= Forth (programming language) =

Forth is an imperative stack @-@ based computer programming language and environment originally designed by Charles " Chuck " Moore . Language features include structured programming , reflection (the ability to modify the program structure during program execution) , concatenative programming (functions are composed with juxtaposition) and extensibility (the programmer can create new commands) . Although not an acronym , the language 's name is sometimes spelled with all capital letters as FORTH , following the customary usage during its earlier years .

A procedural programming language without type checking , Forth features both interactive execution of commands (making it suitable as a shell for systems that lack a more formal operating system) and the ability to compile sequences of commands for later execution . Some Forth implementations (usually early versions or those written to be extremely portable) compile threaded code , but many implementations today generate optimized machine code like other language compilers .

Forth is used in the Open Firmware boot loader , in space applications , such as the Philae spacecraft and other embedded systems which involve interaction with hardware . The bestselling 1986 DOS game Starflight , from Electronic Arts , was written with a custom Forth .

The free software Gforth implementation is actively maintained , as are several commercially supported systems .

= = Overview = =

A Forth environment combines the compiler with an interactive shell , where the user defines and runs subroutines called words . Words can be tested , redefined , and debugged as the source is entered without recompiling or restarting the whole program . All syntactic elements , including variables and basic operators , are defined as words . Forth environments vary in how the resulting program is stored , but ideally running the program has the same effect as manually re @-@ entering the source .

= = = Stacks = = =

Most programming environments with recursive subroutines use a stack for control flow . This structure typically also stores local variables , including subroutine parameters (in call by value system such as C) . Forth often does not have local variables , however , nor is it call @-@ by @-@ value . Instead , intermediate values are kept in a second stack . Words operate directly on the topmost values in the first stack . It may therefore be called the " parameter " or " data " stack , but most often simply " the " stack . The second , function @-@ call stack is then called the " linkage " or " return " stack , abbreviated rstack . Special rstack manipulation functions provided by the kernel allow it to be used for temporary storage within a word , but otherwise it cannot be used to pass parameters or manipulate data .

Most words are specified in terms of their effect on the stack . Typically , parameters are placed on the top of the stack before the word executes . After execution , the parameters have been erased and replaced with any return values . For arithmetic operators , this follows the rule of reverse Polish notation . See below for examples illustrating stack usage .

= = = Maintenance = = =

Forth is a simple yet extensible language ; its modularity and extensibility permit the writing of high @-@ level programs such as CAD systems . Forth has been used successfully in large , complex projects , while applications developed by competent , disciplined professionals have proven to be easily maintained on evolving hardware platforms over decades of use . Forth has a niche both in astronomical and space applications . Forth is still used today in many embedded systems (small computerized devices) because of its portability , efficient memory use , short development time ,

and fast execution speed . It has been implemented efficiently on modern RISC processors , and processors that use Forth as machine language have been produced . Other uses of Forth include the Open Firmware boot ROMs used by Apple , IBM , Sun , and OLPC XO @-@ 1 ; and the FICL @-@ based first stage boot controller of the FreeBSD operating system .

= = History = =

Forth evolved from Charles H. Moore 's personal programming system , which had been in continuous development since 1968 . Forth was first exposed to other programmers in the early 1970s , starting with Elizabeth Rather at the US National Radio Astronomy Observatory . After their work at NRAO , Charles Moore and Elizabeth Rather formed FORTH , Inc. in 1973 , refining and porting Forth systems to dozens of other platforms in the next decade .

Forth is so named because in 1968 " the file holding the interpreter was labeled FOURTH , for 4th (next) generation software ? but the IBM 1130 operating system restricted file names to 5 characters . " Moore saw Forth as a successor to compile @-@ link @-@ go third @-@ generation programming languages , or software for " fourth generation " hardware , not a fourth @-@ generation programming language as the term has come to be used .

