

Camel99

A Forth system for the TI-BASIC programmer

Brian Fox

Manual Revision 0.5
Copyright © 2018

Table of Contents

Introduction	8
About CAMEL Forth	8
What you have	9
Library Files	9
File Format	9
Only Needs the E/A Package.....	9
File Naming Convention (Optional)	9
Forth Terminology	10
Standard Forth vs CAMEL99 Forth.....	12
Library files Available with CAMEL99 Forth	12
The Forth Programming Pyramid.....	13
*Incremental Compiling.....	13
Loading CAMEL99	14
DSK1.START	14
Lesson 1: Jump Right in.....	15
Hello World	15
The Colon Compiler.....	16
Final Thoughts on Hello World	19
Error messages.....	19
Lesson 2: Transition from BASIC to Forth	20
Structured Programming	21
Moving Closer to Forth Style.....	21
Trim the Fat.....	21
Minimal	22
Factoring is the key to good Forth	22
Lesson 3: The DATA Stack	23
What is a computer stack?.....	23
A Few More Math Examples	25
A Fundamental Difference between BASIC and Forth.....	25
Why do we use a Stack?	25

EMIT	26
Stack Diagrams.....	27
Lesson 4: The DO LOOP.....	28
And one more thing... ..	30
If you are very curious.....	30
Strings in Forth.....	31
CAMEL99 String Word Set.....	31
Using CAMEL99 String Words	32
Technical Details about STRINGS.F	33
Other Facts.....	33
Graphics Words.....	34
A New Word for Convenience	36
COLOR in CAMEL99.....	37
Memory Tips and Tricks	38
High CPU RAM.....	38
Low CPU RAM	38
VDP Memory.....	38
SAMS Memory	38
Fetch and Store for each Space	38
Memory Operators	39
CPU RAM Memory Words.....	39
VDP Memory Words	39
SAMS Card Memory	39
Stealing Memory Temporarily	41
Temporary Memory Example Program	43
Multi-Tasking	44
Why Multi-Task?	44
Multi-Tasking Commands	44
TASK SWITCHER TECHNICAL EXPLANATION	46
Techie Stuff for the TMS9900 Nerd	47
WARNING for Assembly Language Users.....	47
CAMEL99 MULTI-TASKING USER AREA.....	48
Example Multi-tasking Code	49

Assembly Language the Easy Way	50
A CODE Word Example	50
Understanding Our Code Word	51
CAMEL99 Assembler vs TI Assembler	52
Forth Assembler Differences	52
Code Comparison.....	52
Special Registers in the Forth Machine.....	53
Proper Use of the TOS Register	53
Structured Branching and Looping	55
Example Programs	56
Random Color Dots	56
Guess the Number in Forth.....	57
GRAPHICS Example: “Denile”	58
Forth Style Version of Denile	61
Important Idea	61
ANS/ISO Forth Glossary	65
6.1 Core words	65
6.2 Core extension words	97
APPENDIX.....	109
Behind the House.....	110
How BASIC Sees a Program.....	110
How Forth sees a program.....	110
Essential Elements of Forth	110
The Forth Virtual Machine	111
Forth Virtual Machine from Different Perspectives	112
Hardware View.....	112
Software Layers.....	112
CAMEL99 Memory Map	41
Program Development in Forth	113
TI BASIC TOOL CHAIN	113
BASIC Process.....	113
Traditional C Tool Chain.....	113
C process	113

Forth Tool Chain.....	114
Forth Process	114
Memory Management in Forth	115

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 ANSI/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. We have made some changes to CAMEL Forth to make it faster for the TI-99.

It is called Camel Forth due to a perception about ANSI/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 a TI-99 E/A5 binary program file.

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 lean Forth Compiler/interpreter. It is a little more than the bare minimum that we need to run Forth. In 8K bytes it has all of the ANS Forth CORE word set except one word and that word is considered obsolete in Forth 2012. And it also has a good number of the CORE Extension Word set.
 - CAMEL99 has 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 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 with .F if they are TI-99 files.

File Format

CAMEL99 Forth assumes that all TI-99 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, start CAMEL99 and INCLUDE your saved text program.

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.
DATA Stack Alias: Parameter 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 DATA 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 DATA stack for ... well... DATA.
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. The colon let's you make words just like the Forth Kernel words.

CELL	A Memory location. The size of 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.
ADDRESS	Only relevant to BASIC in PEEK or POKE	Think of Forth memory as a collection of pigeon holes called CELLS. Each cell has a number. That number is the ADDRESS of the CELL.
BYTE –or- CHAR	One text character in BASIC	In the TI-99 CPU a byte is half of a CELL. CELL= 16 bits CHAR= 8 bits. On the TI-99 the terms mean the same thing. (NOT always true in other computers)
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.

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.

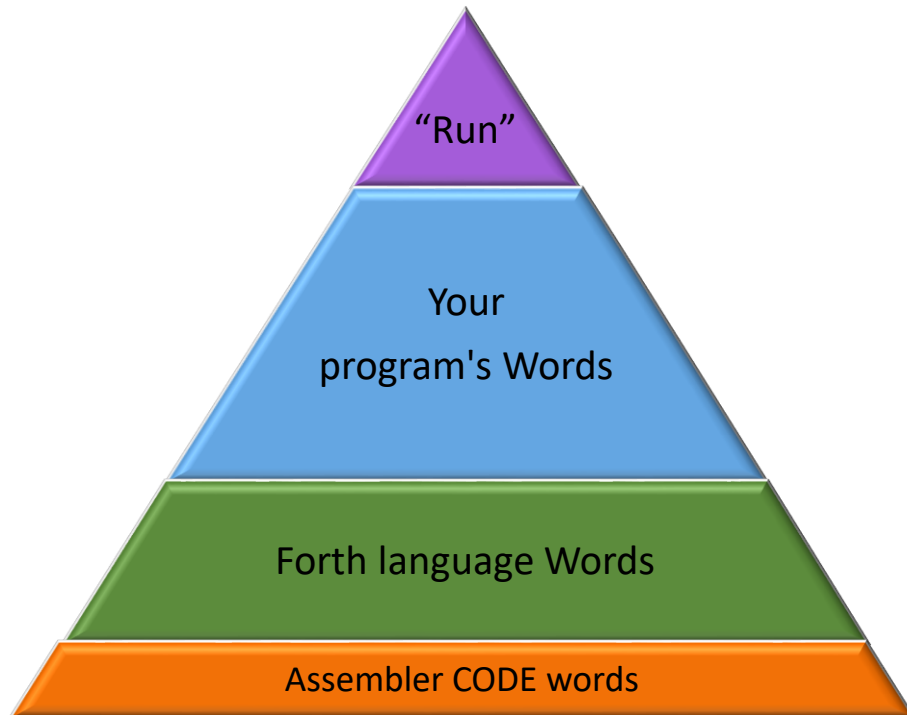
Library files Available with CAMEL99 Forth

80COL.FTH	ELAPSE.FTH	RANDOM.FTH
ANSFILES.FTH	ENUM.FTH	SAMSCARD.FTH
ASM9900.FTH	ERASE.FTH	SEARCH.FTH
BASICHLP.FTH	FLOORED.FTH	SOUND.FTH
BOOLEAN.FTH	GRAFIX.FTH	SPRITES.FTH
BREAK.FTH	GROMS.FTH	SQRT.FTH
BUFFER.FTH	HEAP.FTH	STRINGS.FTH
BYTES.FTH	HEXSTR.FTH	STRPLUS.FTH
CASE.FTH	INCLUDE.FTH	TILOGO.FTH
CELLS.FTH	INLIN99.FTH	TINYHEAP.FTH
CHAR.HSF	INPUT.FTH	TOOLS.FTH
CHARSET.FTH	LOWCASE.FTH	TRIG.FTH
CODE.FTH	MINI-OOF.FTH	UDOTR.FTH
COMPILE.FTH	MORE.FTH	VALUES.FTH
COMPARE.FTH	MOVE.FTH	VDPMEM.FTH
DATABYTE.FTH	MTASK99.FTH	VDPSOUND.FTH
DEFER.FTH	MTOOLS.FTH	VDPTYPE.FTH
DIR.FTH	PATTERN.FTH	VHEAP.FTH

The Forth Programming Pyramid

In BASIC you make your program line by line using only the commands that BASIC gives you.

You program in Forth by adding functionality to the Forth system itself. You make new WORDs that are added to the system word by word and you can test each word as you go if you want to. 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.

*Incremental Compiling

A very important aspect of Forth programming is that you can compile your program part by part.

This allows you to develop the building blocks, compile them and test each routine without needing to complete the entire program. As you complete more of the foundation routines the program gets easier and easier to write because you have made a custom set of words that do exactly what you need.

Loading CAMEL99

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

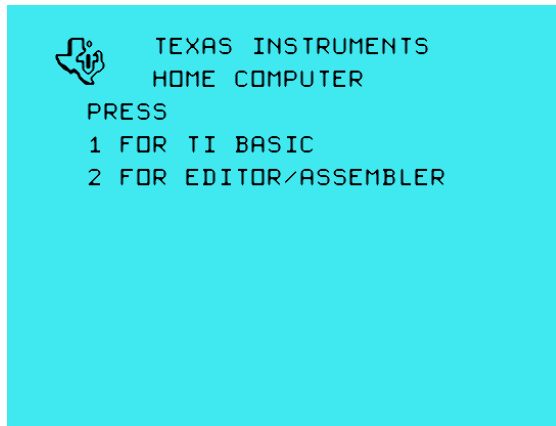


Figure 1 Select 2 For EDITOR/ASSEMBLER



Figure 2 Select 5. RUN PROGRAM FILE



Figure 3 Type DSK2.CAMEL99

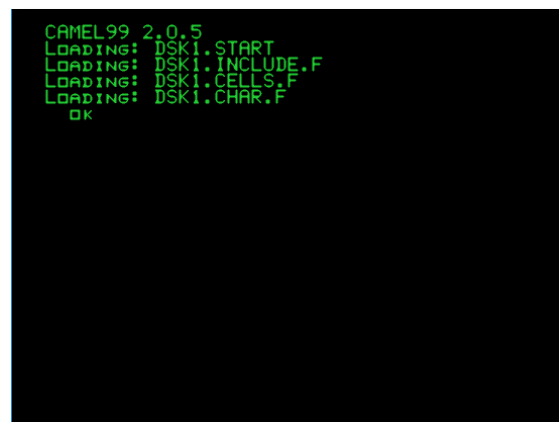


Figure 4 your screen should look like this

Note:

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

DSK1.START

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.

