a value that the sequence **EKEY EKEY>FKEY** may produce when the user presses the corresponding key combination.

See: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— K-CTRL-MASK FACILITY EXT
X:ekeys

(– *u*)

Mask for the CTRL key, that can be **OR**ed with the key value to produce a value that the sequence **EKEY EKEY>FKEY** may produce when the user presses the corresponding key combination.

See: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— K-DELETE FACILITY EXT
X:ekeys

(– *u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "Delete" key.

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: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— K-DOWN FACILITY EXT
X:ekeys

(– *u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "cursor down" key.

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: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— K-END FACILITY EXT
X:ekeys

(– *u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "End" key.

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: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— K-F1 "k-f-1" FACILITY EXT
X:ekeys

(– *u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "F1" key.

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: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— `K-F10` "k-f-10" FACILITY EXT
X:ekeys

(– *u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "F10" key.

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: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— `K-F11` "k-f-11" FACILITY EXT
X:ekeys

(– *u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "F11" key.

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: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— `K-F12` "k-f-12" FACILITY EXT
X:ekeys

(– *u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "F12" key.

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: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— `K-F2` "k-f-2" FACILITY EXT
X:ekeys

(– *u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "F2" key.

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: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— K-F3 “k-f-3” FACILITY EXT
X:ekeys

(– *u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the “F3” key.

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: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— K-F4 “k-f-4” FACILITY EXT
X:ekeys

(– *u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the “F4” key.

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: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— K-F5 “k-f-5” FACILITY EXT
X:ekeys

(– *u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the “F5” key.

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: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— K-F6 “k-f-6” FACILITY EXT
X:ekeys

(– *u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the “F6” key.

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: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— K-F7 “k-f-7” FACILITY EXT
X:ekeys

(– *u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the “F7” key.

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: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— K-F8 “k-f-8” FACILITY EXT
X:ekeys

(– *u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the “F8” key.

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: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— K-F9 “k-f-9” FACILITY EXT
X:ekeys

(– *u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the “F9” key.

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: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— K-HOME FACILITY EXT
X:ekeys

(– *u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the “home” or “Pos1” key.

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: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— K-INSERT FACILITY EXT
X:ekeys

(– *u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "Insert" key.

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: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— K-LEFT FACILITY EXT
X:ekeys

(– *u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "cursor left" key.

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: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— K-NEXT FACILITY EXT
X:ekeys

(– *u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "PgDn" (Page Down) or "Next" key.

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: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— K-PRIOR FACILITY EXT
X:ekeys

(– *u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "PgUp" (Page Up) or "Prior" key.

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: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— K-RIGHT FACILITY EXT
X:ekeys

(– *u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "cursor right" key.

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: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— K-SHIFT-MASK FACILITY EXT
X:ekeys

(– *u*)

Mask for the SHIFT key, that can be **OR**ed with the key value to produce a value that the sequence **EKEY EKEY>FKEY** may produce when the user presses the corresponding key combination.

See: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.—— K-UP FACILITY EXT
X:ekeys

(– *u*)

Leaves the value *u* that the sequence **EKEY EKEY>FKEY** would produce when the user presses the "cursor up" key.

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: **10.6.2.0 EKEY>FKEY**, **A.10.6.2.0 EKEY>FKEY**.

10.6.2.1905 MS FACILITY EXT

(*u* –)

Wait at least *u* milliseconds.

Note: The actual length and variability of the time period depends upon the implementation-defined resolution of the system clock and upon other system and computer characteristics beyond the scope of this Standard.

Rationale: 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.

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).

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

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*
ior	I/O results	1 cell
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.

11.3.1.2 I/O results

I/O results are single-cell numbers indicating the result of I/O operations. A value of zero indicates that the I/O operation completed successfully; other values and their meanings are implementation-defined. Reaching the end of a file shall be reported as zero.

An I/O exception in the execution of a File-Access word that can return an I/O result shall not cause a **THROW**; exception indications are returned in the *ior*.

11.3.1.3 File access methods

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

If the File-Access word set is implemented, the Block word set shall be implemented. Blocks may, but need not, reside in files. When they do:

- Block numbers may be mapped to one or more files by implementation-defined means. An ambiguous condition exists if a requested block number is not currently mapped;
- An **UPDATE**d block that came from a file shall be transferred back to the same file.

11.3.3 Environmental queries

Append table 11.2 to table 3.5.

See: **3.2.6 Environmental queries**.

Table 11.2: Environmental Query Strings

String	Value data type	Constant?	Meaning
FILE	*flag*	no	file word set present
FILE-EXT	*flag*	no	file extensions word set present

11.3.4 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.

Input with **INCLUDED**, **INCLUDE-FILE**, **LOAD** and **EVALUATE** shall be nestable in any order to at least eight levels.

A program that uses more than eight levels of input-file nesting has an environmental dependency. See: **3.3.3.5 Input buffers**, **9 The optional Exception word set**.

11.3.5 Other transient regions

The list of words using memory in transient regions is extended to include **11.6.1.2165 S"**. See: **3.3.3.6 Other transient regions**.

11.3.6 Parsing

When parsing from a text file using a space delimiter, control characters shall be treated the same as the space character.

Lines of at least 128 characters shall be supported. A program that requires lines of more than 128 characters has an environmental dependency.

A program may reposition the parse area within the input buffer by manipulating the contents of **>IN**. More extensive repositioning can be accomplished using **SAVE-INPUT** and **RESTORE-INPUT**.

See: **3.4.1 Parsing**.

11.4 Additional documentation requirements

11.4.1 System documentation

11.4.1.1 Implementation-defined options

- file access methods used by **11.6.1.0765 BIN**, **11.6.1.1010 CREATE-FILE**, **11.6.1.1970 OPEN-FILE**, **11.6.1.2054 R/O**, **11.6.1.2056 R/W**, and **11.6.1.2425 W/O**;
- file exceptions;
- file line terminator (**11.6.1.2090 READ-LINE**);
- file name format (**11.3.1.4 File names**);
- information returned by **11.6.2.1524 FILE-STATUS**;
- input file state after an exception (**11.6.1.1717 INCLUDE-FILE**, **11.6.1.1718 INCLUDED**);
- *ior* values and meaning (**11.3.1.2 I/O results**);
- maximum depth of file input nesting (**11.3.4 Input source**);
- maximum size of input line (**11.3.6 Parsing**);
- methods for mapping block ranges to files (**11.3.2 Blocks in files**);
- number of string buffers provided (**11.6.1.2165 S"**);
- size of string buffer used by **11.6.1.2165 S"**.

11.4.1.2 Ambiguous conditions

- attempting to position a file outside its boundaries (**11.6.1.2142 REPOSITION-FILE**);
- attempting to read from file positions not yet written (**11.6.1.2080 READ-FILE**, **11.6.1.2090 READ-LINE**);
- *fileid* is invalid (**11.6.1.1717 INCLUDE-FILE**);
- I/O exception reading or closing *fileid* (**11.6.1.1717 INCLUDE-FILE**, **11.6.1.1718 INCLUDED**);
- named file cannot be opened (**11.6.1.1718 INCLUDED**);
- requesting an unmapped block number (**11.3.2 Blocks in files**);
- using **11.6.1.2218 SOURCE-ID** when **7.6.1.0790 BLK** is not zero.

x:required

- a file is required while it is being **REQUIRED** (11.6.2.0) or **INCLUDED** (11.6.1.1718).
- a marker is defined outside and executed inside a file or vice versa, and the file is **REQUIRED** (11.6.2.0) 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 **11.6.2.0** REQUIRED (by doing some renaming between the invocations of **REQUIRED**).
- the stack effect of including with **11.6.2.0** REQUIRED the file is not ($i \times x - i \times x$) .

11.4.1.3 Other system documentation

– no additional requirements.

11.4.2 Program documentation

11.4.2.1 Environmental dependencies

– requiring lines longer than 128 characters (**11.3.6 Parsing**);

– using more than eight levels of input-file nesting (**11.3.4 Input source**).

11.4.2.2 Other program documentation

– no additional requirements.

11.5 Compliance and labeling

11.5.1 ANS Forth 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 ANS Forth programs

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

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

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

11.6 Glossary

11.6.1 File Access words

11.6.1.0080 ("paren" FILE

("*ccc*⟨*paren*⟩" –)

Extend the semantics of **6.1.0080 (** to include:

When parsing from a text file, if the end of the parse area is reached before a right parenthesis is found, refill the input buffer from the next line of the file, set **>IN** to zero, and resume parsing, repeating this process until either a right parenthesis is found or the end of the file is reached.

11.6.1.0765 BIN FILE

(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: **11.6.1.2054 R/O**, **11.6.1.2056 R/W**, **11.6.1.2425 W/O**.

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

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

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.

Rationale: Typical use:

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

11.6.1.1190 DELETE-FILE FILE

(*c-addr u* – *ior*)

Delete the file named in the character string specified by *c-addr u*. *ior* is the implementation-defined I/O result code.

11.6.1.1520 FILE-POSITION FILE

(*fileid* – *ud ior*)

ud is the current file position for the file identified by *fileid*. *ior* is the implementation-defined I/O result code. *ud* is undefined if *ior* is non-zero.

11.6.1.1522 FILE-SIZE FILE

(*fileid* – *ud ior*)

ud is the size, in characters, of the file identified by *fileid*. *ior* is the implementation-defined I/O result code. This operation does not affect the value returned by **FILE-POSITION**. *ud* is undefined if *ior* is non-zero.

11.6.1.1717 INCLUDE-FILE FILE

(*i*×*x fileid* – *j*×*x*)

Remove *fileid* from the stack. Save the current input source specification, including the current value of **SOURCE-ID**. Store *fileid* in **SOURCE-ID**. Make the file specified by *fileid* the input source. Store zero in **BLK**. Other stack effects are due to the words included.

Repeat until end of file: read a line from the file, fill the input buffer from the contents of that line, set **>IN** to zero, and interpret.

Text interpretation begins at the file position where the next file read would occur.

When the end of the file is reached, close the file and restore the input source specification to its saved value.

An ambiguous condition exists if *fileid* is invalid, if there is an I/O exception reading *fileid*, or if an I/O exception occurs while closing *fileid*. When an ambiguous condition exists, the status (open or closed) of any files that were being interpreted is implementation-defined.

See: **11.3.4 Input source**.

Rationale: 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.

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 ~~file position where the next file read would occur~~start of the file . X:required

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.

X:required **INCLUDED** may allocate memory in data space before it starts interpreting the file.

See: **11.6.1.1717 INCLUDE-FILE**.

Rationale: Typical use: ... `S" filename" INCLUDED` ...

Testing: See **A.11.6.2.0** REQUIRED.

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.

Rationale: Typical use:

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

11.6.1.2054 R/O “r-o” FILE

(– *fam*)

fam is the implementation-defined value for selecting the “read only” file access method.

See: **11.6.1.1010 CREATE-FILE**, **11.6.1.1970 OPEN-FILE**.

11.6.1.2056 R/W “r-w” FILE

(– *fam*)

fam is the implementation-defined value for selecting the “read/write” file access method.

See: **11.6.1.1010 CREATE-FILE**, **11.6.1.1970 OPEN-FILE**.

11.6.1.2080 READ-FILE FILE

(*c-addr* u_1 *fileid* – u_2 *ior*)

Read u_1 consecutive characters to *c-addr* from the current position of the file identified by *fileid*.

If u_1 characters are read without an exception, *ior* is zero and u_2 is equal to u_1.

If the end of the file is reached before u_1 characters are read, *ior* is zero and u_2 is the number of characters actually read.

If the operation is initiated when the value returned by **FILE-POSITION** is equal to the value returned by **FILE-SIZE** for the file identified by *fileid*, *ior* is zero and u_2 is zero.

If an exception occurs, *ior* is the implementation-defined I/O result code, and u_2 is the number of characters transferred to *c-addr* without an exception.

An ambiguous condition exists if the operation is initiated when the value returned by **FILE-POSITION** is greater than the value returned by **FILE-SIZE** for the file identified by *fileid*, or if the requested operation attempts to read portions of the file not written.

At the conclusion of the operation, **FILE-POSITION** returns the next file position after the last character read.

Rationale: A typical sequential file-processing algorithm might look like:

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

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

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.

Rationale: Implementations are allowed to store the line terminator in the memory buffer in order to allow the use of line reading functions provided by host operating systems, some of which store the terminator. Without this provision, a temporary buffer might be needed.

The two-character limitation is sufficient for the vast majority of existing operating systems. Implementations on host operating systems whose line terminator sequence is longer than two characters may have to take special action to prevent the storage of more than two terminator characters.

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

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

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

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

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

11.6.1.2142 REPOSITION-FILE FILE

(*ud fileid* – *ior*)

Reposition the file identified by *fileid* to *ud*. *ior* is the implementation-defined I/O result code. An ambiguous condition exists if the file is positioned outside the file boundaries.

At the conclusion of the operation, **FILE-POSITION** returns the value *ud*.

11.6.1.2147 RESIZE-FILE FILE

(*ud fileid* – *ior*)

Set the size of the file identified by *fileid* to *ud*. *ior* is the implementation-defined I/O result code.

If the resultant file is larger than the file before the operation, the portion of the file added as a result of the operation might not have been written.

At the conclusion of the operation, **FILE-SIZE** returns the value *ud* and **FILE-POSITION** returns an unspecified value.

See: **11.6.1.2080 READ-FILE**, **11.6.1.2090 READ-LINE**.

11.6.1.2165 S" "s-quote" FILE

Extend the semantics of **6.1.2165 S"** to be:

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

Parse *ccc* delimited by **"** (double quote). Store the resulting string *c-addr u* at a temporary location. The maximum length of the temporary buffer is implementation-dependent but shall be no less than 80 characters. Subsequent uses of **S"** may overwrite the temporary buffer. At least one such buffer shall be provided.

Compilation: (*“ccc⟨quote⟩”* –)

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

Run-time: (– *c-addr u*)

Return *c-addr* and *u* that describe a string consisting of the characters *ccc*.

See: **3.4.1 Parsing**, **6.2.0855 C"**, **6.1.2165 S"**, **11.3.5 Other transient regions**.

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

The interpretation semantics for **S"** are intended to provide a simple mechanism for entering a string in the interpretation state. Since an implementation may choose to provide only one buffer for interpreted strings, an interpreted string is subject to being overwritten by the next execution of **S"** in interpretation state. It is intended that no standard words other than **S"** should in themselves cause the interpreted string to be overwritten. However, since words such as **EVALUATE**, **LOAD**, **INCLUDE-FILE** and **INCLUDED** can result in the interpretation of arbitrary text, possibly including instances of **S"**, the interpreted string may be invalidated by some uses of these words.

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

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

11.6.1.2218 SOURCE-ID “source-i-d” FILE

(– *0* | *-1* | *fileid*)

Extend **6.2.2218 SOURCE-ID** to include text-file input as follows:

SOURCE-ID	Input source
fileid	Text file “*fileid*”
-1	String (via **EVALUATE**)
0	User input device

An ambiguous condition exists if **SOURCE-ID** is used when **BLK** contains a non-zero value.

11.6.1.2425 W/O “w-o” FILE

(– *fam*)

fam is the implementation-defined value for selecting the “write only” file access method.

See: **11.6.1.1010 CREATE-FILE**, **11.6.1.1970 OPEN-FILE**.

11.6.1.2480 WRITE-FILE FILE

(*c-addr u fileid* – *ior*)

Write *u* characters from *c-addr* to the file identified by *fileid* starting at its current position. *ior* is the implementation-defined I/O result code.

At the conclusion of the operation, **FILE-POSITION** returns the next file position after the last character written to the file, and **FILE-SIZE** returns a value greater than or equal to the value returned by **FILE-POSITION**.

See: **11.6.1.2080 READ-FILE**, **11.6.1.2090 READ-LINE**.

11.6.1.2485 `WRITE-LINE` FILE

(*c-addr u fileid* – *ior*)

Write *u* characters from *c-addr* followed by the implementation-dependent line terminator to the file identified by *fileid* starting at its current position. *ior* is the implementation-defined I/O result code.

At the conclusion of the operation, **FILE-POSITION** returns the next file position after the last character written to the file, and **FILE-SIZE** returns a value greater than or equal to the value returned by **FILE-POSITION**.

See: **11.6.1.2080 READ-FILE**, **11.6.1.2090 READ-LINE**.

11.6.2 File-Access extension words

11.6.2.1524 `FILE-STATUS` FILE EXT

(*c-addr u* – *x ior*)

Return the status of the file identified by the character string *c-addr u*. If the file exists, *ior* is zero; otherwise *ior* is the implementation-defined I/O result code. *x* contains implementation-defined information about the file.

11.6.2.1560 `FLUSH-FILE` FILE EXT

(*fileid* – *ior*)

Attempt to force any buffered information written to the file referred to by *fileid* to be written to mass storage, and the size information for the file to be recorded in the storage directory if changed. If the operation is successful, *ior* is zero. Otherwise, it is an implementation-defined I/O result code.

11.6.2.—— `INCLUDE` FILE EXT
X:required

(*i×x "name"* – *j×x*)

Skip leading white space and parse *name* delimited by a white space character. Push the address and length of the *name* on the stack and perform the function of **INCLUDED**.

See: **11.6.1.1718 INCLUDED**.

Rationale: Typical use:

```
INCLUDE filename
```

Implementation:

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

Testing: See **A.11.6.2.0 REQUIRED**.

11.6.2.2125 REFILL FILE EXT

(– *flag*)

Extend the execution semantics of **6.2.2125 REFILL** with the following:

When the input source is a text file, attempt to read the next line from the text-input file. If successful, make the result the current input buffer, set **>IN** to zero, and return *true*. Otherwise return *false*.

See: **6.2.2125 REFILL**, **7.6.2.2125 REFILL**.

11.6.2.2130 RENAME-FILE FILE EXT

(*c-addr*$_1$ *u*$_1$ *c-addr*$_2$ *u*$_2$ – *ior*)

Rename the file named by the character string *c-addr*$_1$ *u*$_1$ to the name in the character string *c-addr*$_2$ *u*$_2$. *ior* is the implementation-defined I/O result code.

11.6.2.—— REQUIRE FILE EXT
X:required

(*i*×*x* *“name”* – *i*×*x*)

Skip leading white space and parse *name* delimited by a white space character. Push the address and length of the *name* on the stack and perform the function of **REQUIRED**.

See: **11.6.2.0 REQUIRED**.

Rationale: Typical use:

```
REQUIRE filename
```

Implementation:

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

Testing: See **A.11.6.2.0 REQUIRED**.

11.6.2.—— REQUIRED FILE EXT
X:required

(*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*) .

Rationale: Typical use:

```
S" filename" REQUIRED
```

Implementation: 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 ;
```

Testing: 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:

```
{ 0
  S" required-helper1.fs" REQUIRED \ Increment TOS
  REQUIRE required-helper1.fs \ Ignore - already loaded
  INCLUDE required-helper1.fs \ Increment TOS
-> 2 }

