# B. Bibliography (informative annex)

## 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.

## Books

Brodie, L. *Starting FORTH*. Englewood Cliffs, NJ: Prentice Hall, 1981.

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.* New York: 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* (2nd 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

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.

# C. Perspective (informative annex)

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 investigate 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], 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

[1] Other common implementation techniques include direct translation to code and other types of tokens.

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, 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.

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.

```
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.**

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 reset.

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.

# D. Compatibility analysis of ANS Forth (informative annex)

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 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:

1) 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.
2) 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.

# E. ANS Forth portability guide (informative annex)

## 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 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

---

[1] These terms are not defined in the standard. They are mentioned here for historical continuity.

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** 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.

# F. Alphabetic list of words (informative annex)

In the following list, the last, four-digit, part of the reference number establishes a sequence corresponding to the alphabetic ordering of all standard words. The first two or three parts indicate the word set and glossary section in which the word is defined.


**6.1.0010** **!** ..... "store" ..... CORE ..... **25**
**6.1.0030** **#** ..... "number-sign" ..... CORE ..... **25**
**6.1.0040** **#>** ..... "number-sign-greater" ..... CORE ..... **25**
**6.1.0050** **#S** ..... "number-sign-s" ..... CORE ..... **25**
**6.2.0060** **#TIB** ..... "number-t-i-b" ..... CORE EXT ..... **51**
**6.1.0070** **'** ..... "tick" ..... CORE ..... **25**
**6.1.0080** **(** ..... "paren" ..... CORE ..... **26**
**11.6.1.0080** **(** ..... "paren" ..... FILE ..... **82**
**13.6.1.0086** **(LOCAL)** ..... "Paren-local-paren" ..... LOCAL ..... **108**
**6.1.0090** ***** ..... "star" ..... CORE ..... **26**
**6.1.0100** ***/** ..... "star-slash" ..... CORE ..... **26**
**6.1.0110** ***/MOD** ..... "star-slash-mod" ..... CORE ..... **26**
**6.1.0120** **+** ..... "plus" ..... CORE ..... **26**
**6.1.0130** **+!