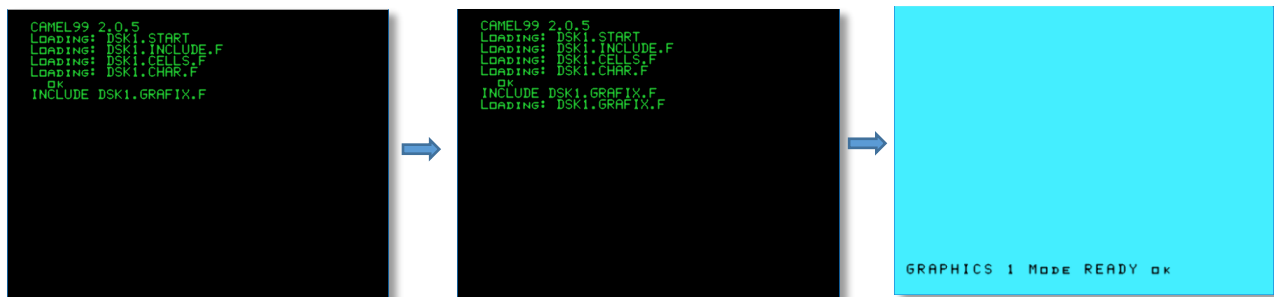
Lesson 1: Jump Right in

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



Hello World

Consider this one-line program in BASIC.

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 has between 1 and 31 characters and does not have a space in it. So here is what the equivalent program looks like in Forth. To make things familiar we chose to name our new Forth word "RUN". Isn't that cute.

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

Watch out! We need this space in the code.

```
TI BASIC READY
>10 PRINT "HELLO WORLD!"
>RUN
HELLO WORLD!
** DONE **
>
```

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

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.



CAMEL99 Forth Hello World Program

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.

There is something I should warn you about with 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. Also when your program finally runs it means it is pretty solid.

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 numbers (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.

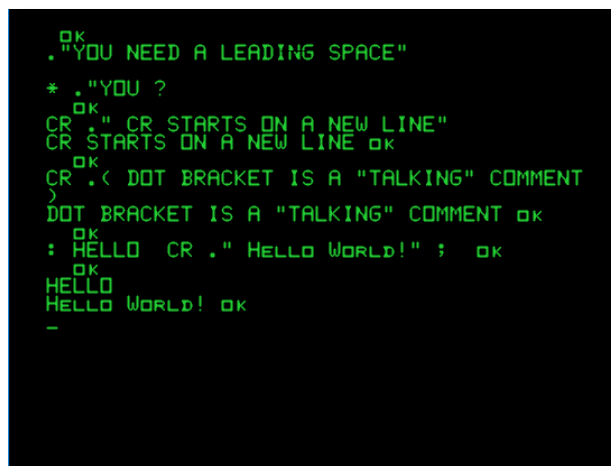
To print a string of characters we use the WORD `."` (pronounced: dot-quote). The Forth interpreter is simple too. Each command must be separated by a space. So `."` must have a space between it and the first character of the text that it will print.

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.BASICHLP.F
```

This will simply include all the files you need at once.

```
INPUT.F      ( $INPUT and #INPUT similar to BASIC)
RANDOM.F      ( RANDOMIZE, RND )
STRINGS.F    ( Strings with BASIC type functions)
GRAFIX.F     ( CLEAR, COLOR, SCREEN, VCHAR, HCHAR etc.)
```

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 **DATA 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 DATA 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 DATA 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 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 dispenser is like a computer stack

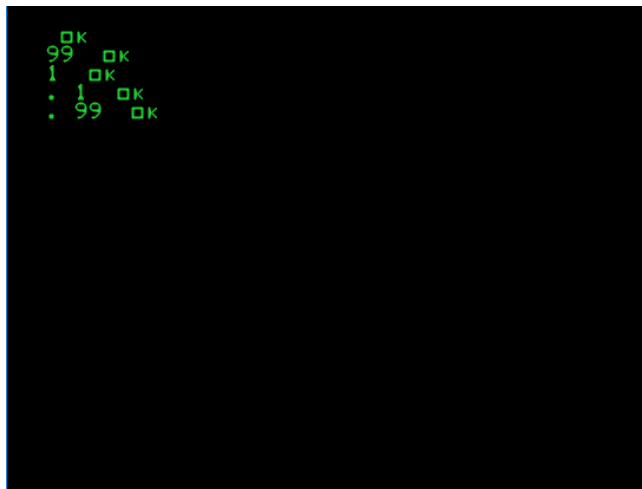


Figure 2 Putting numbers onto the DATA stack

* 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 DATA 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 DATA 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 DATA stack.

Adding on the Stack

```
OK
DECIMAL
99
1
+
. 100
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

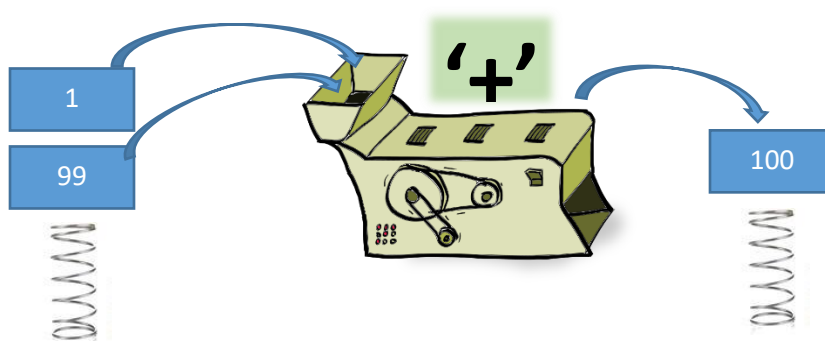
“+” is a Forth WORD. A sub-routine.

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.



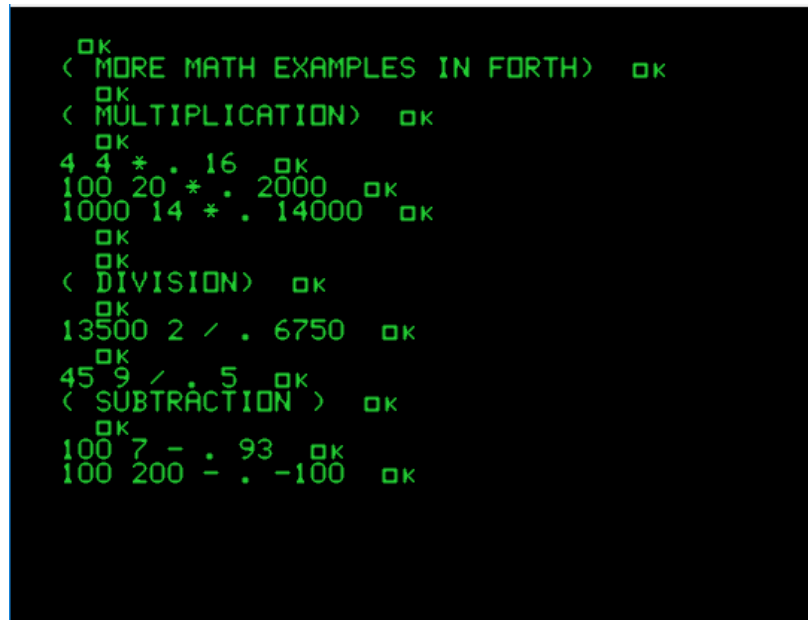
‘+’ takes 2 numbers, adds them and puts the answer on the stack

‘.’ 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 to 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.

A screenshot of a Forth console with a black background and green text. The text shows a series of commands and results. It starts with a prompt 'OK' followed by the command '< MORE MATH EXAMPLES IN FORTH>'. This is followed by another 'OK' and the command '< MULTIPLICATION>'. Then, three multiplication examples are shown: '4 4 * . 16', '100 20 * . 2000', and '1000 14 * . 14000', each followed by 'OK'. Next is a 'OK' and the command '< DIVISION>'. This is followed by two division examples: '13500 2 / . 6750' and '45 9 / . 5', each followed by 'OK'. Then is a 'OK' and the command '< SUBTRACTION>'. Finally, two subtraction examples are shown: '100 7 - . 93' and '100 200 - . -100', each followed by '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
```

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.



```
42  
* 
```

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 which has to find the correct routine to print a string and then finally it writes 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? This is done with the DO LOOP construct which we will see in the next lesson.

Stack Diagrams

Since the stack is so important in Forth code a “convention” of how to keep track of it has been developed.

In order to document the inputs and outputs of Forth we commonly use what is 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 like REM in BASIC. The open bracket ignores everything until it reads a ‘)’ in the code so it can be inserted in the middle of program code.

Notice there are two ‘n’s which are the input numbers. The two hyphens 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 at a minimum, 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

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 ;
```

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.