Because Charles Moore frequently moved from job to job over his career , an early pressure on the developing language was ease of porting to different computer architectures . A Forth system has often been used to bring up new hardware . For example , Forth was the first resident software on the new Intel 8086 chip in 1978 and MacFORTH was the first resident development system for the first Apple Macintosh in 1984 .

FORTH , Inc . ' s microFORTH was developed for the Intel 8080 , Motorola 6800 , and Zilog Z80 microprocessors starting in 1976 . MicroFORTH was later used by hobbyists to generate Forth systems for other architectures , such as the 6502 in 1978 . Wide dissemination finally led to standardization of the language . Common practice was codified in the de facto standards FORTH @-@ 79 and FORTH @-@ 83 in the years 1979 and 1983 , respectively . These standards were unified by ANSI in 1994 , commonly referred to as ANS Forth .

Forth became very popular in the 1980s because it was well suited to the small microcomputers of that time , being compact and portable . At least one home computer , the British Jupiter ACE , had Forth in its ROM @-@ resident operating system . The Canon Cat also used Forth for its system programming , and Rockwell produced single @-@ chip microcomputers with resident Forth kernels , the R65F11 and R65F12 . A complete family tree is at TU @-@ Wien . Insoft GraFORTH was a version of Forth with graphics expansions for the Apple II . ASYST was a Forth expansion for measuring and controlling on PCs .

= = Programmer 's perspective = =

Forth relies heavily on explicit use of a data stack and reverse Polish notation (RPN or postfix notation) , commonly used in calculators from Hewlett @-@ Packard . In RPN , the operator is placed after its operands , as opposed to the more common infix notation where the operator is placed between its operands . Postfix notation makes the language easier to parse and extend ; Forth 's flexibility makes a static BNF grammar inappropriate , and it does not have a monolithic compiler . Extending the compiler only requires writing a new word , instead of modifying a grammar and changing the underlying implementation .

Using RPN , one could get the result of the mathematical expression (25 * 10 + 50) this way :

```
25 10 * 50 + CR .
300 ok
```

This command line first puts the numbers 25 and 10 on the implied stack .

The word * multiplies the two numbers on the top of the stack and replaces them with their product .

Then the number 50 is placed on the stack .

The word + adds it to the previous product . The CR moves the output to a new line (it is only for

formatting purposes and could be omitted but - in most implementations - without it the output would occur on the same line as the input and would be less readable in the example) . Finally , the . command prints the result to the user 's terminal . As everything has completed successfully at that point , the text interpreter then outputs the prompt " ok " and moves to a new line to get more input without needing anything explicit to do that .

Even Forth 's structural features are stack @-@ based . For example :

: FLOOR5 (n -- n ') DUP 6 < IF DROP 5 ELSE 1 - THEN ;

This code defines a new word (again , word is the term used for a subroutine) called FLOOR5 using the following commands : DUP duplicates the number on the stack ; 6 places a 6 on top of the stack ; < compares the top two numbers on the stack (6 and the DUPed input) , and replaces them with a true @-@ or @-@ false value ; IF takes a true @-@ or @-@ false value and chooses to execute commands immediately after it or to skip to the ELSE ; DROP discards the value on the stack ; and THEN ends the conditional . The text in parentheses is a comment , advising that this word expects a number on the stack and will return a possibly changed number . The FLOOR5 word is equivalent to this function written in the C programming language using the ternary operator :

This function is written more succinctly as :

: FLOOR5 (n -- n ') 1- 5 MAX ;

You could run this word as follows :

1 FLOOR5 CR .
5 ok
8 FLOOR5 CR .
7 ok

First the interpreter pushes a number (1 or 8) onto the stack , then it calls FLOOR5 , which pops off this number again and pushes the result . The CR moves the output to a new line (again , this is only here for readability) . Finally , a call to " . " pops the result and prints it to the user 's terminal .