** ..... "plus-store" ..... CORE ..... **27**
**6.1.0140** **+LOOP** ..... "plus-loop" ..... CORE ..... **27**
**6.1.0150** **,** ..... "comma" ..... CORE ..... **27**
**6.1.0160** **-** ..... "minus" ..... CORE ..... **27**
**17.6.1.0170** **-TRAILING** ..... "dash-trailing" ..... STRING ..... **126**
**6.1.0180** **.** ..... "dot" ..... CORE ..... **27**
**6.1.0190** **."** ..... "dot-quote" ..... CORE ..... **28**
**6.2.0200** **.(** ..... "dot-paren" ..... CORE EXT ..... **51**
**6.2.0210** **.R** ..... "dot-r" ..... CORE EXT ..... **51**
**15.6.1.0220** **.S** ..... "dot-s" ..... TOOLS ..... **115**
**6.1.0230** **/** ..... "slash" ..... CORE ..... **28**
**6.1.0240** **/MOD** ..... "slash-mod" ..... CORE ..... **28**
**17.6.1.0245** **/STRING** ..... "slash-string" ..... STRING ..... **126**
**6.1.0250** **0<** ..... "zero-less" ..... CORE ..... **28**
**6.2.0260** **0<>** ..... "zero-not-equals" ..... CORE EXT ..... **51**
**6.1.0270** **0=** ..... "zero-equals" ..... CORE ..... **28**
**6.2.0280** **0>** ..... "zero-greater" ..... CORE EXT ..... **51**
**6.1.0290** **1+** ..... "one-plus" ..... CORE ..... **28**
**6.1.0300** **1-** ..... "one-minus" ..... CORE ..... **29**
**6.1.0310** **2!** ..... "two-store" ..... CORE ..... **29**
**6.1.0320** **2*** ..... "two-star" ..... CORE ..... **29**
**6.1.0330** **2/** ..... "two-slash" ..... CORE ..... **29**
**6.2.0340** **2>R** ..... "two-to-r" ..... CORE EXT ..... **51**
**6.1.0350** **2@** ..... "two-fetch" ..... CORE ..... **29**
**8.6.1.0360** **2CONSTANT** ..... "two-constant" ..... DOUBLE ..... **68**
**6.1.0370** **2DROP** ..... "two-drop" ..... CORE ..... **29**
**6.1.0380** **2DUP** ..... "two-dupe" ..... CORE ..... **29**
**8.6.1.0390** **2LITERAL** ..... "two-literal" ..... DOUBLE ..... **68**
**6.1.0400** **2OVER** ..... "two-over" ..... CORE ..... **29**
**6.2.0410** **2R>** ..... "two-r-from" ..... CORE EXT ..... **52**
**6.2.0415** **2R@** ..... "two-r-fetch" ..... CORE EXT ..... **52**
**8.6.2.0420** **2ROT** ..... "two-rote" ..... DOUBLE EXT ..... **71**
**6.1.0430** **2SWAP** ..... "two-swap" ..... CORE ..... **30**
**8.6.1.0440** **2VARIABLE** ..... "two-variable" ..... DOUBLE ..... **69**

6.1.0450 : ..... "colon" ..... CORE ..... 30
6.2.0455 :NONAME ..... "colon-no-name" ..... CORE EXT ..... 52
6.1.0460 ; ..... "semicolon" ..... CORE ..... 30
15.6.2.0470 ;CODE ..... "semicolon-code" ..... TOOLS EXT ..... 116
6.1.0480 < ..... "less-than" ..... CORE ..... 30
6.1.0490 <# ..... "less-number-sign" ..... CORE ..... 31
6.2.0500 <> ..... "not-equals" ..... CORE EXT ..... 52
6.1.0530 = ..... "equals" ..... CORE ..... 31
6.1.0540 > ..... "greater-than" ..... CORE ..... 31
6.1.0550 >BODY ..... "to-body" ..... CORE ..... 31
12.6.1.0558 >FLOAT ..... "to-float" ..... FLOATING ..... 93
6.1.0560 >IN ..... "to-in" ..... CORE ..... 31