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

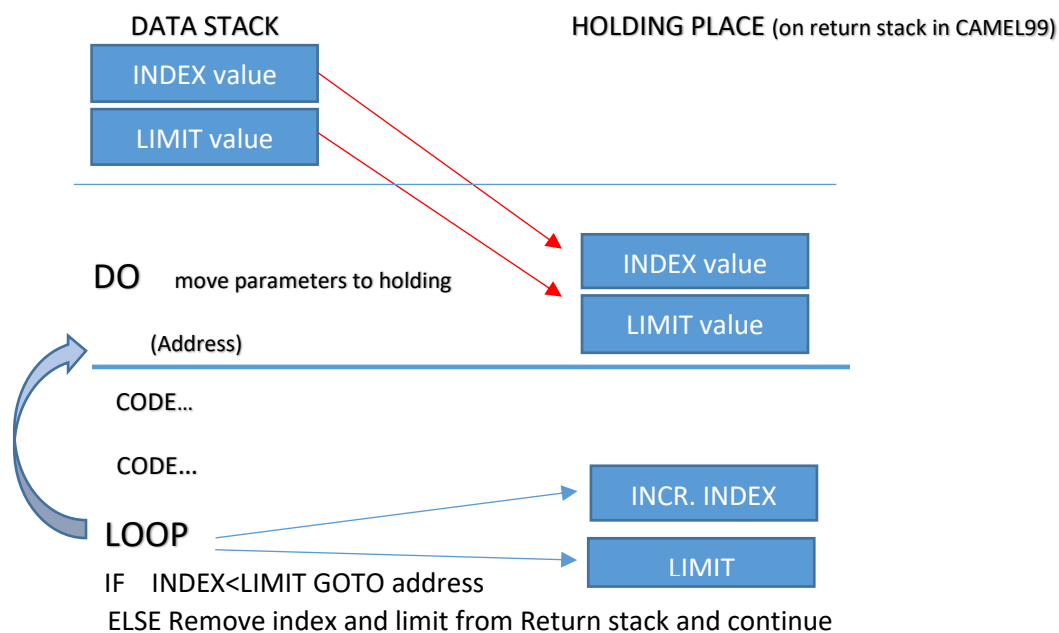
Notice we did not need a variable because we used the stack to pass the number 100 to STARS

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. (Oh yeah and the holding place just happens to be right there on the top of the Return Stack. That's the technical reason we talked about earlier)

Then it compares it to the LIMIT value to the new 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 EXITed to the Forth interpreter.

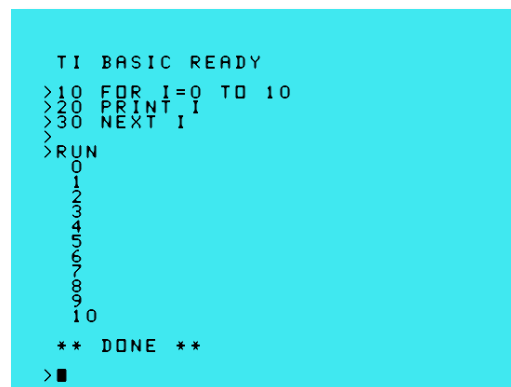
**Not every Forth system uses the return stack so that is why we call it a holding place.*

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



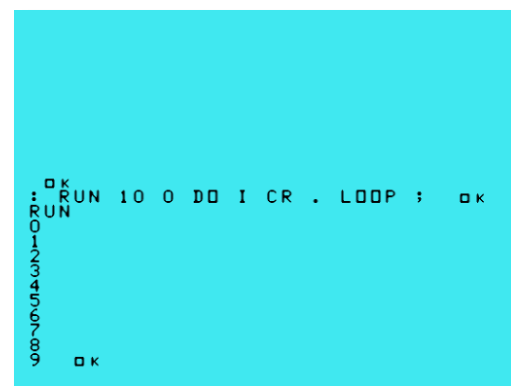
```
TI BASIC READY
>10 FOR I=0 TO 10
>20 PRINT I
>30 NEXT I
>
>RUN
0
1
2
3
4
5
6
7
8
9
10
** DONE **
>■
```

The Forth equivalent is:

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

The word 'I' is a new word for you. 'I' puts the loop's INDEX value on the DATA stack but you know what all the other words mean by now.

And we run this test we see this



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

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 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 ;
```

TI 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 need to know about it, but you can learn about it and understand it if you want to and use the knowledge to make better programs.

That’s a BIG DIFFERENCE with Forth.

If you are very curious

The entire source code for the CAMEL99 kernel is in two files called 9900FAST.HSF and CAMEL2.HSF.

The code is written in a dialect of Forth that was created just to make TI-99 programs with an MS DOS program. It can make your head spin to think about it but there it is for you to read.

You can find them on GITHUB in the SRC folder:

<https://github.com/bfox9900/CAMEL99-V2/tree/master/SRC>

Strings in Forth

One of the more powerful aspects in BASIC is its ability to manipulate text strings. When I first encountered Forth I was really stuck without my strings. So I took it upon myself to make string WORDs that gave me the abilities of BASIC in reverse polish notation. It might hurt your head a little but once you get to use to these words it is almost like being at home in TI BASIC. One thing you might notice is how fast these string functions run compared to TI BASIC.

CAMEL99 String Word Set

Creating String Variables

DIM (n -- <name>) creates a string variable in the dictionary of size n

TI BASIC Function Replicas

```
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=$2 flag=true
<>$      ( $1 $1 -- flag)  $1<>$2 flag=true
>$        ( $1 $2 -- flag)  $1>$2 flag=true
<$        ( $1 $2 -- flag)  $1<$2 flag=true
```

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
```

Init the string stack to the bottom

```
COLLAPSE ( -- )
```

Only need when the program first starts unless your program creates a problem.

Using CAMEL99 String Words

You can use the Forth string words like you would use string functions in TI BASIC. However like other Forth words the input arguments 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      ( running TEST will cause the text to be PUT into B$)

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

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

A$ PRINT
B$ PRINT

FIXED$ PRINT

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

B$ 4 10 SEG$ PRINT \ similar to BASIC but backwards

\ translate BASIC program to CAMEL99 Forth:
10 C$ = SEG$(A$4,10) & SEG$(B$,12,5)
20 PRINT C$

\ This could be on one line, but is vertical for explanation
A$ 4 10 SEG$ \ cut 1st segment
B$ 12 5 SEG$ \ cut 2nd segment
&           \ add them together
C$ PUT      \ put the result into C$
C$ PRINT    \ print C$
```

Important Difference with Forth Strings

After each string function the address of the new string is sitting on the DATA stack! So you don't actually need C\$ to just PRINT the result. You need C\$ to STORE the result. You might need to keep a copy in C\$, which is fine.

BUT ... if you just wanted to print the result you could do this:

```
A$ 4 10 SEG$    B$ 12 5 SEG$    & PRINT
```


Technical Details about STRINGS.F

For those who are curious here is some behind the scenes detail about how we made Forth do strings similar to BASIC. You might never think about this but when you do some string functions that operate on other string functions you need some way to keep the intermediate results. Consider the following BASIC code:

```
10 A$="This is an example of a string"
20 B$=SEG$(A$,1,POS(A$," ",1))
30 PRINT B$
```

When line 20 runs it actually starts working at the POS function because of the rules of evaluation.

So we have to calculate the position of the space first, giving us the number that we use to evaluate the SEG\$ function. This means we need to work with A\$ without altering it. Then we need to cut the SEG\$ with a copy of A\$ and we store the result of that into B\$. Then finally we print B\$

The way to do that is with a temporary storage space in memory where we can work on string but not alter the original. BUT each time a new function is used in the expression we need another copy.

This Forth string package does this similar to BASIC and that is by using a “string stack”

Each time our program runs a string function we push a copy onto the string stack to work on. The string on the TOP of the string stack can be used by your program and it is called TOP\$ but most of the time you don’t need to know this. The new strings are created on the string stack until we have run all the functions in the expression. Then TOP\$ becomes the final answer.

That’s all good but how do we know when to COLLAPSE the stack?

The magic happens when you PUT the final result in a string variable OR if you PRINT the final result with PRINT. With these two words we almost never have a string stack overflow problem.

Example

```
80 DIM A$  80 DIM B$
A$="This is an example of a string"
A$ 1 8 SEG$ ( a new temp string is now on the DATA stack)
           ( A$ was not changed)

( -- TEMP$) PRINT      \ we could PRINT the TEMP string which also collapses the
string stack.
B$ PUT      ( the temp string is PUT into B$ and the string stack is collapsed)
```

Other Facts

- If you want to manually reset the string stack use the word COLLAPSE
- The string stack uses un-allocated memory just above the dictionary and it moves up as the dictionary grows.

Graphics Words

If you include the file DSK1.GRAFIX.F you will have access to words that are common to all TI-BASIC programmers. Most of the BASIC words behave as you would expect in CAMEL99 but the word CHAR is already taken in Standard Forth so we have changed that name per the description below.

CLEAR (--)

Clears the screen and puts the cursor at beginning of the bottom line. I know that I always wanted a way to clear the screen and put the cursor on the top line. You have that too with the Forth word **PAGE**. Use as you see fit.

COLOR (character-set foreground background --)

This one behaves just like BASIC but you cannot set multiple character sets with it. Only one at a time.

COLORS (character-set1 character-set2 foreground background --)

This word lets you change a set of character between set1 and set2 inclusive. You can change the colors of all charsets almost instantly like this:

```
0 32 2 8 COLORS
```

GRAPHICS (--)

Switches the system to 24x32 characters Graphics mode. To get 40 column mode type **TEXT**.

SCREEN (color# --)

Put a color number on the stack and run screen.

CHARDEF (addr char# --)

This word replaces CALL CHAR in BASIC. It works in a way that maximizes speed. Rather than taking a text string and an ascii value, it takes ¹memory address and an ascii value. The memory address must contain 8 bytes that are the number you see in the HEX string in CALL CHAR(). Why an address? Because the memory contains the number pre-converted from text to binary numbers; ready to go. And we have a very fast assembler word called VWRITE that can write those numeric bytes into the Video chip RAM in micro-seconds. The missing piece of the puzzle is how do we convert those text numbers "FFFFABCD044055 etc." into binary number PATTERN in memory? With the word PATTERN: of course!

PATTERN: (u u u u <text> --)

With the word PATTERN: we can make character patterns and give them descriptive names. This word converts the hex number in the code to binary numbers and stores them in the Forth dictionary with a name. Notice you must break the pattern up into 4 pieces. (ie: 4 integers) Example:

```
HEX FFFF FFFF FFFF FFFF PATTERN: ABLOCK
```

Later, anywhere in your program you can set a character's pattern with CHARDEF like this and it happens VERY fast.

