 Table of Contents

# 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, the content of this definition is a string of addresses of previously defined words which will be executed in turn whenever the word being defined is invoked. The definition is terminated by ; (semicolon). For example, here is a definition:

```
: RECEIVE  ( -- addr n )  PAD DUP 32 ACCEPT ;
```

The name of the new word is RECEIVE. The comment (in parentheses) indicates that it requires no parameters and will return an address and count on the data stack. When RECEIVE is executed, it will perform the words in the remainder of the definition in sequence. The word PAD places on the stack the address of a scratch pad used to handle strings. DUP duplicates the top stack item, so we now have two copies of the address. The number 32 is also placed on the stack. The word ACCEPT takes an address (provided by PAD) and length (32) on the stack, accepts from the keyboard a string of up to 32 characters which will be placed at the specified address, and returns the number of characters received. The copy of the scratch-pad address remains on the stack below the count so that the routine that called RECEIVE can use it to pick up the received string.

---

## C.5.2 Push-down stacks

The example above illustrates the use of push-down stacks for passing parameters between Forth words. Forth maintains two push-down stacks, or LIFO lists. These provide communication between Forth words plus an efficient mechanism for controlling logical flow. A stack contains 16-bit items on 8-bit and 16-bit computers, and 32-bit items on 32-bit processors. Double-cell numbers occupy two stack positions, with the most-significant part on top. Items on either stack may be addresses or data items of various kinds. Stacks are of indefinite size, and usually grow towards low memory.

Although the structure of both stacks is the same, they have very different uses. The user interacts most directly with the Data Stack, which contains arguments passed between words. This function replaces the calling sequences used by conventional languages. It is efficient internally, and makes routines intrinsically re-entrant. The second stack is called the Return Stack, as its main function is to hold return addresses for nested definitions, although other kinds of data are sometimes kept there temporarily.

The use of the Data Stack (often called just **the stack**) leads to a notation in which operands precede operators. The word ACCEPT in the example above took an address and count from the stack and left another address there. Similarly, a word called BLANK expects an address and count, and will place the specified number of space characters (20H) in the region starting at that address. Thus,

```
PAD 25 BLANK
```

will fill the scratch region whose address is pushed on the stack by PAD with 25 spaces. Application words are usually defined to work similarly. For example,

```
100 SAMPLES
```

might be defined to record 100 measurements in a data array.

Arithmetic operators also expect values and leave results on the stack. For example, + adds the top two numbers on the stack, replacing them both by their sum. Since results of operations are left on the stack, operations may be strung together without a need to define variables to use for temporary storage.

---

### C.5.3 Interpreters

Forth is traditionally an interpretive system, in that program execution is controlled by data items rather than machine code. Interpreters can be slow, but Forth maintains the high speed required of real-time applications by having two levels of interpretation.

The first is the text interpreter, which parses strings from the terminal or mass storage and looks each word up in the dictionary. When a word is found it is executed by invoking the second level, the address interpreter.

The second is an **address interpreter**. Although not all Forth systems are implemented in this way, it was the first and is still the primary implementation technology. For a small cost in performance, an address interpreter can yield a very compact object program, which has been a major factor in Forth's wide acceptance in embedded systems and other applications where small object size is desirable.

The address interpreter processes strings of addresses or tokens compiled in definitions created by : (colon), by executing the definition pointed to by each. The content of most definitions is a sequence of addresses of previously defined words, which will be executed by the address interpreter in turn. Thus, when the word RECEIVE (defined above) is executed, the word PAD, the word DUP, the literal 32, and the word ACCEPT will be executed in sequence. The process is terminated by the semicolon. This execution requires no dictionary searches, parsing, or other logic, because when RECEIVE was compiled the dictionary was searched for each word, and its address (or other token) was placed in the next successive cell of the entry. The text was not stored in memory, not even in condensed form.

The address interpreter has two important properties. First, it is fast. Although the actual speed depends upon the specific implementation, professional implementations are highly optimized, often requiring only one or two machine instructions per address. On most benchmarks, a good Forth implementation substantially out-performs interpretive languages such as BASIC or LISP, and will compare favorably with other compiled high-level languages.

Second, the address interpreter makes Forth definitions extremely compact, as each reference requires only one cell. In comparison, a subroutine call constructed by most compilers involves instructions for handling the calling sequence (unnecessary in Forth because of the stack) before and after a CALL or JSR instruction and address.

Most of the words in a Forth dictionary will be defined by : (colon) and interpreted by the address interpreter. Most of Forth itself is defined this way.