6.1.0570 >NUMBER ..... "to-number" ..... CORE ..... 31
6.1.0580 >R ..... "to-r" ..... CORE ..... 32
15.6.1.0600 ? ..... "question" ..... TOOLS ..... 115
6.2.0620 ?DO ..... "question-do" ..... CORE EXT ..... 53
6.1.0630 ?DUP ..... "question-dupe" ..... CORE ..... 32
6.1.0650 @ ..... "fetch" ..... CORE ..... 32
6.1.0670 ABORT ..... CORE ..... 32
9.6.2.0670 ABORT ..... EXCEPTION EXT ..... 75
6.1.0680 ABORT" ..... "abort-quote" ..... CORE ..... 32
9.6.2.0680 ABORT" ..... "abort-quote" ..... EXCEPTION EXT ..... 75
6.1.0690 ABS ..... "abs" ..... CORE ..... 32
6.1.0695 ACCEPT ..... CORE ..... 33
6.2.0700 AGAIN ..... CORE EXT ..... 53
15.6.2.0702 AHEAD ..... TOOLS EXT ..... 116
6.1.0705 ALIGN ..... CORE ..... 33
6.1.0706 ALIGNED ..... CORE ..... 33
14.6.1.0707 ALLOCATE ..... MEMORY ..... 111
6.1.0710 ALLOT ..... CORE ..... 33
16.6.2.0715 ALSO ..... SEARCH EXT ..... 123
6.1.0720 AND ..... CORE ..... 33
15.6.2.0740 ASSEMBLER ..... TOOLS EXT ..... 117
10.6.1.0742 AT-XY ..... "at-x-y" ..... FACILITY ..... 77
6.1.0750 BASE ..... CORE ..... 34
6.1.0760 BEGIN ..... CORE ..... 34
11.6.1.0765 BIN ..... FILE ..... 82
6.1.0770 BL ..... "b-l" ..... CORE ..... 34
17.6.1.0780 BLANK ..... STRING ..... 126
7.6.1.0790 BLK ..... "b-l-k" ..... BLOCK ..... 63
7.6.1.0800 BLOCK ..... BLOCK ..... 63
7.6.1.0820 BUFFER ..... BLOCK ..... 64
15.6.2.0830 BYE ..... TOOLS EXT ..... 117
6.1.0850 C! ..... "c-store" ..... CORE ..... 34
6.2.0855 C" ..... "c-quote" ..... CORE EXT ..... 53
6.1.0860 C, ..... "c-comma" ..... CORE ..... 34
6.1.0870 C@ ..... "c-fetch" ..... CORE ..... 34
6.2.0873 CASE ..... CORE EXT ..... 54
9.6.1.0875 CATCH ..... EXCEPTION ..... 74
6.1.0880 CELL+ ..... "cell-plus" ..... CORE ..... 35
6.1.0890 CELLS ..... CORE ..... 35
6.1.0895 CHAR ..... "char" ..... CORE ..... 35
6.1.0897 CHAR+ ..... "char-plus" ..... CORE ..... 35
6.1.0898 CHARS ..... "chars" ..... CORE ..... 35

11.6.1.0900 **CLOSE-FILE** ..... FILE ..... 82
17.6.1.0910 **CMOVE** ..... "c-move" ..... STRING ..... 126
17.6.1.0920 **CMOVE>** ..... "c-move-up" ..... STRING ..... 126
15.6.2.0930 **CODE** ..... TOOLS EXT ..... 117
17.6.1.0935 **COMPARE** ..... STRING ..... 127
6.2.0945 **COMPILE,** ..... "compile-comma" ..... CORE EXT ..... 54
6.1.0950 **CONSTANT** ..... CORE ..... 35
6.2.0970 **CONVERT** ..... CORE EXT ..... 54
6.1.0980 **COUNT** ..... CORE ..... 36
6.1.0990 **CR** ..... "c-r" ..... CORE ..... 36
6.1.1000 **CREATE** ..... CORE ..... 36
11.6.1.1010 **CREATE-FILE** ..... FILE ..... 83
15.6.2.1015 **CS-PICK** ..... "c-s-pick" ..... TOOLS EXT ..... 117
15.6.2.1020 **CS-ROLL** ..... "c-s-roll" ..... TOOLS EXT ..... 118
8.6.1.1040 **D+** ..... "d-plus" ..... DOUBLE ..... 69
8.6.1.1050 **D-** ..... "d-minus" ..... DOUBLE ..... 69
8.6.1.1060 **D.** ..... "d-dot" ..... DOUBLE ..... 69
8.6.1.1070 **D.R** ..... "d-dot-r" ..... DOUBLE ..... 69
8.6.1.1075 **D0<** ..... "d-zero-less" ..... DOUBLE ..... 69