```
ABLOCK 65 CHARDEF
```

¹ These memory addresses are called "pointers" in other languages but Forth keeps things simple

GCHAR (col row -- char)

Moves the cursor to col,row and reads the character on the screen and puts it on the stack.

HCHAR (col row char cnt --)

This word works like BASIC but the top right corner is 0,0 not 1,1. The Forth version is about 10 times faster than BASIC too.

Try this program in BASIC:

```
10 CALL CLEAR
20 CALL HCHAR(1,1,42,768)
30 CALL CLEAR
```

Now INCLUDE the GRAFIX.F and in GRAPHICS Mode type this into the Forth console

```
DECIMAL
0 0 42 768 HCHAR  CLEAR  <enter>
```

This difference is shocking. This is because uses the word VWRITE, a CODE word that is very fast.

VCHAR (col row char cnt --)

This word works like BASIC but top right corner is 0,0 not 1,1. It is about 2.5X faster than BASIC.

CHARPAT (addr char# --)

This word transfers a character pattern from VDP chip memory (the pattern description table) for char# to CPU RAM at the address on the stack.

²PAD is a temporary memory space in Forth that can be used for little jobs like this.

Example usage of CHARPAT below:

```
PAD CHAR A CHARPAT \ read the char pattern for letter A into PAD memory address
PAD CHAR Q CHARDEF \ now re-define the letter Q to look like letter A. ☺
```

Or if you wanted to keep a permanet copy of a character PATTERN you just need to CREATE a memory space that is 4 CELLS in size:

```
CREATE MYPATTERN 4 CELLS ALLOT
MYPATTERN CHAR $ CHARPAT \ now we can use MYPATTERN to CHARDEF characters
```

² PAD in Forth is in dictionary memory. TI-99s PAD is at address >8300 and is only 256 bytes in size

A New Word for Convenience

SET# (char# -- character-set)

I could never remember the correct character SET# for an ASCII CHAR. And then with Forth all the set numbers are different so it was even more confusing. “SET\$#” does this for you.

Look at this example:

```
CHAR A SET# 4 8 COLOR
CHAR X SET# 2 7 COLOR
```

How simple is that? Put in the character that you want to change, use SET# and then the color values. It's so much clearer to understand than CALL COLOR (5, 4, 8)

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

Memory Tips and Tricks

For the TI BASIC programmer most of the details of memory are hidden from view. Forth exposes all of this to you like Assembly Language. The difference with Forth is that there are WORDs that give some structure to the memory and if you need to you can change these WORDs to suit your needs as well.

The little TI-99 has no less than three standard memory spaces and if you use the SAMS Memory Card or use it in the CLASSIC99 emulator there is a fourth memory space that gives you another Mega-byte of space! Here is an overview of these memory spaces and how you use them in CAMEL99 Forth.

High CPU RAM

This is the memory that the CPU can access “natively”. This means it is the memory the computer was designed to use most commonly. The TMS9900 CPU is designed to access 32K 16 bit words of memory in this space. It can also access the individual bytes so you can say it has a 64K byte memory.

This memory space is not ALL available to Forth because the system uses it for other things.

Forth uses the 24K of space from address HEX A000 to FF00 as the dictionary.

Low CPU RAM

There is an 8K block of “LOW MEMORY” and Forth uses this as spare memory called the HEAP.

The top end of that memory block is reserved for use by the Forth stacks and a little space is used to call routines in the expansion card devices. (Device Service Routine or DSR)

See the Memory Map diagram for more detail.

VDP Memory

VDP Memory is 16K bytes that are not connected to the CPU address buss directly but are accessed via memory “ports”. These are special memory address that are like special doorways into the VDP memory space. TI BASIC uses this memory to store your programs if you do not have an expansion memory card.

It is used for Console screen display, the character patterns , sprites and character colors.

SAMS Memory

The SAMS card gives us a big block of memory but it is only visible to the computer in 4K blocks. Out of the box CAMEL99 access this memory in the memory space starting at HEX 6000. This is where TI cartridges are connected to the system if the cartridge has internal ROM or RAM chips. (Extended BASIC for example) Fortunately the Editor Assembler cartridge has no ROM on board so we can use the space for SAMS memory.

Fetch and Store for each Space

Each of these memory areas need to be accessed with different little routines. The Forth solution to this kind of problem is to use the different versions of the standard Forth words for memory so that the languages is consistent even though you are touching different memory devices.

Memory Operators

CPU RAM Memory Words

Here is the template for the words we will use later for different memory spaces.

@ (addr – n) Fetch the integer from memory 'addr' and put it on the stack

C@ (addr – c) Fetch the char (byte) 'c' from memory 'addr' and put it on the stack

! (n addr --) Store n in the memory addr leaving nothing on the stack

C! (c addr --) Store c in the memory addr leaving nothing on the stack

Move Blocks

CMOVE (src dst n --) Move bytes from src address to dst address. **SRC and DST MUST NOT OVER-LAP**

MOVE (src dst n --) Library file: DSK1.MOVE.F This is a little smarter word and it can handle moving memory blocks that overlap each other.

These words work for any memory that is connected to the CPU memory buss. This includes ROM, Low CPU RAM and High CPU RAM and also the High-speed memory called the PAD by TI. (HEX 8300)

ALLOT (n --)

Allocate n bytes of memory in the Dictionary. It's not discussed much but there is no rule that says you cannot ALLOT a negative amount of memory. So you can take memory back if you know what you are doing.

VDP Memory Words

V@ (addr – n) Fetch the integer from VDP memory 'addr' and put it on the stack

VC@ (addr – c) Fetch the char (byte) 'c' from VDP memory 'addr' and put it on the stack

V! (n addr --) Store integer in the VDP memory addr leaving nothing on the stack

VC! (c addr --) Store 8 bit c in the VDP memory addr leaving nothing on the stack

Transfer Blocks to/from VDP RAM

VWRITE (CPU-addr VDP-addr n --) Transfer n bytes from CPU addr to VDP-addr (CODE word)

VREAD (VDP-addr CPU-addr n --) Transfer n bytes from VDP-addr to CPU-addr (CODE word)

SAMS Card Memory

Library File: DSK1.SAMS.F

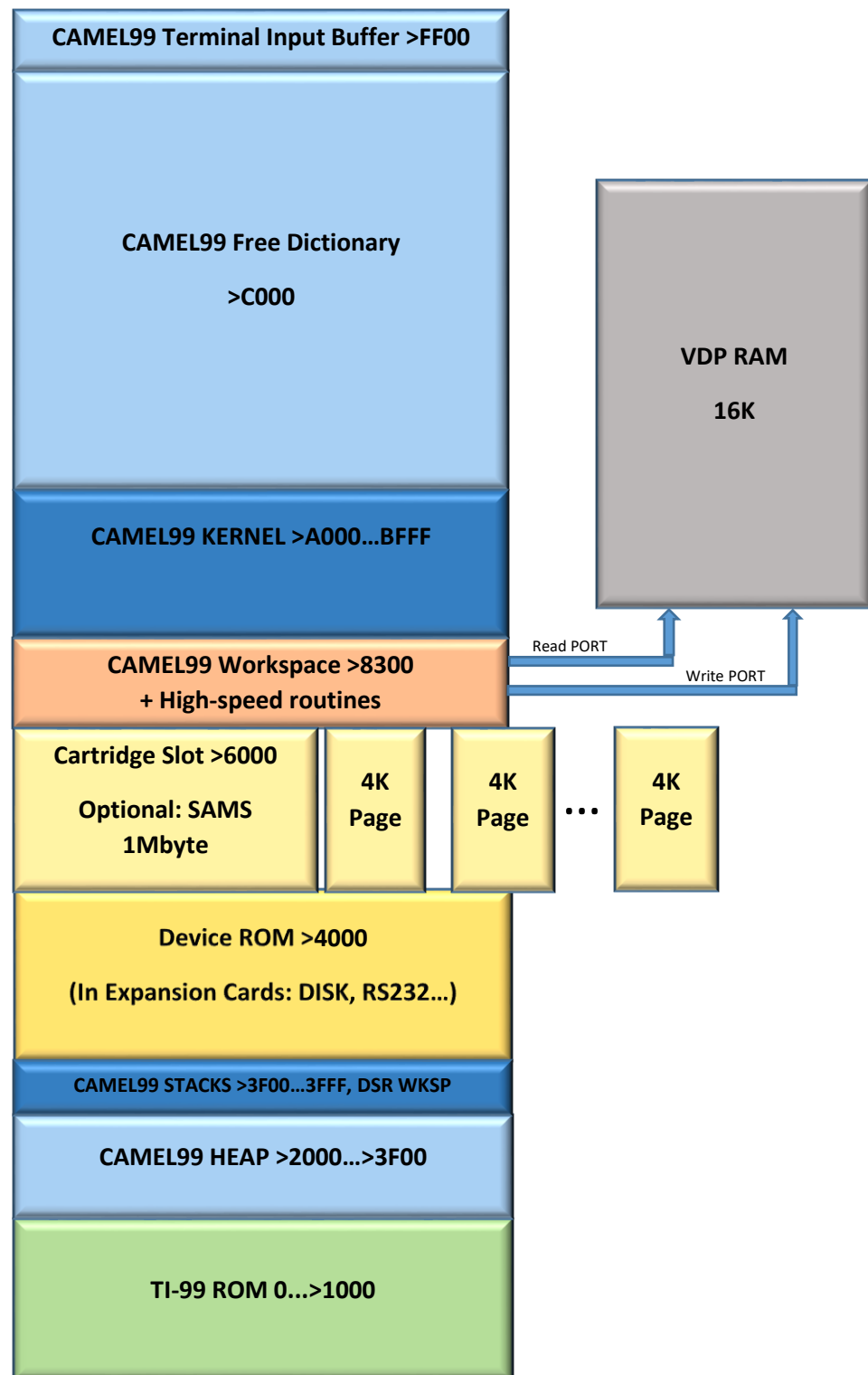
!L (n d --) store n into the double address on the stack

@L (d -- n) fetch the integer at the double integer address on the stack

Note:

We need to use a double int address for SAMS addresses because with a single 16 bit integer we can only reach to 65,635. We need to reach 1,048,576 address locations in the SAMS card.

CAMEL99 Memory Map



Stealing Memory Temporarily

Forth like BASIC lets you add programs to memory until the memory is full. As you add things to the program that memory is permanently allocated or subtracted for what is available.

Sometimes you would like to have a little memory but you don't need it permanently. The Forth dictionary is so simple that it is easy to steal some memory and then put it back.

The end of the Forth dictionary uses one variable to keep track of it. It is called 'DP'. If you fetch the contents of DP you can see the end address like this:

```
DP @ U.
```

```
( You must use U. to print the un-signed value because the number is bigger than 32767)
```

The Forth word 'HERE' is nothing more than a colon definition to fetch DP.

