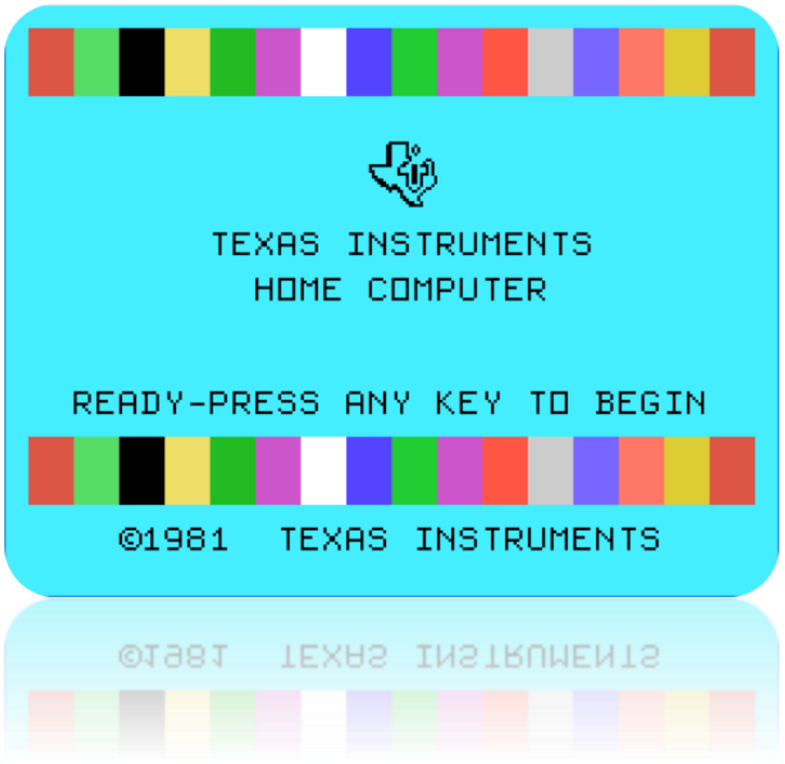


Camel99

A Forth system for the TI-BASIC programmer

Brian Fox

Table of Contents

Introduction

The purpose of CAMEL99 Forth is to assist the programmer familiar with TI-BASIC or Extended BASIC to learn the Forth programming language in general and also learn more about some of the low level details of the TI-99 computer. These details are hidden by the BASIC language but when presented properly are not difficult for an experience programmer to understand.

This document is NOT a tutorial on the Forth programming language. For a better understanding of Forth download a copy of Starting Forth by Leo Brodie, in the updated version at :

https://www.forth.com/starting-forth/1-forth-stacks-dictionary/

It is very important to understand that BASIC and Forth really take a different approach to how the computer should be presented to the human doing the programming. Fortunately Forth was designed to be changeable. The inventor of Forth, Charles Moore felt that no programming language ever had exactly what he needed so he made a language that starts with very little but then lets you add things to it.

Nevertheless it will be impossible to hide some of these big differences but we will try to bridge the gaps for you with additions to the language that make you feel at home.

DON'T BE DISCOURAGED IF SOMETIMES THE CONCEPTS SEEM STRANGE.

Although CAMEL99 is just a regular Forth compiler/interpreter we have given you the ability to load new words into the language that provide "training wheels" for a TI-BASIC programmer. The big difference is that you have the source code for these language extensions and so when the time comes you will be able to see exactly how we created these new words. In the mean time you will be able to create TI-99 programs in a "dialect" of Forth created just for people who know TI-BASIC or Extended BASIC.

About CAMEL Forth

CAMEL99 was created for the TI-99 and is a variation of Camel Forth created by Dr. Brad Rodriguez. Camel Forth was designed to be a portable Forth system that complies with ANS/ISO Forth 94 language specification. Dr. Rodriguez is aware of CAMEL99 and his work in developing Camel Forth was essential in the creation of CAMEL99.

Camel Forth was created to be a transportable Forth system that could be easily ported to different machines. It was not intended to be the speediest system but rather it stressed ease of movement to different platforms.

It is called Camel Forth due to a perception about ANS/ISO Forth. Creating the Forth standard was a challenging process given the fact that Forth as envisioned by its creator Charles Moore, doesn't actually have syntax. If you have heard the joke that "A camel is a horse designed by a committee" then you have an insight into the origin of the name CAMEL FORTH.

Brian Fox

What you have

Camel99 is a Forth interpreter/compiler for the TI-99 computer. In case you are curious about such things, it was created using a Cross-compiler that was also written in Forth.   The Cross-compiler took Forth Assembly language text and Forth language text and converted it all to TMS9000 machine code.

The contents of this package are as follows:

- CAMEL99   program file
    - This program is a TI-99 “program file” (binary image) that can be loaded with the Editor/Assembler cartridge installed in the TI-99 computer. Select option ‘5’ “Load program file” and at the prompt enter the disc no. and KERNEL99
        - Example: DSK1.CAMEL99
    - CAMEL99 is a VERY lean Forth Compiler/interpreter. It is the bare minimum that we need to run Forth. BUT…   what is does have is the ability to load programs that EXTEND the language. That is the Forth “super power”.

Library Files

To create more interesting programs requires routines that provide expanded functionality specific to the TI-99. This “library” code is provided as source code (text) that can be compiled into CAMEL99 to extend the functionality of the system. The TI-99 files are can be edited with the Editor Assembler Editor and saved back to disk. Some of the files are very simple and may provide only one new WORD to the system. Other files give the system words that align with the function of TI-BASIC. All library files end with the extension .FTH if they are PC text files.

File Format

CAMEL99 Forth assumes that all source code files (programs in text form) are “DV80” format files. To open these files in BASIC you would write:

OPEN #1: “DSK1.ASM9900.F”, DISPLAY VARIABLE 80.SEQUENTIAL

Only Needs the E/A Package.

DV80 format is the default format for the TI-Editor in the editor assembler (E/A) package. This means you can begin working with CAMEL99 Forth with only the E/A cartridge. Write your program with the editor, save the file.

File Naming Convention (Optional)

Files that are for the TI-99 have the extension “.F”   This keeps the 8 character file names of MS DOS with 2 characters reserved for the extension. Of course if you don’t like using the .F extension use whatever you prefer. CAMEL99 Forth does not care about the extensions and simply matches the entire 10 characters of the TI-99 file name.

Forth Terminology

Forth is one of the unusual programming languages and as such uses some terminology that requires a little explanation. The good news is that it is all very simple, which is in line with the Forth philosophy. Keep it simple.

Forth name	TI-BASIC Equivalent	Explanation
WORD	sub-routine	A name in the Forth dictionary that runs some code is easy to understand as a sub-routine. Every Forth Word runs some code, even variables and constants so technically they are all just sub-routines… I mean WORDs.
DICTIONARY	Name space. You don’t need to know this in BASIC, but the console keeps a list of all the words that BASIC understands and all the variable names and sub-programs that you create.	The Forth system keeps the names of all the routines in a linked list called the dictionary. Unlike BASIC Forth lets you create other names spaces if that makes your job simpler.
Parameter Stack	Stack. In BASIC you never need to know this, but TI-BASIC has it’s own stack in memory. The TI GPL language even has a separate parameter and return stack like Forth.	Forth uses the parameter stack to communicate between sub-routines. Typically Inputs come from the stack and outputs go back onto the stack for the next routine to pick-up.
Return Stack	Return Stack. TI-BASIC also has a return stack for GOSUB. That’s how it knows what line number to return to without being told.	Forth maintains a separate stack just for return addresses, freeing up the parameter stack for … well… parameters.
CODE WORD	Assembler, ASM. You cannot do assembly language directly from TI-BASIC . With the Editor/Assembler cartridge and programs you can write Assembler and call your programs from BASIC. It is many times more complicated than doing it in Forth.	Forth words CAN BE written in Forth Assembler or machine code. You have to include the ASSEMBLER program in CAMEL99 for Forth to understand 9900 assembly language. Communication between Forth and Assembly is the same as with Forth words. Put things on the stack run the ASM code and put the answer back on the stack. The forth Assembler has words to make that simple.
Colon Definition	Sub-routine, function	A word in Forth that is defined with the ‘:’ (colon) operator. These words are the same as words created in the Forth Kernel.

CELL	A Memory location. The size of a one memory item is not a concern in BASIC	In Forth a CELL is the amount of memory that the CPU needs to hold one of it's natural integers. So a 16 bit CPU like the TI 9900 needs two bytes for a cell. A 32 bit CPU needs 4 bytes. Using CELLs in your program let's your program move across hardware platforms easier.
COMPILER	A program that converts a text file to an object code file.	The same meaning but Forth also has "mini" compilers. For example a little routine that puts a number into the next available memory CELL is called a number COMPILER in the Forth world.

Stack Diagrams

In order to document the inputs and outputs of Forth words a notation has been developed called a "stack diagram". This is simply a comment that shows what should be on the stack when a WORD executes and what will be on the stack when the word completes its execution.

Here is the stack diagram for '+' for example:

```
+     ( n n – n )
```

The '(' bracket is a comment word. It ignores everything until it reads a ')' in the code.

Notice there are two 'n's which are the input numbers. The two hypens separate inputs and outputs.

When '+' is finished running there is only one 'n' remaining on the stack. This is the sum of the two input numbers. It is always a good idea to document your Forth code with Stack diagrams the first time a new word is defined in your program. You won't regret doing it when you need to remember how it all works.

You can put anything you want in your stack diagrams that helps you understand your code. Forth convention has typically used the following:

- n – a signed integer
- u – un-signed integer
- addr – a computer address
- caddr u – a character address and the length. (in other words a string on the stack)
- $ - The author uses this one for a counted string. That is an address where the first byte is the length
- d – a double integer. 32 bits. Not fully implemented in CAMEL99 Forth
- F - A floating point number. No currently supported in CAMEL99 Forth

Standard Forth vs CAMEL99 Forth

Standard Forth is designed to be used by software engineers so it is what we call low level like Assembly Language. You have to build a lot of things yourself. For example there is a limited set of words that work with strings. Standard Forth is more like a box of LEGO bricks. You can make anything but it takes some effort.

Another thing that Standard Forth cannot provide is out of the box support for the TI-99 Graphics chip, the TMS9918 or the sound chip, the TMS9919. Forth provides a few output words like EMIT to put a character on the screen and TYPE which puts a string of characters on the screen. (Type actually just puts EMIT in a loop)

So with CAMEL99 Forth we have provided library files that add the words you would come to expect with a TI-99 Forth system. The GRAPHICS words are in a file called GRAFIX.FTH. The string functions that you have grown so fond of in BASIC are in the file STRINGS.FTH. There is also a file called INPUT.FTH that gives you something very similar to the BASIC input statement.

One big difference with these WORDS in Forth versus BASIC is that you can see how we made them. The source code is there for you to study and improve if you want to do so.

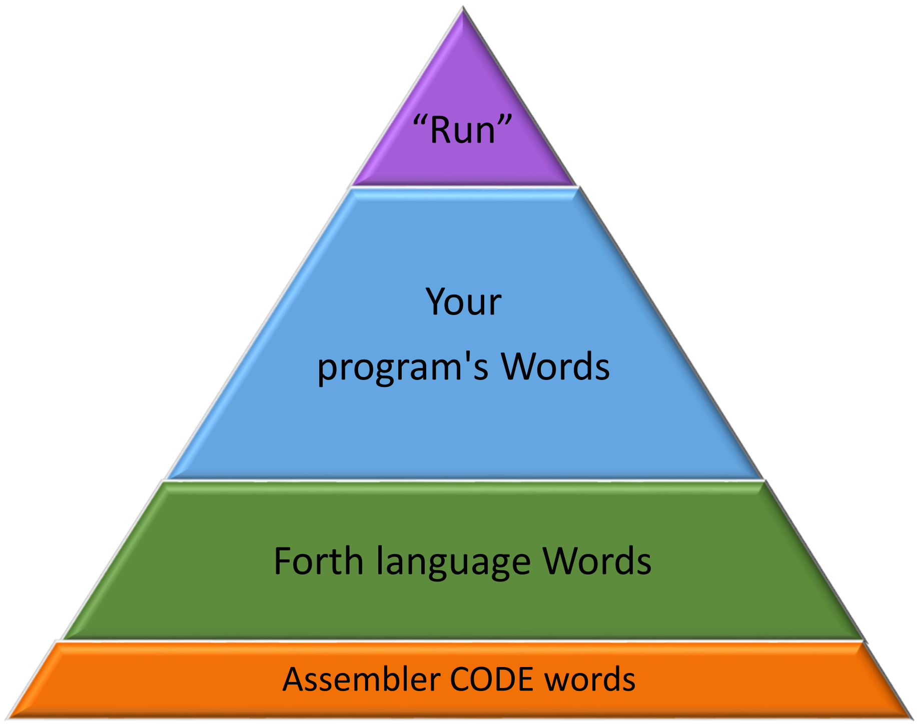
So what we provide in CAMEL99 Forth is some training wheels for the programmer who is new to Forth but familiar with BASIC. You can take them off anytime you want, but they let you start riding right away.