8.6.1.1080 **D0=** ..... "d-zero-equals" ..... DOUBLE ..... 69
8.6.1.1090 **D2*** ..... "d-two-star" ..... DOUBLE ..... 70
8.6.1.1100 **D2/** ..... "d-two-slash" ..... DOUBLE ..... 70
8.6.1.1110 **D<** ..... "d-less-than" ..... DOUBLE ..... 70
8.6.1.1120 **D=** ..... "d-equals" ..... DOUBLE ..... 70
12.6.1.1130 **D>F** ..... "d-to-f" ..... FLOATING ..... 93
8.6.1.1140 **D>S** ..... "d-to-s" ..... DOUBLE ..... 70
8.6.1.1160 **DABS** ..... "d-abs" ..... DOUBLE ..... 70
6.1.1170 **DECIMAL** ..... CORE ..... 36
16.6.1.1180 **DEFINITIONS** ..... SEARCH ..... 122
11.6.1.1190 **DELETE-FILE** ..... FILE ..... 83
6.1.1200 **DEPTH** ..... CORE ..... 36
12.6.2.1203 **DF!** ..... "d-f-store" ..... FLOATING EXT ..... 97
12.6.2.1204 **DF@** ..... "d-f-fetch" ..... FLOATING EXT ..... 98
12.6.2.1205 **DFALIGN** ..... "d-f-align" ..... FLOATING EXT ..... 98
12.6.2.1207 **DFALIGNED** ..... "d-f-aligned" ..... FLOATING EXT ..... 98
12.6.2.1208 **DFLOAT+** ..... "d-float-plus" ..... FLOATING EXT ..... 98
12.6.2.1209 **DFLOATS** ..... "d-floats" ..... FLOATING EXT ..... 98
8.6.1.1210 **DMAX** ..... "d-max" ..... DOUBLE ..... 70
8.6.1.1220 **DMIN** ..... "d-min" ..... DOUBLE ..... 70
8.6.1.1230 **DNEGATE** ..... "d-negate" ..... DOUBLE ..... 71
6.1.1240 **DO** ..... CORE ..... 37
6.1.1250 **DOES>** ..... "does" ..... CORE ..... 37
6.1.1260 **DROP** ..... CORE ..... 37
8.6.2.1270 **DU<** ..... "d-u-less" ..... DOUBLE EXT ..... 71
15.6.1.1280 **DUMP** ..... TOOLS ..... 115
6.1.1290 **DUP** ..... "dupe" ..... CORE ..... 38
15.6.2.1300 **EDITOR** ..... TOOLS EXT ..... 118
10.6.2.1305 **EKEY** ..... "e-key" ..... FACILITY EXT ..... 78
10.6.2.1306 **EKEY>CHAR** ..... "e-key-to-char" ..... FACILITY EXT ..... 78
10.6.2.1307 **EKEY?** ..... "e-key-question" ..... FACILITY EXT ..... 78
6.1.1310 **ELSE** ..... CORE ..... 38
6.1.1320 **EMIT** ..... CORE ..... 38
10.6.2.1325 **EMIT?** ..... "emit-question" ..... FACILITY EXT ..... 78
7.6.2.1330 **EMPTY-BUFFERS** ..... BLOCK EXT ..... 65

6.2.1342 **ENDCASE** ..... "end-case" ..... CORE EXT ..... **54**
6.2.1343 **ENDOF** ..... "end-of" ..... CORE EXT ..... **55**
6.1.1345 **ENVIRONMENT?** ..... "environment-query" ..... CORE ..... **38**
6.2.1350 **ERASE** ..... CORE EXT ..... **55**
6.1.1360 **EVALUATE** ..... CORE ..... **39**
7.6.1.1360 **EVALUATE** ..... BLOCK ..... **64**
6.1.1370 **EXECUTE** ..... CORE ..... **39**
6.1.1380 **EXIT** ..... CORE ..... **39**
6.2.1390 **EXPECT** ..... CORE EXT ..... **55**
12.6.1.1400 **F!** ..... "f-store" ..... FLOATING ..... **93**
12.6.1.1410 **F*** ..... "f-star" ..... FLOATING ..... **93**
12.6.2.1415 **F**** ..... "f-star-star" ..... FLOATING EXT ..... **98**
12.6.1.1420 **F+** ..... "f-plus" ..... FLOATING ..... **93**
12.6.1.1425 **F-** ..... "f-minus" ..... FLOATING ..... **93**
12.6.2.1427 **F.** ..... "f-dot" ..... FLOATING EXT ..... **99**
12.6.1.1430 **F/** ..... "f-slash" ..... FLOATING ..... **94**