= = Facilities = =

Forth has no explicit grammar . The interpreter reads a line of input from the user input device , which is then parsed for a word using spaces as a delimiter ; some systems recognise additional whitespace characters . When the interpreter finds a word , it looks the word up in the dictionary . If the word is found , the interpreter executes the code associated with the word , and then returns to parse the rest of the input stream . If the word isn 't found , the word is assumed to be a number and an attempt is made to convert it into a number and push it on the stack ; if successful , the interpreter continues parsing the input stream . Otherwise , if both the lookup and the number conversion fail , the interpreter prints the word followed by an error message indicating the word is not recognised , flushes the input stream , and waits for new user input .

The definition of a new word is started with the word : (colon) and ends with the word ; (semi @-@ colon) . For example ,

: X DUP 1 + . . ;

will compile the word X , and makes the name findable in the dictionary . When executed by typing 10 X at the console this will print 11 10 .

Most Forth systems include an assembler that allows one to specify words using the processor 's facilities at its lowest level . Mostly the assembler is tucked away in a separate namespace (wordlist) as relatively few users want to use it . Forth assemblers may use a reverse @-@ polish syntax in which the parameters of an instruction precede the instruction , but designs vary widely and are specific to the Forth implementation . A typical reverse @-@ polish assembler prepares the operands on the stack and have the mnemonic copy the whole instruction into memory as the last step . A Forth assembler is by nature a macro assembler , so that it is easy to define an alias for registers according to their role in the Forth system : e.g. " datastackpointer " for the register used as a stack pointer .

= = = Operating system , files , and multitasking = = =

Most Forth systems run under a host operating system such as Microsoft Windows , Linux or a version of Unix and use the host operating system 's file system for source and data files ; the ANSI Forth Standard describes the words used for I / O. All modern Forth systems use normal text files for source , even if they are embedded . An embedded system with a resident compiler gets its source via a serial line .

Classic Forth systems traditionally use neither operating system nor file system . Instead of storing code in files , source code is stored in disk blocks written to physical disk addresses . The word BLOCK is employed to translate the number of a 1K @-@ sized block of disk space into the address of a buffer containing the data , which is managed automatically by the Forth system . Block use has become rare since the mid @-@ 1990s . In a hosted system those blocks too are allocated in a normal file in any case .

Multitasking , most commonly cooperative round @-@ robin scheduling , is normally available (although multitasking words and support are not covered by the ANSI Forth Standard) . The word PAUSE is used to save the current task 's execution context , to locate the next task , and restore its execution context . Each task has its own stacks , private copies of some control variables and a scratch area . Swapping tasks is simple and efficient ; as a result , Forth multitaskers are available even on very simple microcontrollers , such as the Intel 8051 , Atmel AVR , and TI MSP430 .

Other non @-@ standard facilities include a mechanism for issuing calls to the host OS or windowing systems , and many provide extensions that employ the scheduling provided by the operating system . Typically they have a larger and different set of words from the stand @-@ alone Forth 's PAUSE word for task creation , suspension , destruction and modification of priority .

= = = Self @-@ compilation and cross compilation = = =

A fully featured Forth system with all source code will compile itself , a technique commonly called meta @-@ compilation by Forth programmers (although the term doesn 't exactly match meta @-@ compilation as it is normally defined) . The usual method is to redefine the handful of words that place compiled bits into memory . The compiler 's words use specially named versions of fetch and store that can be redirected to a buffer area in memory . The buffer area simulates or accesses a memory area beginning at a different address than the code buffer . Such compilers define words to access both the target computer 's memory , and the host (compiling) computer 's memory .

After the fetch and store operations are redefined for the code space , the compiler , assembler , etc. are recompiled using the new definitions of fetch and store . This effectively reuses all the code of the compiler and interpreter . Then , the Forth system 's code is compiled , but this version is stored in the buffer . The buffer in memory is written to disk , and ways are provided to load it temporarily into memory for testing . When the new version appears to work , it is written over the previous version .