---

### C.5.4 Assembler

Most implementations of Forth include a macro assembler for the CPU on which they run. By using the defining word CODE the programmer can create a definition whose behavior will consist of executing actual machine instructions. CODE definitions may be used to do I/O, implement arithmetic primitives, and do other machine-dependent or time-critical processing. When using CODE the programmer has full control over the CPU, as with any other assembler, and CODE definitions run at full machine speed.

This is an important feature of Forth. It permits explicit computer-dependent code in manageable pieces with specific interfacing conventions that are machine-independent. To move an application to a different processor requires re-coding only the CODE words, which will interact with other Forth words in exactly the same manner.

Forth assemblers are so compact (typically a few Kbytes) that they can be resident in the system (as are the compiler, editor, and other programming tools). This means that the programmer can type in short CODE definitions and execute them immediately. This capability is especially valuable in testing custom hardware.

---

### C.5.5 Virtual memory

The final unique element of Forth is its way of using disk or other mass storage as a form of **virtual memory** for data and program source. As in the case of the address interpreter, this approach is historically characteristic of Forth, but is by no means universal. Disk is divided into 1024-byte blocks. Two or more buffers are provided in memory, into which blocks are read automatically when referred to. Each block has a fixed block number, which in native systems is a direct function of its physical location. If a block is changed in memory, it will be automatically written out when its buffer must be reused. Explicit reads and writes are not needed; the program will find the data in memory whenever it accesses it.

Block-oriented disk handling is efficient and easy for native Forth systems to implement. As a result, blocks provide a completely transportable mechanism for handling program source and data across both native and co-resident versions of Forth on different host operating systems.

Definitions in program source blocks are compiled into memory by the word LOAD. Most implementations include an editor, which formats a block for display into 16 lines of 64 characters each, and provides commands modifying the source. An example of a Forth source block is given in Fig. C.1 below.

Source blocks have historically been an important element in Forth style. Just as Forth definitions may be considered the linguistic equivalent of sentences in natural languages, a block is analogous to a paragraph. A block normally contains definitions related to a common theme, such as **vector arithmetic**. A comment on the top line of the block identifies this theme. An application may selectively load the blocks it needs.

Blocks are also used to store data. Small records can be combined into a block, or large records spread over several blocks. The programmer may allocate blocks in whatever way suits the application, and on native systems can increase performance by organizing data to minimize disk head motion. Several Forth vendors have developed sophisticated file and data base systems based on Forth blocks.

Versions of Forth that run co-resident with a host OS often implement blocks in host OS files. Others use the host files exclusively. The Standard requires that blocks be available on systems providing any disk access method, as they are the only means of referencing disk that can be transportable across both native and co-resident implementations.

---

### C.5.6 Programming environment

Although this Standard does not require it, most Forth systems include a resident editor. This enables a programmer to edit source and recompile it into executable form without leaving the Forth environment. As it is easy to organize an application into layers, it is often possible to recompile only the topmost layer (which is usually the one currently under development), a process which rarely takes more than a few seconds.

Most Forth systems also provide resident interactive debugging aids, not only including words such as those in 15. The optional Programming-Tools word set, but also having the ability to examine and change the contents of VARIABLEs 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 ALLOTted 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.

*

```
Block 180
  0.      ( LED control )
  1.      HEX 40 CONSTANT LIGHTS DECIMAL
```

```
 2. : LIGHT ( n -- )  LIGHTS OUTPUT ;
 3.
 4. VARIABLE DELAY
 5. : SLOW  500 DELAY ! ;
 6. : FAST  100 DELAY ! ;
 7. : COUNTS 256 0 DO I LIGHT DELAY @ MS  LOOP ;
 8.
 9. : LAMP ( n -  )  CREATE ,  DOES> ( a -- n )   @ ;
10. 1 LAMP POWER       2 LAMP HV     4 LAMP TORCH
11. 8 LAMP SAMPLING   16 LAMP IDLING
12.
13. VARIABLE LAMPS
14. : TOGGLE ( n -- ) LAMPS @ XOR DUP LAMPS ! LIGHT ;
15.
```

Figure C.1 - Forth source block containing words that control a set of LEDs.