Library files Available with CAMEL99 Forth

ASM9900.FTH
BOOLEAN.FTH
BREAK.FTH
BUFFER.FTH
BYTES.FTH
CASE.FTH
CHAR.HSF
CHARSET.FTH
CODE.FTH
COMPARE.FTH
DATA.FTH
DATABYTE.FTH
DEFER99.FTH
DOES.FTH
ERASE.FTH
FLOORED.FTH
GRAFIX.FTH
GROMS.FTH
HEAP.FTH
HEXSTR.FTH
INCLUDE.FTH
INLIN99.FTH
INPUT.FTH
LOWCASE.FTH
MINI-OOF.FTH
PATTERN.FTH
RANDOM.FTH
SEARCH.FTH
SOUND.FTH
SPRITES.FTH
SQRT.FTH
STRINGS.FTH
STRPLUS.FTH
TILOGO.FTH
TINYHEAP.FTH
TOOLS.FTH
TRIG.FTH
UDOTR.FTH
VALUES.FTH
VHEAP.FTH

The Forth Programming Pyramid

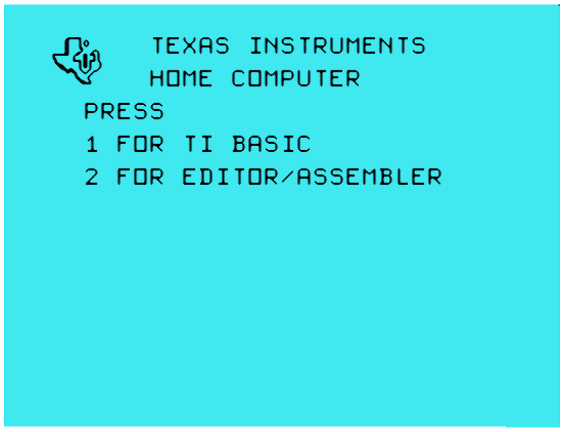
You program in Forth by adding functionality to the Forth system itself. Your WORDs are added to the system word by word and you can test each word as you go if you want. This makes your pyramid of code very solid and reliable.



- Assembler CODE words are the foundation of the language that control the computer hardware
- The Forth language is normally written using a mix of those Assembler words and Forth
- Your program's words are just more words added to the system.
  No different than the Language itself.
- All of this culminates in creating the final word that starts the entire program running which you can call anything you like. RUN is as good as any other.

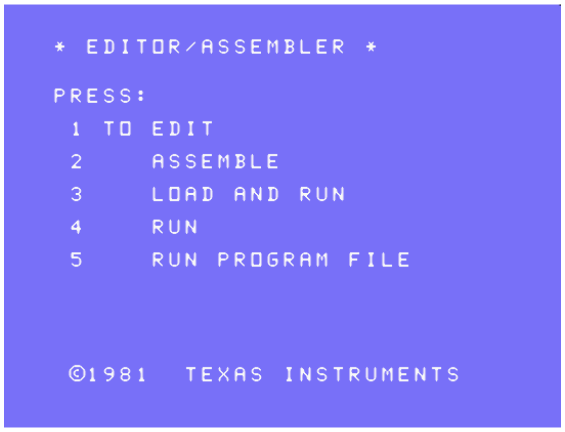
Loading CAMEL99

Insert the TI Editor/Assembler Cartridge into the TI-99 Console. Insert the diskette with CAMEL99.



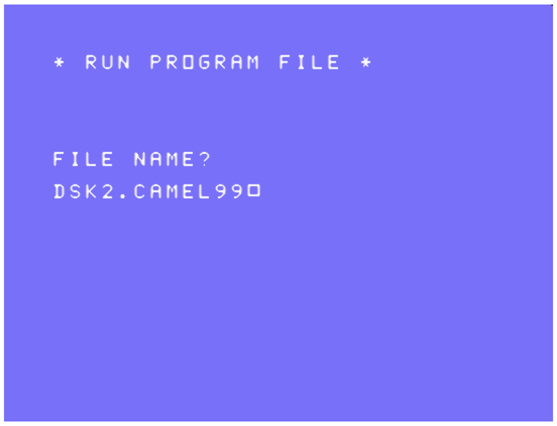
```
TEXAS INSTRUMENTS
HOME COMPUTER
PRESS
1 FOR TI BASIC
2 FOR EDITOR/ASSEMBLER
```

Figure 1 Select 2 For EDITOR/ASSEMBLER



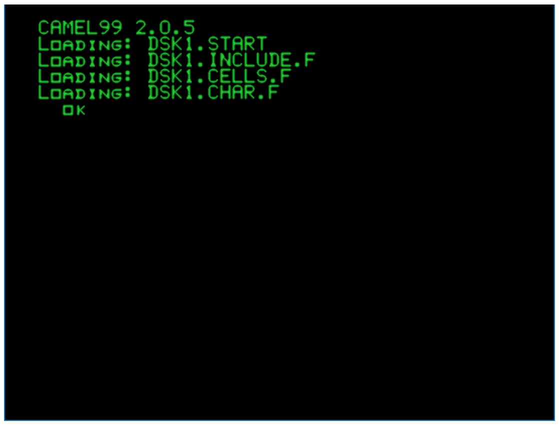
```
* EDITOR/ASSEMBLER *

PRESS:
 1 TO EDIT
 2    ASSEMBLE
 3    LOAD AND RUN
 4    RUN
 5    RUN PROGRAM FILE


 ©1981  TEXAS INSTRUMENTS
```

Figure 2 Select 5. RUN PROGRAM FILE



```
* RUN PROGRAM FILE *


FILE NAME?
DSK2.CAMEL99
```

Figure 3 Type DSK2.CAMEL99



```
CAMEL99 2.0.5
LOADING: DSK1.START
LOADING: DSK1.INCLUDE.F
LOADING: DSK1.CELLS.F
LOADING: DSK1.CHAR.F
  OK
```

Figure 4 your screen should look like this

Note:

Figure 3 shows DSK2.CAMEL99. Use the disk number where CAMEL99 is located on your system

First Things First

You will notice that when CAMEL99 starts it seems to be doing something with files.

What it is doing is adding some extra abilities to itself by compiling a few files. It knows what to do by reading the file called DSK1.START. If you edit DSK1.START with the editor you can make it do whatever you want. This is a little like AUTOEXEC.BAT in MSDOS except that it is not a different language. It's just Forth. If you remove all the text or comment out the code in the START file it will do nothing and just go to the Forth console.

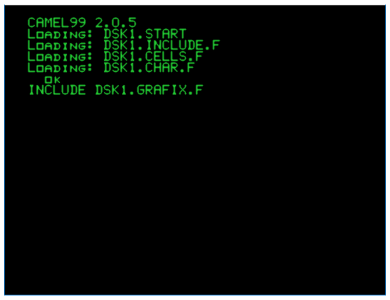
Lesson 1: Jump Right in

Before we learn about the some of the details about Forth let's see if we can calm your nerves a little with a simple program example. And please take a minute to type the example into Forth and try it.

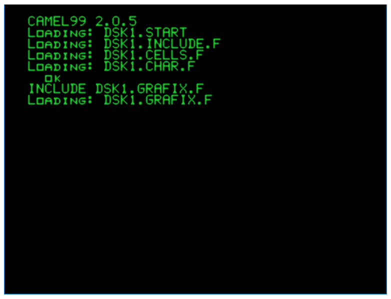
That is one thing that BASIC and Forth have in common. You can test your ideas at the console, one line at a time if you need to because they both have an "interpreter".

First we need to load the GRAPHICS mode words into the system. Here is your first Forth command, "INCLUDE".

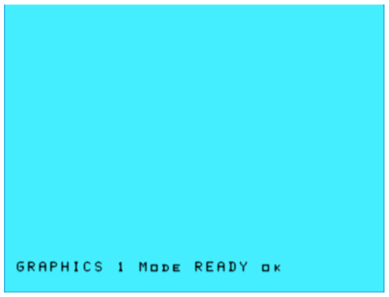
 *In FORTH type: **INCLUDE DSK1.GRAFIX.F <enter>** and after a few seconds you should see the familiar cyan colored, 32 column screen.



```
CAMEL99 2.0.5
Loading: DSK1.START
Loading: DSK1.INCLUDE.F
Loading: DSK1.CELLS.F
Loading: DSK1.CHAR.F
  ok
INCLUDE DSK1.GRAFIX.F
```



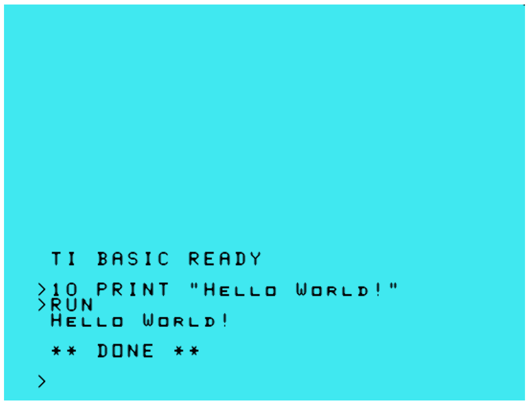
```
CAMEL99 2.0.5
Loading: DSK1.START
Loading: DSK1.INCLUDE.F
Loading: DSK1.CELLS.F
Loading: DSK1.CHAR.F
  ok
INCLUDE DSK1.GRAFIX.F
Loading: DSK1.GRAFIX.F
```



```
GRAPHICS 1 Mode READY ok
```

Hello World

Consider this one-line program in BASIC.



```
 TI BASIC READY
>10 PRINT "Hello World!"
>RUN
 Hello World!

 ** DONE **

>
```

We can see that CAMEL99 is working in the GRAPHICS mode like TI-BASIC with 32 columns and the familiar black on cyan coloring.

10 is the line number in BASIC. A line number is an Identifier for the BASIC interpreter to find the line of code. So you can give BASIC a command like this:

```
> RUN
```

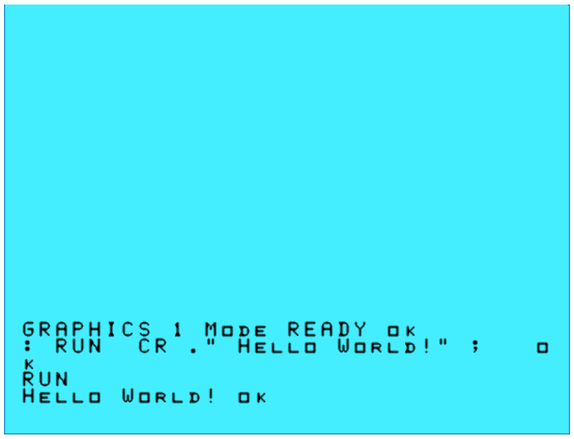
And BASIC will run the program starting from the lowest line number. The Line Number is how BASIC knows where to start running the program.

In Forth there are no line numbers. The Identifier in Forth is called a word. It can be any word as long it is between 1 and 31 characters long does not have a space in it. So here is what the equivalent program looks like in Forth. To make things familiar we called our program "RUN". Isn't that cute.

```
: RUN     CR ." Hello World!" ;
             ^
```

Watch out! We need this space in the code.