Numerous variations of such compilers exist for different environments . For embedded systems , the code may instead be written to another computer , a technique known as cross compilation , over a serial port or even a single TTL bit , while keeping the word names and other non @-@ executing parts of the dictionary in the original compiling computer . The minimum definitions for such a Forth compiler are the words that fetch and store a byte , and the word that commands a Forth word to be executed . Often the most time @-@ consuming part of writing a remote port is constructing the initial program to implement fetch , store and execute , but many modern microprocessors have integrated debugging features (such as the Motorola CPU32) that eliminate this task .

= = Structure of the language = =

The basic data structure of Forth is the " dictionary " which maps " words " to executable code or named data structures . The dictionary is laid out in memory as a tree of linked lists with the links proceeding from the latest (most recently) defined word to the oldest , until a sentinel value ,

usually a NULL pointer , is found . A context switch causes a list search to start at a different leaf . A linked list search continues as the branch merges into the main trunk leading eventually back to the sentinel , the root . There can be several dictionaries . In rare cases such as meta @-@ compilation a dictionary might be isolated and stand @-@ alone . The effect resembles that of nesting namespaces and can overload keywords depending on the context .

A defined word generally consists of head and body with the head consisting of the name field (NF) and the link field (LF) and body consisting of the code field (CF) and the parameter field (PF) .

Head and body of a dictionary entry are treated separately because they may not be contiguous . For example , when a Forth program is recompiled for a new platform , the head may remain on the compiling computer , while the body goes to the new platform . In some environments (such as embedded systems) the heads occupy memory unnecessarily . However , some cross @-@ compilers may put heads in the target if the target itself is expected to support an interactive Forth .

= = = Dictionary entry = = =

The exact format of a dictionary entry is not prescribed , and implementations vary . However , certain components are almost always present , though the exact size and order may vary . Described as a structure , a dictionary entry might look this way :

```
structure
byte : flag \ 3bit flags + length of word 's name
char @-@ array : name \ name 's runtime length isn 't known at compile time
address : previous \ link field , backward ptr to previous word
address : codeword \ ptr to the code to execute this word
any @-@ array : parameterfield \ unknown length of data , words , or opcodes
end @-@ structure forthword
```

The name field starts with a prefix giving the length of the word 's name (typically up to 32 bytes) , and several bits for flags . The character representation of the word 's name then follows the prefix . Depending on the particular implementation of Forth , there may be one or more NUL (' \ 0 ') bytes for alignment .

The link field contains a pointer to the previously defined word . The pointer may be a relative displacement or an absolute address that points to the next oldest sibling .

The code field pointer will be either the address of the word which will execute the code or data in the parameter field or the beginning of machine code that the processor will execute directly . For colon defined words , the code field pointer points to the word that will save the current Forth instruction pointer (IP) on the return stack , and load the IP with the new address from which to continue execution of words . This is the same as what a processor 's call / return instructions does .

= = = Structure of the compiler = = =

The compiler itself is not a monolithic program . It consists of Forth words visible to the system , and usable by a programmer . This allows a programmer to change the compiler 's words for special purposes .

The " compile time " flag in the name field is set for words with " compile time " behavior . Most simple words execute the same code whether they are typed on a command line , or embedded in code . When compiling these , the compiler simply places code or a threaded pointer to the word .

The classic examples of compile @-@ time words are the control structures such as IF and WHILE . Almost all of Forth 's control structures and almost all of its compiler are implemented as compile @-@ time words . Apart from some rarely used control flow words only found in a few implementations , such as a conditional return , all of Forth 's control flow words are executed during compilation to compile various combinations of primitive words along with their branch addresses . For instance , IF and WHILE , and the words that match with those , set up BRANCH (unconditional branch) and ? BRANCH (pop a value off the stack , and branch if it is false) . Counted loop control

flow words work similarly but set up combinations of primitive words that work with a counter , and so on . During compilation , the data stack is used to support control structure balancing , nesting , and back @-@ patching of branch addresses . The snippet :

... DUP 6 < IF DROP 5 ELSE 1 - THEN ...

would be compiled to the following sequence inside a definition :

... DUP LIT 6 < ? BRANCH 5 DROP LIT 5 BRANCH 3 LIT 1 - ...