```
: HERE ( -- ADDR ) DP @ ;
```

There is nothing preventing us in Forth from moving the number in the variable DP! We have the power. So we can use some memory and then put DP back to its old value and instantly reclaim the memory we just used.

Example of Temporary Memory Usage

When changing character patterns in the VDP memory we simply write numbers into the pattern description table for each character. The numbers we write are the ones you see all the time in the CALL CHAR statement. But they must be translated from text "FFFBCDEDE..." into binary numbers before we write them to VDP RAM.

The Forth interpreter can translate text hex numbers into binary numbers easily and we can compile them into CPU memory with the comma.

The Trick

We will remember where the memory starts with HERE and then compile a bunch of character patterns into Dictionary memory. Then we will write all the characters to the VDP RAM in one blast using VWRITE. Finally we will change the value in DP back to our original HERE address removing any trace that we used the memory as far as the Forth system is concerned. Tricky but completely legal in Forth.

Temporary Memory Example Program

\ TI.LOGO GRAPHICS chars patterns from Sometimes99er Atariage.com
\ requires DSK1.GRAFIX.F

```
: TI.LOGO ( -- ) \ print the TI logo on a new line
  CR 11 EMIT 12 EMIT 13 EMIT    \ print 3 characters
  CR 14 EMIT 15 EMIT 16 EMIT    \ print 3 characters
  CR 17 EMIT 18 EMIT 19 EMIT    ; \ print 3 characters
```

\ get a copy of the dictionary address on the DATA stack

HERE (-- addr)

\ Now compile patterns into unused dictionary space

```
0103 , 0303 , 0303 , 0303 ,
FC04 , 0505 , 0406 , 020C ,
0080 , 4040 , 8000 , 0C12 ,
FF80 , C040 , 6038 , 1C0E ,
1921 , 213D , 0505 , 05C4 ,
BA8A , 8ABA , A1A1 , A122 ,
0301 , 0000 , 0000 , 0000 ,
E231 , 1018 , 0C07 , 0300 ,
4C90 , 2040 , 4020 , E000 ,
```

\ write the data from HERE on stack, to VDP memory
\ starting at the location in the pattern description table
\ for character HEX 11.
\ We calc. how many bytes like this
\ 4 CELLS for each character X 9 characters

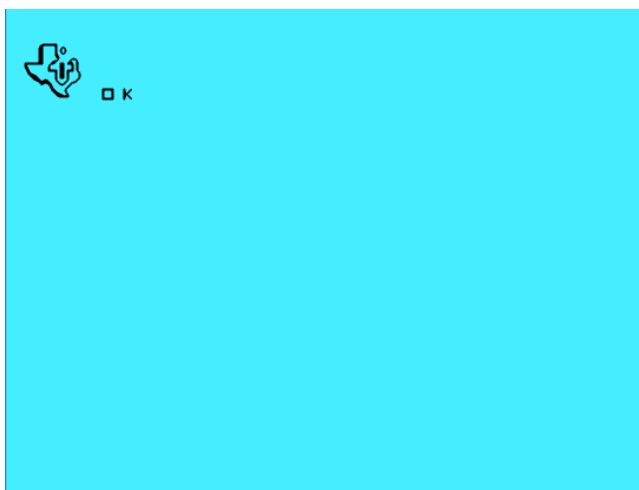
(-- addr) DUP 11]PDT 4 CELLS 9 * VWRITE

\ now put the dictionary pointer back to old HERE (reclaims the memory)

DP !

PAGE TI.LOGO

\ No CPU memory was permanently used in the making of the this program :-)



Multi-Tasking

CAMEL99 has been extended to include a classic Forth cooperative multi-tasker. Forth was always a multi-tasking system in its first commercial versions but somehow this feature was not included in Fig-Forth and other public domain Forth systems. Even today there are no standard words for these functions at this time because there are conflicting implementations in many professional Forth systems. The words chosen for CAMEL99 Forth are commonly used with the exception of FORK, which is a word used by UNIX.

Why Multi-Task?

There are so many times in a program where you want some little routine to just look after itself and keeping doing something repetitively every now and then. In a single task program this means your program must keep track of when to do that thing and then jump to that routine.

With a multi-tasker these things become more like having little machines that do things for you while you concentrate on something else. This is particularly valuable in creating video games where you want the computer generated enemy to operate independently while the hero (the human) battles on for life , liberty and the pursuit of higher scores.

Check on the multi-tasking demo code in this chapter and also in the DEMO programs supplied in the DSK2 folder.

Multi-Tasking Commands

Here are the words that allow you make multi-tasking programs in CAMEL99 Forth.

INIT-MULTI (--)

INIT-MULTI **must** be run once before any other tasks run. INIT-MULTI configures the root task for multi-tasking operation. When you load DSK1.MTASK99.F it runs at the end of the file for convenience.

PAUSE (--)

This is the user level command that causes multi-tasking. You control things by inserting this word where ever you need the program to give time to other tasks. Typically this is done inside loops or before executing an I/O routine. For example the forth delay timer called 'MS' has PAUSE inside its loop so the machine can do other things while delaying on some task. Clever eh?

LOCAL (PID uvar -- addr')

Returns the addr of a user-variable in PID's user area when given the process ID and the User-variable name. Example: TASK3 TFLAG LOCAL @ returns the value of TFLAG in TASK3's memory space.

SLEEP (PID --)

Put a task PID to sleep. This is done by setting the local TFLAG variable to FALSE.

WAKE (PID --)

Wake up a task. This is done by setting the local TFLAG variable to TRUE.

SINGLE (--)

Single changes the action of PAUSE to a NEXT. This disables all MULTI-tasking.

MULTI (--)

MULTI changes the action of PAUSE to YIELD. Yield is the internal routine that switches from one task to the next.

MYSELF (-- PID)

MYSELF returns the Process Identifier (PID) of the currently running task. In CAMEL99 this is the workspace address of the task. Workspace is a TMS9900 term for the memory where the CPU registers are located.

FORK (PID --)

Fork copies the workspace of the root task (USER0) into the workspace of a new task and then modifies it so it is ready to run as a separate task with its own stacks and workspace and Forth registers configured.

ASSIGN (xt pid --)

Assign the execution token (XT) of a Forth word to the task PID. This sets up a task to run the Forth word.

RESTART (pid --)

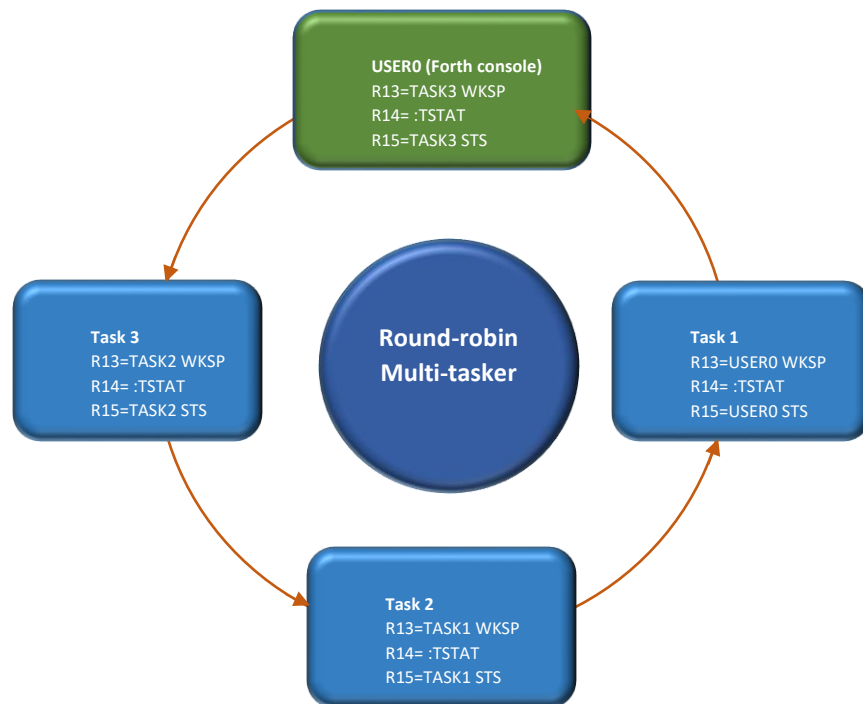
Cause the program to re-initialize and begin running like it was when it was first started up.

TASK SWITCHER TECHNICAL EXPLANATION

The Wondrous TMS9900

Forth multi-taskers use a word, in this case YIELD, that switches from one task "context" to the next "context". TMS9900 uses a fantastic method to hold context called the Workspace. CAMEL99 uses this feature of the CPU to make a very fast context switch from one task to the next.

CAMEL99 initializes the workspace of each task as if it had been called by a BLWP instruction. Each workspace has its return register, R13, set to point to the previous workspace (task). With all the workspaces pointing in a circle, we can use the TMS9900 RTWP instruction to hop back to the previous task in one instruction! How cool is that?



But TMS9900 created a problem. The RTWP instruction will change context immediately given an address and program counter in R13 and R14. This is different than conventional round robin where YIELD remains in a loop testing each task in the linked list, only leaving the loop when a task is awake. (tflag<>0)

*SOLUTION : *Divide YIELD into 2 parts**

Part1 : YIELD

- Do the Forth style context switch at appropriate code boundaries.
- In this case it's just one instruction. RTWP.

Part2 : TSTAT

- Load R14 (Program counter register) of each task with the address of the routine "TSTAT"
- TSTAT will therefore run when the RTWP instruction hops to the new workspace.

- TSTAT tests its own TLAG variable to see if it is awake
- If the task is asleep TSTAT jumps back to YIELD otherwise its runs NEXT, which runs the Forth system for the awake task we just entered.