And here is the result when we enter the code and type RUN…



```
GRAPHICS 1 MODE READY ok
: RUN  CR ." HELLO WORLD!" ;    
ok
RUN
HELLO WORLD! ok
```

CAMEL99 Forth Hello Program

Things to notice:

- We started the program with ':'
    - This turns on the compiler
    - (really)
- We needed to tell Forth to start printing on a new line with the word 'CR'
- PRINT is reduced to just ."
- There needs to be a space after ." because there needs to be at least 1 space between every Forth word
- We ended the program with ';'
    - This is like RETURN in BASIC and it also turns off the compiler.

Congratulation! You wrote AND compiled your first Forth program. If you want to you can try making some new words using the same template and make them print other things.

One thing I should warned you about Forth. You are communicating with the computer very "close to the metal" as they say. There is very little protecting you from crashing the machine. This can be frustrating in the beginning but it is part of why Forth can do so much with so little.

"With great power comes great responsibility."

Fortunately the TI-99 is quickly reset if you make a big mistake. It is your right as a Forth programmer to crash the machine. It proves you have control of everything.

The Colon Compiler

The first thing on the line of our Forth program is the colon character. The ':' in our Forth program is a command. Yes really. To the Forth interpreter colon means:

- Turn on the compiler
- Read the next characters as a string until you get to a space and put that string in the dictionary as a new word in the language
- **COMPILE** all the Forth words that come after the first string into memory until you get to a semi-colon. (';')

Note:

COMPILE does not mean put the text of the words into memory. It means translate the text into a form of computer code. For the curious, CAMEL99 compiles to a list of addresses that contain code that is written in Assembler. There are other ways to compile Forth, but this way is the oldest and simplest.

We call this whole structure a "**colon definition**". This is how easy it is to use the Forth compiler.

The Semi-Colon

The semi-colon at the end of the definition is also a Forth command. It's a sneaky little devil. Semi-colon doesn't care if the compiler is turned on, it runs immediately and turns off the compiler. Semi-colon is called an "**IMMEDIATE**" word because it does NOT compile even if the compiler is turned on but rather it RUNS "immediately".

> Behind the scenes Semi-colon secretly compiles the word EXIT at the end of a definition. EXIT is the equivalent of RETURN in BASIC. Now you can understand why every Forth word is like a BASIC sub-routine.

Compared to BASIC

The BASIC command PRINT you already understand, but you may have never considered that PRINT always starts printing on a new line. The person who designed the BASIC PRINT Command decided, without asking you, that every time you PRINT something it will start on a new line.

CR (Carriage Return)

'CR' means "carriage return" a carryover from the days of mechanical printers. But for Forth it means start on a new line. If you want a new line you do it when you want it with CR.

This brings us to fundamental difference in the Forth way of thinking:

Forth assumes that you know more than the computer does, about what your program needs to do.

This may not seem important but it is**. In Forth you are responsible for everything**

By comparison here is a partial list of the things that TI-BASIC assumes for you and your programs:

1. The language will never need any new commands
2. The language will never need new syntax
3. You always want to type the program in the BASIC editor, line by line.
4. You always want your variables to be floating point numbers (that take 8 bytes each)
5. You will never need to reach directly into the computer's memory and read or write it
6. You will never want different error messages if the program has to stop from some reason
7. All string variable names must end with '$'
8. You never want punctuation characters as variable or sub-program names
9. You will never need any other kind of data except numbers and strings.
10. You always want numbers to be displayed the same way
11. You always want to work with decimal number (base 10)
12. Etc…

So if you want to put some text on a new line you must tell Forth with the 'CR' command. (carriage return) Forth will not assume you want a new line.

DOT QUOTE .”

In BASIC the word PRINT is what computer scientists call ‘overloaded”. It means PRINT has to do more than one thing even though it is just one command.

PRINT has to identify and know how to display:

- numeric variables
- numeric array elements
- literal numbers
- string variables
- string array elements
- string literals

You may be shocked but to a computer each of these things is COMPLETELY different. PRINT has to identify what it was given and then choose the correct internal code to print it. This way of working is one of the reasons that BASIC can be slower than other languages.

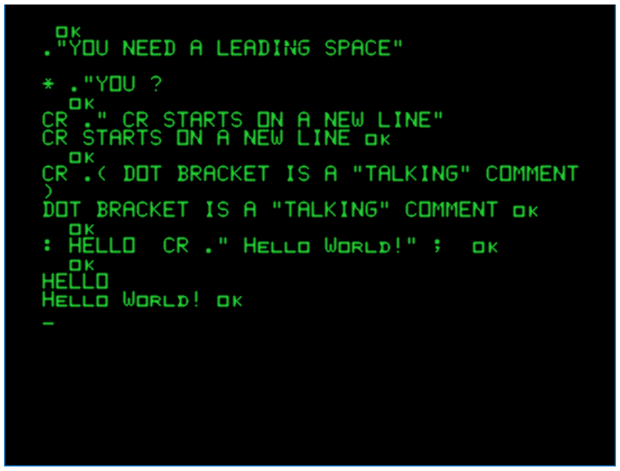
Forth does not work that way. Each word does one thing and usually it is a simple thing.

To print a string of characters we use the WORD .” (pronounced: dot-quote). The Forth interpreter is simple too. Each command must be separated by a space. So .” must have a space between it and the first character of the text that it will print.

Don’t be confused though. .” knows how to read each character that follows and compile them into memory. In CAMEL99 FORTH .” is a “little” bit overloaded to make things easy for the programmer. In the ANS 94 Forth Standard, .” only works while compiling a new definition. In CAMEL99 .” checks the STATE of the system (a variable) and compiles if it should compile and interprets if it should interpret.

Some Dot-quote “Gotchas”

```
: BAD  .”This will abort with an error” ;
     ^^^^^
( In Forth ALL words are separated by a space )

: GOOD .” This will compile perfectly” ;
```



```
 OK
."YOU NEED A LEADING SPACE"

* ."YOU ?
  OK
CR ." CR STARTS ON A NEW LINE"
CR STARTS ON A NEW LINE OK
  OK
CR .( DOT BRACKET IS A "TALKING" COMMENT
)
DOT BRACKET IS A "TALKING" COMMENT OK
  OK
: HELLO  CR ." HELLO WORLD!" ;  OK
  OK
HELLO
HELLO WORLD! OK
_
```

If we want to print some text while in immediate mode (interpreting) we can also use the “talking comment” which is .(and end the text with) This is typically put in source code files to report to the programmer that things are going on while compiling a file.

Examples of how to print text on the screen:

Final Thoughts on Hello World

After you typed our example program in BASIC you started it by typing RUN. In this Forth example we do the same thing in Forth. However Forth did not have a RUN command until we made it … which we did using a COLON DEFINITION.

For clarity FORTH does not need a RUN command. Forth only needs WORDS. If a WORD is in the dictionary Forth will run it. So we could have called our program anything. The inventor of Forth, Chuck Moore, was fond of calling his RUN word "GO". It works just as well in Forth.

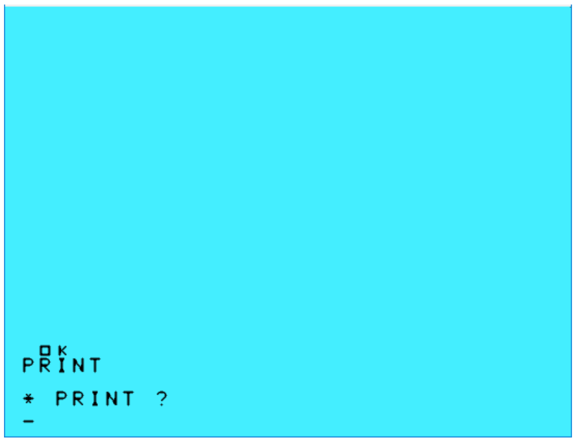
And one more thing…

When TI BASIC is finished running the program it shows you the '>' character.

Forth shows you the 'ok" to tell you everything worked as expected.

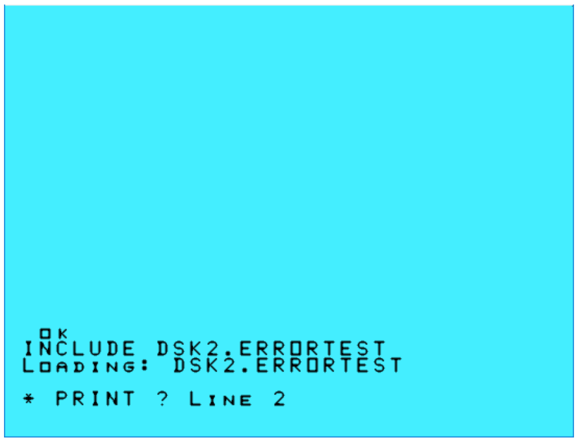
Error messages

If CAMEL99 cannot find a word in the dictionary it tries to convert the text to a number. If it can't convert it to a number it reports a simple error. We have made the error messages a little like TI-BASIC in that they start with an asterisk, then the word that is not found followed by a question mark. And of course they create that annoying HONK sound. It just seemed correct to HONK.



```
OK
PRINT
* PRINT ?
_
```

When you INCLUDE a file and CAMEL99 Forth finds a word it doesn't recognize it shows the same error and also shows you the line number in the file.



```
OK
INCLUDE DSK2.ERRORTEST
LOADING: DSK2.ERRORTEST
* PRINT ? LINE 2
```

Lesson 2: Transition from BASIC to Forth

Look at the program listing below:

```
10 CALL CLEAR
20 PRINT "Hello world!"
30 GOTO 20
```

In this lesson we will begin by adding some language extensions to Forth that make you feel more familiar with this strange Forth world. These extensions are a little like training wheels on a bicycle and eventually you may choose to not use them but they are handy nevertheless.

If they are not already loaded in the system you get the helper words by typing:

```
INCLUDE DSK1.GRAFIX.F <enter>    ( CLEAR, VCHAR, HCHAR and others)
INCLUDE DSK1.STRINGS.F <enter>   ( Strings with BASIC type functions)
INCLUDE DSK1.INPUT.F <enter>     ( $INPUT and #INPUT similar to BASIC)
```

Once the OK prompt comes back on the screen you can type this little program in at the console.

```
: 10>  CLEAR ;
: 20>  " HELLO WORLD" PRINT ;

: RUN   10>  BEGIN  20>  AGAIN ;
```

To a TI-BASIC programmer it looks a little weird. This program is only to help you understand BASIC from Forth's perspective. It should not be taken as a good example of a Forth program.

We have loaded our TI-BASIC helper words (GRAFIX.F and STRINGS.F) into CAMEL99 so we have some familiar word names like BASIC, but they seem to be backwards. Why? That has to do with how Forth uses something called the **Parameter Stack** which we will explain in the next lesson. Also notice we don't have to "CALL" those sub-programs. That's because all Forth words are sub-programs so they are called by Forth by default.

We have used the colon definition that we learned in Lesson 1 to create something that looks like line numbers. PLEASE NOTE: They are NOT line numbers, they are Forth WORDs but they let you see how a line number in BASIC performs the same function as a WORD does in Forth. Line numbers are just an identifier that the computer can use to find code when it needs to run it.

BASIC's line numbers are labels to let the computer find pieces of code

Forth WORDs are labels to let the computer find pieces of code

Let’s review what our new Forth “line-numbers” (words) do:

10>   is obvious. It calls CLEAR, which fills the screen with spaces and puts the cursor on the bottom line.

20>   A new word called ” simply puts literal characters in the Definition that end at the other quote. (Notice the space before the text begins) Literal means the characters are “compiled” into place in memory just as they appear. PRINT is a word from the CAMEL99 strings library. It can take a string made with “ and PRINT it to the screen. (it prints on a new line like BASIC)

RUN - This is where things get more different. We defined a word called RUN. RUN is performing the function of the BASIC interpreter, which is to “RUN” the code in each line number, in order, one at a time. As before Forth does not have a RUN function so we created it.

So first our final definition runs line 10, then it runs line 20 … AGAIN and AGAIN. Notice there is no GOTO.

And by the way….

You noticed that when you typed RUN (cause I know you did) the computer would not stop PRINTing. That’s because you did not program a place to check if the break key was pressed. Remember what we said about you are responsible for everything. Well that’s one of those things.

Press Function QUIT and reload Forth.

Structured Programming

Forth is known as a structured programming language so it does not have GOTO. This may be a very weird thing for people who have only used BASIC.