{ 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 }
```

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

12.2.2.1 Numeric notation

The following notation is used to define the syntax of the external representation of floating-point numbers:

Each component of a floating-point number is defined with a rule consisting of the name of the component (italicized in angle-brackets, e.g., ⟨*sign*⟩, 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., ⟨*digit*⟩; The metacharacter * is used to specify zero or more occurrences of the preceding token (e.g., ⟨*digit*⟩*);

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

Vertical bars separate choices from a list of tokens enclosed with braces (e.g., { + | − }).

12.2.2.2 Stack notation

Floating-point stack notation when the floating-point stack is separate from the data stack is:

(F: *before* − *after*)

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

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.

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. By default the floating-point stack shall be separate from the data and return stacks. A program may determine whether floating-point numbers are kept on the data stack by passing the string "`FLOATING-STACK`" to **`ENVIRONMENT?`**.

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 table 3.5.

See: **3.2.6 Environmental queries**.

12.3.5 Address alignment

Since the address returned by a **`CREATE`**d word is not necessarily aligned for any particular class of floating-point data, a program shall align the address (to be float aligned, single-float aligned, or double-float aligned) before accessing floating-point data at the address.

See: **3.3.3.1 Address alignment**, **12.3.1.1 Addresses**.

Table 12.2: Environmental Query Strings

String	Value data type	Constant?	Meaning
FLOATING	*flag*	no	floating-point word set present
FLOATING-EXT	*flag*	no	floating-point extensions word set present
FLOATING-STACK	*n*	yes	If *n* = zero, floating-point numbers are kept on the data stack; otherwise *n* is the maximum depth of the separate floating-point stack.
MAX-FLOAT	*r*	yes	largest usable floating-point number

12.3.6 Variables

A program may address memory in data space regions made available by **FVARIABLE**. These regions may be non-contiguous with regions subsequently allocated with , (comma) or **ALLOT**. See: **3.3.3.3 Variables**.

12.3.7 Text interpreter input number conversion

If the Floating-Point word set is present in the dictionary and the current base is **DECIMAL**, the input number-conversion algorithm shall be extended to recognize floating-point numbers in this form:

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: **3.4.1.3 Text interpreter input number conversion**, **12.6.1.0558 >FLOAT**.

12.4 Additional documentation requirements

12.4.1 System documentation

12.4.1.1 Implementation-defined options

- format and range of floating-point numbers (**12.3.1 Data types**, **12.6.1.2143 REPRESENT**)
- results of **12.6.1.2143 REPRESENT** when *float* is out of range;
- rounding or truncation of floating-point numbers (**12.3.1.2 Floating-point numbers**);
- size of floating-point stack (**12.3.3 Floating-point stack**);
- width of floating-point stack (**12.3.3 Floating-point stack**).

12.4.1.2 Ambiguous conditions

- **DF@** or **DF!** is used with an address that is not double-float aligned;
- **F@** or **F!** is used with an address that is not float aligned;
- floating point result out of range (e.g., in **12.6.1.1430 F/**);
- **SF@** or **SF!** is used with an address that is not single-float aligned;
- **BASE** is not decimal (**12.6.1.2143 REPRESENT**, **12.6.2.1427 F**, **12.6.2.1513 FE**, **12.6.2.1613 FS**);
- both arguments equal zero (**12.6.2.1489 FATAN2**);
- cosine of argument is zero for **12.6.2.1625 FTAN**;
- *d* can't be precisely represented as *float* in **12.6.1.1130 D>F**;
- dividing by zero (**12.6.1.1430 F/**);
- exponent too big for conversion (**12.6.2.1203 DF!**, **12.6.2.1204 DF@**, **12.6.2.2202 SF!**, **12.6.2.2203 SF@**);
- *float* less than one (**12.6.2.1477 FACOSH**);
- *float* less than or equal to minus-one (**12.6.2.1554 FLNP1**);
- *float* less than or equal to zero (**12.6.2.1553 FLN**, **12.6.2.1557 FLOG**);
- *float* less than zero (**12.6.2.1487 FASINH**, **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**).

12.4.1.3 Other system documentation

- no additional requirements.

12.4.2 Program documentation

12.4.2.1 Environmental dependencies

- requiring the floating-point stack to be larger than six items (**12.3.3 Floating-point stack**).

12.4.2.2 Other program documentation

- no additional requirements.

12.5 Compliance and labeling

12.5.1 ANS Forth 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 ANS Forth programs

The phrase "Requiring the Floating-Point word set" shall be appended to the label of Standard Programs that require the system to provide the Floating-Point word set.

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

The phrase "Requiring the Floating-Point Extensions word set" shall be appended to the label of Standard Programs that require the system to provide all of the Floating-Point and Floating-Point Extensions word sets.

12.6 Glossary

12.6.1 Floating-Point words

12.6.1.0558 `>FLOAT` "to-float" FLOATING

(*c-addr u* – *true* | *false*) (F: – *r* |) or (*c-addr u* – *r true* | *false*)

An attempt is made to convert the string specified by *c-addr* and *u* to internal floating-point representation. If the string represents a valid floating-point number in the syntax below, its value *r* and *true* are returned. If the string does not represent a valid floating-point number only *false* is returned.

A string of blanks should be treated as a special case representing zero.

The syntax of a convertible string

$$
\begin{aligned}
&:= \langle significand \rangle [\langle exponent \rangle] \\
\langle significand \rangle &:= [\langle sign \rangle] \{ \langle digits \rangle [\,.\, \langle digits0 \rangle] \mid .\, \langle digits \rangle \} \\
\langle exponent \rangle &:= \langle marker \rangle \langle digits0 \rangle \\
\langle marker \rangle &:= \{ \langle e\text{-}form \rangle \mid \langle sign\text{-}form \rangle \} \\
\langle e\text{-}form \rangle &:= \langle e\text{-}char \rangle [\langle sign\text{-}form \rangle] \\
\langle sign\text{-}form \rangle &:= \{ + \mid - \} \\
\langle e\text{-}char \rangle &:= \{ \texttt{D} \mid \texttt{d} \mid \texttt{E} \mid \texttt{e} \}
\end{aligned}
$$

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

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

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

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

(*d* –) (F: – *r*) or (*d* – *r*)

r is the floating-point equivalent of *d*. An ambiguous condition exists if *d* cannot be precisely represented as a floating-point value.

12.6.1.1400 `F!` "f-store" FLOATING

(*f-addr* –) (F: *r* –) or (*r f-addr* –)
Store *r* at *f-addr*.

12.6.1.1410 `F*` "f-star" FLOATING

(F: r_1 r_2 – r_3) or (r_1 r_2 – r_3)
Multiply r_1 by r_2 giving r_3.

12.6.1.1420 `F+` "f-plus" FLOATING

(F: r_1 r_2 – r_3) or (r_1 r_2 – r_3)
Add r_1 to r_2 giving the sum r_3.

12.6.1.1425 `F-` "f-minus" FLOATING

(F: r_1 r_2 – r_3) or (r_1 r_2 – r_3)
Subtract r_2 from r_1, giving r_3.

12.6.1.1430 `F/` "f-slash" FLOATING

(F: r_1 r_2 – r_3) or (r_1 r_2 – r_3)
Divide r_1 by r_2, giving the quotient r_3. An ambiguous condition exists if r_2 is zero, or the quotient lies outside of the range of a floating-point number.

12.6.1.1440 `F0<` "f-zero-less-than" FLOATING

(– *flag*) (F: *r* –) or (*r* – *flag*)
flag is true if and only if *r* is less than zero.

12.6.1.1450 `F0=` "f-zero-equals" FLOATING

(– *flag*) (F: *r* –) or (*r* – *flag*)
flag is true if and only if *r* is equal to zero.

12.6.1.1460 `F<` "f-less-than" FLOATING

(– *flag*) (F: r_1 r_2 –) or (r_1 r_2 – *flag*)
flag is true if and only if r_1 is less than r_2.

12.6.1.1470 `F>D` "f-to-d" FLOATING

(– *d*) (F: *r* –) or (*r* – *d*)

d is the double-cell signed-integer equivalent of the integer portion of *r*. The fractional portion of *r* is discarded. An ambiguous condition exists if the integer portion of *r* cannot be precisely represented as a double-cell signed integer.

12.6.1.1472 `F@` "f-fetch" FLOATING

(*f-addr* –) (F: – *r*) or (*f-addr* – *r*)

r is the value stored at *f-addr*.

12.6.1.1479 `FALIGN` "f-align" FLOATING

(–)

If the data-space pointer is not float aligned, reserve enough data space to make it so.

12.6.1.1483 `FALIGNED` "f-aligned" FLOATING

(*addr* – *f-addr*)

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

12.6.1.1492 `FCONSTANT` "f-constant" FLOATING

("⟨*spaces*⟩*name*" –) (F: *r* –) or (*r* "⟨*spaces*⟩*name*" –)

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

name is referred to as an "f-constant".

name Execution: (–) (F: – *r*) or (– *r*)

Place *r* on the floating-point stack.

See: **3.4.1 Parsing**.

Rationale: Typical use: `r` **`FCONSTANT`** *name*

12.6.1.1497 `FDEPTH` "f-depth" FLOATING

(– +*n*)

+*n* is the number of values contained on the default separate floating-point stack. If floating-point numbers are kept 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.

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

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.

12.6.1.1562 FMAX "f-max" FLOATING

(F: r_1 r_2 – r_3) or (r_1 r_2 – r_3)

r_3 is the greater of r_1 and r_2.

12.6.1.1565 `FMIN` "f-min" FLOATING

(F: r_1 r_2 – r_3) or (r_1 r_2 – r_3)

r_3 is the lesser of r_1 and r_2.

12.6.1.1567 `FNEGATE` "f-negate" FLOATING

(F: r_1 – r_2) or (r_1 – r_2)

r_2 is the negation of r_1.

12.6.1.1600 `FOVER` "f-over" FLOATING

(F: r_1 r_2 – r_1 r_2 r_1) or (r_1 r_2 – r_1 r_2 r_1)

Place a copy of r_1 on top of the floating-point stack.

12.6.1.1610 `FROT` "f-rote" FLOATING

(F: r_1 r_2 r_3 – r_2 r_3 r_1) or (r_1 r_2 r_3 – r_2 r_3 r_1)

Rotate the top three floating-point stack entries.

12.6.1.1612 `FROUND` "f-round" FLOATING

(F: r_1 – r_2) or (r_1 – r_2)

Round r_1 to an integral value using the "round to nearest" rule, giving r_2.

See: **12.3.2 Floating-point operations**.

12.6.1.1620 `FSWAP` "f-swap" FLOATING

(F: r_1 r_2 – r_2 r_1) or (r_1 r_2 – r_2 r_1)

Exchange the top two floating-point stack items.

12.6.1.1630 `FVARIABLE` "f-variable" FLOATING

("⟨*spaces*⟩*name*" –)

Skip leading space delimiters. Parse *name* delimited by a space. Create a definition for *name* with the execution semantics defined below. Reserve `1` **`FLOATS`** address units of data space at a float-aligned address.

name is referred to as an "f-variable".

name Execution: (– *f-addr*)

f-addr is the address of the data space reserved by **`FVARIABLE`** when it created *name*. A program is responsible for initializing the contents of the reserved space.

See: **3.4.1 Parsing**.

Rationale: Typical use: **`FVARIABLE`** *name*

12.6.1.2143 REPRESENT FLOATING

(*c-addr u* – *n flag*$_1$ *flag*$_2$) (F: *r* –) or (*r c-addr u* – *n flag*$_1$ *flag*$_2$)

At *c-addr*, place the character-string external representation of the significand of the floating-point number *r*. Return the decimal-base exponent as *n*, the sign as *flag*$_1$ and "valid result" as *flag*$_2$. The character string shall consist of the *u* most significant digits of the significand represented as a decimal fraction with the implied decimal point to the left of the first digit, and the first digit zero only if all digits are zero. The significand is rounded to *u* digits following the "round to nearest" rule; *n* is adjusted, if necessary, to correspond to the rounded magnitude of the significand. If *flag*$_2$ is *true* then *r* was in the implementation-defined range of floating-point numbers. If *flag*$_1$ is *true* then *r* is negative.

An ambiguous condition exists if the value of **BASE** is not decimal ten.

When *flag*$_2$ is *false*, *n* and *flag*$_1$ are implementation defined, as are the contents of *c-addr*. Under these circumstances, the string at *c-addr* shall consist of graphic characters.

See: **3.2.1.2 Digit conversion**, **6.1.0750 BASE**, **12.3.2 Floating-point operations**.

Rationale: 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.

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: **12.3.1.1 Addresses**, **12.3.2 Floating-point operations**.

12.6.2.1204 DF@ "d-f-fetch" FLOATING EXT

(*df-addr* –) (F: – *r*) or (*df-addr* – *r*)

Fetch the 64-bit IEEE double-precision number stored at *df-addr* to the floating-point stack as *r* in the internal representation. If the IEEE double-precision significand has more precision than the internal representation it will be rounded to the internal 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: **12.3.1.1 Addresses**, **12.3.2 Floating-point operations**.

12.6.2.1205 `DFALIGN` "d-f-align" FLOATING EXT

(–)

If the data-space pointer is not double-float aligned, reserve enough data space to make it so.

See: **12.3.1.1 Addresses**.

12.6.2.1207 `DFALIGNED` "d-f-aligned" FLOATING EXT

(*addr* – *df-addr*)

df-addr is the first double-float-aligned address greater than or equal to *addr*.

See: **12.3.1.1 Addresses**.

12.6.2.1208 `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: **12.3.1.1 Addresses**.

12.6.2.1209 `DFLOATS` "d-floats" FLOATING EXT

(n_1 – n_2)

n_2 is the size in address units of n_1 64-bit IEEE double-precision numbers.

12.6.2.1415 `F**` "f-star-star" FLOATING EXT

(F: r_1 r_2 – r_3) or (r_1 r_2 – r_3)

Raise r_1 to the power r_2, giving the product r_3.

12.6.2.1427 `F.` "f-dot" FLOATING EXT

(–) (F: *r* –) or (*r* –)

Display, with a trailing space, the top number on the floating-point stack using fixed-point notation:

[–] ⟨*digits*⟩ **.** ⟨*digits0*⟩

An ambiguous condition exists if the value of **BASE** is not (decimal) ten or if the character string representation exceeds the size of the pictured numeric output string buffer.

See: **12.6.1.0558 >FLOAT**.

Rationale: For example, `1E3` **F** displays `1000.`.

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. An ambiguous condition exists if r_1 is less than zero.

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 radian angle whose tangent is r_1/r_2. An ambiguous condition exists if r_1 and r_2 are zero.

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

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

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

12.6.2.1491 FATANH "f-a-tan-h" FLOATING EXT

(F: r_1 – r_2) or (r_1 – r_2)

r_2 is the floating-point value whose hyperbolic tangent is r_1. An ambiguous condition exists if r_1 is outside the range of -1E0 to 1E0.

12.6.2.1493 FCOS "f-cos" FLOATING EXT

(F: r_1 – r_2) or (r_1 – r_2)

r_2 is the cosine of the radian angle r_1.

12.6.2.1494 FCOSH "f-cosh" FLOATING EXT

(F: r_1 – r_2) or (r_1 – r_2)

r_2 is the hyperbolic cosine of r_1.

12.6.2.1513 FE. "f-e-dot" FLOATING EXT

(–) (F: *r* –) or (*r* –)

Display, with a trailing space, the top number on the floating-point stack using engineering notation, where the significand is greater than or equal to 1.0 and less than 1000.0 and the decimal exponent is a multiple of three.

An ambiguous condition exists if the value of **BASE** is not (decimal) ten or if the character string representation exceeds the size of the pictured numeric output string buffer.

See: **6.1.0750 BASE**, **12.3.2 Floating-point operations**, **12.6.1.2143 REPRESENT**.

12.6.2.1515 FEXP "f-e-x-p" FLOATING EXT

(F: r_1 – r_2) or (r_1 – r_2)

Raise e to the power r_1, giving r_2.

12.6.2.1516 FEXPM1 "f-e-x-p-m-one" FLOATING EXT

(F: r_1 – r_2) or (r_1 – r_2)

Raise e to the power r_1 and subtract one, giving r_2.

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

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

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

Thus we only have two digit accuracy.

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

This is full six digit accuracy.

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

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

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.

Rationale: 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**.

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: **6.1.0750 BASE**, **12.3.2 Floating-point operations**, **12.6.1.2143 REPRESENT**.

12.6.2.1614 `FSIN` "f-sine" FLOATING EXT

(F: r_1 – r_2) or (r_1 – r_2)

r_2 is the sine of the radian angle r_1.

12.6.2.1616 `FSINCOS` "f-sine-cos" FLOATING EXT

(F: r_1 – r_2 r_3) or (r_1 – r_2 r_3)

r_2 is the sine of the radian angle r_1. r_3 is the cosine of the radian angle r_1.

Rationale: See: **A.12.6.2.1489 FATAN2**.

12.6.2.1617 `FSINH` "f-cinch" FLOATING EXT

(F: r_1 – r_2) or (r_1 – r_2)

r_2 is the hyperbolic sine of r_1.

12.6.2.1618 `FSQRT` "f-square-root" FLOATING EXT

(F: r_1 – r_2) or (r_1 – r_2)

r_2 is the square root of r_1. An ambiguous condition exists if r_1 is less than zero.

12.6.2.1625 `FTAN` "f-tan" FLOATING EXT

(F: r_1 – r_2) or (r_1 – r_2)

r_2 is the tangent of the radian angle r_1. An ambiguous condition exists if $\cos(r_1)$ is zero.

12.6.2.1626 `FTANH` "f-tan-h" FLOATING EXT

(F: r_1 – r_2) or (r_1 – r_2)

r_2 is the hyperbolic tangent of r_1.

12.6.2.1640 `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.

Rationale: This provides the three types of "floating point equality" in common use — "close" in absolute terms, exact equality as represented, and "relatively close".

12.6.2.2035 `PRECISION` FLOATING EXT

(– *u*)

Return the number of significant digits currently used by **`F`**, **`FE`**, or **`FS`** as *u*.

12.6.2.2200 `SET-PRECISION` FLOATING EXT

(*u* –)

Set the number of significant digits currently used by **`F`**, **`FE`**, or **`FS`** to *u*.

12.6.2.2202 `SF!` "s-f-store" FLOATING EXT

(*sf-addr* –) (F: *r* –) or (*r sf-addr* –)

Store the floating-point number *r* as a 32-bit IEEE single-precision number at *sf-addr*. If the significand of the internal representation of *r* has more precision than the IEEE single-precision format, it will be rounded using the "round to nearest" rule. An ambiguous condition exists if the exponent of *r* is too large to be accommodated by the IEEE single-precision format.

See: **12.3.1.1 Addresses**, **12.3.2 Floating-point operations**.

12.6.2.2203 `SF@` "s-f-fetch" FLOATING EXT

(*sf-addr* –) (F: – *r*) or (*sf-addr* – *r*)

Fetch the 32-bit IEEE single-precision number stored at *sf-addr* to the floating-point stack as *r* in the internal representation. If the IEEE single-precision significand has more precision than the internal representation, it will be rounded to the internal representation using the "round to nearest" rule. An ambiguous condition exists if the exponent of the IEEE single-precision representation is too large to be accommodated by the internal representation.

See: **12.3.1.1 Addresses**, **12.3.2 Floating-point operations**.

12.6.2.2204 `SFALIGN` "s-f-align" FLOATING EXT

(–)

If the data-space pointer is not single-float aligned, reserve enough data space to make it so.

See: **12.3.1.1 Addresses**.

12.6.2.2206 `SFALIGNED` "s-f-aligned" FLOATING EXT

(*addr* – *sf-addr*)

sf-addr is the first single-float-aligned address greater than or equal to *addr*.

See: **12.3.1.1 Addresses**.

12.6.2.2207 `SFLOAT+` "s-float-plus" FLOATING EXT

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

Add the size in address units of a 32-bit IEEE single-precision number to $sf\text{-}addr_1$, giving $sf\text{-}addr_2$.

See: **12.3.1.1 Addresses**.

12.6.2.2208 `SFLOATS` "s-floats" FLOATING EXT

(n_1 – n_2)

n_2 is the size in address units of n_1 32-bit IEEE single-precision numbers.

See: **12.3.1.1 Addresses**.

13 The optional Locals word set

13.1 Introduction

See: **A.13 Glossary**.

13.2 Additional terms and notation

None.

13.3 Additional usage requirements

13.3.1 Locals

A local is a data object whose execution semantics shall return its value, whose scope shall be limited to the definition in which it is declared, and whose use in a definition shall not preclude reentrancy or recursion.

13.3.2 Environmental queries

Append table 13.1 to table 3.5.

See: **3.2.6 Environmental queries**.

Table 13.1: Environmental Query Strings

String	Value data type	Constant?	Meaning
`#LOCALS`	*n*	yes	maximum number of local variables in a definition
`LOCALS`	*flag*	no	locals word set present
`LOCALS-EXT`	*flag*	no	locals extensions word set present

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 eight 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 eight locals in a single definition has an environmental dependency;

g) Locals may be accessed or updated within control structures, including do-loops;

h) Local names shall not be referenced by **POSTPONE** and **[COMPILE]**.

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

13.4 Additional documentation requirements

13.4.1 System documentation

13.4.1.1 Implementation-defined options

– maximum number of locals in a definition (**13.3.3 Processing locals**, **13.6.2.1795 LOCALS|**).

13.4.1.2 Ambiguous conditions

– executing a named *local* while in interpretation state (**13.6.1.0086 (LOCAL)**);

x:to

– ~~*name* not defined by **VALUE** or **(LOCAL)** (**13.6.1.2295** TO).~~

13.4.1.3 Other system documentation

– no additional requirements.

13.4.2 Program documentation

13.4.2.1 Environmental dependencies

– declaring more than eight locals in a single definition (**13.3.3 Processing locals**).

13.4.2.2 Other program documentation

– no additional requirements.

13.5 Compliance and labeling

13.5.1 ANS Forth 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 ANS Forth programs

The phrase “Requiring the Locals word set” shall be appended to the label of Standard Programs that require the system to provide the Locals word set.

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

The phrase “Requiring the Locals Extensions word set” shall be appended to the label of Standard Programs that require the system to provide all of the Locals and Locals Extensions word sets.

13.6 Glossary

13.6.1 Locals words

13.6.1.0086 `(LOCAL)` "paren-local-paren" LOCAL

Interpretation: Interpretation semantics for this word are undefined.

Execution: (*c-addr u* –)

When executed during compilation, **(LOCAL)** passes a message to the system that has one of two meanings. If *u* is non-zero, the message identifies a new *local* whose definition name is given by the string of characters identified by *c-addr u*. If *u* is zero, the message is "last local" and *c-addr* has no significance.

The result of executing **(LOCAL)** during compilation of a definition is to create a set of named local identifiers, each of which is a definition name, that only have execution semantics within the scope of that definition's source.

local Execution: (– *x*)

Push the local's value, *x*, onto the stack. The local's value is initialized as described in **13.3.3 Processing locals** and may be changed by preceding the local's name with **TO**. An ambiguous condition exists when local is executed while in interpretation state.

Note: This word does not have special compilation semantics in the usual sense because it provides access to a system capability for use by other user-defined words that do have them. However, the locals facility as a whole and the sequence of messages passed defines specific usage rules with semantic implications that are described in detail in section **13.3.3 Processing locals**.

Note: This word is not intended for direct use in a definition to declare that definition's locals. It is instead used by system or user compiling words. These compiling words in turn define their own syntax, and may be used directly in definitions to declare locals. In this context, the syntax for **(LOCAL)** is defined in terms of a sequence of compile-time messages and is described in detail in section **13.3.3 Processing locals**.

Note: ~~The Locals word set modifies the syntax and semantics of **6.2.2295** TO as defined in the Core Extensions word set.~~ x:to

TO *local* Run-time: (*x* –)

Associate the value *x* with the local value *local*.

See: **3.4 The Forth text interpreter** and **6.2.2295** TO.

13.6.1.2295 `TO` LOCAL

~~Extend the semantics of **6.2.2295** TO to be:~~ x:to

Interpretation: (~~*x "⟨spaces⟩name"*~~ –)

~~Skip leading spaces and parse *name* delimited by a space. Store *x* in *name*. An ambiguous condition exists if *name* was not defined by **VALUE**.~~

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 either **VALUE** or **(LOCAL)**.~~

Run-time: (~~*x*~~ –)

~~Store *x* in *name*.~~

Note: ~~An ambiguous condition exists if either **POSTPONE** or **[COMPILE]** is applied to **TO**.~~

See: **3.4.1 Parsing**, ~~**6.2.2295** TO~~, ~~**6.2.2405** VALUE~~, ~~**13.6.1.0086** (LOCAL)~~.

Rationale: ~~Typical use: x **TO** name~~

x:to ~~See: **A.6.2.2295** TO~~.

13.6.2 Locals extension words

13.6.2.1795 `LOCALS|` "locals-bar" LOCAL EXT

Interpretation: Interpretation semantics for this word are undefined.

Compilation: ("⟨*spaces*⟩*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 ~~**13.6.1.2295** TO~~**6.2.2295** TO

x:to .

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

14 The optional Memory-Allocation word set

14.1 Introduction

14.2 Additional terms and notation

None.

14.3 Additional usage requirements

14.3.1 I/O Results data type

I/O results are single-cell numbers indicating the result of I/O operations. A value of zero indicates that the I/O operation completed successfully; other values and their meanings are implementation-defined.

Append table 14.1 to table 3.1.

Table 14.1: Data types

Symbol	*Data type*	*Size on stack*
ior	I/O results	1 cell

14.3.2 Environmental queries

Append table 14.2 to table 3.5.

See: **3.2.6 Environmental queries**.

Table 14.2: Environmental Query Strings

String	Value data type	Constant?	Meaning
`MEMORY-ALLOC`	*flag*	no	memory-allocation word set present
`MEMORY-ALLOC-EXT`	*flag*	no	memory-allocation extensions word set present

14.3.3 Allocated regions

A program may address memory in data space regions made available by **`ALLOCATE`** or **`RESIZE`** and not yet released by **`FREE`**.

See: **3.3.3 Data space**.

14.4 Additional documentation requirements

14.4.1 System documentation

14.4.1.1 Implementation-defined options

– values and meaning of ior (**14.3.1 I/O Results data type**, **14.6.1.0707 `ALLOCATE`**, **14.6.1.1605 `FREE`**, **14.6.1.2145 `RESIZE`**.

14.4.1.2 Ambiguous conditions

– no additional requirements.

14.4.1.3 Other system documentation

– no additional requirements.

14.4.2 Program documentation

– no additional requirements.

14.5 Compliance and labeling

14.5.1 ANS Forth 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 ANS Forth programs

The phrase "Requiring the Memory-Allocation word set" shall be appended to the label of Standard Programs that require the system to provide the Memory-Allocation word set.

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

The phrase "Requiring the Memory-Allocation Extensions word set" shall be appended to the label of Standard Programs that require the system to provide all of the Memory-Allocation and Memory-Allocation Extensions word sets.

14.6 Glossary

14.6.1 Memory-Allocation words

14.6.1.0707 `ALLOCATE` MEMORY

(*u* – *a-addr ior*)

Allocate *u* address units of contiguous data space. The data-space pointer is unaffected by this operation. The initial content of the allocated space is undefined.

If the allocation succeeds, *a-addr* is the aligned starting address of the allocated space and *ior* is zero.

If the operation fails, *a-addr* does not represent a valid address and *ior* is the implementation-defined I/O result code.

See: **6.1.1650 `HERE`**, **14.6.1.1605 `FREE`**, **14.6.1.2145 `RESIZE`**.

14.6.1.1605 FREE MEMORY

(*a-addr* – *ior*)

Return the contiguous region of data space indicated by *a-addr* to the system for later allocation. *a-addr* shall indicate a region of data space that was previously obtained by **ALLOCATE** or **RESIZE**. The data-space pointer is unaffected by this operation.

If the operation succeeds, *ior* is zero. If the operation fails, *ior* is the implementation-defined I/O result code.

See: **6.1.1650 HERE**, **14.6.1.0707 ALLOCATE**, **14.6.1.2145 RESIZE**.

14.6.1.2145 RESIZE MEMORY

($a\text{-}addr_1$ *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: **6.1.1650 HERE**, **14.6.1.0707 ALLOCATE**, **14.6.1.1605 FREE**.

14.6.2 Memory-Allocation extension words

None

15 The optional Programming-Tools word set

15.1 Introduction

This optional word set contains words most often used during the development of applications.

15.2 Additional terms and notation

None.

15.3 Additional usage requirements

15.3.1 Environmental queries

Append table 15.1 to table 3.5.

See: **3.2.6 Environmental queries**.

Table 15.1: Environmental Query Strings

String	Value data type	Constant?	Meaning
`TOOLS`	*flag*	no	programming-tools word set present
`TOOLS-EXT`	*flag*	no	programming-tools extensions word set present
`X:defined`	–	–	the X:defined extension is present

15.3.2 The Forth dictionary

A program using the words **CODE** or **;CODE** associated with assembler code has an environmental dependency on that particular instruction set and assembler notation.

Programs using the words **EDITOR** or **ASSEMBLER** require the Search Order word set or an equivalent implementation-defined capability.

See: **3.3 The Forth dictionary**.

15.4 Additional documentation requirements

15.4.1 System documentation

15.4.1.1 Implementation-defined options

- ending sequence for input following **15.6.2.0470 ;CODE** and **15.6.2.0930 CODE**;
- manner of processing input following **15.6.2.0470 ;CODE** and **15.6.2.0930 CODE**;
- search-order capability for **15.6.2.1300 EDITOR** and **15.6.2.0740 ASSEMBLER** (**15.3.2 The Forth dictionary**);
- source and format of display by **15.6.1.2194 SEE**.

15.4.1.2 Ambiguous conditions

- deleting the compilation word-list (**15.6.2.1580 FORGET**);

- fewer than $u+1$ items on control-flow stack (**15.6.2.1015 CS-PICK**, **15.6.2.1020 CS-ROLL**);
- *name* can't be found (**15.6.2.1580 FORGET**);
- *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**).

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 ANS Forth 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 ANS Forth programs

The phrase "Requiring the Programming-Tools word set" shall be appended to the label of Standard Programs that require the system to provide the Programming-Tools word set.

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

The phrase "Requiring the Programming-Tools Extensions word set" shall be appended to the label of Standard Programs that require the system to provide all of the Programming-Tools and Programming-Tools Extensions word sets.

15.6 Glossary

15.6.1 Programming-Tools words

15.6.1.0220 `.S` "dot-s" TOOLS

(–)

Copy and display the values currently on the data stack. The format of the display is implementation-dependent.

.S may be implemented using pictured numeric output words. Consequently, its use may corrupt the transient region identified by **#>**.

See: **3.3.3.6 Other transient regions**.

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

15.6.1.0600 `?` "question" TOOLS

(*a-addr* –)

Display the value stored at *a-addr*.

? may be implemented using pictured numeric output words. Consequently, its use may corrupt the transient region identified by **#>**.

See: **3.3.3.6 Other transient regions**.

15.6.1.1280 `DUMP` TOOLS

(*addr u* –)

Display the contents of *u* consecutive addresses starting at *addr*. The format of the display is implementation dependent.

DUMP may be implemented using pictured numeric output words. Consequently, its use may corrupt the transient region identified by **#>**.

See: **3.3.3.6 Other transient regions**.

15.6.1.2194 `SEE` TOOLS

(*"⟨spaces⟩name"* –)

Display a human-readable representation of the named word's definition. The source of the representation (object-code decompilation, source block, etc.) and the particular form of the display is implementation defined.

SEE may be implemented using pictured numeric output words. Consequently, its use may corrupt the transient region identified by **#>**.

See: **3.3.3.6 Other transient regions**.

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

15.6.1.2465 `WORDS` TOOLS

(–)

List the definition names in the first word list of the search order. The format of the display is implementation-dependent.

WORDS may be implemented using pictured numeric output words. Consequently, its use may corrupt the transient region identified by **#>**.

See: **3.3.3.6 Other transient regions**.

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

15.6.2 Programming-Tools extension words

15.6.2.0470 **;** `CODE` "semicolon-code" TOOLS EXT

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: *colon-sys* –)

Append the run-time semantics below to the current definition. End the current definition, allow it to be found in the dictionary, and enter interpretation state, consuming *colon-sys*.

Subsequent characters in the parse area typically represent source code in a programming language, usually some form of assembly language. Those characters are processed in an implementation-defined manner, generating the corresponding machine code. The process continues, refilling the input buffer as needed, until an implementation-defined ending sequence is processed.

Run-time: (–) (R: *nest-sys* –)

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

name Execution: (*i*×*x* – *j*×*x*)

Perform the machine code sequence that was generated following **;CODE**.

See: **6.1.1250 DOES>**.

Rationale: Typical use: **:** `namex` ... ⟨*create*⟩ ... **;CODE** ...

where `namex` is a defining word, and ⟨*create*⟩ is **CREATE** or any user defined word that calls **CREATE**.

15.6.2.0702 `AHEAD` TOOLS EXT

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: – *orig*)

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

Run-time: (–)

Continue execution at the location specified by the resolution of *orig*.

15.6.2.0740 ASSEMBLER TOOLS EXT

(–)

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

See: **16 The optional Search-Order word set**.

15.6.2.0830 BYE TOOLS EXT

(–)

Return control to the host operating system, if any.

15.6.2.0930 CODE TOOLS EXT

("⟨*spaces*⟩*name*" –)

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

Subsequent characters in the parse area typically represent source code in a programming language, usually some form of assembly language. Those characters are processed in an implementation-defined manner, generating the corresponding machine code. The process continues, refilling the input buffer as needed, until an implementation-defined ending sequence is processed.

name Execution: ($i \times x$ – $j \times x$)

Execute the machine code sequence that was generated following **CODE**.

See: **3.4.1 Parsing**.

Rationale: 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.

15.6.2.1015 CS-PICK "c-s-pick" TOOLS EXT

Interpretation: Interpretation semantics for this word are undefined.

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

Remove u. Copy $dest_u$ to the top of the control-flow stack. An ambiguous condition exists if there are less than u+1 items, each of which shall be an *orig* or *dest*, on the control-flow stack before **CS-PICK** is executed.

If the control-flow stack is implemented using the data stack, u shall be the topmost item on the data stack.

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

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

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.

Rationale: 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
```