12.6.1.1440 **F0<** ..... "f-zero-less-than" ..... FLOATING ..... **94**
12.6.1.1450 **F0=** ..... "f-zero-equals" ..... FLOATING ..... **94**
12.6.1.1460 **F<** ..... "f-less-than" ..... FLOATING ..... **94**
12.6.1.1470 **F>D** ..... "f-to-d" ..... FLOATING ..... **94**
12.6.1.1472 **F@** ..... "f-fetch" ..... FLOATING ..... **94**
12.6.2.1474 **FABS** ..... "f-abs" ..... FLOATING EXT ..... **99**
12.6.2.1476 **FACOS** ..... "f-a-cos" ..... FLOATING EXT ..... **99**
12.6.2.1477 **FACOSH** ..... "f-a-cosh" ..... FLOATING EXT ..... **99**
12.6.1.1479 **FALIGN** ..... "f-align" ..... FLOATING ..... **94**
12.6.1.1483 **FALIGNED** ..... "f-aligned" ..... FLOATING ..... **94**
12.6.2.1484 **FALOG** ..... "f-a-log" ..... FLOATING EXT ..... **99**
6.2.1485 **FALSE** ..... CORE EXT ..... **55**
12.6.2.1486 **FASIN** ..... "f-a-sine" ..... FLOATING EXT ..... **99**
12.6.2.1487 **FASINH** ..... "f-a-cinch" ..... FLOATING EXT ..... **99**
12.6.2.1488 **FATAN** ..... "f-a-tan" ..... FLOATING EXT ..... **100**
12.6.2.1489 **FATAN2** ..... "f-a-tan-two" ..... FLOATING EXT ..... **100**
12.6.2.1491 **FATANH** ..... "f-a-tan-h" ..... FLOATING EXT ..... **100**
12.6.1.1492 **FCONSTANT** ..... "f-constant" ..... FLOATING ..... **95**
12.6.2.1493 **FCOS** ..... "f-cos" ..... FLOATING EXT ..... **100**
12.6.2.1494 **FCOSH** ..... "f-cosh" ..... FLOATING EXT ..... **100**
12.6.1.1497 **FDEPTH** ..... "f-depth" ..... FLOATING ..... **95**
12.6.1.1500 **FDROP** ..... "f-drop" ..... FLOATING ..... **95**
12.6.1.1510 **FDUP** ..... "f-dupe" ..... FLOATING ..... **95**
12.6.2.1513 **FE.** ..... "f-e-dot" ..... FLOATING EXT ..... **100**
12.6.2.1515 **FEXP** ..... "f-e-x-p" ..... FLOATING EXT ..... **100**
12.6.2.1516 **FEXPM1** ..... "f-e-x-p-m-one" ..... FLOATING EXT ..... **101**
11.6.1.1520 **FILE-POSITION** ..... FILE ..... **83**
11.6.1.1522 **FILE-SIZE** ..... FILE ..... **83**
11.6.2.1524 **FILE-STATUS** ..... FILE EXT ..... **88**
6.1.1540 **FILL** ..... CORE ..... **39**
6.1.1550 **FIND** ..... CORE ..... **39**
16.6.1.1550 **FIND** ..... SEARCH ..... **122**
12.6.1.1552 **FLITERAL** ..... "f-literal" ..... FLOATING ..... **95**
12.6.2.1553 **FLN** ..... "f-l-n" ..... FLOATING EXT ..... **101**
12.6.2.1554 **FLNP1** ..... "f-l-n-p-one" ..... FLOATING EXT ..... **101**
12.6.1.1555 **FLOAT+** ..... "float-plus" ..... FLOATING ..... **95**
12.6.1.1556 **FLOATS** ..... FLOATING ..... **96**
12.6.2.1557 **FLOG** ..... "f-log" ..... FLOATING EXT ..... **101**

12.6.1.1558 **FLOOR** ..... FLOATING **.....96**
7.6.1.1559 **FLUSH** ..... BLOCK **.....64**
11.6.2.1560 **FLUSH-FILE** ..... FILE EXT **.....88**
6.1.1561 **FM/MOD** ..... "f-m-slash-mod" ..... CORE **.....40**
12.6.1.1562 **FMAX** ..... "f-max" ..... FLOATING **.....96**
12.6.1.1565 **FMIN** ..... "f-min" ..... FLOATING **.....96**
12.6.1.1567 **FNEGATE** ..... "f-negate" ..... FLOATING **.....96**