Structured languages do not let you jump anywhere. They provide you with ways to jump but it is well… structured. To create an infinite loop (goes forever) we use the BEGIN/AGAIN structure. AGAIN is a GOTO that can only jump backwards to BEGIN.   Don’t worry there are ways to jump where ever you need to, but you might have to “structure” your program a little differently than you are used to in BASIC.

I hope it is clear in our RUN word that line 20> is going to go on forever because it is between BEGIN and AGAIN.

Moving Closer to Forth Style

Now please do not think I want you to write Forth code that looks like this with BASIC line numbers.

I simply wanted you to see something more familiar. Now that you know what line numbers really do in BASIC, let’s look at how it could look in Forth if we used more descriptive names instead of line numbers:

```
: CLS   CLEAR ;
: HI!   " HELLO WORLD" PRINT ;
: RUN   CLS   BEGIN   HI! AGAIN ;
```

Trim the Fat

In fact we really don’t need the first line because CLS is just calling the word CLEAR. So we can use CLEAR by itself and the program would become:

```
: HI!  " HELLO WORLD" PRINT ;
: RUN  CLEAR  BEGIN  HI! AGAIN ;
```

Minimal

And if we really wanted to save space we could remove the word "HI!" and put it right in our RUN word so it would look like this:

```
: RUN     CLEAR  BEGIN  " HELLO WORLD" PRINT   AGAIN ;
```

Factoring is the key to good Forth

An important part of programming Forth is "factoring". This means removing common "factors" in a program and giving them a name. In BASIC terms this is like using SUB-ROUTINES as much as possible.

For example, if we wanted to use "HI!" in many places in our program we should keep it as a separate word. That way we could say "Hello World" anytime we wanted to with one command. Using the word "HI!" any time after it's defined only adds 2 bytes to our program!

Did that make your head spin a little? It can when you are used to BASIC. Compared to FORTH, BASIC is like a straight-jacket, forcing you do things in very specific ways with few exceptions. Forth gives you much more freedom, which means you have to think a little more about what you want but the program can do almost anything.

Insider Secret

Now that you have a sense of how Forth works here is how we created the "sub-program" CLEAR, in our BASIC helper word set, in the file GRAFIX.F .

```
: CLEAR      PAGE    0 23 AT-XY ;
```

With lesson 1 and lesson 2 under your belt you can understand what we did.

- PAGE is the STANDARD Forth word to clear the screen, but cursor goes to top left.
- AT-XY positions the cursor to column 0, row 23.

By the way it is best to think of Forth as being OPTION BASE 0 for all graphics coordinates, whereas BASIC is OPTION BASE 1. So the upper corner in Forth is coordinate 0,0 and the bottom right corner is 23,31 .

Parameters Come First

What is the deal with all those backwards parameters? That is really weird.

Lesson 3 will explain why parameters come before the operation in Forth.

Lesson 3: The Parameter Stack

- Start CAMEL99 Forth and type along with this lesson at the console.

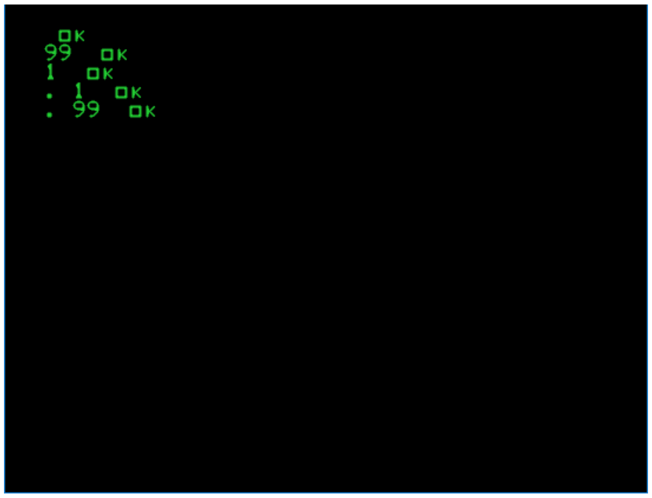
The Forth interpreter exposes you, the programmer, directly to the CPU stack. This is considered impossible or at least dangerous by most computer scientists. In conventional systems the stack is only directly accessed by the CPU or O/S and humans don't touch it directly.

Forth stands that kind of thinking on its head. In fact Forth has two stacks. One stack is for sub-routine returns, called the RETURN STACK. This has the same function as the return stack used by TI BASIC for GOSUB and CALL. It lets the program get back to where it left off after a sub-routine has completed. In Forth everything is a sub-routine so this stack gets used a great deal. We can use the return stack as Forth programmers but more commonly it is the Parameter stack on which we do most things.

What is a computer stack?

One of the easiest ways to understand a stack is to think of it as a cafeteria plate dispenser. These are harder to find in the 21st century but they are a device that that allows a pile of plates to be dispensed one at a time. When you take one plate off the top, the next plate rises up to the top and is available. If you put a plate on the stack the others push down. If you have never seen a plate dispenser we have a photo here for you.

Figure 1 The Victor Plate dispencer is like a computer stack



```
 OK
99   OK
1   OK
. 1   OK
. 99   OK
```

Figure 2 Putting numbers onto the parameter stack

*** At the console type the word DECIMAL**

This makes sure that Forth is working in BASE 10 arithmetic.

Next type 99 and press enter.

Type 1 and press enter.

Type **.** and press enter.

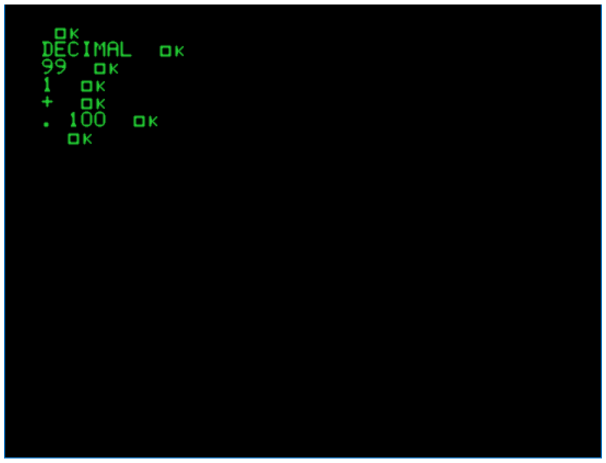
Type . again and press enter.

What happened? Well first you typed 99 and just like a plate in the plate dispenser Forth put the number 99 onto the parameter stack. Then the number 1 went on the stack also. Then you typed "DOT" twice. "DOT" as it is called in Forth takes a number from the PARAMETER Stack and prints it on the output device using the number BASE that is currently set. That is ALL that DOT does. This is true to the Forth philosophy that each word should have one clear, easy to understand function.

But how is the Stack Useful?

Have you ever wished that your BASIC program had a place to put the result of a computation without having to use more named variables. Some kind of a place where you put stuff there and the next subroutine just knew where to get what it needed? Maybe it never occurred to you but that is how to think of the parameter stack.

Adding on the Stack



```
 OK
DECIMAL   OK
99   OK
1   OK
+   OK
. 100   OK
 OK
```

So at the console do the same thing, type 99 <enter> and 1 <enter>

Now this time type '+" <enter>   What happened? Nothing?

Actually something did happen.

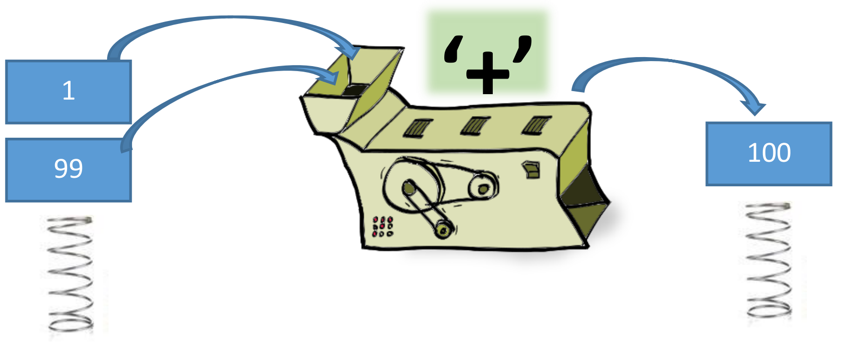
Two numbers were on the stack

"+" is a Forth WORD. A sub-routine.

It added them together and put the answer back on the stack.

"DOT" of course takes a number off the stack and prints it.

FORTH's PLUS (+) is only two assembler instructions on the TI-99 so it is very fast.

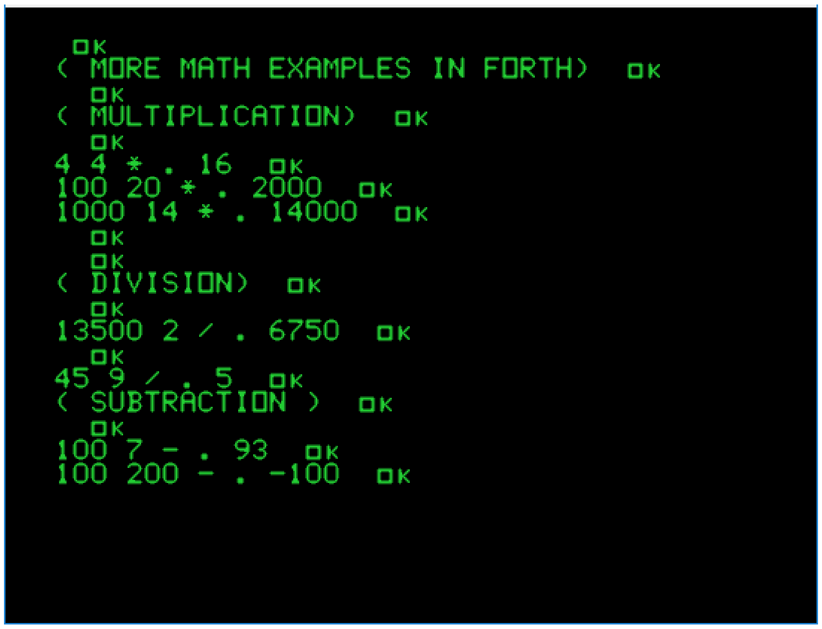


'+' takes 2 numbers, adds then and puts the answer on the stack

'.' Dot is more complicated but uses some clever Forth code to convert a binary number on the stack into a string of ASCII characters and then prints the string on the screen. Unlike in BASIC, you have access to all the WORDS that make "DOT" function. This means you can change the format of numbers look like anything you want. A date, a time of day, money or anything else you need.

A Few More Math Examples

We can do the regular math operations like subtract, multiply and divide. We can also do a few operations that are not typically available in BASIC. Try these ones in this screen on your Forth console.



```
  OK
( MORE MATH EXAMPLES IN FORTH)   OK
  OK
( MULTIPLICATION)  OK
  OK
4 4 * . 16  OK
100 20 * . 2000  OK
1000 14 * . 14000  OK
  OK
  OK
( DIVISION)  OK
  OK
13500 2 / . 6750  OK
  OK
45 9 / . 5  OK
( SUBTRACTION )  OK
  OK
100 7 - . 93  OK
100 200 - . -100  OK
```

A Fundamental Difference between BASIC and Forth

The plus sign, minus sign, division and multiplication signs in BASIC are called an "operators". They are woven tightly into the BASIC interpreter and you cannot change what they do. In Forth these are just WORDs like all the rest of the language so if you want to you are free to make a new '+" word that adds things differently. I would recommend that you use this power wisely. (big grin by the author)

Also TI BASIC only uses floating point numbers. This makes TI BASIC an excellent calculator, but it makes the math operations slower. This is because each floating point number is made of 8 bytes of data. A Forth number in this example is an integer on the TMS9900 CPU just like Assembly language. Since the 9900 is a 16 bit computer each integer can only be in the range of -32767 to 32767. There are ways to add words to work with bigger numbers, but out of the box CAMEL99 is limited to signed numbers in this range or un-signed numbers from 0 to 65,536 just like Assembly language.

Why do we use a Stack?

When Chuck Moore invented Forth he wanted a way to pass parameters to a sub-routine that would not take extra memory space needlessly. He decided the stack was the way to go. Using the stack allows Forth to simply connect words together like Lego blocks or electric circuits. One word takes some inputs from the stack and leaves behind some outputs for the next word to pick up and use and so on…

In fact most other modern languages use a single stack this way to create local variables for sub-routines. The 'C' language, Pascal, Ada and many more create space on the stack every time a sub-routine needs local variables. But the details are hidden from the programmer. In Forth you work with the stack directly.