15.6.2.1300 EDITOR TOOLS EXT

(-)

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

See: **16 The optional Search-Order word set**.

15.6.2.1580 FORGET TOOLS EXT

("⟨*spaces*⟩*name*" -)

Skip leading space delimiters. Parse *name* delimited by a space. Find *name*, then delete *name* from the dictionary along with all words added to the dictionary after *name*. An ambiguous condition exists if *name* cannot be found.

If the Search-Order word set is present, **FORGET** searches the compilation word list. An ambiguous condition exists if the compilation word list is deleted.

An ambiguous condition exists if **FORGET** removes a word required for correct execution. Note: This word is obsolescent and is included as a concession to existing implementations.

See: **3.4.1 Parsing**.

Rationale: Typical use: ... **FORGET** *name* ...

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

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: **3.4 The Forth text interpreter**, **6.1.0450 :**, **6.1.0460 ;**, **6.1.0670 ABORT**, **6.1.2050 QUIT**, **6.1.2250 STATE**, **6.1.2500 [**, **6.1.2540]**, **6.2.0455 :NONAME**, **15.6.2.0470 ;CODE**.

15.6.2.—— [DEFINED] "bracket-defined" TOOLS EXT
X:defined

Compilation: Perform the execution semantics given below.

Execution: ("⟨*spaces*⟩*name* ..." – *flag*)

Skip leading space delimiters. Parse name delimited by a space. Return a true flag if *name* is the name of a word that can be found (according to the rules in the system's **FIND**); otherwise return a false flag. **[DEFINED]** is an immediate word.

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

15.6.2.2531 [ELSE] "bracket-else" TOOLS EXT

Compilation: Perform the execution semantics given below.

Execution: ("⟨*spaces*⟩*name* ..." –)

Skipping leading spaces, parse and discard space-delimited words from the parse area, including nested occurrences of **[IF]** ... **[THEN]** and **[IF]** ... **[ELSE]** ... **[THEN]**, until the word **[THEN]** has been parsed and discarded. If the parse area becomes exhausted, it is refilled as with **REFILL**. **[ELSE]** is an immediate word.

See: **3.4.1 Parsing**.

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

15.6.2.2532 `[IF]` "bracket-if" TOOLS EXT

Compilation: Perform the execution semantics given below.

Execution: (*flag* | *flag* "⟨*spaces*⟩*name* ... " –)

If *flag* is true, do nothing. Otherwise, skipping leading spaces, parse and discard space-delimited words from the parse area, including nested occurrences of **[IF]** ... **[THEN]** and **[IF]** ... **[ELSE]** ... **[THEN]**, until either the word **[ELSE]** or the word **[THEN]** has been parsed and discarded. If the parse area becomes exhausted, it is refilled as with **REFILL**. **[IF]** is an immediate word.

An ambiguous condition exists if **[IF]** is **POSTPONE**d, or if the end of the input buffer is reached and cannot be refilled before the terminating **[ELSE]** or **[THEN]** is parsed.

See: **3.4.1 Parsing**.

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

15.6.2.2533 `[THEN]` "bracket-then" TOOLS EXT

Compilation: Perform the execution semantics given below.

Execution: (–)

Does nothing. **[THEN]** is an immediate word.

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

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

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

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

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

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

```
:  [ELSE] ( – )
  1 BEGIN                                  \ level
    BEGIN  BL WORD COUNT  DUP  WHILE       \ level adr len
      2DUP S" [IF]" COMPARE 0= IF          \ level adr len
        2DROP 1+                           \ level'
      ELSE                                 \ level adr len
        2DUP S" [ELSE]" COMPARE 0= IF      \ level adr len
```

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

15.6.2.—— `[UNDEFINED]` "bracket-undefined" TOOLS EXT
X:defined

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.

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

16 The optional Search-Order word set

16.1 Introduction

16.2 Additional terms and notation

compilation word list: The word list into which new definition names are placed.

search order: A list of word lists specifying the order in which the dictionary will be searched.

16.3 Additional usage requirements

16.3.1 Data types

Word list identifiers are implementation-dependent single-cell values that identify word lists.

Append table 16.1 to table 3.1.

Table 16.1: Data types

Symbol	*Data type*	*Size on stack*
wid	word list identifiers	1 cell

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

16.3.2 Environmental queries

Append table 16.2 to table 3.5.

See: **3.2.6 Environmental queries**.

Table 16.2: Environmental Query Strings

String	Value data type	Constant?	Meaning
`SEARCH-ORDER`	*flag*	no	search-order word set present
`SEARCH-ORDER-EXT`	*flag*	no	search-order extensions word set present
`WORDLISTS`	*n*	yes	maximum number of word lists usable in the search order

16.3.3 Finding definition names

When searching a word list for a definition name, the system shall search each word list from its last definition to its first. The search may encompass only a single word list, as with **`SEARCH-WORDLIST`**, or all the word lists in the search order, as with the text interpreter and **`FIND`**.

Changing the search order shall only affect the subsequent finding of definition names in the dictionary. A system with the Search-Order word set shall allow at least eight word lists in the search order.

An ambiguous condition exists if a program changes the compilation word list during the compilation of a definition or before modification of the behavior of the most recently compiled definition with **`;CODE`**, **`DOES>`**, or **`IMMEDIATE`**.

A program that requires more than eight word lists in the search order has an environmental dependency.

See: **3.4.2 Finding definition names**.

16.3.4 Contiguous regions

The regions of data space produced by the operations described in **3.3.3.2 Contiguous regions** may be non-contiguous if **WORDLIST** is executed between allocations.

16.4 Additional documentation requirements

16.4.1 System documentation

16.4.1.1 Implementation-defined options

- maximum number of word lists in the search order (**16.3.3 Finding definition names**, **16.6.1.2197 SET-ORDER**);
- minimum search order (**16.6.1.2197 SET-ORDER**, **16.6.2.1965 ONLY**).

16.4.1.2 Ambiguous conditions

- changing the compilation word list (**16.3.3 Finding definition names**);
- search order empty (**16.6.2.2037 PREVIOUS**);
- too many word lists in search order (**16.6.2.0715 ALSO**).

16.4.1.3 Other system documentation

- no additional requirements.

16.4.2 Program documentation

16.4.2.1 Environmental dependencies

- requiring more than eight word-lists in the search order (**16.3.3 Finding definition names**).

16.4.2.2 Other program documentation

- no additional requirements.

16.5 Compliance and labeling

16.5.1 ANS Forth 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 ANS Forth programs

The phrase "Requiring the Search-Order word set" shall be appended to the label of Standard Programs that require the system to provide the Search-Order word set.

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

The phrase "Requiring the Search-Order Extensions word set" shall be appended to the label of Standard Programs that require the system to provide all of the Search-Order and Search-Order Extensions word sets.

16.6 Glossary

16.6.1 Search-Order words

16.6.1.1180 `DEFINITIONS` SEARCH

(–)

Make the compilation word list the same as the first word list in the search order. Specifies that the names of subsequent definitions will be placed in the compilation word list. Subsequent changes in the search order will not affect the compilation word list.

See: **16.3.3 Finding definition names**.

16.6.1.1550 `FIND` SEARCH

Extend the semantics of **6.1.1550 FIND** to be:

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

Find the definition named in the counted string at *c-addr*. If the definition is not found after searching all the word lists in the search order, return *c-addr* and zero. If the definition is found, return *xt*. If the definition is immediate, also return one (*1*); otherwise also return minus-one (*-1*). For a given string, the values returned by **FIND** while compiling may differ from those returned while not compiling.

See: **3.4.2 Finding definition names**, **6.1.0070 '**, **6.1.1550 FIND**, **6.1.2033 POSTPONE**, **6.1.2510 ['']**, **D.6.7 Immediacy**.

16.6.1.1595 `FORTH-WORDLIST` SEARCH

(– *wid*)

Return *wid*, the identifier of the word list that includes all standard words provided by the implementation. This word list is initially the compilation word list and is part of the initial search order.

16.6.1.1643 `GET-CURRENT` SEARCH

(– *wid*)

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

16.6.1.1647 GET-ORDER SEARCH

(– $wid_n \ldots wid_1\ n$)

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.

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.

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

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

16.6.1.2195 SET-CURRENT SEARCH

(*wid* –)

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

16.6.1.2197 SET-ORDER SEARCH

($wid_n \ldots wid_1\ n$ –)

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.

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.

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

```
WORDLIST CONSTANT ROOT   ROOT SET-CURRENT
:  DO-VOCABULARY  ( – ) \ Implementation factor
  DOES>  @ >R          ( ) ( R: widnew )
   GET-ORDER SWAP DROP ( wid1 ...  widn-1 n )
   R> SWAP SET-ORDER
;
:  DISCARD  ( x1 ...  xu u – ) \ Implementation factor
  0 ?DO DROP LOOP             \ DROP u+1 stack items
;
CREATE FORTH   FORTH-WORDLIST , DO-VOCABULARY
:  VOCABULARY    ( name – ) WORDLIST CREATE ,   DO-VOCABULARY
;
:  ALSO          ( – ) GET-ORDER   OVER SWAP 1+ SET-ORDER
;
:  PREVIOUS      ( – ) GET-ORDER   SWAP DROP 1- SET-ORDER
;
:  DEFINITIONS   ( – ) GET-ORDER   OVER SET-CURRENT DISCARD
;
:  ONLY          ( – ) ROOT ROOT   2 SET-ORDER ;
\ Forth-83 version; just removes ONLY
:  SEAL ( – ) GET-ORDER 1- SET-ORDER DROP ;
\ F83 and F-PC version; leaves only CONTEXT
:  SEAL ( – ) GET-ORDER OVER 1 SET-ORDER DISCARD ;
```

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

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

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

```
16 CONSTANT #VOCS
VARIABLE #ORDER
CREATE CONTEXT   #VOCS CELLS ALLOT
:  GET-ORDER  ( – wid1 ...  widn n )
  #ORDER @ 0 ?DO
```

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

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

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, $wid_{\text{FORTH-WORDLIST}}$.

16.6.2.1965 ONLY SEARCH EXT

(-)

Set the search order to the implementation-defined minimum search order. The minimum search order shall include the words **FORTH-WORDLIST** and **SET-ORDER**.

16.6.2.1985 ORDER SEARCH EXT

(-)

Display the word lists in the search order in their search order sequence, from first searched to last searched. Also display the word list into which new definitions will be placed. The display format is implementation dependent.

ORDER may be implemented using pictured numeric output words. Consequently, its use may corrupt the transient region identified by **#>**.

See: **3.3.3.6 Other transient regions**.

16.6.2.2037 PREVIOUS SEARCH EXT

(–)

Transform the search order consisting of wid_n, ... wid_2, wid_1 (where wid_1 is searched first) into wid_n, ... wid_2. An ambiguous condition exists if the search order was empty before **PREVIOUS** was executed.

17 The optional String word set

17.1 Introduction

17.2 Additional terms and notation

None.

17.3 Additional usage requirements

Append table 17.1 to table 3.5.

See: **3.2.6 Environmental queries**.

Table 17.1: Environmental Query Strings

String	Value data type	Constant?	Meaning
`STRING`	*flag*	no	string word set present
`STRING-EXT`	*flag*	no	string extensions word set present

17.4 Additional documentation requirements

None.

17.5 Compliance and labeling

17.5.1 Compliance and labeling

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 ANS Forth 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.

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.

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

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*.

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.

Contrast with: **17.6.1.0920 CMOVE>**.

Rationale: 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.

Typical use: Assume a character string at address 100: "ABCD". Then after

```
100 DUP    CHAR+    3 CMOVE
```

the string at address 100 is "AAAA".

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

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.

Contrast with:: **17.6.1.0910 CMOVE**.

Rationale: 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.

Typical use: Assume a character string at address 100: "ABCD". Then after

```
100 DUP CHAR+ SWAP 3 CMOVE>
```

the string at address 100 is "DDDD".

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.

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

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*$_1$.

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

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.

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

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

17.6.2 String extension words

None.

Annex A
(informative)
Rationale

A.1 Introduction

A.1.1 Purpose

A.1.2 Scope

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

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

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

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

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

A.1.3 Document organization

A.1.3.1 Word sets

From the beginning, the X3J14 Technical Committee faced not only conflicting ideas as to what "real" Forth is, but also conflicting needs of the various groups within the Forth community. At one extreme were those who pressed for a "bare" Forth. At the other extreme were those who wanted a "fat" Forth.

Many were somewhere in between. All were convinced of the rightness of their own position and of the wrongness of at least one of the two extremes. The committee's composition reflected this full range of interests.

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

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

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

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

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

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

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

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

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

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

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

A.2 Terms and notation

A.2.1 Definitions of terms

ambiguous condition

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

cross compiler

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

data field

In earlier standards, data fields were known as "parameter fields".

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

word set

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

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

A.2.2 Notation

A.2.2.1 Stack notation

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

A.3 Usage requirements

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

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

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

A.3.1 Data-types

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

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

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

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

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

A.3.1.2 Character types

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

a) **Standard Character Set**

1) The storage unit for the character data type (**C@**, **C!**, **FILL**, etc.) must be able to contain unsigned numbers from 0 through 255.
2) An implementation is not required to restrict character storage to that range, but a Standard Program without environmental dependencies cannot assume the ability to store numbers outside that range in a "char" location.
3) The allowed number representations are two's-complement, one's-complement, and signed-magnitude. Note that all of these number systems agree on the representation of positive numbers.
4) Since a "*char*" can store small positive numbers and since the character data type is a subrange of the unsigned integer data type, **C!** must store the *n* least-significant bits of a cell ($8 <= n <=$ bits/cell). Given the enumeration of allowed number representations and their known encodings, "**TRUE** xx **C!** xx **C@**" must leave a stack item with some number of bits set, which will thus will be accepted as non-zero by **IF**.
5) For the purposes of input (**KEY**, **ACCEPT**, etc.) and output (**EMIT**, **TYPE**, etc.), the encoding between numbers and human-readable symbols is ISO646/IRV (ASCII) within the range from 32 to 126 (space to ~). EBCDIC is out (most "EBCDIC" computer systems support ASCII too). Outside that range, it is up to the implementation. The obvious implementation choice is to use ASCII control characters for the range from 0 to 31, at least for the "displayable" characters in that range (TAB, RETURN, LINEFEED, FORMFEED). However, this is not as clear-cut as it may seem, because of the variation between operating systems on the treatment of those characters. For example, some systems TAB to 4 character boundaries, others to 8 character boundaries, and others to preset tab stops. Some systems perform an automatic linefeed after

a carriage return, others perform an automatic carriage return after a linefeed, and others do neither.

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

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

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

b) **Storage and retrieval**

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

c) **Manipulation on the stack**

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

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

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

The trend in ANS Forth is to move toward the consistent use of the "*c-addr u*" representation of strings on the stack. The use of the alternate "address of counted string" stack representation is discouraged. The traditional Forth words **WORD** and **FIND** continue to use the "address of counted string" representation for historical reasons. The new word **C"**, added as a porting aid for existing programs, also uses the counted string representation.

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

A.3.1.3.5 Execution tokens

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

```
' 1+ and
' CHAR+
```

might return the same value.

A.3.1.4 Cell-pair types

a) **Storage and retrieval**

Two operators are provided to fetch and store cell pairs:

```
2@ 2!
```

b) **Manipulation on the stack**

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

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

c) **Comparison**

The following comparison operations are universally valid for cell pairs:

D= D0=

A.3.1.4.1 Double-Cell Integers

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

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

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

D+ D- UM/MOD DU<

A.3.1.4.2 Character strings

See: **A.3.1.3.4 Counted strings**.

A.3.2 The Implementation environment

A.3.2.1 Numbers

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

As a consequence of these behaviors, the likely ranges of signed and unsigned numbers for implementations hosted on each of the permissible arithmetic architectures is:

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

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

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

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

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

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

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

A.3.2.1.2 Digit conversion

For example, an implementation might convert the characters "a" through "z" identically to the characters "A" through "Z", or it might treat the characters " [" through "~" as additional digits with decimal values 36 through 71, respectively.

A.3.2.2 Arithmetic

A.3.2.2.1 Integer division

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

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

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

A.3.2.3 Stacks

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

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

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

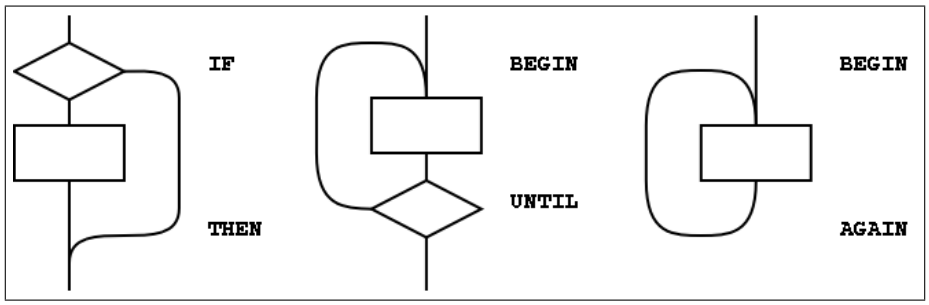

Figure A.1: The basic control-flow patterns

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

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

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

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

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

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

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

```
:  WHILE  ( dest - orig dest / flag - )
  \ conditional exit from loops
  POSTPONE IF       \ conditional forward brach
```

```
  1 CS-ROLL          \ keep dest on top
; IMMEDIATE

:  REPEAT  ( orig dest - / - )
  \ resolve a single WHILE and return to BEGIN
  POSTPONE AGAIN     \ uncond.  backward branch to dest
  POSTPONE THEN      \ resolve forward branch from orig
; IMMEDIATE

:  ELSE  ( orig1 - orig2 / - )
  \ resolve IF supplying alternate execution
  POSTPONE AHEAD     \ unconditional forward branch orig2
  1 CS-ROLL          \ put orig1 back on top
  POSTPONE THEN      \ resolve forward branch from orig1
; IMMEDIATE
```

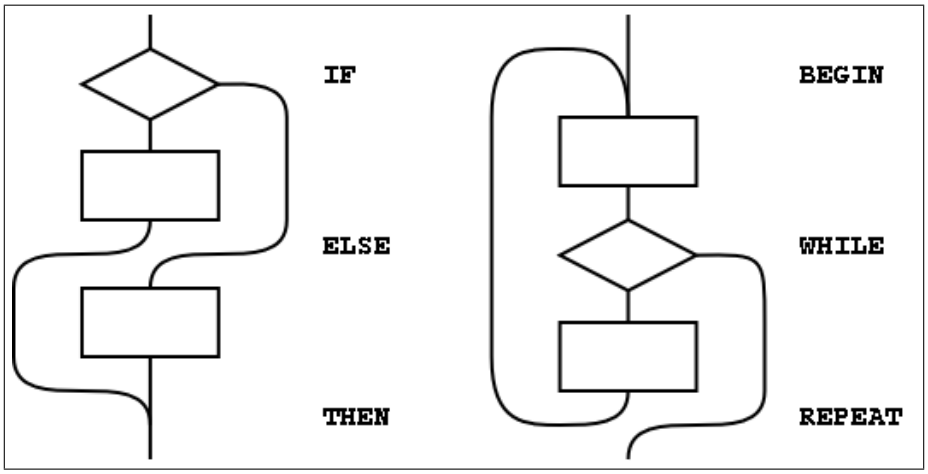

Figure A.2: Additional basic control-flow patterns

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

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

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

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

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

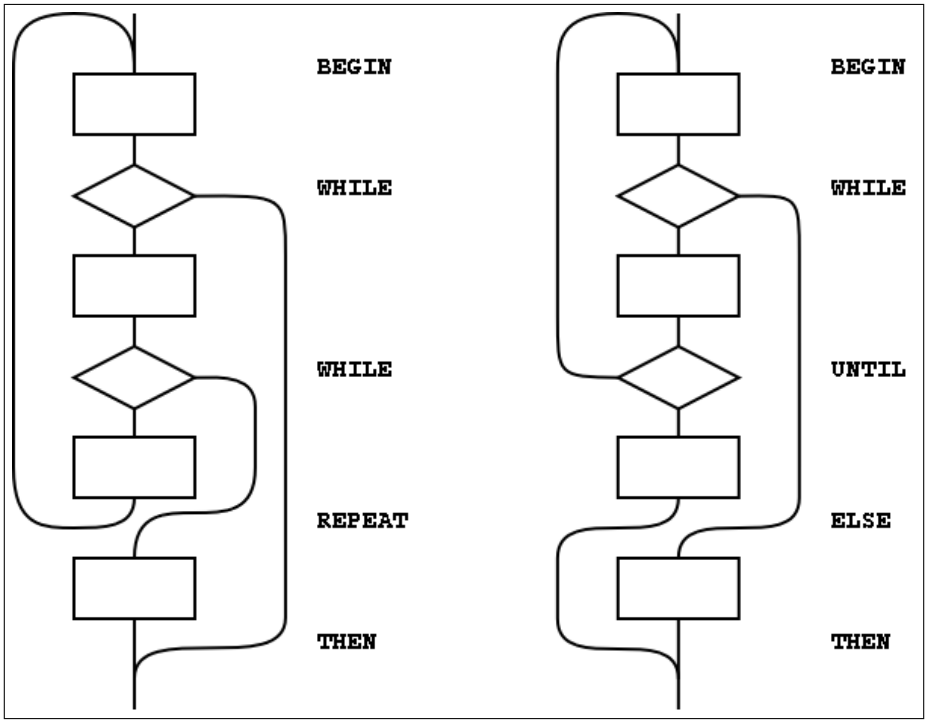

Figure A.3: Extended control-flow patterns

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

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

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

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

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

```
; IMMEDIATE
```

A.3.2.3.3 Return stack

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

A.3.2.4 Environmental queries

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

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

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

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

A.3.3 The Forth dictionary

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

A.3.3.1 Name space

A.3.3.1.2 Definition names

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

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

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

A.3.3.2 Code space

A.3.3.3 Data space

The words **`#TIB`**, **`>IN`**, **`BASE`**, **`BLK`**, **`SCR`**, **`SOURCE`**, **`SOURCE-ID`**, **`STATE`**, and **`TIB`** contain information used by the Forth system in its operation and may be of use to the application. Any assumption made by

the application about data available in the Forth system it did not store other than the data just listed is an environmental dependency.