15.6.2.1580 **FORGET** ..... TOOLS EXT **.....118**
16.6.2.1590 **FORTH** ..... SEARCH EXT **.....123**
16.6.1.1595 **FORTH-WORDLIST** ..... SEARCH **.....122**
12.6.1.1600 **FOVER** ..... "f-over" ..... FLOATING **.....96**
14.6.1.1605 **FREE** ..... MEMORY **.....112**
12.6.1.1610 **FROT** ..... "f-rote" ..... FLOATING **.....96**
12.6.1.1612 **FROUND** ..... "f-round" ..... FLOATING **.....96**
12.6.2.1613 **FS.** ..... "f-s-dot" ..... FLOATING EXT **.....101**
12.6.2.1614 **FSIN** ..... "f-sine" ..... FLOATING EXT **.....101**
12.6.2.1616 **FSINCOS** ..... "f-sine-cos" ..... FLOATING EXT **.....102**
12.6.2.1617 **FSINH** ..... "f-cinch" ..... FLOATING EXT **.....102**
12.6.2.1618 **FSQRT** ..... "f-square-root" ..... FLOATING EXT **.....102**
12.6.1.1620 **FSWAP** ..... "f-swap" ..... FLOATING **.....96**
12.6.2.1625 **FTAN** ..... "f-tan" ..... FLOATING EXT **.....102**
12.6.2.1626 **FTANH** ..... "f-tan-h" ..... FLOATING EXT **.....102**
12.6.1.1630 **FVARIABLE** ..... "f-variable" ..... FLOATING **.....97**
12.6.2.1640 **F~** ..... "f-proximate" ..... FLOATING EXT **.....102**
16.6.1.1643 **GET-CURRENT** ..... SEARCH **.....122**
16.6.1.1647 **GET-ORDER** ..... SEARCH **.....122**
6.1.1650 **HERE** ..... CORE **.....40**
6.2.1660 **HEX** ..... CORE EXT **.....55**
6.1.1670 **HOLD** ..... CORE **.....40**
6.1.1680 **I** ..... CORE **.....40**
6.1.1700 **IF** ..... CORE **.....40**
6.1.1710 **IMMEDIATE** ..... CORE **.....41**
11.6.1.1717 **INCLUDE-FILE** ..... FILE **.....83**
11.6.1.1718 **INCLUDED** ..... FILE **.....84**
6.1.1720 **INVERT** ..... CORE **.....41**
6.1.1730 **J** ..... CORE **.....41**
6.1.1750 **KEY** ..... CORE **.....41**
10.6.1.1755 **KEY?** ..... "key-question" ..... FACILITY **.....77**
6.1.1760 **LEAVE** ..... CORE **.....41**
7.6.2.1770 **LIST** ..... BLOCK EXT **.....65**
6.1.1780 **LITERAL** ..... CORE **.....42**
7.6.1.1790 **LOAD** ..... BLOCK **.....64**
13.6.2.1795 **LOCALS|** ..... "locals-bar" ..... LOCAL EXT **.....109**
6.1.1800 **LOOP** ..... CORE **.....42**
6.1.1805 **LSHIFT** ..... "l-shift" ..... CORE **.....42**
6.1.1810 **M*** ..... "m-star" ..... CORE **.....42**
8.6.1.1820 **M*/** ..... "m-star-slash" ..... DOUBLE **.....71**
8.6.1.1830 **M+** ..... "m-plus" ..... DOUBLE **.....71**
6.2.1850 **MARKER** ..... CORE EXT **.....56**
6.1.1870 **MAX** ..... CORE **.....42**
6.1.1880 **MIN** ..... CORE **.....42**
6.1.1890 **MOD** ..... CORE **.....43**
6.1.1900 **MOVE** ..... CORE **.....43**
10.6.2.1905 **MS** ..... FACILITY EXT **.....78**

6.1.1910 **NEGATE** ..... CORE ..... 43
6.2.1930 **NIP** ..... CORE EXT ..... 56
6.2.1950 **OF** ..... CORE EXT ..... 56
16.6.2.1965 **ONLY** ..... SEARCH EXT ..... 124
11.6.1.1970 **OPEN-FILE** ..... FILE ..... 84
6.1.1980 **OR** ..... CORE ..... 43
16.6.2.1985 **ORDER** ..... SEARCH EXT ..... 124
6.1.1990 **OVER** ..... CORE ..... 43