Using the stack also makes testing a sub-routine very easy because if you do it the Forth way you don't need to create variables for the inputs and outputs. Let's look at a simple example.

For this example we will learn some new words.

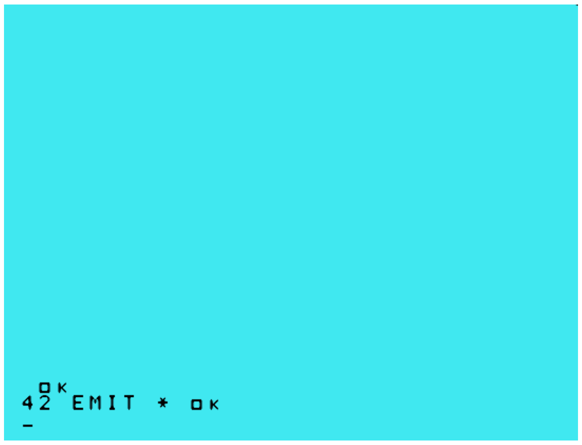
EMIT

The word EMIT is the simplest output word you will ever encounter. It takes one character from the stack and outputs it to the screen.

What use could such a simple thing be you might ask? As with most Forth words it is not a means to an end but a little building block to create something else.

Try this at the CAMEL99 console:

```
CLEAR
42 EMIT
```



```
 OK
42 EMIT * OK
_
```

You should see something like this on your screen. Emit took 42 which is the ASCII value for asterisk and wrote it to the screen.

Let's compare what happens if we did the equivalent in BASIC. BASIC does not give us such a primitive little word.

To do this in BASIC we would type:

```
PRINT CHR$(42)
```

CHR$() in BASIC takes 42 and creates a string in the VIDEO RAM with a string length of 1 character followed by the number 42. Then it runs PRINT to write the ASCII string "'*" on the screen.

In comparison EMIT takes the number 42 from the CPU memory (where the stack resides) and writes it directly to the Video RAM at the current cursor position and then advances the cursor position variables. So EMIT takes much less time to do the same as BASIC

For fun try any other numbers with EMIT and see what you get.

Now type this:

```
: STAR  42 EMIT ; <enter>
STAR <enter>
```

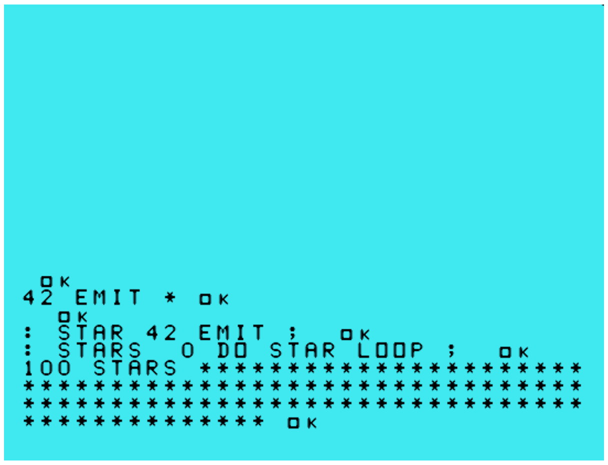
And you should see a "star" on your screen. You created a new word in the Forth dictionary!

What if we wanted many stars but we never knew how many we might need? Let me introduce you to the DO LOOP construct.

Lesson 4: The DO LOOP

In BASIC you use the FOR NEXT loop to do things a specified number of times. The Forth equivalent is the DO LOOP. Here is the solution to our previous question.

```
: STARS         0  DO  STAR LOOP ;
```



```
OK
42 EMIT * OK
 OK
: STAR 42 EMIT ; OK
: STARS 0 DO STAR LOOP ;  OK
100 STARS ***********************
*******************************
*******************************
**************  OK
```

Note: You must have STAR defined in the system before you can create STARS

To test our new word at the console you only have to do this:

```
100 STARS
```

And you get something like this on the screen.

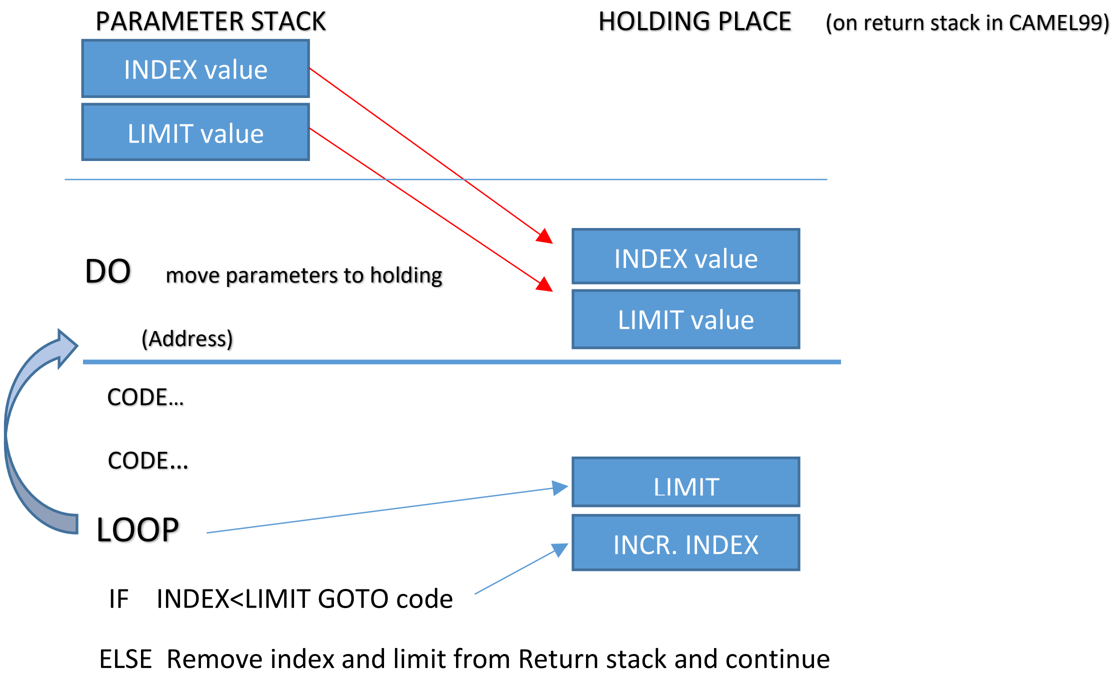
Notice we did not need a variable because we used the stack to pass the number 100 to STARS

Not exactly the same as FOR/NEXT

DO LOOP is a little different than BASIC's FOR NEXT loops.

The word DO accepts 2 numbers from the stack. The first number is called the LIMIT and the number on the top of the stack is called the INDEX. Yes it's seems backwards but there is a technical reason for that so we have to accept it for now.

DO simply takes both numbers and puts them in a *holding place. In CAMEL99 Forth that's on the return stack. For the curious DO also makes a note of the address it is sitting at, so that LOOP can jump back to that address later.



Then the code after DO runs as you would expect until the word LOOP is encountered. 'LOOP' adds 1 to the INDEX value in the *holding place and compares it to the LIMIT value in the holding place.

If the two numbers are different, LOOP jumps back to where the code is after DO. If the numbers are the same, LOOP jumps to code that follows LOOP. In our example 'STARS' all that was there was the semi-colon so we returned to the Forth interpreter.

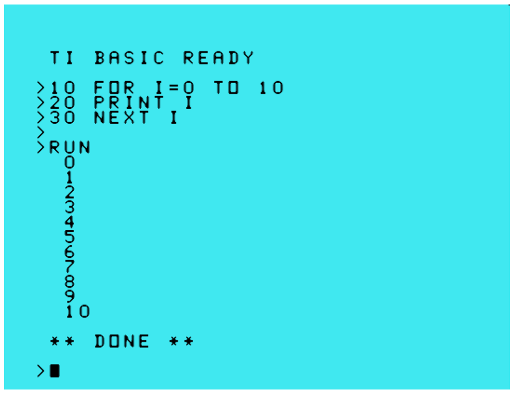
**Not every Forth system uses the return stack so that is why we call it a holding place.*

The IMPORTANT Difference from FOR NEXT

Consider the code:

```
10 FOR I=0 TO 10
20 PRINT I
30 NEXT I
```

If we run this in TI-BASIC we see this



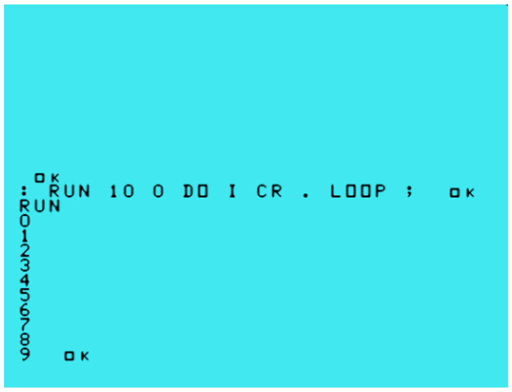
```
  TI BASIC READY
>10 FOR I=0 TO 10
>20 PRINT I
>30 NEXT I
>
>RUN
   0
   1
   2
   3
   4
   5
   6
   7
   8
   9
   10

  ** DONE **

>
```

The Forth equivalent is:

```
:  RUN     10 0 DO  CR I .  LOOP ;
```

(The word 'I' puts the loop's INDEX value on the stack but you know what all the other words mean by now)

And we run this test we see this



```
 OK
: RUN 10 0 DO I CR . LOOP ;   OK
RUN
0
1
2
3
4
5
6
7
8
9   OK
```

As you can now understand if you make the loop parameters 10 to 0 , Forth will count up 0,1,2,3,4,5,6,7,8,9 but when the INDEX=10, Forth will exit the loop because INDEX=LIMIT at the point. This is by design because Forth, like Assembler, works with addresses that start at a base Address plus X, where X is starts at "0".

No Safety NET

The DO LOOP in Forth as with everything is stripped down. There is no protection from mistakes. If you use zero as a parameter for STARS the loop will go on forever. You can try it if you don't believe me but you will have to reset your TI-99.

Modern Forth has added the word '?DO' to prevent that accidental mistake. ?DO compares the 2 parameters on the stack before it starts the loop. If the parameters are equal, it skips the looping completely. If we wanted to be safe from people putting in a zero we could write:

```
: STARS    0 ?DO  STAR LOOP ;
```

Try it as well and you see that typing “0 STARS” gives you no stars on the screen.

What’s the point of this stripping down to a bare minimum?

One word. **SPEED!**

And if we compare equivalent empty loop programs which removes screen scrolling we get:

```
10 FOR I = 0 TO 10000
20 NEXT I


: RUN   10001 0 DO LOOP ;
```

BASIC runs in 29 seconds and CAMEL99 runs in .8 seconds or 36 times faster.

Forth’s DO LOOP is 36X faster than FOR NEXT in TI-BASIC

And one more thing…

The other thing I hope you can see is that you now understand how DO LOOP actually works inside the Forth Virtual Machine. This never happens with BASIC. It is a black box.

That’s a BIG Difference

In BASIC even though you may know how to use the language you never see the details of how BASIC does what it does.

In Forth much of the system is written in Forth. I know that sounds weird but it is true. So you can actually see “under the hood” (under the bonnet in the UK) and see the code that does all this stuff.

You don’t <u>**need**</u> to know about it, but you can learn about it and understand it if you want to.

That’s a BIG DIFFERENCE with Forth.