There is no point in specifying (in the Standard) both what is and what is not addressable. A Standard Program may NOT address:

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

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

A.3.3.3.1 Address alignment

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

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

A.3.3.3.2 Contiguous regions

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

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

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

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

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

Similarly,

```
CREATE DATA   1000 ALLOT
```

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

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

This array can be indexed like this:

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

A.3.3.3.6 Other transient regions

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

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

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

A.3.4 The Forth text interpreter

A.3.4.1 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.3.4.1.2 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.3.4.2 Compilation

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

A.4 Documentation requirements

A.4.1 System documentation

A.4.2 Program documentation

A.5 Compliance and labeling

A.5.1 ANS Forth systems

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

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

A.5.2 ANS Forth programs

A.5.2.2 Program labeling

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

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

On the other hand, as a program requires increasing levels of resources, there will probably be sucessively fewer systems on which it will execute sucessfully. An algorithm requiring an array of 10^9 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 an "ANS Forth Program" is unlikely to be labeled correctly.

A.6 Glossary

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

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

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

In the *rationale* (r) version of the document, the rationale text for each of the word sets are given here. While the rationale for the individual words are given in the word's definition.

A.6.2 Core extension words

The words in this collection fall into several categories:

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

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

A.7 The optional Block word set

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

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

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

A.7.2 Additional terms

block

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

A.7.6 Glossary

A.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.9 The optional Exception word set

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

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

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

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

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

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

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

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

```
VARIABLE HANDLER   0 HANDLER !   \ last exception handler
:  CATCH    ( xt -- exception# | 0 ) \ return addr on stack
  SP@ >R         ( xt )       \ save data stack pointer
  HANDLER @ >R   ( xt )       \ and previous handler
```

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

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

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

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

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

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

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

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

In systems with other stacks in addition to the data and return stacks, the implementation of **CATCH** and **THROW** must save and restore those stack pointers as well. Such an "extended version" can be built on top of this basic implementation. For example, with another stack pointer accessed with `FP@` and `FP!` only **CATCH** needs to be redefined:

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

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

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

A.9.3 Additional usage requirements

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

A.9.3.6 Exception handling

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

A.9.6 Glossary

A.10 The optional Facility word set

A.10.6 Glossary

A.11 The optional File-Access word set

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

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

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

Most of the existing implementations studied use simple English words for common host file functions:

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

Specific rationales for members of this word set follow.

A.11.3 Additional usage requirements

A.11.3.2 Blocks in files

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

A.11.6 Glossary

A.12 The optional Floating-Point word set

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

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

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

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

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

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

A.12.3 Additional usage requirements

A.12.3.5 Address alignment

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

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

A.12.3.7 Text interpreter input number conversion

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

A.12.6 Glossary

A.13 The optional Locals word set

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

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

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

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

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

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

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

- The syntax defined by this Standard and used in the systems of Creative Solutions, Inc.:

```
:  LOCALS| ( "name...name |" - )
  BEGIN
    BL WORD   COUNT OVER C@
    [CHAR] | - OVER 1 - OR   WHILE
    (LOCAL)
  REPEAT 2DROP   0 0 (LOCAL)
; IMMEDIATE
:  EXAMPLE ( n - n**2 n**3 )
  LOCALS| N |   N DUP N *  DUP N * ;
```

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

```
:  LOCAL ( "name" - ) BL WORD COUNT (LOCAL) ;  IMMEDIATE
:  END-LOCALS ( - )   0 0 (LOCAL) ;  IMMEDIATE
:  EXAMPLE ( n - n n**2 n**3 )
  LOCAL N  END-LOCALS   N DUP N *   DUP N * ;
```

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

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

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

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

c)      100 300 10 JOE .
```

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

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

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

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

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

```
d)      :  ZERO   0 POSTPONE LITERAL POSTPONE LOCAL ; IMMEDIATE

e)      :  MOE  ( a b )
           ZERO TEMP  LOCAL B  1+ LOCAL A+  ZERO ANSWER ;

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

Here are two definitions with various violations of rule 13.3.3.2a. In e) the declaration of TEMP is legal and creates a local whose initial value is zero. It's OK because the executable code that ZERO generates precedes the first use of **(LOCAL)** in the definition. However, the **1+** preceding the declaration of A+ is illegal. Likewise the use of ZERO to define ANSWER is illegal because it generates executable code between uses of **(LOCAL)**. Finally, MOE terminates illegally (no END-LOCALS). BOB inf) violates the rule against declaring two sets of locals.

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

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

ANN in g) violates two rules. The **IF** ... **THEN** around the declaration of its locals violates 13.3.3.2b, and the copy of B left on the return stack before declaring locals violates 13.3.3.2c. JANE in h) violates 13.3.3.2d by accessing locals after placing the sum of A and B on the return stack without first removing that sum.

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

CHRIS in i) illustrates three violations of 13.3.3.2e. The attempt to **EXECUTE** the local called A is inconsistent with some implementations. The store into B via **>BODY** is likely to cause tragic results with many implementations; moreover, if locals are in registers they can't be addressed as memory no matter what is written.

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

A.13.3 Additional usage requirements

Rule 13.3.3.2d 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.

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

Access to previously declared local variables is prohibited by Section 13.3.3.2d until any data placed onto the return stack by the application has been removed, due to the possible use of the return stack for storage of locals.

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

Access to Locals within **DO**...**LOOP**s is expressly permitted as an additional requirement of conforming systems by Section 13.3.3.2g. Although words, such as **(LOCAL)**, written by a System Implementor, may require inside knowledge of the internal structure of the return stack, such knowledge is not required of a user of compliant Forth systems.

A.13.6 Glossary

A.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 Technical Committee believes that providing a stan-

dard 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.16 The optional Search-Order word set

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

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

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

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

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

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

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

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

A.16.2 Additional terms

search order

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

A.16.3.3 Finding definition names

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

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

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

A.16.6 Glossary

A.17 The optional String word set

A.17.6 Glossary

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. ~~Several members of the Forth Standards Team have also been members of the X3J14 Technical Committee.~~ x:forward

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. *Starting FORTH* (2nd ed). Englewood Cliffs, NJ: Prentice Hall, 1987.

Brodie, L. *Thinking FORTH*. Englewood Cliffs, NJ: Prentice Hall, 1984.

Feierbach, G. and Thomas, P. *Forth Tools & Applications*. Reston, VA: Reston Computer Books, 1985.

Haydon, Dr. Glen B. *All About FORTH, Third 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, Third Edition*. Rochester, New York: Institute of Applied Forth Research, 1987.

McCabe, C. K. *Forth Fundamentals* (2 volumes). Oregon: Dilithium Press, 1983.

Pountain, R. *Object Oriented Forth*. London, England: Academic Press, 1987.

Ouverson, Marlin, editor. *Dr. Dobbs Toolbook of Forth*. Redwood City, CA: M&T Press, Vol. 1, 1986; Vol. 2, 1987.

Terry, J. D. *Library of Forth Routines and Utilities*. New York: Shadow Lawn Press, 1986.

Tracy, M. and Anderson, A. *Mastering FORTH* (revised ed). 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)
Perspective

The purpose of this section is to provide an informal overview of Forth as a language, illustrating its history, most prominent features, usage, and common implementation techniques. Nothing in this section should be considered as binding upon either implementors or users. A list of books and articles is given in Annex B for those interested in learning more about Forth.

C.1 Features of Forth

Forth provides an interactive programming environment. Its primary uses have been in scientific and industrial applications such as instrumentation, robotics, process control, graphics and image processing, artificial intelligence and business applications. The principal advantages of Forth include rapid, interactive software development and efficient use of computer hardware.

Forth is often spoken of as a language because that is its most visible aspect. But in fact, Forth is both more and less than a conventional programming language: more in that all the capabilities normally associated with a large portfolio of separate programs (compilers, editors, etc.) are included within its range and less in that it lacks (deliberately) the complex syntax characteristic of most high-level languages.

The original implementations of Forth were stand-alone systems that included functions normally performed by separate operating systems, editors, compilers, assemblers, debuggers and other utilities. A single simple, consistent set of rules governed this entire range of capabilities. Today, although very fast stand-alone versions are still marketed for many processors, there are also many versions that run co-resident with conventional operating systems such as MS-DOS and UNIX.

Forth is not derived from any other language. As a result, its appearance and internal characteristics may seem unfamiliar to new users. But Forth's simplicity, extreme modularity, and interactive nature offset the initial strangeness, making it easy to learn and use. A new Forth programmer must invest some time mastering its large command repertoire. After a month or so of full-time use of Forth, that programmer could understand more of its internal working than is possible with conventional operating systems and compilers.

The most unconventional feature of Forth is its *extensibility*. The programming process in Forth consists of defining new "words". actually new commands in the language. These may be defined in terms of previously defined words, much as one teaches a child concepts by explaining them in terms of previously understood concepts. Such words are called "high-level definitions". Alternatively, new words may also be defined in assembly code, since most Forth implementations include an assembler for the host processor.

This extensibility facilitates the development of special application languages for particular problem areas or disciplines.

Forth's extensibility goes beyond just adding new commands to the language. With equivalent ease, one can also add new *kinds* of words. That is, one may create a word which itself will define words. In creating such a defining word the programmer may specify a specialized behavior for the words it will create which will be effective at compile time, at run-time, or both. This capability allows one to define specialized data types, with complete control over both structure and behavior. Since the run-time behavior of such words may be defined either in high-level or in code, the words created by this new defining word are equivalent to all other kinds of Forth words in performance. Moreover, it is even easy to add new *compiler directives* to implement special kinds of loops or other control structures.

Most professional implementations of Forth are written in Forth. Many Forth systems include a "meta-compiler" which allows the user to modify the internal structure of the Forth system itself.

C.2 History of Forth

Forth was invented by Charles H. Moore. A direct outgrowth of Moore's work in the 1960's, the first program to be called Forth was written in about 1970. The first complete implementation was used in 1971 at the National Radio Astronomy Observatory's 11-meter radio telescope in Arizona. This system was responsible for pointing and tracking the telescope, collecting data and recording it on magnetic tape, and supporting an interactive graphics terminal on which an astronomer could analyze previously recorded data. The multi-tasking nature of the system allowed all these functions to be performed concurrently, without timing conflicts or other interference — a very advanced concept for that time.

The system was so useful that astronomers from all over the world began asking for copies. Its use spread rapidly, and in 1976 Forth was adopted as a standard language by the International Astronomical Union.

In 1973, Moore and colleagues formed FORTH, Inc. to explore commercial uses of the language. FORTH, Inc. developed multi-user versions of Forth on minicomputers for diverse projects ranging from data bases to scientific applications such as image processing. In 1977, FORTH, Inc. developed a version for the newly introduced 8-bit microprocessors called "microFORTH", which was successfully used in embedded microprocessor applications in the United States, Britain and Japan.

Stimulated by the volume marketing of microFORTH, a group of computer hobbyists in Northern California became interested in Forth, and in 1978 formed the Forth Interest Group (FIG). They developed a simplified model which they implemented on several microprocessors and published listings and disks at very low cost. Interest in Forth spread rapidly, and today there are chapters of the Forth Interest Group throughout the U.S. and in over fifteen countries.

By 1980, a number of new Forth vendors had entered the market with versions of Forth based upon the FIG model. Primarily designed for personal computers, these relatively inexpensive Forth systems have been distributed very widely.

C.3 Hardware implementations of Forth

The internal architecture of Forth simulates a computer with two stacks, a set of registers, and other standardized features. As a result, it was almost inevitable that someone would attempt to build a hardware representation of an actual Forth computer.

In the early 1980's, Rockwell produced a 6502-variant with Forth primitives in on-board ROM, the Rockwell 65F11. This chip has been used successfully in many embedded microprocessor applications. In the mid-1980's Zilog developed the z8800 (Super8) which offered ENTER (nest), EXIT (unnest) and NEXT in microcode.

In 1981, Moore undertook to design a chip-level implementation of the Forth virtual machine. Working first at FORTH, Inc. and subsequently with the start-up company NOVIX, formed to develop the chip, Moore completed the design in 1984, and the first prototypes were produced in early 1985. More recently, Forth processors have been developed by Harris Semiconductor Corp., Johns Hopkins University, and others.

C.4 Standardization efforts

The first major effort to standardize Forth was a meeting in Utrecht in 1977. The attendees produced a preliminary standard, and agreed to meet the following year. The 1978 meeting was also attended by members of the newly formed Forth Interest Group. In 1979 and 1980 a series of meetings attended by both users and vendors produced a more comprehensive standard called Forth 79.

Although Forth 79 was very influential, many Forth users and vendors found serious flaws in it, and in 1983 a new standard called Forth 83 was released.

Encouraged by the widespread acceptance of Forth 83, a group of users and vendors met in 1986 to inves-

tigate the feasibility of an American National Standard. The X3J14 Technical Committee for ANS Forth held its first meeting in 1987. This Standard is the result.

C.5 Programming in Forth

Forth is an English-like language whose elements (called "words") are named data items, procedures, and defining words capable of creating data items with customized characteristics. Procedures and defining words may be defined in terms of previously defined words or in machine code, using an embedded assembler.

Forth "words" are functionally analogous to subroutines in other languages. They are also equivalent to commands in other languages — Forth blurs the distinction between linguistic elements and functional elements.

Words are referred to either from the keyboard or in program source by name. As a result, the term "word" is applied both to program (and linguistic) units and to their text names. In parsing text, Forth considers a word to be any string of characters bounded by spaces. There are a few special characters that cannot be included in a word or start a word: space (the universal delimiter), CR (which ends terminal input), and backspace or DEL (for backspacing during keyboard input). Many groups adopt naming conventions to improve readability. Words encountered in text fall into three categories: defined words (i.e., Forth routines), numbers, and undefined words. For example, here are four words:

```
HERE   DOES>   !    8493
```

The first three are standard-defined words. This means that they have entries in Forth's dictionary, described below, explaining what Forth is to do when these words are encountered. The number "8493" will presumably not be found in the dictionary, and Forth will convert it to binary and place it on its push-down stack for parameters. When Forth encounters an undefined word and cannot convert it to a number, the word is returned to the user with an exception message.

Architecturally, Forth words adhere strictly to the principles of "structured programming":

- Words must be defined before they are used.
- Logical flow is restricted to sequential, conditional, and iterative patterns. Words are included to implement the most useful program control structures.
- The programmer works with many small, independent modules (words) for maximum testability and reliability.

Forth is characterized by five major elements: a dictionary, two push-down stacks, interpreters, an assembler, and virtual storage. Although each of these may be found in other systems, the combination produces a synergy that yields a powerful and flexible system.

C.5.1 The Forth dictionary

A Forth program is organized into a dictionary that occupies most of the memory used by the system. This dictionary is a threaded list of variable-length items, each of which defines a word. The content of each definition depends upon the type of word (data item, constant, sequence of operations, etc.). The dictionary is extensible, usually growing toward high memory. On some multi-user systems individual users have private dictionaries, each of which is connected to a shared system dictionary.

Words are added to the dictionary by "defining words", of which the most commonly used is **:** (colon). When **:** is executed, it constructs a definition for the word that follows it. In classical implementations[1],

[1] Other common implementation techniques include direct translation to code and other types of tokens.

the content of this definition is a string of addresses of previously defined words which will be executed in turn whenever the word being defined is invoked. The definition is terminated by **;** (semicolon). For example, here is a definition:

```
:  RECEIVE ( - addr n ) PAD DUP 32 ACCEPT ;
```

The name of the new word is RECEIVE. The comment (in parentheses) indicates that it requires no parameters and will return an address and count on the data stack. When RECEIVE is executed, it will perform the words in the remainder of the definition in sequence. The word **PAD** places on the stack the address of a scratch pad used to handle strings. **DUP** duplicates the top stack item, so we now have two copies of the address. The number 32 is also placed on the stack. The word **ACCEPT** takes an address (provided by **PAD**) and length (32) on the stack, accepts from the keyboard a string of up to 32 characters which will be placed at the specified address, and returns the number of characters received. The copy of the scratch-pad address remains on the stack below the count so that the routine that called RECEIVE can use it to pick up the received string.

C.5.2 Push-down stacks

The example above illustrates the use of push-down stacks for passing parameters between Forth words. Forth maintains two push-down stacks, or LIFO lists. These provide communication between Forth words plus an efficient mechanism for controlling logical flow. A stack contains 16-bit items on 8-bit and 16-bit computers, and 32-bit items on 32-bit processors. Double-cell numbers occupy two stack positions, with the most-significant part on top. Items on either stack may be addresses or data items of various kinds. Stacks are of indefinite size, and usually grow towards low memory.

Although the structure of both stacks is the same, they have very different uses. The user interacts most directly with the Data Stack, which contains arguments passed between words. This function replaces the calling sequences used by conventional languages. It is efficient internally, and makes routines intrinsically re-entrant. The second stack is called the Return Stack, as its main function is to hold return addresses for nested definitions, although other kinds of data are sometimes kept there temporarily.

The use of the Data Stack (often called just “the stack”) leads to a notation in which operands precede operators. The word **ACCEPT** in the example above took an address and count from the stack and left another address there. Similarly, a word called **BLANK** expects an address and count, and will place the specified number of space characters (20H) in the region starting at that address. Thus,

```
PAD 25 BLANK
```

will fill the scratch region whose address is pushed on the stack by **PAD** with 25 spaces. Application words are usually defined to work similarly. For example,

```
100 SAMPLES
```

might be defined to record 100 measurements in a data array.

Arithmetic operators also expect values and leave results on the stack. For example, **+** adds the top two numbers on the stack, replacing them both by their sum. Since results of operations are left on the stack, operations may be strung together without a need to define variables to use for temporary storage.

C.5.3 Interpreters

Forth is traditionally an interpretive system, in that program execution is controlled by data items rather than machine code. Interpreters can be slow, but Forth maintains the high speed required of real-time applications by having two levels of interpretation.

The first is the text interpreter, which parses strings from the terminal or mass storage and looks each word up in the dictionary. When a word is found it is executed by invoking the second level, the address interpreter.

The second is an "address interpreter" Although not all Forth systems are implemented in this way, it was the first and is still the primary implementation technology. For a small cost in performance, an address interpreter can yield a very compact object program, which has been a major factor in Forth's wide acceptance in embedded systems and other applications where small object size is desirable.

The address interpreter processes strings of addresses or tokens compiled in definitions created by **:** (colon), by executing the definition pointed to by each. The content of most definitions is a sequence of addresses of previously defined words, which will be executed by the address interpreter in turn. Thus, when the word RECEIVE (defined above) is executed, the word **PAD**, the word **DUP**, the literal 32, and the word **ACCEPT** will be executed in sequence. The process is terminated by the semicolon. This execution requires no dictionary searches, parsing, or other logic, because when RECEIVE was *compiled* the dictionary was searched for each word, and its address (or other token) was placed in the next successive cell of the entry. The text was not stored in memory, not even in condensed form.

The address interpreter has two important properties. First, it is fast. Although the actual speed depends upon the specific implementation, professional implementations are highly optimized, often requiring only one or two machine instructions per address. On most benchmarks, a good Forth implementation substantially out-performs interpretive languages such as BASIC or LISP, and will compare favorably with other compiled high-level languages.

Second, the address interpreter makes Forth definitions extremely compact, as each reference requires only one cell. In comparison, a subroutine call constructed by most compilers involves instructions for handling the calling sequence (unnecessary in Forth because of the stack) before and after a CALL or JSR instruction and address.

Most of the words in a Forth dictionary will be defined by **:** (colon) and interpreted by the address interpreter. Most of Forth itself is defined this way.

C.5.4 Assembler

Most implementations of Forth include a macro assembler for the CPU on which they run. By using the defining word **CODE** the programmer can create a definition whose behavior will consist of executing actual machine instructions. **CODE** definitions may be used to do I/O, implement arithmetic primitives, and do other machine-dependent or time-critical processing. When using **CODE** the programmer has full control over the CPU, as with any other assembler, and **CODE** definitions run at full machine speed.

This is an important feature of Forth. It permits explicit computer-dependent code in manageable pieces with specific interfacing conventions that are machine-independent. To move an application to a different processor requires re-coding only the **CODE** words, which will interact with other Forth words in exactly the same manner.

Forth assemblers are so compact (typically a few Kbytes) that they can be resident in the system (as are the compiler, editor, and other programming tools). This means that the programmer can type in short **CODE** definitions and execute them immediately. This capability is especially valuable in testing custom hardware.

C.5.5 Virtual memory

The final unique element of Forth is its way of using disk or other mass storage as a form of "virtual memory" for data and program source. As in the case of the address interpreter, this approach is historically characteristic of Forth, but is by no means universal. Disk is divided into 1024-byte blocks. Two or more buffers are provided in memory, into which blocks are read automatically when referred to. Each block

has a fixed block number, which in native systems is a direct function of its physical location. If a block is changed in memory, it will be automatically written out when its buffer must be reused. Explicit reads and writes are not needed; the program will find the data in memory whenever it accesses it.

Block-oriented disk handling is efficient and easy for native Forth systems to implement. As a result, blocks provide a completely transportable mechanism for handling program source and data across both native and co-resident versions of Forth on different host operating systems.

Definitions in program source blocks are compiled into memory by the word **LOAD**. Most implementations include an editor, which formats a block for display into 16 lines of 64 characters each, and provides commands modifying the source. An example of a Forth source block is given in Fig. C.1 below.

Source blocks have historically been an important element in Forth style. Just as Forth definitions may be considered the linguistic equivalent of sentences in natural languages, a block is analogous to a paragraph.

A block normally contains definitions related to a common theme, such as "vector arithmetic". A comment on the top line of the block identifies this theme. An application may selectively load the blocks it needs.

Blocks are also used to store data. Small records can be combined into a block, or large records spread over several blocks. The programmer may allocate blocks in whatever way suits the application, and on native systems can increase performance by organizing data to minimize disk head motion. Several Forth vendors have developed sophisticated file and data base systems based on Forth blocks.

Versions of Forth that run co-resident with a host OS often implement blocks in host OS files. Others use the host files exclusively. The Standard requires that blocks be available on systems providing any disk access method, as they are the only means of referencing disk that can be transportable across both native and co-resident implementations.

C.5.6 Programming environment

Although this Standard does not require it, most Forth systems include a resident editor. This enables a programmer to edit source and recompile it into executable form without leaving the Forth environment. As it is easy to organize an application into layers, it is often possible to recompile only the topmost layer (which is usually the one currently under development), a process which rarely takes more than a few seconds.

Most Forth systems also provide resident interactive debugging aids, not only including words such as those in **15 The optional Programming-Tools word set**, but also having the ability to examine and change the contents of **VARIABLE**s and other data items and to execute from the keyboard most of the component words in both the underlying Forth system and the application under development.