```
\ This is the entire task switcher for CAMEL99 Forth.
CODE YIELD ( -- )
    BEGIN,          \ we're in the current task
    RTWP,           \ one instruction switches context
1: _TSTAT          \ In the New TASK, store NEW workspace in R1
    32 (R1) R0 MOV, \ Read local TFLAG variable to see if I am awake
    NE UNTIL,       \ loop thru tasks until TFLAG is <> 0
    NEXT,           \ task is awake, run its own next Forth word
ENDCODE
```

Techie Stuff for the TMS9900 Nerd

This multi-tasker takes advantage of the unique TMS9900 memory to memory architecture to create a 20uS task switcher. The WP register in the 9900 CPU points to the current WORKSPACE which is normally just the registers.

We extend the “workspace” concept so that the WP register is the base address of:

- 16 registers
- The tasks “USER variables” (variables that are local to a task)
- The task’s DATA Stack
- The task’s Return Stack

This entire memory area is called a USER AREA.

With this system therefore, the WP register in the CPU becomes the USER POINTER (UP) of a conventional Forth multi-tasker.

Using WP to point to the USER area also lets us use the Workspace register architecture further. We can use registers 13,14 and 15 to link to another workspace and use the RTWP instruction to change tasks in 1 instruction! A very neat trick.

ALSO, the CPU registers become user variables 0..15 of each task so we can use them just like Forth variables.

WARNING for Assembly Language Users

BLWP,RTWP, R13 and R14 have been stolen by this MULTI-TASKER.

If you want to write code words that use BLWP/RTWP you must either save the contents of R13 and R14 before using BLWP or create a dummy workspace, then LWPI to the DUMMY workspace and BLWP to your new workspace. Then get back to Forth by reversing the process.

CAMEL99 MULTI-TASKING USER AREA

HEX

```
0 USER R0    LOCAL general purpose register      ( workspace begins)
2 USER R1    LOCAL general purpose register
4 USER R2    LOCAL general purpose register
6 USER R3    LOCAL general purpose register
8 USER R4    LOCAL Top of stack cache
0A USER R5   LOCAL overflow for mult. & div.,
0C USER R6   LOCAL DATA stack pointer ('SP')
0E USER R7   LOCAL return stack pointer  ('RP')
10 USER R8   LOCAL Forth working register  ('W')
11 USER R9   LOCAL Forth interpreter pointer ('IP')
12 USER R10  LOCAL Forth's "NEXT" routine cached in R10
14 USER R11  LOCAL 9900 sub-routine return register
18 USER R12  LOCAL 9900 CRU register
1A USER R13  LOCAL task link
1C USER R14  LOCAL Program counter: ALWAYS runs TSTAT routine
1F USER R15  LOCAL Status Register

\ -----registers end, USER variables begin-----

20 USER: TFLAG           \ TASK flag awake/asleep status
22 USER: JOB             \ Forth word that runs in a task
24 USER: DP
26 USER: HP
28 USER: CSP
2A USER: BASE
2C USER: >IN
\ 2E USER: 'EMIT         \ vector for char. output routine
\ 30 USER: 'CR           \ vector for carriage return
\ 32 USER: 'KEY          \ vector for key input >8332
\ 34 USER: 'TYPE         \ vector for block output
\ 36 USER: 'PAGE         \ vector for screen clear
38 USER: LP             \ LEAVE stack pointer.
3A USER: SOURCE-ID      \ =true if EVALUATE is running, false otherwise
3C USER: 'SOURCE        \ WATCH OUT! This is 2variable, occupies 3C and 3E
\ 3E USER:              \ 2nd cell of 'SOURCE
\ 40 USER: CURRENT
\ 42 USER: CONTEXT
\ 44 USER:
\ 46 USER:              \ last free space in hi speed RAM (PAD)
```


Example Multi-tasking Code

```
INIT-MULTI
CREATE TASK1 USIZE ALLOT    TASK1 FORK ( setup task1's user area)

CREATE TASK2 USIZE ALLOT    TASK2 FORK ( setup task2's user area)


VARIABLE X
VARIABLE Y

: THING1 BEGIN    1 X +!    PAUSE AGAIN ;
: THING2 BEGIN   -1 Y +!    PAUSE AGAIN ;


' THING1 TASK1 ASSIGN
' THING2 TASK2 ASSIGN


MULTI          ( enable task switcher)
TASK1 WAKE
TASK2 WAKE


\ when you type this you should see the variables changing quickly
X @ .
Y @ .
```

Assembly Language the Easy Way

One of the big secrets about Forth is the Forth Assembler. It gives you a way to learn Assembly language in a way that is so much less painful than the traditional Assembler process. You can learn by writing tiny routines that do simple things and test them in the interpreter. WHAT? That's impossible you say.

Well... you must still understand how the TMS9900 CPU operates and any other system details that your program uses but it follows the "How do you eat an elephant principle?" Answer: Piece by piece.

Consider this traditional Assembly language work flow:

1. Write your program in the editor, save as SOURCE file
2. Assemble the program with the Assembler, giving OBJECT file.
3. *Link the OBJECT file with the LINKER giving BINARY file
4. Load the BINARY file into RAM and watch it CRASH
5. Goto 1

*TI-99 joins step 3 and 4 with a special "LINKING LOADER" program so they were aware of this issue.

Now consider how you make an Assembly language program in Forth"

1. Start CAMEL99 Forth
2. Load the Assembler (becomes part of Forth)
3. Write a short CODE word in Assembly language
4. Test the CODE word by typing its name

A CODE Word Example

Let's imagine we wanted a way to increment a Forth variable by two at maximum speed. It turns out that the 9900 CPU can do that in one instruction. Very fast. Let's test this idea in the Forth console.

```
INCLUDE DSK1.TOOLS.F      \ give us the programmer tools
INCLUDE DSK1.ASM9900.F    \ give us the Assembler words

\ "two-plus-store" adds 2 to the contents in addr
\ TOS is the CPU register where CAMEL99 caches the "top of stack" value

CODE 2+!      ( addr -- ) *TOS INCT,    NEXT,    ENDCODE

\ test it at the Forth console
VARIABLE X

X ? 0 ok
X 2+!
X ? 2 ok
X 2+!
X ? 4
```

Seems to work. We are done!

"2+!" is now just another word in the Forth dictionary but it runs really fast.

Understanding Our Code Word

```
CODE 2+! ( addr -- )
```

CODE is a Forth word that creates a new “code” word in the dictionary. A CODE word is slightly different than words created with colon (:). CODE tells Forth that this new word will be followed by real machine instructions and not a list of addresses like colon definition words.

```
*TOS INCT,
```

This is the one instruction in our code word example. As mentioned TOS is the “top-of-stack” CPU register. It happens to be Register 4 (R4) but we renamed it for clarity in our code. It only takes a colon definition to do that. ☺

INCT, is the Assembler name for the “INCREMENT BY TWO” instruction. All the CPU instructions if the Forth assembler end with a comma. This is for two reasons:

1. The comma is a Forth word that compiles a number into memory. Having a comma at the end of our instructions reminds us that the instruction also compiles some numbers into memory. (In fact the Assembler uses Forth’s “comma” to do the job)
2. It keeps the names of the Assembler instructions different than the rest of the Forth words we make. No name conflicts with very little effort.

```
NEXT,
```

For our purposes NEXT, is not a real instruction for the TMS9900 CPU but rather it is a name for a routine that jumps to the NEXT Forth word that will run in the system. Technically speaking NEXT, is something called a MACRO-instruction. This means it inserts some Assembly language instructions into your program, but usually you don’t need to remember the details of the instructions it inserts.

If your Assembler word needs to return to Forth it MUST end with the NEXT, macro-instruction

For the very curious here are the instructions that it inserts at the end of your Assembly language word.

```
*R10 B, \ branch to the address in Register 10.
```

```
ENDCODE
```

ENDCODE is a housekeeping word. You don’t really need to use it, but it works with CODE to test if your ASSEMBLY routine left anything behind on the DATA stack. If it finds something left behind, that means there is a syntax error in your Assembler code so it will ABORT the system and give you the message:

```
* UNFINISHED ?
```

CAMEL99 Assembler vs TI Assembler

There are some really big differences between these two systems. First of all the TI-Assembler is a program. It's pretty complicated and it has to read through your program file a couple of times to collect all the information it needs to make a binary file. The Forth Assembler can't even be called a program. It is actually a collection of tiny "assemblers". Each word is a little program that knows how to Assembler one kind of instruction. That's about as different as you can get.

Forth Assembler Differences

1. Arguments on the left, instructions on the right. Typical of Forth words, the little assembler words take their inputs (registers and numbers) from the stack so the instruction comes last
2. No Assembler "Directives". Since you are in the Forth world a REF in TI Assembler is a Variable or a named memory location. A DEF is just another Forth word that you might execute in your program.
3. Forth Assembler is part of a unified environment with the interpreter and the compiler.

Code Comparison

We have made an effort to enhance the original TI Forth Assembler to make the code closer to TI Assembler but it is not identical.

<pre>\ ASM9900 Usage \ ----- \ src. dst. \ ---- ---- \ *R13 R2 MOV, *R13+ R2 MOV,</pre>	<pre>* TI Syntax Equivalent * ----- * label src. dst. * ---- ---- ---- * * MOV *R13, R2 * MOV *R13+,R2</pre>
---	--

We simply use Forth to create data areas. X and ARRAY will return their address just like a Label in TI Assembler .

<pre>VARIABLE X VARIABLE Y X @@ R12 ADD, X @@ Y @@ MOV, HEX CREATE ARRAY 100 CELLS ALLOT ARRAY (R13) R2 MOV,</pre>	<pre>X DATA 0 Y DATA 0 A X@,R12 MOV X@,Y@ ARRAY BSS >100*2 MOV @ARRAY(R13),R2</pre>
---	---

**We don't need the ampersand (@) in the Forth Assembler when we use indirect/index addressing mode.*

Special Registers in the Forth Machine

Forth takes over the CPU and transforms it into something called a Virtual Machine. This means the TMS9900 begins to act like a Forth CPU with two stacks. To do this Forth uses some of the CPU's registers for dedicated purposes. Here are their names, what they do and how to use them in the CAMEL99 Assembler. I am only going to show you the code for how we created the names because you know what colon definition is by now.