COLOR in CAMEL99

Although the hardware supports color numbers from 0 to 15, we made CAMEL99 use the same color values as TI BASIC just to keep things simple. These are set in the code in GRAFIX.F

1	Transparent	9	Medium Red
2	Black	10	Light Red
3	Medium Green	11	Dark Yellow
4	Light Green	12	Light Yellow
5	Dark Blue	13	Dark Red
6	Light Blue	14	Magenta
7	Dark Red	15	Gray
8	Cyan	16	White

The COLOR WORD is expanded compared to TI-BASIC. The hardware supports 255 characters. BASIC only gives you access to ASCII characters 32.. 159 (127 chars in total). A sided effect of this is that the Character code numbers are different in Forth. *Please* look over the new Set numbers vs BASIC

Char. Code	Forth Set#	Basic Set#
0-7	0	N/A
8-15	1	N/A
16-23	2	N/A
24-31	3	N/A
32-39	4	1
40-47	5	2
48-55	6	3
56-63	7	4
64-71	8	5
72-79	9	6
80-87	10	7
88-95	11	8
96-103	12	9
104-111	13	10
112-119	14	11
120-127	15	12
128-135	16	13
136-143	17	14
144-151	18	15
152-159	19	16
160-167	20	N/A
168-175	21	N/A
176-183	22	N/A
184-191	23	N/A
192-199	24	N/A
200-207	25	N/A
208-215	27	N/A
216-223	28	N/A
224-231	29	N/A
232-239	30	N/A
240-247	31	N/A
248-255	32	N/A

COLOR Command

This works just like BASIC except you can only set 1 character set at a time.

```
2  2 8 COLOR   \ Character set 2,  BLACK foreground, CYAN background
```

COLORS, a New Command

COLORS is a word that lets you set the color values of more than one character set. Forth requires the correct number of parameters on the stack for each word so we cannot use COLOR to do more than one character set. COLORS will set the color values for all character sets between the first parameter and the second parameter.

Example:

```
4 8 4 2 COLORS \ sets colors for character sets #4,5,6,7,8
               \ to 4 foreground, 2 background
```

SCREEN

Just like BASIC.

```
12 SCREEN  \ screen color set to Light yellow
```

TI BASIC Strings in Forth (STRINGS.F)

```
LEN       ( $ -- n )            return the length of $
SEG$      ( $ n1 n2 -- top$)    create new string: start=n1,size=n2
STR$      ( n -- top$)          create new string of no. 'n'
VAL$      ( $ - # )             convert $ to a number on data stack
CHR$      ( ascii# -- top$ )    create new string from ascii#, len=1
ASC       ( $ -- char)          return ascii value of 1st character
 &        ( $1 $2 -- top$)      concatenate $1 and $2
POS$      ( $1 $2 -- $1 $2 n )  find position of $2 in $1
```

String Comparison

```
COMPARE$  ( $1 $2 -- flag)   flag meaning: 0 $1=$2,  -1  $1<$2  1 $1>$2
=$        ( $1 $1 -- flag)
<>$       ( $1 $1 -- flag)
>$        ( $1 $2 -- flag)
<$        ( $1 $2 -- flag)
```

IMMEDIATE mode string assignment

```
: ="      ( $addr -- <text> )
: =""     ( $addr -- )
```

Moving Strings

```
COPY$     ( $1 $2 -- )  move $1 to $2.  Does not cleanup string stack
PUT       ( $1 $2 -- )  move $1 to $2.  Cleans up string stack
```

String Output

```
PRINT$    ( $ -- )  prints with No new line, Does not cleanup string stack
PRINT     ( $ -- )  prints with new line. Cleans up string stack
```

Create a String Literal

```
: "       ( -- <text>)  Works in immediate mode and compiling mode
```

Allocate String Space

```
DIM   ( n -- )   creates a string variable in the dictionary of size n
```

Creating String Variables

```
DECIMAL
 32 DIM A$    \ called A$ to mimic BASIC. You can use any name
100 DIM B$

\ You can also used the colon compiler to create string constants.
\ They cannot be changed.(You could try to changed them but it will crash)

: FIXED$  " This string is compiled into memory and cannot be changed" ;
```

Using CAMEL99 String Words

You can use the Forth string words like you would use string functions in TI BASIC but they make your head hurt the first few times you try them. This is because they are like other Forth words the input argument go first and then the string function follows.

Example String Code:

```
80 DIM$ A$   \ like number variables you must create these first
80 DIM$ B$
80 DIM$ C$

A$ =” A$ is being given text when the code is loading”

: TEST ( -- ) “ B$ is being loaded with PUT in a definition” B$ PUT ;

TEST  \ this will cause the text to be PUT into B$

A$ PRINT
B$ PRINT

A$ B$ &  PRINT     \ this prints A$&B&

B$ 4 10 SEG$ PRINT \ very similar to BASIC

\ translate this BASIC statement to CAMEL99 Forth:

\ C$ = SEG$(A$4,10) & SEG$(B$,12,5)

A$ 4 10 SEG$  B$ 12 5 SEG$ &  C$ PUT
```

Example Programs in BASIC and Forth

Random Color Dots

From the TI BASIC Reference Manual.

```
100 REM  Random Color Dots
110 RANDOMIZE
120 CALL CLEAR
130 FOR C=2 TO 16
140 CALL COLOR(C,C,C)
150 NEXT C
160 N=INT(24*RND+1)
170 Y=110*(2^(1/12))^N
180 CHAR=INT(120*RND)*40
190 ROW=INT(23*RND)+1
200 COL=INT(31*RND)+1
210 CALL SOUND(-500,Y,2)
220 CALL HCHAR(ROW,COL,CHAR)
230 GOTO 160
```

Here is an equivalent program in CAMEL99 Forth with training wheels included and extra comments for explanation. (Comments don't go into your program in Forth)

```
\ Random Color Dots
INCLUDE DSK1.RND.F       \ here we add the features we need
INCLUDE DSK1.GRAFIX.F
INCLUDE DSK1.SOUND.F

\ rather than use variables we can make words with the same names
\ that calculate the numbers we need and leave them on the stack

\ *SIMPLER version: give a frequency between 110 and 1110 Hz
: Y   ( -- n ) 1000 RND 110 + ;  \ does not calc. notes.

: CHR ( -- n )   80 RND 32 + ;
: ROW ( -- n )   24 RND ;
: COL ( -- n )   32 RND ;

\ Sound is factored into pieces in CAMEL99 (HZ DB MS MUTE)
\ but we can create a similar word to BASIC's SOUND
\ for 1 voice at a time.

: SOUND  ( delay vol freq -- ) GEN1 HZ  DB  MS MUTE ;

: RUN ( -- )
      CLEAR
      14 2 ?DO
         I I I COLOR
      LOOP
      BEGIN
         GEN1 80 -2 Y SOUND   \ "GEN1" select generator 1
         COL ROW CHR 1 HCHAR
         ?TERMINAL
      UNTIL ;
```

Guess the Number in Forth

```
\ Demonstrate Forth style. Create words to make the program.

DECIMAL
VARIABLE TRIES
VARIABLE GUESS

: ASK   ( -- )
         CR CR
         TRIES @ 0=
         IF    ." Guess a number between 1 and 10: "
         ELSE  ." Try Again: "
         THEN ;

DECIMAL
: RANGE  ( n -- ? )
         1 11 WITHIN 0=
         IF CR ." That's not valid so... " THEN ;

: GET-GUESS ( -- ) GUESS #INPUT  ;

: REPLY  ( the# guess -- n)
         GUESS @              \ fetch GUESS variable and DUP
         DUP RANGE            \ make a DUP & check if the guess is in range
         2DUP <>              \ compare the# and the guess for not equal
         IF CR HONK ." No, it's not " DUP .
         THEN ;

: .TRIES ( -- )
         TRIES @ DUP .
         1 = IF ." try!" ELSE ." tries!" THEN ;

: FINISH ( -- )
          CR
          CR BEEP 50 MS  BEEP ." Yes it was " .
          CR ." You got it in " .TRIES
          CR ;

: Y/N?   ( -- flag) \ this is VERY different than BASIC
          KEY [CHAR] Y =    \ wait for a key, compare to "Y"
          IF   FALSE        \ if it is 'Y' put FALSE on stack
          ELSE TRUE         \ any other key, put TRUE on the stack
          THEN ;            \ then end the sub-routine

: PLAYAGAIN? ( -- flag)
          CR ." Want to play again? (Y/N)"  Y/N?    ;

: RUN ( -- )
       BEGIN
          CLEAR
          0 TRIES !
          10 RND 1+ ( -- rnd#) \ no variable, just leave on stack
          BEGIN
             ASK
             GET-GUESS
             REPLY
             1 TRIES +!
          OVER = UNTIL
          FINISH
          PLAYAGAIN?
       UNTIL
       CR ." OK, thanks for playing!" ;
```

GRAPHICS Example “Denile”

```
10 CALL CLEAR
20 FOR L=65 TO 70
30 READ Q$
40 CALL CHAR(L,Q$)
50 NEXT L
60 DATA
010207091F247F92,8040E090F824FE49,FF92FF24FF92FF49,AA55448920024801,000217357CFC44AA,0
008081C2A081414
70 CALL CHAR(104,"0083C7AEFBEFBDF7")
80 CALL CHAR(105,"00078F5DF7DF7BEF")
90 CALL CHAR(106,"000E1FBAEFBFF6DF")
100 CALL CHAR(107,"001C3E75DF7FEDBF")
110 CALL CHAR(108,"00387CEABFFEDB7F")
120 CALL CHAR(109,"0070F8D57FFDB7FE")
130 CALL CHAR(110,"00E0F1ABFEFB6FFD")
140 CALL CHAR(111,"00C1E357FDF7DEFB")
150 CALL COLOR(10,6,5)
160 X=13
170 C=1
180 PRINT TAB(X+1);"AB"
190 C$=C$&"CC"
200 B$="A"&C$&"B"
210 PRINT TAB(X);B$
220 C=C+1
230 X=X-1
240 IF C=13 THEN 250 ELSE 190
250 CALL HCHAR(24,1,68,32)
260 CALL HCHAR(23,1,69)
270 CALL HCHAR(23,2,70)
280 PRINT
290 PRINT
295 PRINT
296 CALL HCHAR(24,1,68,32)
300 T=104
310 Y=106
320 T=T+1
330 IF T>111 THEN 340 ELSE 350
340 T=104
350 Y=Y+2
360 IF Y>111 THEN 370 ELSE 380
370 Y=104
380 CALL HCHAR(22,1,T,32)
390 CALL HCHAR(23,1,Y,32)
400 GOTO 320
```