The combination of resident editor, integrated debugging tools, and direct executability of most defined words leads to a very interactive programming style, which has been shown to shorten development time.

C.5.7 Advanced programming features

One of the unusual characteristics of Forth is that the words the programmer defines in building an application become integral elements of the language itself, adding more and more powerful application-oriented features.

For example, Forth includes the words **VARIABLE** and **2VARIABLE** to name locations in which data may be stored, as well as **CONSTANT** and **2CONSTANT** to name single and double-cell values. Suppose a programmer finds that an application needs arrays that would be automatically indexed through a number of two-cell items. Such an array might be called `2ARRAY`. The prefix "2" in the name indicates that each element in this array will occupy two cells (as would the contents of a **2VARIABLE** or **2CONSTANT**). The prefix "2", however, has significance only to a human and is no more significant to the text interpreter than any other character that may be used in a definition name.

Such a definition has two parts, as there are two "behaviors" associated with this new word 2ARRAY, one at compile time, and one at run or execute time. These are best understood if we look at how 2ARRAY is used to define its arrays, and then how the array might be used in an application. In fact, this is how one would design and implement this word.

Beginning the top-down design process, here's how we would like to use 2ARRAY:

```
100 2ARRAY RAW   50 2ARRAY REFINED
```

In the first case, we are defining an array 100 elements long, whose name is RAW. In the second, the array is 50 elements long, and is named REFINED. In each case, a size parameter is supplied to 2ARRAY on the data stack (Forth's text interpreter automatically puts numbers there when it encounters them), and the name of the word immediately follows. This order is typical of Forth defining words.

When we use RAW or REFINED, we would like to supply on the stack the index of the element we want, and get back the address of that element on the stack. Such a reference would characteristically take place in a loop. Here's a representative loop that accepts a two-cell value from a hypothetical application word DATA and stores it in the next element of RAW:

```
:  ACQUIRE 100 0 DO DATA I RAW 2!  LOOP ;
```

The name of this definition is ACQUIRE. The loop begins with **DO**, ends with **LOOP**, and will execute with index values running from 0 through 99. Within the loop, DATA gets a value. The word **I** returns the current value of the loop index, which is the argument to RAW. The address of the selected element, returned by RAW, and the value, which has remained on the stack since DATA, are passed to the word **2!** (pronounced "two-store"), which stores two stack items in the address.

Now that we have specified exactly what 2ARRAY does and how the words it defines are to behave, we are ready to write the two parts of its definition:

```
:  2ARRAY  ( n - )
  CREATE   2* CELLS ALLOT
  DOES>    ( i a - a')   SWAP   2* CELLS +   ;
```

The part of the definition before the word **DOES>** specifies the "compile-time" behavior, that is, what the 2ARRAY will do when it us used to define a word such as RAW. The comment indicates that this part expects a number on the stack, which is the size parameter. The word **CREATE** constructs the definition for the new word. The phrase **2* CELLS** converts the size parameter from two-cell units to the internal addressing units of the system (normally characters). **ALLOT** then allocates the specified amount of memory to contain the data to be associated with the newly defined array.

The second line defines the "run-time" behavior that will be shared by all words defined by 2ARRAY, such as RAW and REFINED. The word **DOES>** terminates the first part of the definition and begins the second part. A second comment here indicates that this code expects an index and an address on the stack, and will return a different address. The index is supplied on the stack by the caller (of RAW in the example), while the address of the content of a word defined in this way (the **ALLOT**ted region) is automatically pushed on top of the stack before this section of the code is to be executed. This code works as follows: **SWAP** reverses the order of the two stack items, to get the index on top. **2* CELLS** converts the index to the internal addressing units as in the compile-time section, to yield an offset from the beginning of the array. The word **+** then adds the offset to the address of the start of the array to give the effective address, which is the desired result.

```
Block 180
 0.  ( LED control )
 1.  HEX 40 CONSTANT LIGHTS DECIMAL
 2.  :  LIGHT ( n - ) LIGHTS OUTPUT ;
 3.
 4.  VARIABLE DELAY
 5.  :  SLOW 500 DELAY ! ;
 6.  :  FAST 100 DELAY ! ;
 7.  :  COUNTS 256 0 DO I LIGHT   DELAY @ MS   LOOP ;
 8.
 9.  :  LAMP ( n - )   CREATE ,   DOES> ( a - n )   @ ;
10.  1 LAMP POWER     2 LAMP HV     4 LAMP TORCH
11.  8 LAMP SAMPLING 16 LAMP IDLING
12.
13.  VARIABLE LAMPS
14.  :  TOGGLE ( n - ) LAMPS @ XOR DUP LAMPS ! LIGHT ;
15.
```

Figure C.1: Forth source block containing words that control a set of LEDs

Given this basic definition, one could easily modify it to do more sophisticated things. For example, the compile-time code could be changed to initialize the array to zeros, spaces, or any other desired initial value. The size of the array could be compiled at its beginning, so that the run-time code could compare the index against it to ensure it is within range, or the entire array could be made to reside on disk instead of main memory. *None of these changes would affect the run-time usage we have specified in any way*. This illustrates a little of the flexibility available with these defining words.

C.5.8 A programming example

Figure C.1 contains a typical block of Forth source. It represents a portion of an application that controls a bank of eight LEDs used as indicator lamps on an instrument, and indicates some of the ways in which Forth definitions of various kinds combine in an application environment. This example was coded for a STD-bus system with an 8088 processor and a millisecond clock, which is also used in the example.

The LEDs are interfaced through a single 8-bit port whose address is 40H. This location is defined as a **CONSTANT** on Line 1, so that it may be referred to by name; should the address change, one need only adjust the value of this constant. The word `LIGHTS` returns this address on the stack. The definition `LIGHT` takes a value on the stack and sends it to the device. The nature of this value is a bit mask, whose bits correspond directly to the individual lights.

Thus, the command `255 LIGHT` will turn on all lights, while `0 LIGHT` will turn them all off.

Lines 4 – 7 contain a simple diagnostic of the sort one might type in from the terminal to confirm that everything is working. The variable `DELAY` contains a delay time in milliseconds — execution of the word `DELAY` returns the address of this variable. Two values of `DELAY` are set by the definitions `SLOW` and `FAST`, using the Forth operator **!** (pronounced "store") which takes a value and an address, and stores the value in the address. The definition `COUNTS` runs a loop from 0 through 255 (Forth loops of this type are exclusive at the upper end of the range), sending each value to the lights and then waiting for the period specified by `DELAY`. The word **@** (pronounced "fetch") fetches a value from an address, in this case the address supplied by `DELAY`. This value is passed to **MS**, which waits the specified number of milliseconds. The result of executing `COUNTS` is that the lights will count from 0 to 255 at the desired rate. To run this, one would type:

```
SLOW COUNTS   or   FAST COUNTS
```

at the terminal.

Line 9 provides the capability of naming individual lamps. In this application they are being used as indicator lights. The word LAMP is a defining word which takes as an argument a mask which represents a particular lamp, and compiles it as a named entity. Lines 10 and 11 contain five uses of LAMP to name particular indicators. When one of these words such as POWER is executed, the mask is returned on the stack. In fact, the behavior of defining a value such that when the word is invoked the value is returned, is identical to the behavior of a Forth **CONSTANT**. We created a new defining word here, however, to illustrate how this would be done.

Finally, on lines 13 and 14, we have the words that will control the light panel. LAMPS is a variable that contains the current state of the lamps. The word TOGGLE takes a mask (which might be supplied by one of the LAMP words) and changes the state of that particular lamp, saving the result in LAMPS.

In the remainder of the application, the lamp names and TOGGLE are probably the only words that will be executed directly. The usage there will be, for example:

```
POWER TOGGLE   or   SAMPLING TOGGLE
```

as appropriate, whenever the system indicators need to be changed.

The time to compile this block of code on that system was about half a second, including the time to fetch it from disk. So it is quite practical (and normal practice) for a programmer to simply type in a definition and try it immediately.

In addition, one always has the capability of communicating with external devices directly. The first thing one would do when told about the lamps would be to type:

```
HEX FF 40 OUTPUT
```

and see if all the lamps come on. If not, the presumption is that something is amiss with the hardware, since this phrase directly transmits the "all ones" mask to the device. This type of direct interaction is useful in applications involving custom hardware, as it reduces hardware debugging time.

C.6 Multiprogrammed systems

Multiprogrammed Forth systems have existed since about 1970. The earliest public Forth systems propagated the "hooks" for this capability despite the fact that many did not use them. Nevertheless the underlying assumptions have been common knowledge in the community, and there exists considerable common ground among these multiprogrammed systems. These systems are not just language processors, but contain operating system characteristics as well. Many of these integrated systems run entirely stand-alone, performing all necessary operating system functions.

Some Forth systems are very fast, and can support both multi-tasking and multi-user operation even on computers whose hardware is usually thought incapable of such advanced operation. For example, one producer of telephone switchboards is running over 50 tasks on a Z80. There are several multiprogrammed products for PC's, some of which even support multiple users. Even on computers that are commonly used in multi-user operations, the number of users that can be supported may be much larger than expected. One large data-base application running on a single 68000 has over 100 terminals updating and querying its data-base, with no significant degradation.

Multi-user systems may also support multiple programmers, each of which has a private dictionary, stacks, and a set of variables controlling that task. The private dictionary is linked to a shared, re-entrant dictionary

containing all the standard Forth functions. The private dictionary can be used to develop application code which may later be integrated into the shared dictionary. It may also be used to perform functions requiring text interpretation, including compilation and execution of source code.

C.7 Design and management considerations

Just as the choice of building materials has a strong effect on the design and construction of a building, the choice of language and operating system will affect both application design and project management decisions.

Conventionally, software projects progress through four stages: analysis, design, coding, and testing. A Forth project necessarily incorporates these activities as well. Forth is optimized for a project-management methodology featuring small teams of skilled professionals. Forth encourages an iterative process of "successive prototyping" wherein high-level Forth is used as an executable design tool, with "stubs" replacing lower-level routines as necessary (e.g., for hardware that isn't built yet).

In many cases successive prototyping can produce a sounder, more useful product. As the project progresses, implementors learn things that could lead to a better design. Wiser decisions can be made if true relative costs are known, and often this isn't possible until prototype code can be written and tried.

Using Forth can shorten the time required for software development, and reduce the level of effort required for maintenance and modifications during the life of the product as well.

C.8 Conclusion

Forth has produced some remarkable achievements in a variety of application areas. In the last few years its acceptance has grown rapidly, particularly among programmers looking for ways to improve their productivity and managers looking for ways to simplify new software-development projects.

Annex D
(informative)
Compatibility analysis of ANS Forth

Prior to ANS Forth, there were several industry standards for Forth. The most influential are listed here in chronological order, along with the major differences between ANS Forth and the most recent, Forth 83.

D.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".

D.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.

D.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.

D.4 Recent developments

Since the Forth-83 Standard was published, the computer industry has undergone rapid and profound changes. The speed, memory capacity, and disk capacity of affordable personal computers have increased by factors of more than 100. 8-bit processors have given way to 16-bit processors, and now 32-bit processors are commonplace.

The operating systems and programming-language environments of small systems are much more powerful than they were in the early 80's.

The personal-computer marketplace has changed from a predominantly "hobbyist" market to a mature business and commercial market.

Improved technology for designing custom microprocessors has resulted in the design of numerous "Forth chips", computers optimized for the execution of the Forth language.

The market for ROM-based embedded control computers has grown substantially.

In order to take full advantage of this evolving technology, and to better compete with other programming languages, many recent Forth implementations have ignored some of the "rules" of previous Forth standards. In particular:

- 32-bit Forth implementations are now common.
- Some Forth systems adopt the address-alignment restrictions of the hardware on which they run.
- Some Forth systems use native-code generation, microcode generation, and optimization techniques, rather than the traditional "threaded code".
- Some Forth systems exploit segmented addressing architectures, placing portions of the Forth "dictionary" in different segments.
- More and more Forth systems now run in the environment of another "standard" operating system, using OS text files for source code, rather than the traditional Forth "blocks".
- Some Forth systems allow external operating system software, windowing software, terminal concentrators, or communications channels to handle or preprocess user input, resulting in deviations from the input editing, character set availability, and screen management behavior prescribed by Forth 83.

Competitive pressure from other programming languages (predominantly "C") and from other Forth vendors have led Forth vendors to optimizations that do not fit in well with the "virtual machine model" implied by existing Forth standards.

D.5 ANS Forth approach

The ANS Forth committee addressed the serious fragmentation of the Forth community caused by the differences between Forth 79 and Forth 83, and the divergence from either of these two industry standards caused by marketplace pressures.

Consequently, the committee has chosen to base its compatibility decisions not upon a strict comparison with the Forth-83 Standard, but instead upon consideration of the variety of existing implementations, especially those with substantial user bases and/or considerable success in the marketplace.

The committee feels that, if ANS Forth prescribes stringent requirements upon the virtual machine model, as did the previous standards, then many implementors will chose not to comply with ANS Forth. The committee hopes that ANS Forth will serve to unify rather than to further divide the Forth community, and thus has chosen to encompass rather than invalidate popular implementation techniques.

Many of the changes from Forth 83 are justified by this rationale. Most fall into the category that "an ANS Forth Standard Program may not assume x", where "x" is an entitlement resulting from the virtual machine model prescribed by the Forth-83 Standard. The committee feels that these restrictions are reasonable, especially considering that a substantial number of existing Forth implementations do not correctly implement the Forth-83 virtual model, thus the Forth-83 entitlements exist "in theory" but not "in practice".

Another way of looking at this is that while ANS Forth acknowledges the diversity of current Forth practice, it attempts to document the similarity therein. In some sense, ANS Forth is thus a "description of reality" rather than a "prescription for a particular virtual machine".

Since there is no previous American National Standard for Forth, the action requirements prescribed by section 3.4 of X3/SD-9, "Policy and Guidelines", regarding previous standards do not apply.

The following discussion describes differences between ANS Forth and Forth 83. In most cases, Forth 83 is representative of Forth 79 and FIG Forth for the purposes of this discussion. In many of these cases, however, ANS Forth is more representative of the existing state of the Forth industry than the previously-published standards.

D.6 Differences from Forth 83

D.6.1 Stack width

Forth 83 specifies that stack items occupy 16 bits. This includes addresses, flags, and numbers. ANS Forth specifies that stack items are at least 16 bits; the actual size must be documented by the implementation.

Words affected: all arithmetic, logical and addressing operators

Reason: 32-bit machines are becoming commonplace. A 16-bit Forth system on a 32-bit machine is not competitive.

Impact: Programs that assume 16-bit stack width will continue to run on 16-bit machines; ANS Forth does not require a different stack width, but simply allows it. Many programs will be unaffected (but see "address unit").

Transition/Conversion: Programs which use bit masks with the high bits set may have to be changed, substituting either an implementation-defined bit-mask constant, or a procedure to calculate a bit mask in a stack-width-independent way. Here are some procedures for constructing width-independent bit masks:

```
1 CONSTANT LO-BIT
TRUE 1 RSHIFT   INVERT   CONSTANT HI-BIT
: LO-BITS ( n - mask ) 0 SWAP 0 ?DO 1 LSHIFT LO-BIT OR LOOP ;
:  HI-BITS ( n - mask ) 0 SWAP 0 ?DO 1 RSHIFT HI-BIT OR LOOP
;
```

Programs that depend upon the "modulo 65536" behavior implicit in 16-bit arithmetic operations will need to be rewritten to explicitly perform the modulus operation in the appropriate places. The committee

believes that such assumptions occur infrequently. Examples: some checksum or CRC calculations, some random number generators and most fixed-point fractional math.

D.6.2 Number representation

Forth 83 specifies two's-complement number representation and arithmetic. ANS Forth also allows one's-complement and signed-magnitude.

Words affected: all arithmetic and logical operators, **LOOP**, **+LOOP**.

Reason: Some computers use one's-complement or signed-magnitude. The committee did not wish to force Forth implementations for those machines to emulate two's-complement arithmetic, and thus incur severe performance penalties. The experience of some committee members with such machines indicates that the usage restrictions necessary to support their number representations are not overly burdensome.

Impact: An ANS Forth Standard Program may declare an "environmental dependency on two's-complement arithmetic". This means that the otherwise-Standard Program is only guaranteed to work on two's-complement machines. Effectively, this is not a severe restriction, because the overwhelming majority of current computers use two's-complement. The committee knows of no Forth-83 compliant implementations for non-two's-complement machines at present, so existing Forth-83 programs will still work on the same class of machines on which they currently work.

Transition/Conversion: Existing programs wishing to take advantage of the possibility of ANS Forth Standard Systems on non-two's-complement machines may do so by eliminating the use of arithmetic operators to perform logical functions, by deriving bit-mask constants from bit operations as described in the section about stack width, by restricting the usage range of unsigned numbers to the range of positive numbers, and by using the provided operators for conversion from single numbers to double numbers.

D.6.3 Address units

Forth 83 specifies that each unique address refers to an 8-bit byte in memory. ANS Forth specifies that the size of the item referred to by each unique address is implementation-defined, but, by default, is the size of one character. Forth 83 describes many memory operations in terms of a number of bytes. ANS Forth describes those operations in terms of a number of either characters or address units.

Words affected: those with "address unit" arguments

Reason: Some machines, including the most popular Forth chip, address 16-bit memory locations instead of 8-bit bytes.

Impact: Programs may choose to declare an environmental dependency on byte addressing, and will continue to work on the class of machines for which they now work. In order for a Forth implementation on a word-addressed machine to be Forth 83 compliant, it would have to simulate byte addressing at considerable cost in speed and memory efficiency. The committee knows of no such Forth-83 implementations for such machines, thus an environmental dependency on byte addressing does not restrict a Standard Program beyond its current de facto restrictions.

Transition/Conversion: The new **CHARS** and **CHAR+** address arithmetic operators should be used for programs that require portability to non-byte-addressed machines. The places where such conversion is necessary may be identified by searching for occurrences of words that accept a number of address units as an argument (e.g., **MOVE**, **ALLOT**).

D.6.4 Address increment for a cell is no longer two

As a consequence of Forth-83's simultaneous specification of 16-bit stack width and byte addressing, the number two could reliably be used in address calculations involving memory arrays containing items from the stack. Since ANS Forth requires neither 16-bit stack width nor byte addressing, the number two is no longer necessarily appropriate for such calculations.

Words affected: `@ ! +!` `2+` `2*` `2-` `+LOOP`

Reason: See reasons for "Address Units" and "Stack Width"

Impact: In this respect, existing programs will continue to work on machines where a stack cell occupies two address units when stored in memory. This includes most machines for which Forth 83 compliant implementations currently exist. In principle, it would also include 16-bit-word-addressed machines with 32-bit stack width, but the committee knows of no examples of such machines.

Transition/Conversion: The new `CELLS` and `CELL+` address arithmetic operators should be used for portable programs. The places where such conversion is necessary may be identified by searching for the character "2" and determining whether or not it is used as part of an address calculation. The following substitutions are appropriate within address calculations:

Old	New
`2+` or `2 +`	`CELL+`
`2*` or `2 *`	`CELLS`
`2-` or `2 -`	`1 CELLS -`
`2/` or `2 /`	`1 CELLS /`
`2`	`1 CELLS`

The number "2" by itself is sometimes used for address calculations as an argument to `+LOOP`, when the loop index is an address. When converting the word `2/` which operates on negative dividends, one should be cognizant of the rounding method used.

D.6.5 Address alignment

Forth 83 imposes no restriction upon the alignment of addresses to any boundary. ANS Forth specifies that a Standard System may require alignment of addresses for use with various "`@`" and "`!`" operators.

Words Affected: `! +! 2! 2@ @ ? ,`

Reason: Many computers have hardware restrictions that favor the use of aligned addresses. On some machines, the native memory-access instructions will cause an exception trap if used with an unaligned address. Even on machines where unaligned accesses do not cause exception traps, aligned accesses are usually faster.

Impact: All of the ANS Forth words that return addresses suitable for use with aligned "`@`" and "`!`" words must return aligned addresses. In most cases, there will be no problem. Problems can arise from the use of user-defined data structures containing a mixture of character data and cell-sized data.

Many existing Forth systems, especially those currently in use on computers with strong alignment requirements, already require alignment. Much existing Forth code that is currently in use on such machines has already been converted for use in an aligned environment.

Transition/Conversion: There are two possible approaches to conversion of programs for use on a system requiring address alignment.

The easiest approach is to redefine the system's aligned "`@`" and "`!`" operators so that they do not require alignment. For example, on a 16-bit little-endian byte-addressed machine, unaligned "`@`" and "`!`" could be defined:

```
:  @ ( addr - x ) DUP C@ SWAP CHAR+ C@ 8 LSHIFT OR ;
:  !  ( x addr - ) OVER 8 RSHIFT OVER CHAR+ C! C! ;
```

These definitions, and similar ones for "**+!**", "**2@**", "**2!**", "**,**", and "**?**" as needed, can be compiled before an unaligned application, which will then work as expected.

This approach may conserve memory if the application uses substantial numbers of data structures containing unaligned fields.

Another approach is to modify the application's source code to eliminate unaligned data fields. The ANS Forth words **ALIGN** and **ALIGNED** may be used to force alignment of data fields. The places where such alignment is needed may be determined by inspecting the parts of the application where data structures (other than simple variables) are defined, or by "smart compiler" techniques (see the "Smart Compiler" discussion below).

This approach will probably result in faster application execution speed, at the possible expense of increased memory utilization for data structures.

Finally, it is possible to combine the preceding techniques by identifying exactly those data fields that are unaligned, and using "unaligned" versions of the memory access operators for only those fields. This "hybrid" approach affects a compromise between execution speed and memory utilization.

D.6.6 Division/modulus rounding direction

Forth 79 specifies that division rounds toward 0 and the remainder carries the sign of the dividend. Forth 83 specifies that division rounds toward negative infinity and the remainder carries the sign of the divisor. ANS Forth allows either behavior for the division operators listed below, at the discretion of the implementor, and provides a pair of division primitives to allow the user to synthesize either explicit behavior.

Words Affected: **/ MOD /MOD */MOD */**

Reason: The difference between the division behaviors in Forth 79 and Forth 83 was a point of much contention, and many Forth implementations did not switch to the Forth 83 behavior. Both variants have vocal proponents, citing both application requirements and execution efficiency arguments on both sides. After extensive debate spanning many meetings, the committee was unable to reach a consensus for choosing one behavior over the other, and chose to allow either behavior as the default, while providing a means for the user to explicitly use both behaviors as needed. Since implementors are allowed to choose either behavior, they are not required to change the behavior exhibited by their current systems, thus preserving correct functioning of existing programs that run on those systems and depend on a particular behavior. New implementations could choose to supply the behavior that is supported by the native CPU instruction set, thus maximizing execution speed, or could choose the behavior that is most appropriate for the intended application domain of the system.

Impact: The issue only affects programs that use a negative dividend with a positive divisor, or a positive dividend with a negative divisor. The vast majority of uses of division occur with both a positive dividend and a positive divisor; in that case, the results are the same for both allowed division behaviors.

Transition/Conversion: For programs that require a specific rounding behavior with division operands of mixed sign, the division operators used by the program may be redefined in terms of one of the new ANS Forth division primitives **SM/REM** (symmetrical division, i.e., round toward zero) or **FM/MOD** (floored division, i.e., round toward negative infinity). Then the program may be recompiled without change. For example, the Forth 83 style division operators may be defined by:

```
:  /MOD   ( n1 n2 - n3 n4 )     >R S>D R> FM/MOD ;
:  MOD    ( n1 n2 - n3 )        /MOD DROP ;
:  /      ( n1 n2 - n3 )        /MOD SWAP DROP ;
:  */MOD  ( n1 n2 n3 - n4 n5 )  >R M* R> FM/MOD ;
:  */     ( n1 n2 n3 - n4 n5 )  */MOD SWAP DROP ;
```

D.6.7 Immediacy

Forth 83 specified that a number of "compiling words" are "immediate", meaning that they are executed instead of compiled during compilation. ANS Forth is less specific about most of these words, stating that their behavior is only defined during compilation, and specifying their results rather than their specific compile-time actions.

To force the compilation of a word that would normally be executed, Forth 83 provided the words `COMPILE`, used with non-immediate words, and **`[COMPILE]`**, used with immediate words. ANS Forth provides the single word **`POSTPONE`**, which is used with both immediate and non-immediate words, automatically selecting the appropriate behavior.