```
\ R0..R3 are available for your programs.
\ R5 can be used inside your CODE words but will be erase when you return to Forth

\ R4 re-named to TOS, caches the TOP item in the DATA stack
4 CONSTANT TOS
: (TOS)      TOS () ;
: *TOS      TOS ** ;
: *TOS+     TOS *+ ;

\ R6 called SP, points to the 2nd item on the DATA stack (see TOS)
6 CONSTANT SP
: (SP)      SP () ;
: *SP      SP ** ;
: *SP+     SP *+ ;

7 CONSTANT RP
: (RP)      RP () ;
: *RP      RP ** ;
: *RP+     RP *+ ;

\ W is a temp register used by the address interpreter
\ W can be used inside your CODE words but will be erase when you return toForth

8 CONSTANT W
: (W)      W () ;
: *W      W ** ;
: *W+     W *+ ;

\ R9 renamed to IP is the Forth "Instruction Pointer"
9 CONSTANT IP
: (IP)      IP () ;
: *IP      IP ** ;
: *IP+     IP *+ ;

\ R10 holds the address to the code for the Address interpreter
: *R10      10 ** ;

\ R11 is a TMS9900 register which hold the return address after a BL instruction
: *R11      11 ** ;

\ R12 is used for CRU addresses only in CAMEL99. See CRU! CRU@ in glossary

\ R13..R15 are used by the multi-tasker.
If your program is single tasked then they are FREE to use them.
```

Proper Use of the TOS Register

CAMEL99 keeps the top item of the DATA stack cached in Register 4. This make the system about 8 to 10% faster than if we kept TOS in memory. When you are writing your own Assembler code you must understand how to handle keeping the TOS register up to date. Here are some code examples from the

Kernel that give you examples of how to do it. We can't anticipate every situation but these give you a good start.

Condition 1

This simplest code words are when there is one input on the TOS and one output on the TOS. This means you don't have to worry about refreshing the TOS register. Below see the definition of FETCH, which takes an address from the TOS and replaces it with the contents of the address.

```
CODE @      ( addr -- n )
            *TOS TOS MOV,
            NEXT,
            ENDCODE
```

Condition 2

The second condition you might encounter is where you take 2 input arguments, process them and return 1 output. The Plus word in Forth works like that. Notice we take the 2nd item by referencing the SP register with indirect addressing mode.

*The clever trick here is provide by using the auto-incrementing action of the TMS9900 (*SP+)

The DATA stack grows downward in memory. When you ADD and auto-increment SP, the SP register has 2 added to it. This is like dropping the second item off the stack so the stack housing-keeping is automatic. Nice!

```
CODE: +      ( u1 u2 -- u )
            *SP+ TOS ADD,      \ ADD 2nd item to TOS and incr stack pointer.
            NEXT,
            END-CODE
```

Condition 3

This situation is one where you need a cell to put your output data into. This requires that you push the current value in the TOS register onto the data stack in memory. This takes 2 instructions on the 9900. One to DECT SP and then a MOV operation to MOV R4 to *SP. We have rolled these into a macro instruction called PUSH, . See how it is used below in the R@ word:

```
CODE: R@      ( -- w )
            TOS PUSH,      \ PUSH the current TOS onto DATA stack
            *RP TOS MOV,    \ Move the contents of top of return stack to TOS
            NEXT,
            END-CODE
```

Condition 4

The final example is one where you have consumed the data in the TOS register and you need refill it with the 2nd item on the stack. You commonly do this at the end of your Assembler word, just before you return to Forth with the NEXT, macro-instruction.

```
CODE: ON      ( adr -- )
            *TOS SETO,      \ set all bits in address in TOS to ones
            TOS POP,        \ Refill the TOS from data stack with POP,
            NEXT,
            END-CODE
```

Structured Branching and Looping

One of the cleverest things about the traditional Forth Assembler is the use of branching and looping the looks very much like regular Forth. In TI Assembler you have labels and your program jumps to those labels very much like using GOTO in BASIC. This is not cool for structured programming geeks.

Here are two pieces of Assembler code that create the same binary code:

```
CODE TEST1                                \ structured Forth assembler code
    BEGIN,
        R7 1000 ADDI,
        R7 R1 CMP,
        GTE IF,
        R6 R6 CLR,
        ENDIF,
        R8 R7 SUB,
    AGAIN,
ENDCODE

CODE TEST2                                \ un-structured Forth Assembler(not in CAMEL99)
@@1:    R7 1000 ADDI,    \ begin
        R7 R1 CMP,
        @@2 JL,        \ GTE if
        R6 R6 CLR,
@@2:    R8 R7 SUB,
        @@1 JMP,        \ again
ENDCODE
```

Notice we have BEGIN, AGAIN, for infinite loops and we have IF, ENDIF, for branching and yes there is an ELSE, clause as well.

The ASM9900 assembler also gives you BEGIN, UNTIL, loops and BEGIN, WHILE, REPEAT, loops.

These are not Indirect threaded code, but real machine instructions that assemble into your code.

For a little machine like the TI-99 this is an extremely powerful Assembler.

Example Programs

Random Color Dots

From the TI BASIC Reference Manual. This program shows off TI-BASIC'S math abilities by calculating real notes on the musical scale.

```
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)           ( N is the note value)
170 Y=110*(2^(1/12))^N        ( this calculates a musical note frequency)
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)

It does not calculate musical notes because it uses integer math not floating point math.

```
\ Random Color Dots
INCLUDE DSK1.RANDOM.F
INCLUDE DSK1.SOUND.F
INCLUDE DSK1.CHARSET.F
INCLUDE DSK1.GRAFIX.F
DECIMAL
: SET-COLORS ( -- )
  19 4 DO I I I COLOR LOOP ; \ Forth has different color sets

\ rather than use variables we make smart words with the same names
\ that calculate the numbers we need and leave them on the stack
: Y ( -- n ) 1000 RND 110 + ; \ does not calc. musical notes.
: CHR ( -- n ) 80 RND 32 + ;
: ROW ( -- n ) 23 RND ;
: COL ( -- n ) 31 RND ;

\ We create a SOUND word from primitives HZ DB MS MUTE
: SOUND ( dur freq att --) DB HZ MS MUTE ;

: RUN ( -- )
  RANDOMIZE
  CLEAR
  SET-COLORS
  BEGIN
    GEN1 100 Y -2 SOUND \ Select sound Generator #1
    COL ROW CHR 1 HCHAR
    ?TERMINAL \ check for the break key
  UNTIL
  8 SCREEN \ restore things like BASIC
  4 19 2 1 COLORS
  CHARSET ;
```


Guess the Number in Forth

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

```
INCLUDE DSK1.INPUT.F

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
    PAGE
    0 TRIES !
    10 RND 1+ ( -- rnd#) \ no variable, just leave on stack
    BEGIN
      ASK
      GET-GUESS
      REPLY
      1 TRIES +!
      OVER = UNTIL      \ loop until reply=rnd# on stack
    FINISH
    PLAYAGAIN?          \ pressing any key but "Y" gives a TRUE
    UNTIL               \ keeps going UNTIL stack is TRUE
  CR ." OK, thanks for playing!" ;
```

GRAPHICS Example: "Denile"

Original program was published by RETROSPECT, Atariage.com

```
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

This “literal” translation is for you to see equivalent statements in BASIC and Forth. It is not ideal use of the Forth language. See the next example for some Forth “style” changes to the code.

```
\ LITERAL TRANSLATION OF DENILE.BAS using variables
\ Original 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 FBEB 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,"0083C7AEFBEBBDF7")
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 ( slow Forth down)
  KEY? ( check if key pressed)
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.
;

```

A Note on Variables

Now that you have seen a literal translation from BASIC to Forth you can see how Forth handles variables with fetch and store (@,!) You might not like this as much as the algebraic notation in BASIC but you now understand how they work. VARIABLES in Forth and BASIC are very similar in that they are “GLOBAL”. This means that the program can reach them at any time. Global variables are not popular with computer scientists. In Forth with the code being so modular with small routines the risks are minimal. The only other real downside of using variables in Forth is that your code can only be used by one task if it uses variables unless you keep a separate copy of the variable for each task. (USER variables do this job for us)

Use More Words

A Forth solution to help manage tricky bits of code, is to use more WORDs. Small routines that are easy to understand and if done well make the code read more like a description.

Things to notice in this new version:

1. All the character patterns have been given descriptive names
2. We create words that print out just one character but they also have descriptive names
 - a. See: .BRICK .SAND etc...
3. The loop to draw the pyramid is now very simple because we created words to plot the characters on the screen by name and words to output a string of BRICK characters of any length.
 - a. See: BRICKS
4. The loop to make the river flow is very simple because we created words to increment a variable to the next character in the sequence but also to wrap back to beginning character automatically. This is what is meant by “extending the language”.
 - a. See: LIMIT 1+@ 2+@

Important Idea

Number four above is perhaps the most difficult thing for newbies to Forth to get used to. The overhead to call a sub-routine is quite low, by design, so that it makes sense to create small routines that do very simple things. These simple routines are difficult to mess up because they are so simple. Then we use those small routines as a “language” to solve the problem at hand. It takes a while to stop writing sub-routines that are one page long and start writing one line routines that work together as a specialty language.

Make small simple words that work together as a tiny language