Denile in Forth

```
\ LITERAL TRANSLATION OF DENILE.BAS using variables
\ Orignal program by RETROSPECT, Atariage.com

INCLUDE DSK1.GRAFIX.F
INCLUDE DSK1.STRINGS.F
INCLUDE DSK1.CHARSET.F

\ 60 DATA 010207091F247F92,8040E090F824FE49,FF92FF24FF92FF49,AA55448920024801,
\         000217357CFC44AA,0008081C2A081414

\ Character DATA patterns are named in CAMEL99 Forth
HEX
0102 0709 1F24 7F92  PATTERN: CHAR65    \ these should be better names
8040 E090 F824 FE49  PATTERN: CHAR66
FF92 FF24 FF92 FF49  PATTERN: CHAR67
AA55 4489 2002 4801  PATTERN: CHAR68
0002 1735 7CFC 44AA  PATTERN: Camel      \ these are good names
0008 081C 2A08 1414  PATTERN: LittleMan
0083 C7AE FBEF BDF7  PATTERN: WATER104
0007 8F5D F7DF 7BEF  PATTERN: WATER105
000E 1FBA EFBF F6DF  PATTERN: WATER106
001C 3E75 DF7F EDBF  PATTERN: WATER107
0038 7CEA BFFE DB7F  PATTERN: WATER108
0070 F8D5 7FFD B7FE  PATTERN: WATER109
00E0 F1AB FEFB 6FFD  PATTERN: WATER110
00C1 E357 FDF7 DEFB  PATTERN: WATER111

DECIMAL
: CHANGE-CHARS ( -- )
\ CHARDEF takes a defined pattern and the ascii number
       CHAR65     65 CHARDEF \ 20 FOR L=65 TO 70
       CHAR66     66 CHARDEF \ 30 READ Q$
       CHAR67     67 CHARDEF \ 40 CALL CHAR(L,Q$)
       CHAR68     68 CHARDEF \ ...
       Camel      69 CHARDEF \ ...
       LittleMan  70 CHARDEF \ 50 NEXT L
       WATER104  104 CHARDEF \ 70 CALL CHAR(104,"0083C7AEFBEFBDF7")
       WATER105  105 CHARDEF \ 80 CALL CHAR(105,"00078F5DF7DF7BEF")
       WATER106  106 CHARDEF \ 90 CALL CHAR(106,"000E1FBAEFBFF6DF")
       WATER107  107 CHARDEF \ 100 CALL CHAR(107,"001C3E75DF7FEDBF")
       WATER108  108 CHARDEF \ 110 CALL CHAR(108,"00387CEABFFEDB7F")
       WATER109  109 CHARDEF \ 120 CALL CHAR(109,"0070F8D57FFDB7FE")
       WATER110  110 CHARDEF \ 130 CALL CHAR(110,"00E0F1ABFEFB6FFD")
       WATER111  111 CHARDEF \ 140 CALL CHAR(111,"00C1E357FDF7DEFB")
;

\ ALL variables and strings must be defined first
VARIABLE X   VARIABLE C  VARIABLE T  VARIABLE Y

32 DIM A$    \ Not standard Forth. Language extension
32 DIM B$    \ BTW Strings can have any name. Imagine that...
32 DIM C$

: TAB ( n -- ) 0 ?DO SPACE LOOP ; \ we don't have a TAB word so make one

: PYRAMID
   A$ =""  B$ =""  C$ =""      \ clear these strings
  14 X !                       \ 160 X=13
   1 C !                       \ 170 C=1
   CR X @ 1+ TAB ." AB"        \ 180 PRINT TAB(X+1);"AB"
 BEGIN                         \ " needs a space, '&' is RPN :-)
      C$ " CC" &        C$ PUT  \ 190 C$=C$&"CC"
     " A" C$ &  " B" & B$ PUT  \ 200 B$="A"&C$&"B"
      CR   X @ TAB  B$ PRINT   \ 210 PRINT TAB(X);B$
      1 C +!                   \ 220 C=C+1
     -1 X +!                   \ 230 X=X-1
      C @ 14 =                 \ 240 IF C=13 THEN 250 ELSE 190
 UNTIL
    0 23 68 32 HCHAR           \ 250 CALL HCHAR(24,1,68,32)
    0 22 69 1  HCHAR           \ 260 CALL HCHAR(23,1,69)
    1 22 70 1  HCHAR           \ 270 CALL HCHAR(23,2,70)
    CR                         \ 280 PRINT
    CR                         \ 290 PRINT
    CR                         \ 295 PRINT
    0 23 68 32 HCHAR           \ 296 CALL HCHAR(24,1,68,32)
;

: RIVER    \ flow the river loop
    104 T !                    \ 300 T=104
    106 Y !                    \ 310 Y=106
 BEGIN
    1 T +!                     \ 320 T=T+1
    T @ 111 >                  \ 330 IF T>111 THEN 340 ELSE 350
    IF  104 T ! THEN           \ 340 T=104
    2 Y +!                     \ 350 Y=Y+2
    Y @ 111 >                  \ 360 IF Y>111 THEN 370 ELSE 380
    IF  104 Y ! THEN           \ 370 Y=104
    0 21 T @ 32 HCHAR          \ 380 CALL HCHAR(22,1,T,32)
    0 22 Y @ 32 HCHAR          \ 390 CALL HCHAR(23,1,Y,32)
    100 MS
    KEY?
 UNTIL                         \ 400 GOTO 320
;

: RUN   \ Forth doesn't have RUN so we make one
   CLEAR                  \ 10 CALL CLEAR
   4 SCREEN               \ BASIC does when running.
   CHANGE-CHARS           \ line 20 to 140
   13 6 5 COLOR           \ 150 CALL COLOR[10,6,5]
                           ( Forth has different character sets)

   PYRAMID                \ 500 GOSUB PYRAMID
   RIVER                  \ 600 GOSUB RIVERFLOW

   8 SCREEN               \ BASIC does this automatically
   CHARSET                \ Forth must be told.
;
```

APPENDIX

Background Information

Behind the House

How BASIC Sees a Program

A BASIC programmer never needs to think about this but if you wanted to create a BASIC interpreter you would need to think about it for an overview of how it would work:

```
TOP:  Wait for input text and the <enter> key
      Does text start with a line number?
      YES:  Start the line editor. Accept text until <enter>
            And put it into the program line.

      No:   Lookup the command, Execute the command,
            Is it a RUN command?
                  YES: GOTO Dorun
                  NO:  Execute the Command
GOTO TOP

DORUN:
Find the lowest line number
While there are more line numbers:
      Interpret the code in the line
      GOTO the next line number
IF there are no more line numbers THEN STOP
GOTO TOP
```

How Forth sees a program

```
BEGIN:
  ACCEPT: input text until <enter> key

  WHILE there are words in the string
   PARSE: first space delimited string in text
       Is it in the dictionary of WORDS?
    YES:   ARE WE COMPILING?
           YES: compile the command
            NO: EXECUTE the command

    NO:  IS IT A NUMBER?
         YES: Are we compiling?
  Yes: compile the number a as literal
   NO:  Put the number onto the stack

   NO: Don't know what this is. Print error message

AGAIN: (goto begin)
```

That's it for Forth. Everything is WORD or a number. If the word cannot be found, it tries to convert to a number. If that fails we abort and restart the interpreter loop.

Essential Elements of Forth

The Forth language was create by Charles Moore and Chuck as his friends call him, and he is something of a radical genius. The programs he was creating were controlling real world hardware. He needed to be close to the silicon to get the telescopes, fabric factories and many other systems working efficiently

BUT he also needed to be an efficient programmer. He found that Assembler programming gave the control he needed but took way too long, while conventional compiled languages of the day like Fortran, forced him away from the hardware which meant he was fighting to get back the control he needed for the programs he had to make.

Like BASIC, Forth can hide a lot of dirty details about the TI-99 from us, but unlike BASIC, we can drop down below the hiding layer anytime we need to and even program in Assembler if we need maximum speed. Chuck created a computer vision in software that suited his needs. Here is the list of things Chuck needed in a computer:

1. Memory (RAM)
2. Disk storage (originally Forth did not use files, just raw disk blocks)
3. A Central Processing Unit (CPU) to do calculations
4. A parameter stack (to hold parameters, what else)
5. A return stack so that he could call a sub-routine and return back (GOSUB)

That is all Chuck thought a computer needed so he built what we now call a "virtual machine". That is just a program that acts like a computer of your design. That's how Forth was born.

The Forth Virtual Machine

A virtual machine is a computer that is created in software. You may be familiar with the concept of a virtual machine if you have any experience with Java. The Java virtual machine is a single stack computer architecture but it is a really a program. It allows the Java language to be portable to many different hardware platforms, because all you need to do is re-write the Java VM for the new hardware and Java programs can run on it.

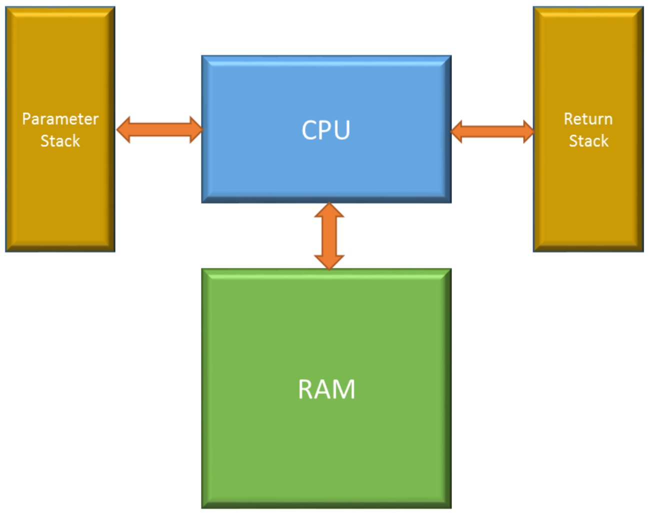
Forth is much older than Java but performs a similar function. In the writings about Chuck Moore, it is told that when IBM created the 360 mainframe, they struggled to get an operating system working built for it. (See the Mythical Man Month, by Fred Brooks (ISBN 0-201-00650-2) to see how many years it took them). Chuck Moore had access to an IBM 360/50 mainframe computer and ported his Multi-tasking Forth system to it in less than a week and had it doing things the IBM engineers did not know it was capable of doing.

The Forth VM is unique in that it uses two stacks. In a conventional machine, virtual or otherwise, the stack is used to keep return addresses and also as temporary memory, typically for local variables in a sub-routine. Moore's innovation was to separate those two functions. A "parameter" stack keeps all the data that is being acted upon and a separate "return stack" is used to keep track of sub-routine returns.

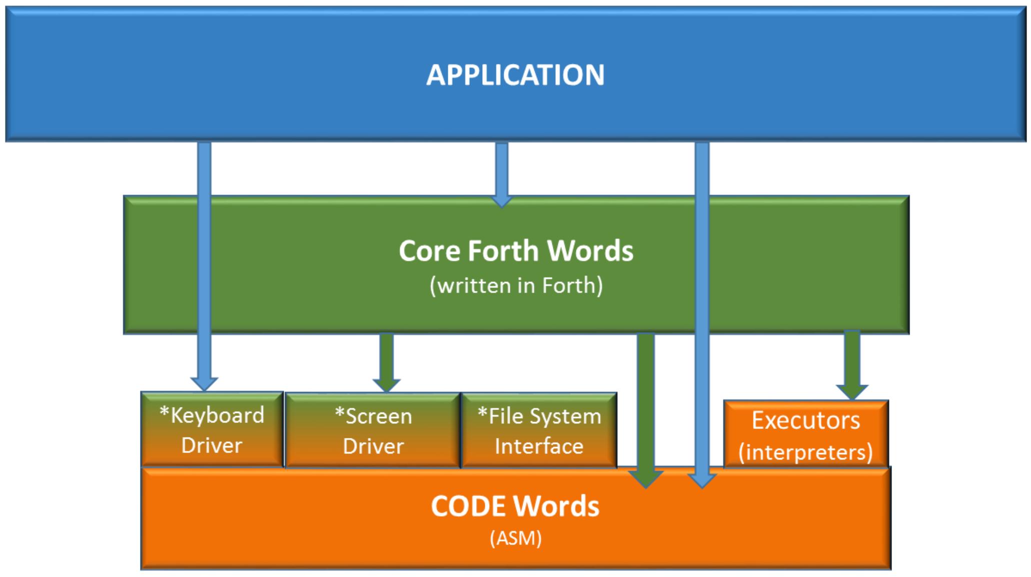
In typical Forth style however you are free to use either stack as your imagination sees fit. For example if you need temporary storage in the middle of a Forth word, you are free to push data onto the return stack as long as you clean it up before you end the word which is when it has to return from whence it came!

Forth Virtual Machine from Different Perspectives

Hardware View



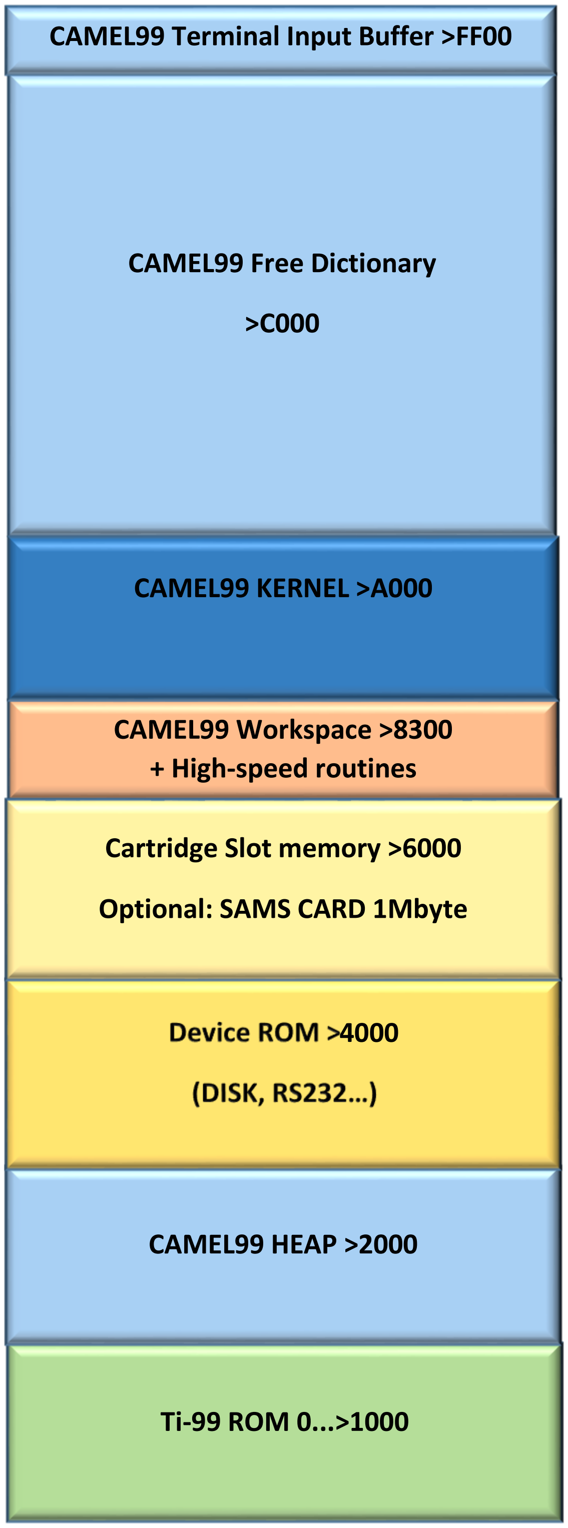
Software Layers



* Written in ASM or Forth or both

- Forth allows your application to call any routine that has a name in the Dictionary.
- Large applications typically create an "Application specific language" (ASL) to abstract the problem to a higher level. That is the preferred method to program in Forth in the author's opinion
- Executors are special code routines that perform operations for each type of Forth word.