Words Affected: `COMPILE` **`[COMPILE]`** **`[']`** **`'`**

Reason: The designation of particular words as either immediate or not depends upon the implementation technique chosen for the Forth system. With traditional "threaded code" implementations, the choice was generally quite clear (with the single exception of the word **`LEAVE`**), and the standard could specify which words should be immediate. However, some of the currently popular implementation techniques, such as native-code generation with optimization, require the immediacy attribute on a different set of words than the set of immediate words of a threaded code implementation. ANS Forth, acknowledging the validity of these other implementation techniques, specifies the immediacy attribute in as few cases as possible.

When the membership of the set of immediate words is unclear, the decision about whether to use `COMPILE` or **`[COMPILE]`** becomes unclear. Consequently, ANS Forth provides a "general purpose" replacement word **`POSTPONE`** that serves the purpose of the vast majority of uses of both `COMPILE` and **`[COMPILE]`**, without requiring that the user know whether or not the "postponed" word is immediate.

Similarly, the use of **`'`** and **`[']`** with compiling words is unclear if the precise compilation behavior of those words is not specified, so ANS Forth does not permit a Standard Program to use **`'`** or **`[']`** with compiling words.

The traditional (non-immediate) definition of the word `COMPILE` has an additional problem. Its traditional definition assumes a threaded code implementation technique, and its behavior can only be properly described in that context. In the context of ANS Forth, which permits other implementation techniques in addition to threaded code, it is very difficult, if not impossible, to describe the behavior of the traditional `COMPILE`. Rather than changing its behavior, and thus breaking existing code, ANS Forth does not include the word `COMPILE`. This allows existing implementations to continue to supply the word `COMPILE` with its traditional behavior, if that is appropriate for the implementation.

Impact: **`[COMPILE]`** remains in ANS Forth, since its proper use does not depend on knowledge of whether or not a word is immediate (Use of **`[COMPILE]`** with a non-immediate word is and has always been a no-op). Whether or not you need to use **`[COMPILE]`** requires knowledge of whether or not its target word is immediate, but it is always safe to use **`[COMPILE]`**. **`[COMPILE]`** is no longer in the (required) core word set, having been moved to the Core Extensions word set, but the committee anticipates that most vendors will supply it anyway.

In nearly all cases, it is correct to replace both **`[COMPILE]`** and `COMPILE` with **`POSTPONE`**. Uses of **`[COMPILE]`** and `COMPILE` that are not suitable for "mindless" replacement by **`POSTPONE`** are quite infrequent, and fall into the following two categories:

- Use of **`[COMPILE]`** with non-immediate words. This is sometimes done with the words **`'`** (tick, which was immediate in Forth 79 but not in Forth 83) and **`LEAVE`** (which was immediate in Forth 83 but not in Forth 79), in order to force the compilation of those words without regard to whether you are using a Forth 79 or Forth 83 system.
- Use of the phrase `COMPILE` **`[COMPILE]`** ⟨*immediate word*⟩ to "doubly postpone" an immediate word.

Transition/Conversion: Many ANS Forth implementations will continue to implement both **[COMPILE]** and COMPILE in forms compatible with existing usage. In those environments, no conversion is necessary.

For complete portability, uses of COMPILE and **[COMPILE]** should be changed to **POSTPONE**, except in the rare cases indicated above. Uses of **[COMPILE]** with non-immediate words may be left as-is, and the program may declare a requirement for the word **[COMPILE]** from the Core Extensions word set, or the **[COMPILE]** before the non-immediate word may be simply deleted if the target word is known to be non-immediate.

Uses of the phrase COMPILE **[COMPILE]** ⟨*immediate-word*⟩ may be handled by introducing an "intermediate word" (XX in the example below) and then postponing that word. For example:

```
:  ABC COMPILE [COMPILE] IF ;
```

changes to:

```
:  XX POSTPONE IF ;
:  ABC POSTPONE XX ;
```

A non-standard case can occur with programs that "switch out of compilation state" to explicitly compile a thread in the dictionary following a COMPILE. For example:

```
:  XYZ COMPILE [ ' ABC , ] ;
```

This depends heavily on knowledge of exactly how COMPILE and the threaded-code implementation works. Cases like this cannot be handled mechanically; they must be translated by understanding exactly what the code is doing, and rewriting that section according to ANS Forth restrictions.

Use the phrase **POSTPONE [COMPILE]** to replace **[COMPILE] [COMPILE]**.

D.6.8 Input character set

Forth 83 specifies that the full 7-bit ASCII character set is available through **KEY**. ANS Forth restricts it to the graphic characters of the ASCII set, with codes from hex 20 to hex 7E inclusive.

Words Affected: **KEY**

Reason: Many system environments "consume" certain control characters for such purposes as input editing, job control, or flow control. A Forth implementation cannot always control this system behavior.

Impact: Standard Programs which require the ability to receive particular control characters through **KEY** must declare an environmental dependency on the input character set.

Transition/Conversion: For maximum portability, programs should restrict their required input character set to only the graphic characters. Control characters may be handled if available, but complete program functionality should be accessible using only graphic characters.

As stated above, an environmental dependency on the input character set may be declared. Even so, it is recommended that the program should avoid the requirement for particularly-troublesome control characters, such as control-S and control-Q (often used for flow control, sometimes by communication hardware whose presence may be difficult to detect), ASCII NUL (difficult to type on many keyboards), and the distinction between carriage return and line feed (some systems translate carriage returns into line feeds, or vice versa).

D.6.9 Shifting with UM/MOD

Given Forth-83's two's-complement nature, and its requirement for floored (round toward minus infinity) division, shifting is equivalent to division. Also, two's-complement representation implies that unsigned division by a power of two is equivalent to logical right-shifting, so **UM/MOD** could be used to perform a logical right-shift.

Words Affected: **UM/MOD**

Reason: The problem with **UM/MOD** is a result of allowing non-two's-complement number representations, as already described.

ANS Forth provides the words **LSHIFT** and **RSHIFT** to perform logical shifts. This is usually more efficient, and certainly more descriptive, than the use of **UM/MOD** for logical shifting.

Impact: Programs running on ANS Forth systems with two's-complement arithmetic (the majority of machines), will not experience any incompatibility with **UM/MOD**. Existing Forth-83 Standard programs intended to run on non-two's-complement machines will not be able to use **UM/MOD** for shifting on a non-two's-complement ANS Forth system. This should not affect a significant number of existing programs (perhaps none at all), since the committee knows of no existing Forth-83 implementations on non-two's-complement machines.

Transition/Conversion: A program that requires **UM/MOD** to behave as a shift operation may declare an environmental dependency on two's-complement arithmetic.

A program that cannot declare an environmental dependency on two's-complement arithmetic may require editing to replace incompatible uses of **UM/MOD** with other operators defined within the application.

D.6.10 Vocabularies / wordlists

ANS Forth does not define the words VOCABULARY, CONTEXT, and CURRENT, which were present in Forth 83. Instead, ANS Forth defines a primitive word set for search order specification and control, including words which have not existed in any previous standard.

Forth-83's "**ALSO**/**ONLY**" experimental search order word set is specified for the most part as the extension portion of the ANS Forth Search Order word set.

Words Affected: VOCABULARY CONTEXT CURRENT

Reason: Vocabularies are an area of much divergence among existing systems. Considering major vendors' systems and previous standards, there are at least 5 different and mutually incompatible behaviors of words defined by VOCABULARY. Forth 83 took a step in the direction of "run-time search-order specification" by declining to specify a specific relationship between the hierarchy of compiled vocabularies and the run-time search order. Forth 83 also specified an experimental mechanism for run-time search-order specification, the **ALSO**/**ONLY** scheme. **ALSO**/**ONLY** was implemented in numerous systems, and has achieved some measure of popularity in the Forth community.

However, several vendors refuse to implement it, citing technical limitations. In an effort to address those limitations and thus hopefully make **ALSO**/**ONLY** more palatable to its critics, the committee specified a simple "primitive word set" that not only fixes some of the objections to **ALSO**/**ONLY**, but also provides sufficient power to implement **ALSO**/**ONLY** and all of the other search-order word sets that are currently popular.

The Forth 83 **ALSO**/**ONLY** word set is provided as an optional extension to the search-order word set. This allows implementors that are so inclined to provide this word set, with well-defined standard behavior, but does not compel implementors to do so. Some vendors have publicly stated that they

will not implement **ALSO**/**ONLY**, no matter what, and one major vendor stated an unwillingness to implement ANS Forth at all if **ALSO**/**ONLY** is mandated. The committee feels that its actions are prudent, specifying **ALSO**/**ONLY** to the extent possible without mandating its inclusion in all systems, and also providing a primitive search-order word set that vendors may be more likely to implement, and which can be used to synthesize **ALSO**/**ONLY**.

Transition/Conversion: Since Forth 83 did not mandate precise semantics for VOCABULARY, existing Forth-83 Standard programs cannot use it except in a trivial way. Programs can declare a dependency on the existence of the Search Order word set, and can implement whatever semantics are required using that word set's primitives. Forth 83 programs that need **ALSO**/**ONLY** can declare a dependency on the Search Order Extensions word set, or can implement the extensions in terms of the Search Order word set itself.

D.6.11 Multiprogramming impact

Forth 83 marked words with "multiprogramming impact" by the letter "M" in the first lines of their descriptions. ANS Forth has removed the "M" designation from the word descriptions, moving the discussion of multiprogramming impact to this non-normative annex.

Words affected: none

Reason: The meaning of "multiprogramming impact" is precise only in the context of a specific model for multiprogramming. Although many Forth systems do provide multiprogramming capabilities using a particular round-robin, cooperative, block-buffer sharing model, that model is not universal. Even assuming the classical model, the "M" designations did not contain enough information to enable writing of applications that interacted in a multiprogrammed system.

Practically speaking, the "M" designations in Forth 83 served to document usage rules for block buffer addresses in multiprogrammed systems. These addresses often become meaningless after a task has relinquished the CPU for any reason, most often for the purposes of performing I/O, awaiting an event, or voluntarily sharing CPU resources using the word PAUSE. It was essential that portable applications respect those usage rules to make it practical to run them on multiprogrammed systems; failure to adhere to the rules could easily compromise the integrity of other applications running on those systems as well as the applications actually in error. Thus, "M" appeared on all words that by design gave up the CPU, with the understanding that other words NEVER gave it up.

These usage rules have been explicitly documented in the Block word set where they are relevant. The "M" designations have been removed entirely.

Impact: In practice, none.

In the sense that any application that depends on multiprogramming must consist of at least two tasks that share some resource(s) and communicate between themselves, Forth 83 did not contain enough information to enable writing of a standard program that DEPENDED on multiprogramming. This is also true of ANS Forth.

Non-multiprogrammed applications in Forth 83 were required to respect usage rules for **BLOCK** so that they could be run properly on multiprogrammed systems. The same is true of ANS Forth.

The only difference is the documentation method used to define the **BLOCK** usage rules. The Technical Committee believes that the current method is clearer than the concept of "multiprogramming impact".

Transition/Conversion: none needed.

D.6.12 Words not provided in executable form

ANS Forth allows an implementation to supply some words in source code or "load as needed" form, rather than requiring all supplied words to be available with no additional programmer action.

Words affected: all

Reason: Forth systems are often used in environments where memory space is at a premium. Every word included in the system in executable form consumes memory space. The committee believes that allowing standard words to be provided in source form will increase the probability that implementors will provide complete ANS Forth implementations even in systems designed for use in constrained environments.

Impact: In order to use a Standard Program with a given ANS Forth implementation, it may be necessary to precede the program with an implementation-dependent "preface" to make "source form" words executable. This is similar to the methods that other computer languages require for selecting the library routines needed by a particular application.

In languages like C, the goal of eliminating unnecessary routines from the memory image of an application is usually accomplished by providing libraries of routines, using a "linker" program to incorporate only the necessary routines into an executable application. The method of invoking and controlling the linker is outside the scope of the language definition.

Transition/Conversion: Before compiling a program, the programmer may need to perform some action to make the words required by that program available for execution.

Annex E
(informative)
ANS Forth portability guide

E.1 Introduction

The most popular architectures used to implement Forth have had byte-addressed memory, 16-bit operations, and two's-complement number representation. The Forth-83 Standard dictates that these particular features must be present in a Forth-83 Standard system and that Forth-83 programs may exploit these features freely.

However, there are many beasts in the architectural jungle that are bit addressed or cell addressed, or prefer 32-bit operations, or represent numbers in one's complement. Since one of Forth's strengths is its usefulness in "strange" environments on "unusual" hardware with "peculiar" features, it is important that a Standard Forth run on these machines too.

A primary goal of the ANS Forth Standard is to increase the types of machines that can support a Standard Forth. This is accomplished by allowing some key Forth terms to be implementation-defined (e.g., how big is a cell?) and by providing Forth operators (words) that conceal the implementation. This frees the implementor to produce the Forth system that most effectively utilizes the native hardware. The machine independent operators, together with some programmer discipline, enable a programmer to write Forth programs that work on a wide variety of machines.

The remainder of this Annex provides guidelines for writing portable ANS Forth programs. The first section describes ways to make a program hardware independent. It is difficult for someone familiar with only one machine architecture to imagine the problems caused by transporting programs between dissimilar machines. Consequently, examples of specific architectures with their respective problems are given. The second section describes assumptions about Forth implementations that many programmers make, but can't be relied upon in a portable program.

E.2 Hardware peculiarities

E.2.1 Data/memory abstraction

Data and memory are the stones and mortar of program construction. Unfortunately, each computer treats data and memory differently. The ANS Forth Systems Standard gives definitions of 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, ANS 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/memory peculiarities of different computers.

E.2.2 Definitions

Three terms defined by ANS Forth are address unit, cell, and character. The address space of an ANS 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. Such "byte addressed" machines include the Intel 8086 and Motorola 68000 families. However, other address unit sizes exist. There are machines that are bit addressed and machines that are 4-bit nibble addressed. There are also machines with address units larger than 8-bits. For example, several Forth-in-hardware computers are cell addressed.

The cell is the fundamental data type of a Forth system. A cell can be a single-cell integer or a memory

address. Forth's parameter and return stacks are stacks of cells. Forth 83 specifies that a cell is 16-bits. In ANS Forth the size of a cell is an implementation-defined number of address units. Thus, an ANS 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. Also 18-bit machines, 36-bit machines, etc., could support ANS Forth systems with 18 or 36-bit cells respectively. In all of these systems, **DUP** does the same thing: it duplicates the top of the data stack. **!** (store) behaves consistently too: given two cells on the data stack it stores the second cell in the memory location designated by the top cell.

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.

E.2.3 Addressing memory

ANS Forth eliminates many portability problems by using the above definitions. One of the most common portability problems is addressing successive cells in memory. Given the memory address of a cell, how do you find the address of the next cell? In Forth 83 this is easy: `2 +`. This code assumes that memory is addressed in 8-bit units (bytes) and a cell is 16-bits wide. 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. ANS Forth 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. A defining word to create an array of cells using Forth 83 would be:

```
: ARRAY  CREATE  2* ALLOT  DOES> SWAP 2* + ;
```

Use of **2*** to scale the array index assumes byte addressing and 16-bit cells again. As in the example above, different versions of the code would be needed for different machines. ANS Forth provides a portable scaling operator named **CELLS**. Given a number *n*, **CELLS** returns the number of address units needed to hold *n* cells. A portable definition of array is:

```
: ARRAY  CREATE  CELLS ALLOT
  DOES> SWAP CELLS + ;
```

There are also portability problems with addressing arrays of characters. In Forth 83 (and in the most common ANS Forth implementations), the size of a character will equal the size of an address unit. Consequently 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 are cases where a character is larger than an address unit. Examples include (1) systems with small address units (e.g., bit- and nibble-addressed systems), and (2) systems with large character sets (e.g., 16-bit characters on a byte-addressed machine). **CHAR+** and **CHARS** operators, analogous to **CELL+** and **CELLS** are available to allow maximum portability.

ANS Forth generalizes the definition of some Forth words that operate on chunks 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)
```

The memory-block-move word also uses address units:

```
source destination 8 CELLS MOVE ( move 8 cells)
```

E.2.4 Alignment problems

Not all addresses are created equal. Many processors have restrictions on the addresses that can be used by memory access instructions. This Standard does not require an implementor of an ANS Forth to make alignment transparent; on the contrary, it requires (in Section **3.3.3.1 Address alignment**) that an ANS Forth program assume that character and cell alignment may be required.

One of the most common problems caused by alignment restrictions is in creating tables containing both characters and cells. When `,` (comma) or `C,` is used to initialize a table, data is 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 16-bit fetches to even addresses, `CREATE` would leave the data space pointer at an even address, the 1 `C,` would make the data space pointer odd, and `,` (comma) would violate the address restriction by storing `X` at an odd 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.

E.3 Number representation

Different computers represent numbers in different ways. An awareness of these differences can help a programmer avoid writing a program that depends on a particular representation.

E.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. Other machines do the opposite — the least-significant part is stored at the lowest address. For example, the following code for a 16-bit 8086 “little endian” Forth would produce the answer 34 (hex):

```
VARIABLE FOO   HEX 1234 FOO !   FOO C@
```

The same code on a 16-bit 68000 “big endian” Forth would produce the answer 12 (hex). 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.

E.3.2 ALU organization

Different computers use different bit patterns to represent integers. Possibilities include binary representations (two's complement, one's complement, sign magnitude, etc.) and decimal representations (BCD, etc.). Each of these formats creates advantages and disadvantages in the design of a computer's arithmetic logic unit (ALU). The most commonly used representation, two's complement, is popular because of the simplicity of its addition and subtraction algorithms.

Programmers who have grown up on two's complement machines tend to become intimate with their representation of numbers and take some properties of that representation for granted. For example, a trick to find the remainder of a number divided by a power of two is to mask off some bits with **AND**. A common application of this trick is to test a number for oddness using 1 **AND**. However, this will not work on a one's complement machine if the number is negative (a portable technique is 2 **MOD**).

The remainder of this section is a (non-exhaustive) list of things to watch for when portability between machines with binary representations other than two's complement is desired.

To convert a single-cell number to a double-cell number, ANS Forth provides the operator **S>D**. To convert a double-cell number to single-cell, Forth programmers have traditionally used **DROP**. However, this trick doesn't work on sign-magnitude machines. For portability a **D>S** operator is available. Converting an unsigned single-cell number to a double-cell number can be done portably by pushing a zero on the stack.

E.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.

E.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 are called the name, link, code, and parameter fields[1]. No method for accessing these fields has been found that works across all of the Forth implementations currently in use. Therefore, ANS Forth severely restricts how the fields may be used. Specifically, a portable ANS 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 way for a Standard Program to modify the value of a constant or to change the meaning of a colon definition. The **DOES>** part of a defining word operates on a data field. Since only **CREATE**d words have data fields, **DOES>** can only be paired with **CREATE** or words that call **CREATE**.

In ANS 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

[1] These terms are not defined in the Standard. They are mentioned here for historical continuity.

execute the word ticked or compiled into the current definition with **COMPILE,**. The token can also be stored in a variable and used later. Finally, if the word ticked was defined via **CREATE**, **>BODY** converts the execution token into the word's data-field address.

One thing that definitely cannot be done with an execution token is use **!** or **,** to store it into the object code of a Forth definition. This technique is sometimes used in implementations where the object code is a list of addresses (threaded code) and an execution token is also an address. However, ANS Forth permits native code implementations where this will not work.

E.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, however. On some systems, especially Forth-in-hardware systems, the stacks might be in a part of 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 since the system may use the return stack to hold return addresses and loop-control parameters. Section **3.2.3.3 Return stack** of the Standard has a list of restrictions.

E.5 ROMed application disciplines and conventions

When a Standard System provides a data space which is uniformly readable and writeable we may term this environment "RAM-only".

Programs designed for ROMed application must divide data space into at least two parts: a writeable and readable uninitialized part, called "RAM", and a read-only initialized part, called "ROM". A third possibility, a writeable and readable initialized part, normally called "initialized RAM", is not addressed by this discipline. A Standard Program must explicitly initialize the RAM data space as needed.

The separation of data space into RAM and ROM is meaningful only during the generation of the ROMed program. If the ROMed program is itself a standard development system, it has the same taxonomy as an ordinary RAM-only system.

The words affected by conversion from a RAM-only to a mixed RAM and ROM environment are:

, (comma) **ALIGN ALIGNED ALLOT C, CREATE HERE UNUSED**

(**VARIABLE** always accesses the RAM data space.)

With the exception of **,** (comma) and **C,** these words are meaningful in both RAM and ROM data space. To select the data space, these words could be preceded by selectors RAM and ROM. For example:

```
ROM   CREATE ONES   32 ALLOT   ONES 32 1 FILL   RAM
```

would create a table of ones in the ROM data space. The storage of data into RAM data space when generating a program for ROM would be an ambiguous condition.

A straightforward implementation of these selectors would maintain separate address counters for each space. A counter value would be returned by **HERE** and altered by **,** (comma), **C,**, **ALIGN**, and **ALLOT**, with RAM and ROM simply selecting the appropriate address counter. This technique could be extended to additional partitions of the data space.

E.6 Summary

The ANS Forth Standard cannot and should not force anyone to write a portable program. In situations where performance is paramount, the programmer is encouraged to use every trick in the book. On the other hand, if portability to a wide variety of systems is needed, ANS Forth provides the tools to accomplish this. There is probably 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 F
(informative)
Test Suite

F.1 Introduction

test

After the publication of the original ANS Forth document (ANSI X3.215-1994), John Hayes of the Johns Hopkins University / Applied Physics Laboratory developed a ~~validation~~test suite. In November of 1995 the test suite was released.

While this suite does test many aspects of the core system, it is not comprehensive. A standard system must be capable of passing all of the tests within the suite. A system can not claim to be standard simply because it is able to pass this test suite.

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 "`{`" (open brace) followed by a code sequence to test. This is followed by "`->`", the expected results, and "`}`" (close brace) . For example, the following:

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

tests that one plus one indeed equals two. The "`{`" does nothing; it just makes the test look pretty. 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 "`}`" 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. In the example test, the expected stack depth is one, the expected value is two, and the test passes.

The tester 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. An example is the test of the ANS Forth CORE word set. ~~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 basic stack manipulations and arithmetic. Ultimately, it tests the Forth interpreter and compiler.~~

F.3 Tester Source

The following source code provides the ~~test harness described in section ??~~tester .

```
\ From:     John Hayes S1I
\ Subject:  tester.fr
\ Date:     Mon, 27 Nov 95 13:10:09 PST

\ (C) 1995 JOHNS HOPKINS UNIVERSITY / APPLIED PHYSICS LABORATORY
\ MAY BE DISTRIBUTED FREELY AS LONG AS THIS COPYRIGHT NOTICE REMAINS.
\ VERSION 1.1
HEX

\ 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 !

:  EMPTY-STACK   \ ( ...  - ) EMPTY STACK: HANDLES UNDERFLOWED STACK TOO.
  DEPTH ?DUP IF DUP 0< IF NEGATE 0 DO 0 LOOP ELSE 0 DO DROP LOOP THEN THEN
;

:  ERROR   \ ( 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 EVERY THING ELSE
;

VARIABLE ACTUAL-DEPTH       \ STACK RECORD
  CREATE ACTUAL-RESULTS 20 CELLS ALLOT

: {       \ ( - ) SYNTACTIC SUGAR.
  ;

:  ->      \ ( ...  - ) RECORD DEPTH AND CONTENT OF STACK.
  DEPTH DUP ACTUAL-DEPTH !            \ RECORD DEPTH
  ?DUP IF                             \ IF THERE IS SOMETHING ON STACK
    0 DO ACTUAL-RESULTS I CELLS + !  LOOP   \ SAVE THEM
  THEN ;

:  }       \ ( ...  - ) COMPARE STACK (EXPECTED) CONTENTS WITH SAVED
          \ (ACTUAL) CONTENTS.
  DEPTH ACTUAL-DEPTH @ = IF           \ IF DEPTHS MATCH
    DEPTH ?DUP IF                   \ IF THERE IS SOMETHING ON THE STACK
      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 ;

:  TESTING   \ ( - ) TALKING COMMENT.
  SOURCE VERBOSE @
  IF DUP >R TYPE CR R> >IN !
  ELSE >IN ! DROP
  THEN ;
```

F.4 Core Tests

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 basic stack manipulations and arithmetic. Ultimately, it tests the Forth interpreter and compiler. test

The tests presented here are John Hayes' original test which accompany the tester package. Additional test have been given in the rationale for individual words.

~~In this and following sections we present validation tests for specific words. A standard system would be expected to pass *all* of the validation tests documented here.~~

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

test

```
\ From:     John Hayes S1I
\ Subject:  core.fr
\ Date:     Mon, 27 Nov 95 13:10

\ (C) 1995 JOHNS HOPKINS UNIVERSITY / APPLIED PHYSICS LABORATORY
\ MAY BE DISTRIBUTED FREELY AS LONG AS THIS COPYRIGHT NOTICE REMAINS.
\ VERSION 1.2
\ THIS PROGRAM TESTS THE CORE WORDS OF AN ANS FORTH SYSTEM.
\ THE PROGRAM ASSUMES A TWO'S COMPLEMENT IMPLEMENTATION WHERE
\ THE RANGE OF SIGNED NUMBERS IS -2^(N-1) ...  2^(N-1)-1 AND
\ THE RANGE OF UNSIGNED NUMBERS IS 0 ...  2^(N)-1.
\ I HAVEN'T FIGURED OUT HOW TO TEST KEY, QUIT, ABORT, OR ABORT"...
\ I ALSO HAVEN'T THOUGHT OF A WAY TO TEST ENVIRONMENT?...