The numbers after BRANCH represent relative jump addresses . LIT is the primitive word for pushing a " literal " number onto the data stack .

= = = = Compilation state and interpretation state = = = =

The word : (colon) parses a name as a parameter , creates a dictionary entry (a colon definition) and enters compilation state . The interpreter continues to read space @-@ delimited words from the user input device . If a word is found , the interpreter executes the compilation semantics associated with the word , instead of the interpretation semantics . The default compilation semantics of a word are to append its interpretation semantics to the current definition .

The word ; (semi @-@ colon) finishes the current definition and returns to interpretation state . It is an example of a word whose compilation semantics differ from the default . The interpretation semantics of ; (semi @-@ colon) , most control flow words , and several other words are undefined in ANS Forth , meaning that they must only be used inside of definitions and not on the interactive command line .

The interpreter state can be changed manually with the words [(left @-@ bracket) and] (right @-@ bracket) which enter interpretation state or compilation state , respectively . These words can be used with the word LITERAL to calculate a value during a compilation and to insert the calculated value into the current colon definition . LITERAL has the compilation semantics to take an object from the data stack and to append semantics to the current colon definition to place that object on the data stack .

In ANS Forth , the current state of the interpreter can be read from the flag STATE which contains the value true when in compilation state and false otherwise . This allows the implementation of so @-@ called state @-@ smart words with behavior that changes according to the current state of the interpreter .

= = = = Immediate words = = = =

The word IMMEDIATE marks the most recent colon definition as an immediate word , effectively replacing its compilation semantics with its interpretation semantics . Immediate words are normally executed during compilation , not compiled but this can be overridden by the programmer , in either state . ; is an example of an immediate word . In ANS Forth , the word POSTPONE takes a name as a parameter and appends the compilation semantics of the named word to the current definition even if the word was marked immediate . Forth @-@ 83 defined separate words COMPILE and [COMPILE] to force the compilation of non @-@ immediate and immediate words , respectively .

= = = = Unnamed words and execution tokens = = = =

In ANS Forth , unnamed words can be defined with the word : NONAME which compiles the following words up to the next ; (semi @-@ colon) and leaves an execution token on the data stack . The execution token provides an opaque handle for the compiled semantics , similar to the function pointers of the C programming language .

Execution tokens can be stored in variables . The word EXECUTE takes an execution token from the data stack and performs the associated semantics . The word COMPILE , (compile @-@ comma) takes an execution token from the data stack and appends the associated semantics to the current definition .

The word ' (tick) takes the name of a word as a parameter and returns the execution token

associated with that word on the data stack . In interpretation state , ' RANDOM @-@ WORD EXECUTE is equivalent to RANDOM @-@ WORD .

= = = = Parsing words and comments = = = =

The words : (colon) , POSTPONE , ' (tick) are examples of parsing words that take their arguments from the user input device instead of the data stack . Another example is the word ((paren) which reads and ignores the following words up to and including the next right parenthesis and is used to place comments in a colon definition . Similarly , the word \ (backslash) is used for comments that continue to the end of the current line . To be parsed correctly , ((paren) and \ (backslash) must be separated by whitespace from the following comment text .

= = = Structure of code = = =

In most Forth systems , the body of a code definition consists of either machine language , or some form of threaded code . The original Forth which follows the informal FIG standard (Forth Interest Group) , is a TIL (Threaded Interpretive Language) . This is also called indirect @-@ threaded code , but direct @-@ threaded and subroutine threaded Forths have also become popular in modern times . The fastest modern Forths use subroutine threading , insert simple words as macros , and perform peephole optimization or other optimizing strategies to make the code smaller and faster .