CAMEL99 Memory Map



Program Development in Forth

For BASIC programmers reading about Forth for the first time it can be quite confusing and complicated. On the other hand the Forth process seems to have important parts missing for people used to the C programming language.

For those used to TI-BASIC there might seem like too many moving pieces. BASIC is an “all-in-one” system. You never leave BASIC so it seems very simple.

TI BASIC TOOL CHAIN

1.	TI BASIC Console	Interpreter for BASIC commands and line editor

BASIC Process

1. Type program into BASIC console
2. RUN program
3. LIST program and edit errors
4. Go back to 1 until complete
5. Save program to disk

Usage: load the file when you need it and type ‘RUN’

The C language on the other hand is more complicated.

Traditional C Tool Chain

1.	Editor	for creating source code text files in the ‘C’ language
2.	Preprocessor	for creating constants and macros from the C source code
3.	Compiler	for translating the ‘C’ source code into Assembler
4.	Assembler	for translating the Assembler files to object code
5.	Linker	for creating a finished runnable programs from object files
6.	Debugger	for inspecting the internal operation of a running program

C process

1. Create the ‘C’ source file for your program with the editor
2. Save to disk
3. Run the compiler on the C file which also runs the pre-processor creating ASM file
4. Run the assembler on the ASM file creating object file
5. Run the linker on the object file(s) creating loadable/executable file
6. Run the executable file in the Debugger to search for errors
7. Go back to 1 until complete

Usage: Run the executable program when needed

Forth is somewhere between BASIC AND ‘C’.

Forth Tool Chain

1.	Forth Console	Interpreter and Compiler
2.	Editor	For creating source code text files
3.	Assembler	**Optionally** loaded into the Forth system to allow writing Assembler code

Forth Process

1. Create the source code for a small part of the program with the EDITOR
   a. Save the file to disk
2. Load the disk file into Forth system
3. Test in the Forth interpreter
   a. Run each WORD (sub-routine) individually to test for correct behaviour
   b. examine variables and memory as needed
4. Go back to 1 and include more code **until** everything works

Usage:

1. Load the source file and run it by typing the name of the WORD you want to run.

Memory Management in Forth

Originally published by the Author in Atariage.com, Mon Oct 9, 2017

The Forth language is commonly used for the same type of programs that people might choose to use Assembler. Typically they are embedded programs for electronic devices, measurement instruments, drum machines, satellites and even electric toothbrushes. Forth works close to the hardware but gives you ways to make your own simple language so in short order, you are using your own high level functions to get the job done. This can make it faster to get a program ready and out the door in Forth versus doing it all in Assembler.

This tutorial illustrates how Forth manages memory, starting with nothing more than the raw memory addresses, just like you would see in Assembler programming. The difference is that with a few quick definitions, you create a simple memory management system. Using that memory management system you can build named variables, constants buffers and arrays.

To begin we have to assume we already have a Forth compiler somewhere that lets us add new routines, or as Forth calls them WORDs to the Forth Dictionary. The dictionary is actually just the memory space that we will be managing in this demonstration.

The Forth compiler is nothing more than the ':' and ';' WORDS. To compile a new word that does nothing in Forth you would type:

```
: MY_NEW_WORD   ;
```

This would create a word in the Forth dictionary but since there is no other code after the name it does nothing.

We also need our compiler to have the Forth word 'VARIABLE'.   With these things in place we can create our simple memory management system.

We will create an empty memory location that will hold the next available memory address that we can use. In Forth this is called the dictionary pointer or DP for short we declare it like this:

```
VARIABLE DP         \ holds the address of the next free memory location
```

Next we will create a function that returns the address that is held in DP. Forth calls that function 'HERE' as in "Where is the next available memory location?" Forth says "HERE".

HERE uses the 'FETCH' operator which is the ampersand, '@' in Forth.

```
: HERE  ( -- addr)   DP @ ;    \ fetch the address in DP
```

Another thing that we will need to do is move the "dictionary pointer" by changing the value in DP so Forth creates a function that takes a number from the stack and adds it to DP using the function '+!" (Pronounced "plus-store") Plus-store is very much like '+=' for those familiar with 'C'.

Forth calls this DP altering function 'ALLOT' and it is defined like this:

```
: ALLOT ( n --)    DP +!  ;  \ add n to value in variable DP.
\ in other words allocate dictionary space (Pretty simple huh)
```

So with these three definitions we have the beginnings of a simple memory manager. We can now say things in our program like:

```
HEX 2000 HERE 20 CMOVE     \ move $20 bytes from HEX 2000 to HERE
```

\ Now move DP forward to "allocate" the space we just wrote to:

```
  20 ALLOT
```

 By using "20 ALLOT". HERE is moved past our little 20 byte space so we won't tromp on it later. So we have allocated 20 bytes of memory for our own use. To make it really practical we should have recorded the address of HERE somewhere because HERE is now pointing to a new address.

Getting Fancy

We can combine HERE and ALLOT and the STORE word '!' in Forth to make an operator that 'compiles" a number into memory. Forth uses the comma ',' for this function and the definition is simple now.

```
: ,    ( n -- )   HERE !      \ store n at HERE,
2 ALLOT ;   \ allocate 2 bytes (for a 16 bit computer)
```

To use the comma we simply type:

```
 99 , 100 , 101 , 4096 ,
```

And the numbers go into memory like magic!

And of course we have a similar word that is used for bytes or characters called 'C,'. (pronounced "c-comma") It works the same way a comma.

```
: C,   ( c --)    HERE C!  1 ALLOT ;
```

Getting Creative

There is a Forth word called CREATE that lets us add a new word to the dictionary. Words made with 'CREATE' simply return the dictionary memory address after the name. So with our new system making a variable called 'X' is as simple as:

```
CREATE X   2 ALLOT
```

In fact using the colon compiler we can take it to a higher level still:

```
: VARIABLE   CREATE    0 , ; \ create a name and compile 0 in memory
```

Notice how we used the comma to automate the memory allocation of one unit of memory and initialize it to zero. Now our programs can say:

```
VARIABLE X
VARIABLE Y
VARIABLE Z
```

And if we type:

```
CREATE ABUFFER      50 ALLOT
```

We have created a named memory space that we can use to hold data. Invoking the name ‘ABUFFER’ in our program will give us the beginning address of that buffer.

But why stop there? Use the compiler to make it better:

```
: BUFFER:      CREATE     ALLOT  ;
```

Now it’s all done with one word!

```
50 BUFFER: ABUFFER
```

We could also ‘CREATE’ an array of numbers in memory like this:

```
CREATE MYNUMS   0 , 11 , 22 , 33 , 44 , 55 , 66 , 77 , 88 , 99 ,
```

There are fancier ways to access this array but for this tutorial we will keep it simple. To get at these numbers in the array, we simply need to compute the address of the number we want. Since each number has been given 2 bytes of memory or 1 CELL as Forth calls it, the math is easy.

Given an index number, we multiply the index by the memory size, in bytes, of the CPU and add the result to the base address.

Let’s do it with the colon compiler:

```
: ]MYNUMS   ( index -- addr) CELLS  MYNUMS + ;
```

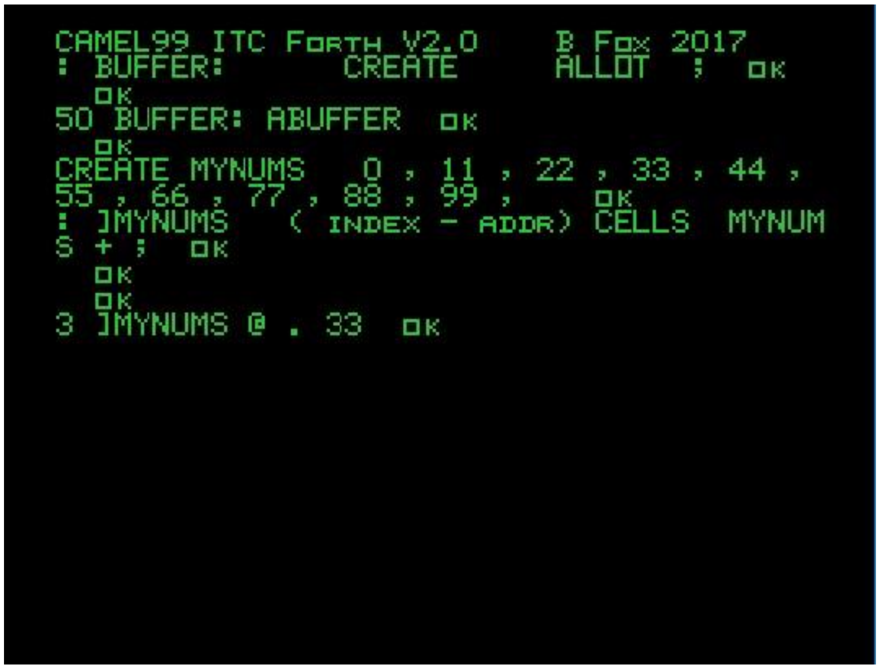
Explanation:

- CELLS is a forth function that multiplies a number by the memory address size of the CPU.
- As in x2 for a 16 bit CPU, x4 for 32 bit CPUs or x8 for a 64 bit computer.
- In this case it will multiply the index value on the stack
- MYNUMS returns the base address of the array
- ‘+’ simply adds the two numbers together giving us the address we need.

We can now fetch and see the value of any number in the array like this:

```
3 ]MYNUMS @ .  \ the ‘.’ word prints the number on the top of the stack
```

The screen capture shows all of this entered at Forth console:



```
CAMEL99 ITC FORTH V2.0    B Fox 2017
: BUFFER:      CREATE     ALLOT  ;  ok
   ok
50 BUFFER: ABUFFER  ok
   ok
CREATE MYNUMS   0 , 11 , 22 , 33 , 44 ,
55 , 66 , 77 , 88 , 99 ,      ok
: ]MYNUMS   ( INDEX - ADDR) CELLS  MYNUM
S + ;  ok
   ok
   ok
3 ]MYNUMS @ . 33  ok
```

Conclusion

So this tutorial gives you examples of how Forth builds itself up from almost nothing to higher levels of programming. This approach is used to create the memory management words that you can use but it's that the Forth compiler uses these same WORDs internally to compile words and numbers into memory and even to ASSEMBLE op-codes into memory in the Forth Assembler.

I know you are asking "Where did the compiler come from in the first place?" Well that's a bit more black-belt level programming but the principals are the very same. You can start in Assembler or C and make a few primitive routines. Then you use those routines to make higher level WORDs. People have even built Forth compilers in Java and LISP. The method however is always the same. You begin with simple pieces and combine them to build something better and eventually you have built the compiler. It's not for the faint-of-heart but the cool thing is that it is possible to understand every aspect of the system.