TESTING CORE WORDS
HEX
```

F.4.1 Basic Assumptions

```
TESTING BASIC ASSUMPTIONS

{ -> }                \ START WITH CLEAN SLATE
( TEST IF ANY BITS ARE SET; ANSWER IN BASE 1 )
{ :  BITSET? IF 0 0 ELSE 0 THEN ; -> }
{  0 BITSET? -> 0 }       ( ZERO IS ALL BITS CLEAR )
{  1 BITSET? -> 0 0 }     ( OTHER NUMBER HAVE AT LEAST ONE BIT )
{ -1 BITSET? -> 0 0 }
```

F.4.2 Booleans

```
TESTING BOOLEANS: INVERT AND OR XOR

{ 0 0 AND -> 0 }
{ 0 1 AND -> 0 }
{ 1 0 AND -> 0 }
{ 1 1 AND -> 1 }

{ 0 INVERT 1 AND -> 1 }
{ 1 INVERT 1 AND -> 0 }

0        CONSTANT 0S
0 INVERT CONSTANT 1S

{ 0S INVERT -> 1S }
{ 1S INVERT -> 0S }

{ 0S 0S AND -> 0S }
{ 0S 1S AND -> 0S }
{ 1S 0S AND -> 0S }
```

```
{ 1S 1S AND -> 1S }

{ 0S 0S OR -> 0S }
{ 0S 1S OR -> 1S }
{ 1S 0S OR -> 1S }
{ 1S 1S OR -> 1S }

{ 0S 0S XOR -> 0S }
{ 0S 1S XOR -> 1S }
{ 1S 0S XOR -> 1S }
{ 1S 1S XOR -> 0S }
```

F.4.3 Shifts

```
TESTING 2* 2/ LSHIFT RSHIFT

( WE TRUST 1S, INVERT, AND BITSSET?; WE WILL CONFIRM RSHIFT LATER )
1S 1 RSHIFT INVERT CONSTANT MSB
{ MSB BITSSET? -> 0 0 }

{ 0S 2* -> 0S }
{ 1 2* -> 2 }
{ 4000 2* -> 8000 }
{ 1S 2* 1 XOR -> 1S }
{ MSB 2* -> 0S }

{ 0S 2/ -> 0S }
{ 1 2/ -> 0 }
{ 4000 2/ -> 2000 }
{ 1S 2/ -> 1S }          \ MSB PROPOGATED
{ 1S 1 XOR 2/ -> 1S }
{ MSB 2/ MSB AND -> MSB }

{ 1 0 LSHIFT -> 1 }
{ 1 1 LSHIFT -> 2 }
{ 1 2 LSHIFT -> 4 }
{ 1 F LSHIFT -> 8000 }          \ BIGGEST GUARANTEED SHIFT
{ 1S 1 LSHIFT 1 XOR -> 1S }
{ MSB 1 LSHIFT -> 0 }

{ 1 0 RSHIFT -> 1 }
{ 1 1 RSHIFT -> 0 }
{ 2 1 RSHIFT -> 1 }
{ 4 2 RSHIFT -> 1 }
{ 8000 F RSHIFT -> 1 }              \ BIGGEST
{ MSB 1 RSHIFT MSB AND -> 0 }       \ RSHIFT ZERO FILLS MSBS
{ MSB 1 RSHIFT 2* -> MSB }
```

F.4.4 Comparisons

```
TESTING COMPARISONS: 0= = 0< < > U< MIN MAX

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>

{ 0 0= -> <TRUE> }
{ 1 0= -> <FALSE> }
{ 2 0= -> <FALSE> }
{ -1 0= -> <FALSE> }
{ MAX-UINT 0= -> <FALSE> }
{ MIN-INT 0= -> <FALSE> }
{ MAX-INT 0= -> <FALSE> }

{ 0 0 = -> <TRUE> }
{ 1 1 = -> <TRUE> }
{ -1 -1 = -> <TRUE> }
{ 1 0 = -> <FALSE> }
{ -1 0 = -> <FALSE> }
{ 0 1 = -> <FALSE> }
{ 0 -1 = -> <FALSE> }


{ 0 0< -> <FALSE> }
{ -1 0< -> <TRUE> }
{ MIN-INT 0< -> <TRUE> }
{ 1 0< -> <FALSE> }
{ MAX-INT 0< -> <FALSE> }

{ 0 1 < -> <TRUE> }
{ 1 2 < -> <TRUE> }
{ -1 0 < -> <TRUE> }
{ -1 1 < -> <TRUE> }
{ MIN-INT 0 < -> <TRUE> }
{ MIN-INT MAX-INT < -> <TRUE> }
{ 0 MAX-INT < -> <TRUE> }
{ 0 0 < -> <FALSE> }
{ 1 1 < -> <FALSE> }
{ 1 0 < -> <FALSE> }
{ 2 1 < -> <FALSE> }
{ 0 -1 < -> <FALSE> }
{ 1 -1 < -> <FALSE> }
{ 0 MIN-INT < -> <FALSE> }
{ MAX-INT MIN-INT < -> <FALSE> }
{ MAX-INT 0 < -> <FALSE> }

{ 0 1 > -> <FALSE> }
{ 1 2 > -> <FALSE> }
{ -1 0 > -> <FALSE> }
{ -1 1 > -> <FALSE> }
{ MIN-INT 0 > -> <FALSE> }
{ MIN-INT MAX-INT > -> <FALSE> }
{ 0 MAX-INT > -> <FALSE> }
{ 0 0 > -> <FALSE> }
{ 1 1 > -> <FALSE> }
{ 1 0 > -> <TRUE> }
```

```
{ 2 1 > -> <TRUE> }
{ 0 -1 > -> <TRUE> }
{ 1 -1 > -> <TRUE> }
{ 0 MIN-INT > -> <TRUE> }
{ MAX-INT MIN-INT > -> <TRUE> }
{ MAX-INT 0 > -> <TRUE> }

{ 0 1 U< -> <TRUE> }
{ 1 2 U< -> <TRUE> }
{ 0 MID-UINT U< -> <TRUE> }
{ 0 MAX-UINT U< -> <TRUE> }
{ MID-UINT MAX-UINT U< -> <TRUE> }
{ 0 0 U< -> <FALSE> }
{ 1 1 U< -> <FALSE> }
{ 1 0 U< -> <FALSE> }
{ 2 1 U< -> <FALSE> }
{ MID-UINT 0 U< -> <FALSE> }
{ MAX-UINT 0 U< -> <FALSE> }
{ MAX-UINT MID-UINT U< -> <FALSE> }

{ 0 1 MIN -> 0 }
{ 1 2 MIN -> 1 }
{ -1 0 MIN -> -1 }
{ -1 1 MIN -> -1 }
{ MIN-INT 0 MIN -> MIN-INT }
{ MIN-INT MAX-INT MIN -> MIN-INT }
{ 0 MAX-INT MIN -> 0 }
{ 0 0 MIN -> 0 }
{ 1 1 MIN -> 1 }
{ 1 0 MIN -> 0 }
{ 2 1 MIN -> 1 }
{ 0 -1 MIN -> -1 }
{ 1 -1 MIN -> -1 }
{ 0 MIN-INT MIN -> MIN-INT }
{ MAX-INT MIN-INT MIN -> MIN-INT }
{ MAX-INT 0 MIN -> 0 }


{ 0 1 MAX -> 1 }
{ 1 2 MAX -> 2 }
{ -1 0 MAX -> 0 }
{ -1 1 MAX -> 1 }
{ MIN-INT 0 MAX -> 0 }
{ MIN-INT MAX-INT MAX -> MAX-INT }
{ 0 MAX-INT MAX -> MAX-INT }
{ 0 0 MAX -> 0 }
{ 1 1 MAX -> 1 }
{ 1 0 MAX -> 1 }
{ 2 1 MAX -> 2 }
{ 0 -1 MAX -> 0 }
{ 1 -1 MAX -> 1 }
{ 0 MIN-INT MAX -> 0 }
{ MAX-INT MIN-INT MAX -> MAX-INT }
{ MAX-INT 0 MAX -> MAX-INT }
```

F.4.5 Stack Operators

```
TESTING STACK OPS: 2DROP 2DUP 2OVER 2SWAP ?DUP DEPTH DROP DUP OVER ROT SWAP

{ 1 2 2DROP -> }
{ 1 2 2DUP -> 1 2 1 2 }
{ 1 2 3 4 2OVER -> 1 2 3 4 1 2 }
{ 1 2 3 4 2SWAP -> 3 4 1 2 }
{ 0 ?DUP -> 0 }
{ 1 ?DUP -> 1 1 }
{ -1 ?DUP -> -1 -1 }
{ DEPTH -> 0 }
{ 0 DEPTH -> 0 1 }
{ 0 1 DEPTH -> 0 1 2 }
{ 0 DROP -> }
{ 1 2 DROP -> 1 }
{ 1 DUP -> 1 1 }
{ 1 2 OVER -> 1 2 1 }
{ 1 2 3 ROT -> 2 3 1 }
{ 1 2 SWAP -> 2 1 }
```

F.4.6 Return Stack Operators

```
TESTING >R R> R@

{ :  GR1 >R R> ; -> }
{ :  GR2 >R R@ R> DROP ; -> }
{ 123 GR1 -> 123 }
{ 123 GR2 -> 123 }
{ 1S GR1 -> 1S }      ( RETURN STACK HOLDS CELLS )
```

F.4.7 Addition and Subtraction

```
TESTING ADD/SUBTRACT: + - 1+ 1- ABS NEGATE

{ 0 5 + -> 5 }
{ 5 0 + -> 5 }
{ 0 -5 + -> -5 }
{ -5 0 + -> -5 }
{ 1 2 + -> 3 }
{ 1 -2 + -> -1 }
{ -1 2 + -> 1 }
{ -1 -2 + -> -3 }
{ -1 1 + -> 0 }
{ MID-UINT 1 + -> MID-UINT+1 }

{ 0 5 - -> -5 }
{ 5 0 - -> 5 }
{ 0 -5 - -> 5 }
{ -5 0 - -> -5 }
{ 1 2 - -> -1 }
{ 1 -2 - -> 3 }
{ -1 2 - -> -3 }
{ -1 -2 - -> 1 }
```

```
{ 0 1 - -> -1 }
{ MID-UINT+1 1 - -> MID-UINT }

{ 0 1+ -> 1 }
{ -1 1+ -> 0 }
{ 1 1+ -> 2 }
{ MID-UINT 1+ -> MID-UINT+1 }

{ 2 1- -> 1 }
{ 1 1- -> 0 }
{ 0 1- -> -1 }
{ MID-UINT+1 1- -> MID-UINT }

{ 0 NEGATE -> 0 }
{ 1 NEGATE -> -1 }
{ -1 NEGATE -> 1 }
{ 2 NEGATE -> -2 }
{ -2 NEGATE -> 2 }

{ 0 ABS -> 0 }
{ 1 ABS -> 1 }
{ -1 ABS -> 1 }
{ MIN-INT ABS -> MID-UINT+1 }
```

F.4.8 Multiplication

```
TESTING MULTIPLY: S>D * M* UM*

{ 0 S>D -> 0 0 }
{ 1 S>D -> 1 0 }
{ 2 S>D -> 2 0 }
{ -1 S>D -> -1 -1 }
{ -2 S>D -> -2 -1 }
{ MIN-INT S>D -> MIN-INT -1 }
{ MAX-INT S>D -> MAX-INT 0 }

{ 0 0 M* -> 0 S>D }
{ 0 1 M* -> 0 S>D }
{ 1 0 M* -> 0 S>D }
{ 1 2 M* -> 2 S>D }
{ 2 1 M* -> 2 S>D }
{ 3 3 M* -> 9 S>D }
{ -3 3 M* -> -9 S>D }
{ 3 -3 M* -> -9 S>D }
{ -3 -3 M* -> 9 S>D }
{ 0 MIN-INT M* -> 0 S>D }
{ 1 MIN-INT M* -> MIN-INT S>D }
{ 2 MIN-INT M* -> 0 1S }
{ 0 MAX-INT M* -> 0 S>D }
{ 1 MAX-INT M* -> MAX-INT S>D }
{ 2 MAX-INT M* -> MAX-INT 1 LSHIFT 0 }
{ MIN-INT MIN-INT M* -> 0 MSB 1 RSHIFT }
{ MAX-INT MIN-INT M* -> MSB MSB 2/ }
{ MAX-INT MAX-INT M* -> 1 MSB 2/ INVERT }

{ 0 0 * -> 0 }         \ TEST IDENTITIES
```

```
{ 0 1 * -> 0 }
{ 1 0 * -> 0 }
{ 1 2 * -> 2 }
{ 2 1 * -> 2 }
{ 3 3 * -> 9 }
{ -3 3 * -> -9 }
{ 3 -3 * -> -9 }
{ -3 -3 * -> 9 }

{ MID-UINT+1 1 RSHIFT 2 * -> MID-UINT+1 }
{ MID-UINT+1 2 RSHIFT 4 * -> MID-UINT+1 }
{ MID-UINT+1 1 RSHIFT MID-UINT+1 OR 2 * -> MID-UINT+1 }


{ 0 0 UM* -> 0 0 }
{ 0 1 UM* -> 0 0 }
{ 1 0 UM* -> 0 0 }
{ 1 2 UM* -> 2 0 }
{ 2 1 UM* -> 2 0 }
{ 3 3 UM* -> 9 0 }

{ MID-UINT+1 1 RSHIFT 2 UM* -> MID-UINT+1 0 }
{ MID-UINT+1 2 UM* -> 0 1 }
{ MID-UINT+1 4 UM* -> 0 2 }
{ 1S 2 UM* -> 1S 1 LSHIFT 1 }
{ MAX-UINT MAX-UINT UM* -> 1 1 INVERT }
```

F.4.9 Division

```
TESTING DIVIDE: FM/MOD SM/REM UM/MOD /MOD / MOD */ */MOD

{ 0 S>D 1 FM/MOD -> 0 0 }
{ 1 S>D 1 FM/MOD -> 0 1 }
{ 2 S>D 1 FM/MOD -> 0 2 }
{ -1 S>D 1 FM/MOD -> 0 -1 }
{ -2 S>D 1 FM/MOD -> 0 -2 }
{ 0 S>D -1 FM/MOD -> 0 0 }
{ 1 S>D -1 FM/MOD -> 0 -1 }
{ 2 S>D -1 FM/MOD -> 0 -2 }
{ -1 S>D -1 FM/MOD -> 0 1 }
{ -2 S>D -1 FM/MOD -> 0 2 }
{ 2 S>D 2 FM/MOD -> 0 1 }
{ -1 S>D -1 FM/MOD -> 0 1 }
{ -2 S>D -2 FM/MOD -> 0 1 }
{ 7 S>D 3 FM/MOD -> 1 2 }
{ 7 S>D -3 FM/MOD -> -2 -3 }
{ -7 S>D 3 FM/MOD -> 2 -3 }
{ -7 S>D -3 FM/MOD -> -1 2 }
{ MAX-INT S>D 1 FM/MOD -> 0 MAX-INT }
{ MIN-INT S>D 1 FM/MOD -> 0 MIN-INT }
{ MAX-INT S>D MAX-INT FM/MOD -> 0 1 }
{ MIN-INT S>D MIN-INT FM/MOD -> 0 1 }
{ 1S 1 4 FM/MOD -> 3 MAX-INT }
{ 1 MIN-INT M* 1 FM/MOD -> 0 MIN-INT }
{ 1 MIN-INT M* MIN-INT FM/MOD -> 0 1 }
```

```
{ 2 MIN-INT M* 2 FM/MOD -> 0 MIN-INT }
{ 2 MIN-INT M* MIN-INT FM/MOD -> 0 2 }
{ 1 MAX-INT M* 1 FM/MOD -> 0 MAX-INT }
{ 1 MAX-INT M* MAX-INT FM/MOD -> 0 1 }
{ 2 MAX-INT M* 2 FM/MOD -> 0 MAX-INT }
{ 2 MAX-INT M* MAX-INT FM/MOD -> 0 2 }
{ MIN-INT MIN-INT M* MIN-INT FM/MOD -> 0 MIN-INT }
{ MIN-INT MAX-INT M* MIN-INT FM/MOD -> 0 MAX-INT }
{ MIN-INT MAX-INT M* MAX-INT FM/MOD -> 0 MIN-INT }
{ MAX-INT MAX-INT M* MAX-INT FM/MOD -> 0 MAX-INT }

{ 0 S>D 1 SM/REM -> 0 0 }
{ 1 S>D 1 SM/REM -> 0 1 }
{ 2 S>D 1 SM/REM -> 0 2 }
{ -1 S>D 1 SM/REM -> 0 -1 }
{ -2 S>D 1 SM/REM -> 0 -2 }
{ 0 S>D -1 SM/REM -> 0 0 }
{ 1 S>D -1 SM/REM -> 0 -1 }
{ 2 S>D -1 SM/REM -> 0 -2 }
{ -1 S>D -1 SM/REM -> 0 1 }
{ -2 S>D -1 SM/REM -> 0 2 }
{ 2 S>D 2 SM/REM -> 0 1 }
{ -1 S>D -1 SM/REM -> 0 1 }
{ -2 S>D -2 SM/REM -> 0 1 }
{ 7 S>D 3 SM/REM -> 1 2 }
{ 7 S>D -3 SM/REM -> 1 -2 }
{ -7 S>D 3 SM/REM -> -1 -2 }
{ -7 S>D -3 SM/REM -> -1 2 }
{ MAX-INT S>D 1 SM/REM -> 0 MAX-INT }
{ MIN-INT S>D 1 SM/REM -> 0 MIN-INT }
{ MAX-INT S>D MAX-INT SM/REM -> 0 1 }
{ MIN-INT S>D MIN-INT SM/REM -> 0 1 }
{ 1S 1 4 SM/REM -> 3 MAX-INT }
{ 2 MIN-INT M* 2 SM/REM -> 0 MIN-INT }
{ 2 MIN-INT M* MIN-INT SM/REM -> 0 2 }
{ 2 MAX-INT M* 2 SM/REM -> 0 MAX-INT }
{ 2 MAX-INT M* MAX-INT SM/REM -> 0 2 }
{ MIN-INT MIN-INT M* MIN-INT SM/REM -> 0 MIN-INT }
{ MIN-INT MAX-INT M* MIN-INT SM/REM -> 0 MAX-INT }
{ MIN-INT MAX-INT M* MAX-INT SM/REM -> 0 MIN-INT }
{ MAX-INT MAX-INT M* MAX-INT SM/REM -> 0 MAX-INT }

{ 0 0 1 UM/MOD -> 0 0 }
{ 1 0 1 UM/MOD -> 0 1 }
{ 1 0 2 UM/MOD -> 1 0 }
{ 3 0 2 UM/MOD -> 1 1 }
{ MAX-UINT 2 UM* 2 UM/MOD -> 0 MAX-UINT }
{ MAX-UINT 2 UM* MAX-UINT UM/MOD -> 0 2 }
{ MAX-UINT MAX-UINT UM* MAX-UINT UM/MOD -> 0 MAX-UINT }

:  IFFLOORED
  [ -3 2 / -2 = INVERT ] LITERAL IF POSTPONE \ THEN ;
:  IFSYM
  [ -3 2 / -1 = INVERT ] LITERAL IF POSTPONE \ THEN ;
```

```
\ THE SYSTEM MIGHT DO EITHER FLOORED OR SYMMETRIC DIVISION.
\ SINCE WE HAVE ALREADY TESTED M*, FM/MOD, AND SM/REM WE CAN USE THEM
\ IN TEST.

IFFLOORED   :  T/MOD >R S>D R> FM/MOD ;
IFFLOORED   :  T/ T/MOD SWAP DROP ;
IFFLOORED   :  TMOD T/MOD DROP ;
IFFLOORED   :  T*/MOD >R M* R> FM/MOD ;
IFFLOORED   :  T*/ T*/MOD SWAP DROP ;

IFSYM       :  T/MOD >R S>D R> SM/REM ;
IFSYM       :  T/ T/MOD SWAP DROP ;
IFSYM       :  TMOD T/MOD DROP ;
IFSYM       :  T*/MOD >R M* R> SM/REM ;
IFSYM       :  T*/ T*/MOD SWAP DROP ;

{ 0 1 /MOD -> 0 1 T/MOD }
{ 1 1 /MOD -> 1 1 T/MOD }
{ 2 1 /MOD -> 2 1 T/MOD }
{ -1 1 /MOD -> -1 1 T/MOD }
{ -2 1 /MOD -> -2 1 T/MOD }
{ 0 -1 /MOD -> 0 -1 T/MOD }
{ 1 -1 /MOD -> 1 -1 T/MOD }
{ 2 -1 /MOD -> 2 -1 T/MOD }
{ -1 -1 /MOD -> -1 -1 T/MOD }
{ -2 -1 /MOD -> -2 -1 T/MOD }
{ 2 2 /MOD -> 2 2 T/MOD }
{ -1 -1 /MOD -> -1 -1 T/MOD }
{ -2 -2 /MOD -> -2 -2 T/MOD }
{ 7 3 /MOD -> 7 3 T/MOD }
{ 7 -3 /MOD -> 7 -3 T/MOD }
{ -7 3 /MOD -> -7 3 T/MOD }
{ -7 -3 /MOD -> -7 -3 T/MOD }
{ MAX-INT 1 /MOD -> MAX-INT 1 T/MOD }
{ MIN-INT 1 /MOD -> MIN-INT 1 T/MOD }
{ MAX-INT MAX-INT /MOD -> MAX-INT MAX-INT T/MOD }
{ MIN-INT MIN-INT /MOD -> MIN-INT MIN-INT T/MOD }

{ 0 1 / -> 0 1 T/ }
{ 1 1 / -> 1 1 T/ }
{ 2 1 / -> 2 1 T/ }
{ -1 1 / -> -1 1 T/ }
{ -2 1 / -> -2 1 T/ }
{ 0 -1 / -> 0 -1 T/ }
{ 1 -1 / -> 1 -1 T/ }
{ 2 -1 / -> 2 -1 T/ }
{ -1 -1 / -> -1 -1 T/ }
{ -2 -1 / -> -2 -1 T/ }
{ 2 2 / -> 2 2 T/ }
{ -1 -1 / -> -1 -1 T/ }
{ -2 -2 / -> -2 -2 T/ }
{ 7 3 / -> 7 3 T/ }
{ 7 -3 / -> 7 -3 T/ }
{ -7 3 / -> -7 3 T/ }
{ -7 -3 / -> -7 -3 T/ }
```

```
{ MAX-INT 1 / -> MAX-INT 1 T/ }
{ MIN-INT 1 / -> MIN-INT 1 T/ }
{ MAX-INT MAX-INT / -> MAX-INT MAX-INT T/ }
{ MIN-INT MIN-INT / -> MIN-INT MIN-INT T/ }

{ 0 1 MOD -> 0 1 TMOD }
{ 1 1 MOD -> 1 1 TMOD }
{ 2 1 MOD -> 2 1 TMOD }
{ -1 1 MOD -> -1 1 TMOD }
{ -2 1 MOD -> -2 1 TMOD }
{ 0 -1 MOD -> 0 -1 TMOD }
{ 1 -1 MOD -> 1 -1 TMOD }
{ 2 -1 MOD -> 2 -1 TMOD }
{ -1 -1 MOD -> -1 -1 TMOD }
{ -2 -1 MOD -> -2 -1 TMOD }
{ 2 2 MOD -> 2 2 TMOD }
{ -1 -1 MOD -> -1 -1 TMOD }
{ -2 -2 MOD -> -2 -2 TMOD }
{ 7 3 MOD -> 7 3 TMOD }
{ 7 -3 MOD -> 7 -3 TMOD }
{ -7 3 MOD -> -7 3 TMOD }
{ -7 -3 MOD -> -7 -3 TMOD }
{ MAX-INT 1 MOD -> MAX-INT 1 TMOD }
{ MIN-INT 1 MOD -> MIN-INT 1 TMOD }
{ MAX-INT MAX-INT MOD -> MAX-INT MAX-INT TMOD }
{ MIN-INT MIN-INT MOD -> MIN-INT MIN-INT TMOD }