= = = Data objects = = =

When a word is a variable or other data object , the CF points to the runtime code associated with the defining word that created it . A defining word has a characteristic " defining behavior " (creating a dictionary entry plus possibly allocating and initializing data space) and also specifies the behavior of an instance of the class of words constructed by this defining word . Examples include :

VARIABLE

Names an uninitialized , one @-@ cell memory location . Instance behavior of a VARIABLE returns its address on the stack .

CONSTANT

Names a value (specified as an argument to CONSTANT) . Instance behavior returns the value .

CREATE

Names a location ; space may be allocated at this location , or it can be set to contain a string or other initialized value . Instance behavior returns the address of the beginning of this space .

Forth also provides a facility by which a programmer can define new application @-@ specific defining words , specifying both a custom defining behavior and instance behavior . Some examples include circular buffers , named bits on an I / O port , and automatically indexed arrays .

Data objects defined by these and similar words are global in scope . The function provided by local variables in other languages is provided by the data stack in Forth (although Forth also has real local variables) . Forth programming style uses very few named data objects compared with other languages ; typically such data objects are used to contain data which is used by a number of words or tasks (in a multitasked implementation) .

Forth does not enforce consistency of data type usage ; it is the programmer 's responsibility to use appropriate operators to fetch and store values or perform other operations on data .

= = Programming = =

Words written in Forth are compiled into an executable form . The classical " indirect threaded " implementations compile lists of addresses of words to be executed in turn ; many modern systems generate actual machine code (including calls to some external words and code for others expanded in place) . Some systems have optimizing compilers . Generally speaking , a Forth

program is saved as the memory image of the compiled program with a single command (e.g. , RUN) that is executed when the compiled version is loaded .

During development , the programmer uses the interpreter in REPL mode to execute and test each little piece as it is developed . Most Forth programmers therefore advocate a loose top @-@ down design , and bottom @-@ up development with continuous testing and integration .

The top @-@ down design is usually separation of the program into " vocabularies " that are then used as high @-@ level sets of tools to write the final program . A well @-@ designed Forth program reads like natural language , and implements not just a single solution , but also sets of tools to attack related problems .

= = Code examples = =

= = = Hello world = = =

One possible implementation :

```
: HELLO ( -- ) CR . " Hello , world ! " ;
HELLO < cr >
Hello , world !
```

The word CR (Carriage Return) causes the following output to be displayed on a new line . The parsing word . " (dot @-@ quote) reads a double @-@ quote delimited string and appends code to the current definition so that the parsed string will be displayed on execution . The space character separating the word . " from the string Hello , world ! is not included as part of the string . It is needed so that the parser recognizes . " as a Forth word .

A standard Forth system is also an interpreter , and the same output can be obtained by typing the following code fragment into the Forth console :

```
CR . ( Hello , world ! )
```

. ((dot @-@ paren) is an immediate word that parses a parenthesis @-@ delimited string and displays it . As with the word . " the space character separating . (from Hello , world ! is not part of the string .

The word CR comes before the text to print . By convention , the Forth interpreter does not start output on a new line . Also by convention , the interpreter waits for input at the end of the previous line , after an ok prompt . There is no implied " flush @-@ buffer " action in Forth 's CR , as sometimes is in other programming languages .

= = = Mixing states of compiling and interpreting = = =

Here is the definition of a word EMIT @-@ Q which when executed emits the single character Q :

```
: EMIT @-@ Q 81 ( the ASCII value for the character ' Q ' ) EMIT ;
```

This definition was written to use the ASCII value of the Q character (81) directly . The text between the parentheses is a comment and is ignored by the compiler . The word EMIT takes a value from the data stack and displays the corresponding character .