6.2.2000 **PAD** ..... CORE EXT ..... 56
10.6.1.2005 **PAGE** ..... FACILITY ..... 77
6.2.2008 **PARSE** ..... CORE EXT ..... 57
6.2.2030 **PICK** ..... CORE EXT ..... 57
6.1.2033 **POSTPONE** ..... CORE ..... 43
12.6.2.2035 **PRECISION** ..... FLOATING EXT ..... 102
16.6.2.2037 **PREVIOUS** ..... SEARCH EXT ..... 124
6.2.2040 **QUERY** ..... CORE EXT ..... 57
6.1.2050 **QUIT** ..... CORE ..... 44
11.6.1.2054 **R/O** ..... "r-o" ..... FILE ..... 84
11.6.1.2056 **R/W** ..... "r-w" ..... FILE ..... 85
6.1.2060 **R>** ..... "r-from" ..... CORE ..... 44
6.1.2070 **R@** ..... "r-fetch" ..... CORE ..... 44
11.6.1.2080 **READ-FILE** ..... FILE ..... 85
11.6.1.2090 **READ-LINE** ..... FILE ..... 86
6.1.2120 **RECURSE** ..... CORE ..... 44
6.2.2125 **REFILL** ..... CORE EXT ..... 57
7.6.2.2125 **REFILL** ..... BLOCK EXT ..... 65
11.6.2.2125 **REFILL** ..... FILE EXT ..... 88
11.6.2.2130 **RENAME-FILE** ..... FILE EXT ..... 88
6.1.2140 **REPEAT** ..... CORE ..... 45
11.6.1.2142 **REPOSITION-FILE** ..... FILE ..... 86
12.6.1.2143 **REPRESENT** ..... FLOATING ..... 97
14.6.1.2145 **RESIZE** ..... MEMORY ..... 112
11.6.1.2147 **RESIZE-FILE** ..... FILE ..... 86
6.2.2148 **RESTORE-INPUT** ..... CORE EXT ..... 57
6.2.2150 **ROLL** ..... CORE EXT ..... 58
6.1.2160 **ROT** ..... "rote" ..... CORE ..... 45
6.1.2162 **RSHIFT** ..... "r-shift" ..... CORE ..... 45
6.1.2165 **S"** ..... "s-quote" ..... CORE ..... 45
11.6.1.2165 **S"** ..... "s-quote" ..... FILE ..... 87
6.1.2170 **S>D** ..... "s-to-d" ..... CORE ..... 45
7.6.1.2180 **SAVE-BUFFERS** ..... BLOCK ..... 64
6.2.2182 **SAVE-INPUT** ..... CORE EXT ..... 58
7.6.2.2190 **SCR** ..... "s-c-r" ..... BLOCK EXT ..... 65
17.6.1.2191 **SEARCH** ..... STRING ..... 127
16.6.1.2192 **SEARCH-WORDLIST** ..... SEARCH ..... 123
15.6.1.2194 **SEE** ..... TOOLS ..... 115
16.6.1.2195 **SET-CURRENT** ..... SEARCH ..... 123
16.6.1.2197 **SET-ORDER** ..... SEARCH ..... 123
12.6.2.2200 **SET-PRECISION** ..... FLOATING EXT ..... 102
12.6.2.2202 **SF!** ..... "s-f-store" ..... FLOATING EXT ..... 103
12.6.2.2203 **SF@** ..... "s-f-fetch" ..... FLOATING EXT ..... 103
12.6.2.2204 **SFALIGN** ..... "s-f-align" ..... FLOATING EXT ..... 103
12.6.2.2206 **SFALIGNED** ..... "s-f-aligned" ..... FLOATING EXT ..... 103
12.6.2.2207 **SFLOAT+** ..... "s-float-plus" ..... FLOATING EXT ..... 103

12.6.2.2208 **SFLOATS** ..... "s-floats" ..... ..... FLOATING EXT ..... **104**
6.1.2210 **SIGN** ..... ..... ..... CORE ..... **46**
17.6.1.2212 **SLITERAL** ..... ..... ..... STRING ..... **127**
6.1.2214 **SM/REM** ..... "s-m-slash-rem" ..... ..... CORE ..... **46**
6.1.2216 **SOURCE** ..... ..... ..... CORE ..... **46**
6.2.2218 **SOURCE-ID** ..... "source-i-d" ..... ..... CORE EXT ..... **58**
11.6.1.2218 **SOURCE-ID** ..... "source-i-d" ..... ..... FILE ..... **87**
6.1.2220 **SPACE** ..... ..... ..... CORE ..... **46**