{ 0 2 1 */ -> 0 2 1 T*/ }
{ 1 2 1 */ -> 1 2 1 T*/ }
{ 2 2 1 */ -> 2 2 1 T*/ }
{ -1 2 1 */ -> -1 2 1 T*/ }
{ -2 2 1 */ -> -2 2 1 T*/ }
{ 0 2 -1 */ -> 0 2 -1 T*/ }
{ 1 2 -1 */ -> 1 2 -1 T*/ }
{ 2 2 -1 */ -> 2 2 -1 T*/ }
{ -1 2 -1 */ -> -1 2 -1 T*/ }
{ -2 2 -1 */ -> -2 2 -1 T*/ }
{ 2 2 2 */ -> 2 2 2 T*/ }
{ -1 2 -1 */ -> -1 2 -1 T*/ }
{ -2 2 -2 */ -> -2 2 -2 T*/ }
{ 7 2 3 */ -> 7 2 3 T*/ }
{ 7 2 -3 */ -> 7 2 -3 T*/ }
{ -7 2 3 */ -> -7 2 3 T*/ }
{ -7 2 -3 */ -> -7 2 -3 T*/ }
{ MAX-INT 2 MAX-INT */ -> MAX-INT 2 MAX-INT T*/ }
{ MIN-INT 2 MIN-INT */ -> MIN-INT 2 MIN-INT T*/ }

{ 0 2 1 */MOD -> 0 2 1 T*/MOD }
{ 1 2 1 */MOD -> 1 2 1 T*/MOD }
{ 2 2 1 */MOD -> 2 2 1 T*/MOD }
{ -1 2 1 */MOD -> -1 2 1 T*/MOD }
{ -2 2 1 */MOD -> -2 2 1 T*/MOD }
{ 0 2 -1 */MOD -> 0 2 -1 T*/MOD }
{ 1 2 -1 */MOD -> 1 2 -1 T*/MOD }
```

```
{ 2 2 -1 */MOD -> 2 2 -1 T*/MOD }
{ -1 2 -1 */MOD -> -1 2 -1 T*/MOD }
{ -2 2 -1 */MOD -> -2 2 -1 T*/MOD }
{ 2 2 2 */MOD -> 2 2 2 T*/MOD }
{ -1 2 -1 */MOD -> -1 2 -1 T*/MOD }
{ -2 2 -2 */MOD -> -2 2 -2 T*/MOD }
{ 7 2 3 */MOD -> 7 2 3 T*/MOD }
{ 7 2 -3 */MOD -> 7 2 -3 T*/MOD }
{ -7 2 3 */MOD -> -7 2 3 T*/MOD }
{ -7 2 -3 */MOD -> -7 2 -3 T*/MOD }
{ MAX-INT 2 MAX-INT */MOD -> MAX-INT 2 MAX-INT T*/MOD }
{ MIN-INT 2 MIN-INT */MOD -> MIN-INT 2 MIN-INT T*/MOD }
```

F.4.10 Memory

```
TESTING HERE , @ !  CELL+ CELLS C, C@ C! CHAR+ CHARS 2@ 2!  ALIGN ALIGNED +!  ALLOT

HERE 1 ALLOT
HERE
CONSTANT 2NDA
CONSTANT 1STA
{ 1STA 2NDA U< -> <TRUE> }   \ HERE MUST GROW WITH ALLOT
{ 1STA 1+ -> 2NDA }          \ ...  BY ONE ADDRESS UNIT
( MISSING TEST: NEGATIVE ALLOT )

HERE 1 ,
HERE 2 ,
CONSTANT 2ND
CONSTANT 1ST
{ 1ST 2ND U< -> <TRUE> }   \ HERE MUST GROW WITH ALLOT
{ 1ST CELL+ -> 2ND }       \ ...  BY ONE CELL
{ 1ST 1 CELLS + -> 2ND }
{ 1ST @ 2ND @ -> 1 2 }
{ 5 1ST ! -> }
{ 1ST @ 2ND @ -> 5 2 }
{ 6 2ND ! -> }
{ 1ST @ 2ND @ -> 5 6 }
{ 1ST 2@ -> 6 5 }
{ 2 1 1ST 2!  -> }
{ 1ST 2@ -> 2 1 }
{ 1S 1ST ! 1ST @ -> 1S }   \ CAN STORE CELL-WIDE VALUE

HERE 1 C,
HERE 2 C,
CONSTANT 2NDC
CONSTANT 1STC
{ 1STC 2NDC U< -> <TRUE> }   \ HERE MUST GROW WITH ALLOT
{ 1STC CHAR+ -> 2NDC }       \ ...  BY ONE CHAR
{ 1STC 1 CHARS + -> 2NDC }
{ 1STC C@ 2NDC C@ -> 1 2 }
{ 3 1STC C! -> }
{ 1STC C@ 2NDC C@ -> 3 2 }
{ 4 2NDC C! -> }
{ 1STC C@ 2NDC C@ -> 3 4 }

ALIGN 1 ALLOT HERE ALIGN HERE 3 CELLS ALLOT
CONSTANT A-ADDR CONSTANT UA-ADDR
```

```
{ UA-ADDR ALIGNED -> A-ADDR }
{ 1 A-ADDR C! A-ADDR C@ -> 1 }
{ 1234 A-ADDR ! A-ADDR @ -> 1234 }
{ 123 456 A-ADDR 2!  A-ADDR 2@ -> 123 456 }
{ 2 A-ADDR CHAR+ C! A-ADDR CHAR+ C@ -> 2 }
{ 3 A-ADDR CELL+ C! A-ADDR CELL+ C@ -> 3 }
{ 1234 A-ADDR CELL+ !  A-ADDR CELL+ @ -> 1234 }
{ 123 456 A-ADDR CELL+ 2!  A-ADDR CELL+ 2@ -> 123 456 }

:  BITS ( X - U )
  0 SWAP BEGIN DUP WHILE DUP MSB AND IF >R 1+ R> THEN 2* REPEAT DROP ;
( CHARACTERS >= 1 AU, <= SIZE OF CELL, >= 8 BITS )
{ 1 CHARS 1 < -> <FALSE> }
{ 1 CHARS 1 CELLS > -> <FALSE> }
( TBD: HOW TO FIND NUMBER OF BITS? )

( CELLS >= 1 AU, INTEGRAL MULTIPLE OF CHAR SIZE, >= 16 BITS )
{ 1 CELLS 1 < -> <FALSE> }
{ 1 CELLS 1 CHARS MOD -> 0 }
{ 1S BITS 10 < -> <FALSE> }

{ 0 1ST ! -> }
{ 1 1ST +!  -> }
{ 1ST @ -> 1 }
{ -1 1ST +!  1ST @ -> 0 }
```

F.4.11 Characters

```
TESTING CHAR [CHAR] [ ] BL S"

{ BL -> 20 }
{ CHAR X -> 58 }
{ CHAR HELLO -> 48 }
{ :  GC1 [CHAR] X ; -> }
{ :  GC2 [CHAR] HELLO ; -> }
{ GC1 -> 58 }
{ GC2 -> 48 }
{ :  GC3 [ GC1 ] LITERAL ; -> }
{ GC3 -> 58 }
{ :  GC4 S" XY" ; -> }
{ GC4 SWAP DROP -> 2 }
{ GC4 DROP DUP C@ SWAP CHAR+ C@ -> 58 59 }
```

F.4.12 Dictionary

```
TESTING ' ['] FIND EXECUTE IMMEDIATE COUNT LITERAL POSTPONE STATE

{ :  GT1 123 ; -> }
{ ' GT1 EXECUTE -> 123 } { :  GT2 ['] GT1 ; IMMEDIATE -> }
{ GT2 EXECUTE -> 123 } 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
{ GT1STRING FIND -> ' GT1 -1 }
{ GT2STRING FIND -> ' GT2 1 }
( HOW TO SEARCH FOR NON-EXISTENT WORD? ) { :  GT3 GT2 LITERAL ; -> }
{ GT3 -> ' GT1 } { GT1STRING COUNT -> GT1STRING CHAR+ 3 }
```

```
{ :  GT4 POSTPONE GT1 ; IMMEDIATE -> }
{ :  GT5 GT4 ; -> }
{ GT5 -> 123 } { :  GT6 345 ; IMMEDIATE -> }
{ :  GT7 POSTPONE GT6 ; -> }
{ GT7 -> 345 }

{ :  GT8 STATE @ ; IMMEDIATE -> }
{ GT8 -> 0 }
{ :  GT9 GT8 LITERAL ; -> }
{ GT9 0= -> <FALSE> }
```

F.4.13 Flow Control

```
TESTING IF ELSE THEN BEGIN WHILE REPEAT UNTIL RECURSE

{ :  GI1 IF 123 THEN ; -> }
{ :  GI2 IF 123 ELSE 234 THEN ; -> }
{ 0 GI1 -> }
{ 1 GI1 -> 123 }
{ -1 GI1 -> 123 }
{ 0 GI2 -> 234 }
{ 1 GI2 -> 123 }
{ -1 GI1 -> 123 }

{ :  GI3 BEGIN DUP 5 < WHILE DUP 1+ REPEAT ; -> }
{ 0 GI3 -> 0 1 2 3 4 5 }
{ 4 GI3 -> 4 5 }
{ 5 GI3 -> 5 }
{ 6 GI3 -> 6 }

{ :  GI4 BEGIN DUP 1+ DUP 5 > UNTIL ; -> }
{ 3 GI4 -> 3 4 5 6 }
{ 5 GI4 -> 5 6 }
{ 6 GI4 -> 6 7 }

{ :  GI5 BEGIN DUP 2 > WHILE
  DUP 5 < WHILE DUP 1+ REPEAT
  123 ELSE 345 THEN ; -> }
{ 1 GI5 -> 1 345 }
{ 2 GI5 -> 2 345 }
{ 3 GI5 -> 3 4 5 123 }
{ 4 GI5 -> 4 5 123 }
{ 5 GI5 -> 5 123 }

{ :  GI6 ( N - 0,1,..,N ) DUP IF DUP >R 1- RECURSE R> THEN ; -> }
{ 0 GI6 -> 0 }
{ 1 GI6 -> 0 1 }
{ 2 GI6 -> 0 1 2 }
{ 3 GI6 -> 0 1 2 3 }
{ 4 GI6 -> 0 1 2 3 4 }
```

F.4.14 Counted Loops

```
TESTING DO LOOP +LOOP I J UNLOOP LEAVE EXIT

{ :  GD1 DO I LOOP ; -> }
```

```
{ 4 1 GD1 -> 1 2 3 }
{ 2 -1 GD1 -> -1 0 1 }
{ MID-UINT+1 MID-UINT GD1 -> MID-UINT }

{ :  GD2 DO I -1 +LOOP ; -> }
{ 1 4 GD2 -> 4 3 2 1 }
{ -1 2 GD2 -> 2 1 0 -1 }
{ MID-UINT MID-UINT+1 GD2 -> MID-UINT+1 MID-UINT }

{ :  GD3 DO 1 0 DO J LOOP LOOP ; -> }
{ 4 1 GD3 -> 1 2 3 }
{ 2 -1 GD3 -> -1 0 1 }
{ MID-UINT+1 MID-UINT GD3 -> MID-UINT }

{ :  GD4 DO 1 0 DO J LOOP -1 +LOOP ; -> }
{ 1 4 GD4 -> 4 3 2 1 }
{ -1 2 GD4 -> 2 1 0 -1 }
{ MID-UINT MID-UINT+1 GD4 -> MID-UINT+1 MID-UINT }

{ :  GD5 123 SWAP 0 DO I 4 > IF DROP 234 LEAVE THEN LOOP ; -> }
{ 1 GD5 -> 123 }
{ 5 GD5 -> 123 }
{ 6 GD5 -> 234 }

{ :  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 ; -> }
{ 1 GD6 -> 1 }
{ 2 GD6 -> 3 }
{ 3 GD6 -> 4 1 2 }
```

F.4.15 Defining Words

```
TESTING DEFINING WORDS: :  ; CONSTANT VARIABLE CREATE DOES> >BODY

{ 123 CONSTANT X123 -> }
{ X123 -> 123 }
{ :  EQU CONSTANT ; -> }
{ X123 EQU Y123 -> }
{ Y123 -> 123 }

{ VARIABLE V1 -> }
{ 123 V1 !  -> }
{ V1 @ -> 123 }

{ :  NOP : POSTPONE ; ; -> }
{ NOP NOP1 NOP NOP2 -> }
{ NOP1 -> }
{ NOP2 -> }

{ :  DOES1 DOES> @ 1 + ; -> }
{ :  DOES2 DOES> @ 2 + ; -> }
{ CREATE CR1 -> }
{ CR1 -> HERE }
```

```
{ ' CR1 >BODY -> HERE }
{ 1 , -> }
{ CR1 @ -> 1 }
{ DOES1 -> }
{ CR1 -> 2 }
{ DOES2 -> }
{ CR1 -> 3 }

{ : WEIRD: CREATE DOES> 1 + DOES> 2 + ; -> }
{ WEIRD: W1 -> }
{ ' W1 >BODY -> HERE }
{ W1 -> HERE 1 + }
{ W1 -> HERE 2 + }
```

F.4.16 Evaluate

```
TESTING EVALUATE

: GE1 S" 123" ; IMMEDIATE
: GE2 S" 123 1+" ; IMMEDIATE
: GE3 S" : GE4 345 ;" ;
: GE5 EVALUATE ; IMMEDIATE

{ GE1 EVALUATE -> 123 }       ( TEST EVALUATE IN INTERP. STATE )
{ GE2 EVALUATE -> 124 }
{ GE3 EVALUATE -> }
{ GE4 -> 345 }

{ : GE6 GE1 GE5 ; -> }        ( TEST EVALUATE IN COMPILE STATE )
{ GE6 -> 123 }
{ : GE7 GE2 GE5 ; -> }
{ GE7 -> 124 }
```

F.4.17 Parser Input Source Control

```
TESTING SOURCE >IN WORD

: GS1 S" SOURCE" 2DUP EVALUATE
  >R SWAP >R = R> R> = ;
{ GS1 -> <TRUE> <TRUE> }

VARIABLE SCANS
: RESCAN? -1 SCANS +! SCANS @ IF 0 >IN ! THEN ;

{ 2 SCANS !
345 RESCAN?
-> 345 345 }

: GS2 5 SCANS ! S" 123 RESCAN?" EVALUATE ;
{ GS2 -> 123 123 123 123 123 }

: GS3 WORD COUNT SWAP C@ ;
{ BL GS3 HELLO -> 5 CHAR H }
{ CHAR " GS3 GOODBYE" -> 7 CHAR G }
{ BL GS3
DROP -> 0 }         \ BLANK LINE RETURN ZERO-LENGTH STRING
```

```
:  GS4 SOURCE >IN ! DROP ;
{ GS4 123 456
-> }
```

F.4.18 Number Patterns

```
TESTING <# # #S #> HOLD SIGN BASE >NUMBER HEX DECIMAL

:  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 ;

:  GP1 <# 41 HOLD 42 HOLD 0 0 #> S" BA" S= ;
{ GP1 -> <TRUE> }

:  GP2 <# -1 SIGN 0 SIGN -1 SIGN 0 0 #> S" -" S= ;
{ GP2 -> <TRUE> }

:  GP3 <# 1 0 # # #> S" 01" S= ;
{ GP3 -> <TRUE> }

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


:  GP4 <# 1 0 #S #> S" 1" S= ;
{ GP4 -> <TRUE> }

:  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 ! ;
{ GP5 -> <TRUE> }

:  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 ;
{ GP6 -> <TRUE> }

:  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 ! ;
{ GP7 -> <TRUE> }

\ >NUMBER TESTS


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 ;

{ 0 0 GN' 0' >NUMBER -> 0 0 GN-CONSUMED }
{ 0 0 GN' 1' >NUMBER -> 1 0 GN-CONSUMED }
{ 1 0 GN' 1' >NUMBER -> BASE @ 1+ 0 GN-CONSUMED }
{ 0 0 GN' -' >NUMBER -> 0 0 GN-STRING }   \ SHOULD FAIL TO CONVERT THESE
{ 0 0 GN' +' >NUMBER -> 0 0 GN-STRING }
{ 0 0 GN' .' >NUMBER -> 0 0 GN-STRING }

:  >NUMBER-BASED
  BASE @ >R BASE ! >NUMBER R> BASE ! ;

{ 0 0 GN' 2' 10 >NUMBER-BASED -> 2 0 GN-CONSUMED }
{ 0 0 GN' 2' 2 >NUMBER-BASED -> 0 0 GN-STRING }
{ 0 0 GN' F' 10 >NUMBER-BASED -> F 0 GN-CONSUMED }
{ 0 0 GN' G' 10 >NUMBER-BASED -> 0 0 GN-STRING }
{ 0 0 GN' G' MAX-BASE >NUMBER-BASED -> 10 0 GN-CONSUMED }
{ 0 0 GN' Z' MAX-BASE >NUMBER-BASED -> 23 0 GN-CONSUMED }

:  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 ! ;

{ 0 0 2 GN1 -> 0 0 0 }
{ MAX-UINT 0 2 GN1 -> MAX-UINT 0 0 }
{ MAX-UINT DUP 2 GN1 -> MAX-UINT DUP 0 }
{ 0 0 MAX-BASE GN1 -> 0 0 0 }
{ MAX-UINT 0 MAX-BASE GN1 -> MAX-UINT 0 0 }
{ MAX-UINT DUP MAX-BASE GN1 -> MAX-UINT DUP 0 }
```

```
:  GN2 \ ( - 16 10 )
  BASE @ >R HEX BASE @ DECIMAL BASE @ R> BASE ! ;
{ GN2 -> 10 A }
```

F.4.19 Memory Movement

```
TESTING FILL MOVE

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@ ;

{ FBUF 0 20 FILL -> }
{ SEEBUF -> 00 00 00 }

{ FBUF 1 20 FILL -> }
{ SEEBUF -> 20 00 00 }

{ FBUF 3 20 FILL -> }
{ SEEBUF -> 20 20 20 }

{ FBUF FBUF 3 CHARS MOVE -> }       \ BIZARRE SPECIAL CASE
{ SEEBUF -> 20 20 20 }

{ SBUF FBUF 0 CHARS MOVE -> }
{ SEEBUF -> 20 20 20 }

{ SBUF FBUF 1 CHARS MOVE -> }
{ SEEBUF -> 12 20 20 }

{ SBUF FBUF 3 CHARS MOVE -> }
{ SEEBUF -> 12 34 56 }

{ FBUF FBUF CHAR+ 2 CHARS MOVE -> }
{ SEEBUF -> 12 12 34 }

{ FBUF CHAR+ FBUF 2 CHARS MOVE -> }
{ SEEBUF -> 12 34 34 }
```

F.4.20 Output

```
TESTING OUTPUT: .  ." CR EMIT SPACE SPACES TYPE U.

:  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
;

{ OUTPUT-TEST -> }
```

F.4.21 Input

```
TESTING INPUT: 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
;

{ ACCEPT-TEST -> }
```

F.4.22 Dictionary Search Rules

```
TESTING DICTIONARY SEARCH RULES

{ :  GDX 123 ;   :  GDX GDX 234 ; -> }
{ GDX -> 123 234 }
```

Annex G
(informative)
Change Log

05 Original document based on the dpANS99a basis document distributed as part of the review undertaken by the X3/X3J14 TC in 1999.

06.1 Included changes approved at the Santander meeting, 21–23 October, 2005:

1 Introduction

1) Added reference to the validation suite in annex F.
2) Annex F (Alphabetic list of words) now annex G.

2 Terms, notations and references:

1) Added "Extension Designator" to description of the glossary index line.

3 Usage Requirements

1) Added X:extension-query proposal: Section **3.2.7 Extension queries**, including table **3.6 Forth 200*x* Extensions**.

4 Documentation requirements

1) Added X:deferrd ambiguous conditions.

6 Glossary

1) Added "validation" section to glossary entries, taken from John Hayes' tester suite.
2) Added X:deferred proposal: **6.2.0 `ACTION-OF`**, **6.2.0 `DEFER`**, **6.2.0 `DEFER!`**, **6.2.0 `DEFER@`** and **6.2.0 `IS`**.
3) Added X:parse-name proposal: **6.2.0 `PARSE-NAME`**.

15 Tools Word Set

1) Added X:defined proposal: **15.6.2.0 `[DEFINED]`** and **15.6.2.0 `[UNDEFINED]`**.

F Added the "Validation" annex, with John Hayes' introduction to his tester suite.

G Annex F (Alphabetic list of words) now annex G.

06.2 a) Added validation and reference implementations for new words: **6.2.0 `ACTION-OF`**, **6.2.0 `DEFER`**, **6.2.0 `DEFER!`**, **6.2.0 `DEFER@`**, **6.2.0 `IS`**, **6.2.0 `PARSE-NAME`**, **15.6.2.0 `[DEFINED]`** and **15.6.2.0 `[UNDEFINED]`**.

b) Annex G was sorted numerically, this was changed to a full Alphabetical sort.

07.1 Included changes approved at the Cambridge meeting, 14–15 September, 2006:

– Significant reworking of LaTeX source with a view to publication of source code, and to ease the parsing of the document source.
– Added new "Forward" and "Proposals Process" as defined in X:forward.
– Replaced "X3 Membership" with "200*x* Membership" as defined in X:forward.

3 Usage Requirements

a) Added X:defined, X:ekeys and X:required to table **3.6 Forth 200*x* Extensions**

4 Documentation requirements

a) Added ambiguous conditions for X:required proposal.
b) Altered ambiguous conditions for X:to proposal.

6 Glossary

a) Renamed “Validation” section to “Testing”.
b) Reference implementations labelled with “Implementation”.
c) Replaced “gotten” with “become” in rationale of 6.2.1850MARKER.
d) Applied X:to proposal: **6.2.2295 TO** and **6.2.2405 VALUE**.

10 Added X:ekeys proposal: **10.6.2.0 EKEY>FKEY**, **10.6.2.0 K-DELETE**, **10.6.2.0 K-DOWN**, **10.6.2.0 K-END**, **10.6.2.0 K-HOME**, **10.6.2.0 K-INSERT**, **10.6.2.0 K-LEFT**, **10.6.2.0 K-NEXT**, **10.6.2.0 K-PRIOR**, **10.6.2.0 K-RIGHT**, **10.6.2.0 K-UP**, **10.6.2.0 K-ALT-MASK**, **10.6.2.0 K-CTRL-MASK**, **10.6.2.0 K-SHIFT-MASK**, **10.6.2.0 K-F1**, **10.6.2.0 K-F2**, **10.6.2.0 K-F3**, **10.6.2.0 K-F4**, **10.6.2.0 K-F5**, **10.6.2.0 K-F6**, **10.6.2.0 K-F7**, **10.6.2.0 K-F8**, **10.6.2.0 K-F9**, **10.6.2.0 K-F10**, **10.6.2.0 K-F11** and **10.6.2.0 K-F12**.

11 Added X:required proposal: **11.6.1.1718 INCLUDED**, **11.6.2.0 REQUIRE**, **11.6.2.0 REQUIRED** and **11.6.2.0 INCLUDE**.

13 Applied X:to proposal: Abmiguous condition, **13.6.1.0086 (LOCAL)** **13.6.1.2295 TO** and **13.6.2.1795 LOCALS|**

B Added ANSI X3.215-1994 *ANS Forth* and ISO/IEC 15145:1997 *ISO Forth* to “Industry standards” (X:forward).

F Added John Hayes’ core test suite, changing the introduction in the process.

G Added this Change Log.

H Original annex F (alphabetic list of words) is now annex H.

Still to do:

– Separate FP stack

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.

X:ekeys
X:ekeys
X:ekeys
X:ekeys
X:ekeys
X:ekeys
X:ekeys
X:ekeys
X:ekeys
X:ekeys
X:ekeys
X:ekeys
X:ekeys
X:ekeys
X:ekeys
X:ekeys

Forth 200*x* Standards Committee

http://www.forth200x.org/