Lines 4 - 7 contain a simple diagnostic of the sort one might type in from the terminal to confirm that everything is working. The variable DELAY contains a delay time in milliseconds - execution of the word DELAY returns the address of this variable. Two values of DELAY are set by the definitions SLOW and FAST, using the Forth operator ! (pronounced **store**) which takes a value and an address, and stores the value in the address. The definition COUNTS runs a loop from 0 through 255 (Forth loops of this type are exclusive at the upper end of the range), sending each value to the lights and then waiting for the period specified by DELAY. The word @ (pronounced **fetch**) fetches a value from an address, in this case the address supplied by DELAY. This value is passed to MS, which waits the specified number of milliseconds. The result of executing COUNTS is that the lights will count from 0 to 255 at the desired rate. To run this, one would type:

```
SLOW COUNTS   or   FAST COUNTS
```

at the terminal.

Line 9 provides the capability of naming individual lamps. In this application they are being used as indicator lights. The word LAMP is a defining word which takes as an argument a mask which represents a particular lamp, and compiles it as a named entity. Lines 10 and 11 contain five uses of LAMP to name particular indicators. When one of these words such as POWER is executed, the mask is returned on the stack. In fact, the behavior of defining a value such that when the word is invoked the value is returned, is identical to the behavior of a Forth CONSTANT. We created a new defining word here, however, to illustrate how this would be done.

Finally, on lines 13 and 14, we have the words that will control the light panel. LAMPS is a variable that contains the current state of the lamps. The word TOGGLE takes a mask (which might be supplied by one of the LAMP words) and changes the state of that particular lamp, saving the result in LAMPS.

In the remainder of the application, the lamp names and TOGGLE are probably the only words that will be executed directly. The usage there will be, for example:

```
POWER TOGGLE   or   SAMPLING TOGGLE
```

as appropriate, whenever the system indicators need to be changed.

The time to compile this block of code on that system was about half a second, including the time to fetch it from disk. So it is quite practical (and normal practice) for a programmer to simply type in a definition and try it immediately.

In addition, one always has the capability of communicating with external devices directly. The first thing one would do when told about the lamps would be to type:

```
HEX FF 40 OUTPUT
```

and see if all the lamps come on. If not, the presumption is that something is amiss with the hardware, since this phrase directly transmits the **all ones** mask to the device. This type of direct interaction is useful in applications involving custom hardware, as it reduces hardware debugging time.

---

## C.6 Multiprogrammed systems

Multiprogrammed Forth systems have existed since about 1970. The earliest public Forth systems propagated the **hooks** for this capability despite the fact that many did not use them. Nevertheless the underlying assumptions have been common knowledge in the community, and there exists considerable common ground among these

multiprogrammed systems. These systems are not just language processors, but contain operating system characteristics as well. Many of these integrated systems run entirely stand-alone, performing all necessary operating system functions.

Some Forth systems are very fast, and can support both multi-tasking and multi-user operation even on computers whose hardware is usually thought incapable of such advanced operation. For example, one producer of telephone switchboards is running over 50 tasks on a Z80. There are several multiprogrammed products for PC's, some of which even support multiple users. Even on computers that are commonly used in multi-user operations, the number of users that can be supported may be much larger than expected. One large data-base application running on a single 68000 has over 100 terminals updating and querying its data-base, with no significant degradation.

Multi-user systems may also support multiple programmers, each of which has a private dictionary, stacks, and a set of variables controlling that task. The private dictionary is linked to a shared, re-entrant dictionary containing all the standard Forth functions. The private dictionary can be used to develop application code which may later be integrated into the shared dictionary. It may also be used to perform functions requiring text interpretation, including compilation and execution of source code.

---

## C.7 Design and management considerations

Just as the choice of building materials has a strong effect on the design and construction of a building, the choice of language and operating system will affect both application design and project management decisions.

Conventionally, software projects progress through four stages: analysis, design, coding, and testing. A Forth project necessarily incorporates these activities as well. Forth is optimized for a project-management methodology featuring small teams of skilled professionals. Forth encourages an iterative process of **successive prototyping** wherein high-level Forth is used as an executable design tool, with **stubs** replacing lower-level routines as necessary (e.g., for hardware that isn't built yet).

In many cases successive prototyping can produce a sounder, more useful product. As the project progresses, implementors learn things that could lead to a better design. Wiser decisions can be made if true relative costs are known, and often this isn't possible until prototype code can be written and tried.

Using Forth can shorten the time required for software development, and reduce the level of effort required for maintenance and modifications during the life of the product as well.

---

## C.8 Conclusion

Forth has produced some remarkable achievements in a variety of application areas. In the last few years its acceptance has grown rapidly, particularly among programmers looking for ways to improve their productivity and managers looking for ways to simplify new software-development projects.

---

 Table of Contents
 Next Section