6.1.2230 **SPACES** ..... ..... ..... CORE ..... **46**
6.2.2240 **SPAN** ..... ..... ..... CORE EXT ..... **58**
6.1.2250 **STATE** ..... ..... ..... CORE ..... **46**
15.6.2.2250 **STATE** ..... ..... ..... TOOLS EXT ..... **118**
6.1.2260 **SWAP** ..... ..... ..... CORE ..... **46**
6.1.2270 **THEN** ..... ..... ..... CORE ..... **47**
9.6.1.2275 **THROW** ..... ..... ..... EXCEPTION ..... **75**
7.6.2.2280 **THRU** ..... ..... ..... BLOCK EXT ..... **65**
6.2.2290 **TIB** ..... "t-i-b" ..... ..... CORE EXT ..... **58**
10.6.2.2292 **TIME&DATE** ..... "time-and-date" ..... ..... FACILITY EXT ..... **78**
6.2.2295 **TO** ..... ..... ..... CORE EXT ..... **59**
13.6.1.2295 **TO** ..... ..... ..... LOCAL ..... **109**
6.2.2298 **TRUE** ..... ..... ..... CORE EXT ..... **59**
6.2.2300 **TUCK** ..... ..... ..... CORE EXT ..... **59**
6.1.2310 **TYPE** ..... ..... ..... CORE ..... **47**
6.1.2320 **U.** ..... "u-dot" ..... ..... CORE ..... **47**
6.2.2330 **U.R** ..... "u-dot-r" ..... ..... CORE EXT ..... **59**
6.1.2340 **U<** ..... "u-less-than" ..... ..... CORE ..... **47**
6.2.2350 **U>** ..... "u-greater-than" ..... ..... CORE EXT ..... **59**
6.1.2360 **UM*** ..... "u-m-star" ..... ..... CORE ..... **47**
6.1.2370 **UM/MOD** ..... "u-m-slash-mod" ..... ..... CORE ..... **48**
6.1.2380 **UNLOOP** ..... ..... ..... CORE ..... **48**
6.1.2390 **UNTIL** ..... ..... ..... CORE ..... **48**
6.2.2395 **UNUSED** ..... ..... ..... CORE EXT ..... **59**
7.6.1.2400 **UPDATE** ..... ..... ..... BLOCK ..... **65**
6.2.2405 **VALUE** ..... ..... ..... CORE EXT ..... **60**
6.1.2410 **VARIABLE** ..... ..... ..... CORE ..... **48**
11.6.1.2425 **W/O** ..... "w-o" ..... ..... FILE ..... **87**
6.1.2430 **WHILE** ..... ..... ..... CORE ..... **49**
6.2.2440 **WITHIN** ..... ..... ..... CORE EXT ..... **60**
6.1.2450 **WORD** ..... ..... ..... CORE ..... **49**
16.6.1.2460 **WORDLIST** ..... ..... ..... SEARCH ..... **123**
15.6.1.2465 **WORDS** ..... ..... ..... TOOLS ..... **116**
11.6.1.2480 **WRITE-FILE** ..... ..... ..... FILE ..... **87**
11.6.1.2485 **WRITE-LINE** ..... ..... ..... FILE ..... **88**
6.1.2490 **XOR** ..... "x-or" ..... ..... CORE ..... **49**
6.1.2500 **[** ..... "left-bracket" ..... ..... CORE ..... **49**
6.1.2510 **[']** ..... "bracket-tick" ..... ..... CORE ..... **50**
6.1.2520 **[CHAR]** ..... "bracket-char" ..... ..... CORE ..... **50**
6.2.2530 **[COMPILE]** ..... "bracket-compile" ..... ..... CORE EXT ..... **60**
15.6.2.2531 **[ELSE]** ..... "bracket-else" ..... ..... TOOLS EXT ..... **119**
15.6.2.2532 **[IF]** ..... "bracket-if" ..... ..... TOOLS EXT ..... **119**
15.6.2.2533 **[THEN]** ..... "bracket-then" ..... ..... TOOLS EXT ..... **119**
6.2.2535 **** ..... "backslash" ..... ..... CORE EXT ..... **60**
7.6.2.2535 **** ..... "backslash" ..... ..... BLOCK EXT ..... **66**
6.1.2540 **]** ..... "right-bracket" ..... ..... CORE ..... **50**