The following redefinition of EMIT @-@ Q uses the words [(left @-@ bracket) ,] (right @-@ bracket) , CHAR and LITERAL to temporarily switch to interpreter state , calculate the ASCII value of the Q character , return to compilation state and append the calculated value to the current colon definition :

```
: EMIT @-@ Q [ CHAR Q ] LITERAL EMIT ;
```

The parsing word CHAR takes a space @-@ delimited word as parameter and places the value of its first character on the data stack . The word [CHAR] is an immediate version of CHAR . Using [CHAR] , the example definition for EMIT @-@ Q could be rewritten like this :

```
: EMIT @-@ Q [ CHAR ] Q EMIT ; \ Emit the single character ' Q'
```

This definition used \ (backslash) for the describing comment .

Both CHAR and [CHAR] are predefined in ANS Forth . Using IMMEDIATE and POSTPONE , [

CHAR] could have been defined like this :
: [CHAR] CHAR POSTPONE LITERAL ; IMMEDIATE

= = = A complete RC4 cipher program = = =

In 1987 , Ron Rivest developed the RC4 cipher @-@ system for RSA Data Security , Inc . The code is extremely simple and can be written by most programmers from the description :
We have an array of 256 bytes , all different . Every time the array is used it changes by swapping two bytes . The swaps are controlled by counters i and j , each initially 0 . To get a new i , add 1 . To get a new j , add the array byte at the new i . Exchange the array bytes at i and j . The code is the array byte at the sum of the array bytes at i and j . This is XORed with a byte of the plaintext to encrypt , or the ciphertext to decrypt . The array is initialized by first setting it to 0 through 255 . Then step through it using i and j , getting the new j by adding to it the array byte at i and a key byte , and swapping the array bytes at i and j . Finally , i and j are set to 0 . All additions are modulo 256 .
The following Standard Forth version uses Core and Core Extension words only .

```
0 value ii 0 value jj
0 value KeyAddr 0 value KeyLen
create SArray 256 allot \ state array of 256 bytes
: KeyArray KeyLen mod KeyAddr ;
: get _ byte + c @ ;
: set _ byte + c ! ;
: as _ byte 255 and ;
: reset _ ij 0 TO ii 0 TO jj ;
: i _ update 1 + as _ byte TO ii ;
: j _ update ii SArray get _ byte + as _ byte TO jj ;
: swap _ s _ ij
jj SArray get _ byte
ii SArray get _ byte jj SArray set _ byte
ii SArray set _ byte
;
: rc4 _ init ( KeyAddr KeyLen -- )
256 min TO KeyLen TO KeyAddr
256 0 DO i i SArray set _ byte LOOP
reset _ ij
BEGIN
ii KeyArray get _ byte jj + j _ update
swap _ s _ ij
ii 255 < WHILE
ii i _ update
REPEAT
reset _ ij
;
: rc4 _ byte
ii i _ update jj j _ update
swap _ s _ ij
ii SArray get _ byte jj SArray get _ byte + as _ byte SArray get _ byte xor
;
```

This is one of many ways to test the code :

```
hex
create AKey 61 c , 8A c , 63 c , D2 c , FB c ,
: test cr 0 DO rc4 _ byte . LOOP cr ;
AKey 5 rc4 _ init
2C F9 4C EE DC 5 test \ output should be : F1 38 29 C9 DE
```

= = Implementations = =

Because the Forth virtual machine is simple to implement and has no standard reference implementation , there are numerous implementations of the language . In addition to supporting the standard varieties of desktop computer systems (POSIX , Microsoft Windows , Mac OS X) , many of these Forth systems also target a variety of embedded systems . Listed here are some of the more prominent systems which conform to the 1994 ANS Forth standard .

Gforth - a portable ANS Forth implementation from the GNU Project

SwiftForth - native code desktop and embedded Forths by Forth , Inc . , originators of the language ;

VFX Forth - highly @-@ optimizing native code Forth

Open Firmware - a bootloader and BIOS standard based on ANS Forth