```

\ FORTH STYLE TRANSLATION OF DENILE.BAS
\ Original program by RETROSPECT, Atariage.com

INCLUDE DSK1.GRAFIX.F
\ INCLUDE DSK1.STRING.S.F \ We don't strings for this version.
INCLUDE DSK1.CHARSET.F

\ Character DATA patterns are named in CAMEL99 Forth
HEX
0102 0709 1F24 7F92  PATTERN: RSLOPE
8040 E090 F824 FE49  PATTERN: LSLOPE
FF92 FF24 FF92 FF49  PATTERN: STONE
AA55 4489 2002 4801  PATTERN: SAND
0002 1735 7CFC 44AA  PATTERN: Camel
0008 081C 2A08 1414  PATTERN: LittleMan
0083 C7AE FBEB BDF7  PATTERN: WAVES1
0007 8F5D F7DF 7BEF  PATTERN: WAVES2
000E 1FBA EFBF F6DF  PATTERN: WAVES3
001C 3E75 DF7F EDBF  PATTERN: WAVES4
0038 7CEA BFFE DB7F  PATTERN: WAVES5
0070 F8D5 7FFD B7FE  PATTERN: WAVES6
00E0 F1AB FEFB 6FFD  PATTERN: WAVES7
00C1 E357 FDF7 DEFB  PATTERN: WAVES8

DECIMAL
: CHANGE-CHARS ( -- )
\ CHARDEF takes a defined pattern and the ascii number
\ We didn't have to but we used the word [CHAR] to make the code read easier.
\ [CHAR] is used in compiling. It is an IMMEDIATE WORD
\ It PARSES a Character from the input and returns the
\ ASCII number of the character.
    RSLOPE [CHAR] A CHARDEF
    LSLOPE [CHAR] B CHARDEF
    SAND [CHAR] C CHARDEF
    STONE [CHAR] D CHARDEF
    Camel [CHAR] E CHARDEF
    LittleMan [CHAR] F CHARDEF

    WAVES1 [CHAR] h CHARDEF
    WAVES2 [CHAR] i CHARDEF
    WAVES3 [CHAR] j CHARDEF
    WAVES4 [CHAR] k CHARDEF
    WAVES5 [CHAR] l CHARDEF
    WAVES6 [CHAR] m CHARDEF
    WAVES7 [CHAR] n CHARDEF
    WAVES8 [CHAR] o CHARDEF
;

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

\ the original version of PYRAMID used string CONCATENATION
\ to combine the characters needed to draw each row of stones
\ with the correct slope on each end.
\ String concatenation is slow because it copies strings in memory.

\ In Forth we should make some new words to do the job for us.
: .LSLOPE ( -- ) [CHAR] A EMIT ; \ print 1 left slope brick
: .RSLOPE ( -- ) [CHAR] B EMIT ; \ print 1 right slope brick
: .BRICK ( -- ) [CHAR] D EMIT ; \ print 1 BRICK

\ print n*2 BRICKS. If n=0 ?DO will do nothing.

```

```

\ we multitply n by 2 to create the pyramid rows
: BRICKS ( n -- ) 2* 0 ?DO .BRICK LOOP ;

: .SAND ( -- ) 0 23 [CHAR] C 32 HCHAR ;

: PYRAMID ( -- )
  12 0 \ The pyramid is 12 rows deep
  DO
    CR 15 I - TAB \ newline & tab to centre of screen
    .LSLOPE I BRICKS .RSLOPE \ print line of BRICKs
  LOOP
  CR .SAND
  CR \ make room for the river Nile
  CR
  CR .SAND ;

: .MAN ( -- ) 1 19 [CHAR] F 1 HCHAR ;
: .CAMEL ( -- ) 0 19 [CHAR] E 1 HCHAR ;

\ This is an example of Forth Style.
\ Factor out the common LIMITER routine so you can use it twice.
\
: LIMITER ( n -- n')
  DUP 111 > \ is n greater than ascii 111
  IF DROP 104 \ reset it to 104
  THEN ;

\ Forth Style:
\ Create custom variable incremeters, that limit themselves!
\
: 1+@ ( variable -- n ) \ INCR variable, limit to 104..111, return value
  DUP @ 1+ LIMITER DUP ROT ! ;

: 2+@ ( variable -- n ) \ INCR variable by 2, limit to 104..111
  DUP @ 2+ LIMITER DUP ROT ! ;

\ These two variables keep track of the wave characters
\ We could do this on the stack but it would be harder to understand
VARIABLE T
VARIABLE Y

: INIT-WAVES ( -- )
  103 T ! 104 Y ! ;

\ everytime this word runs the variables auto-increment but stay within 104..111
: .WAVES ( -- )
  0 21 T 1+@ 32 HCHAR
  0 22 Y 2+@ 32 HCHAR ;

\ Now the water flow loop is so simple
: FLOW ( -- )
  INIT-WAVES
  BEGIN
    .WAVES
    100 MS \ ms delays in ... milli-seconds ☺
    KEY?
  UNTIL ;

```

```

: RUN      ( -- )
           CLEAR
           12 SCREEN          \ light yellow screen for Sand ☺
           CHANGE-CHARS
           13 6 5 COLOR
           PYRAMID .CAMEL .MAN
           FLOW

           8 SCREEN           \ this resets the screen like BASIC
           4 19 2 1 COLORS    \ reset printable character's colors
           CHARSET            \ reset character patterns
;

```


ANS/ISO Forth Glossary

The following information is from AMERICAN NATIONAL STANDARD ANSI X3.215-1994 American National Standard for Information Systems. CAMEL99 contains all Forth CORE words except ENVIRONMENT? which is deprecated in Forth 2012 anyway. Some words are not in the KERNEL but can be INCLUDED into the system from a source code file. These words are so noted in their entry.

6.1 Core words

6.1.0010 **!**

store CORE

(x a-addr --)

Store x at a-addr.

See: [3.3.3.1](#) Address alignment

6.1.0030 **#**

number-sign CORE

(ud1 -- ud2)

Divide ud1 by the number in [BASE](#) giving the quotient ud2 and the remainder n. (n is the least-significant digit of ud1.) Convert n to external form and add the resulting character to the beginning of the pictured numeric output string. An ambiguous condition exists if # executes outside of a [<# #>](#) delimited number conversion.

See: [6.1.0050 #S](#)

6.1.0040 **#>**

number-sign-greater CORE

(xd -- c-addr u)

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

See: [6.1.0030 #](#) , [6.1.0050 #S](#) , [6.1.0490 <#](#)

6.1.0050 **#s**

number-sign-s CORE

(ud1 -- ud2)

Convert one digit of ud1 according to the rule for <#>. Continue conversion until the quotient is zero. ud2 is zero. An ambiguous condition exists if #S executes outside of a [<# #>](#) delimited number conversion.

```
6.1.0070 '  
tick CORE  
  ( "<spaces>name" -- xt )
```

Skip leading space delimiters. Parse name delimited by a space. Find name and return xt, the execution token for name. ~~An ambiguous condition exists if name is not found.~~

CAMEL99: When tick does not find the word is replies: * Not Found ?

When interpreting, ' **xyz EXECUTE** is equivalent to xyz.

See: [3.4](#) The Forth text interpreter, [3.4.1](#) Parsing, [A.6.1.2033 POSTPONE](#) ,
[6.1.2510 \['\]](#) , [A.6.1.0070 '](#) , [D.6.7](#) Immediacy.

```
6.1.0080 (  
paren CORE  
  Compilation: Perform the execution semantics given below.  
  Execution: ( "ccc<paren>" -- )
```

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

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

See: [3.4.1](#) Parsing, [11.6.1.0080 \(](#) , [A.6.1.0080 \(](#)

```
6.1.0090 *  
star CORE  
  ( n1|u1 n2|u2 -- n3|u3 )
```

Multiply n1|u1 by n2|u2 giving the product n3|u3.

```
6.1.0100 */  
star-slash CORE  
  ( n1 n2 n3 -- n4 )
```

Multiply n1 by n2 producing the intermediate double-cell result d. Divide d by n3 giving the single-cell quotient n4. An ambiguous condition exists if n3 is zero or if the quotient n4 lies outside the range of a signed number. If d and n3 differ in sign, the implementation-defined result returned will be the same as that returned by either the phrase **> R M* R> FM/MOD SWAP DROP** or the phrase **> R M* R> SM/REM SWAP DROP** .

See: [3.2.2.1](#) Integer division

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

```
( n1 n2 n3 -- n4 n5 )
```

Multiply $n1$ by $n2$ producing the intermediate double-cell result d . Divide d by $n3$ producing the single-cell remainder $n4$ and the single-cell quotient $n5$. An ambiguous condition exists if $n3$ is zero, or if the quotient $n5$ lies outside the range of a single-cell signed integer. If d and $n3$ differ in sign, the implementation-defined result returned will be the same as that returned by either the phrase `> R M* R> FM/MOD` or the phrase `> R M* R> SM/REM`.

See: [3.2.2.1](#) Integer division

6.1.0120 **+**

plus CORE

```
( n1|u1 n2|u2 -- n3|u3 )
```

Add $n2|u2$ to $n1|u1$, giving the sum $n3|u3$.

See: [3.3.3.1](#) Address alignment

6.1.0130 **+**!

plus-store CORE

```
( n|u a-addr -- )
```

Add $n|u$ to the single-cell number at $a\text{-}addr$.

See: [3.3.3.1](#) Address alignment

6.1.0140 **+LOOP**

plus-loop CORE

Interpretation: Interpretation semantics for this word are undefined.
Compilation: (C: do-sys --)

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

```
Run-time: ( n -- ) ( R: loop-sys1 -- | loop-sys2 )
```

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

See: [6.1.1240 DO](#) , [6.1.1680 I](#) , [6.1.1760 LEAVE](#) , [A.6.1.0140 +LOOP](#)

6.1.0150 ,
comma CORE
(x --)

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

See: [3.3.3](#) Data space, [3.3.3.1](#) Address alignment, [A.6.1.0150](#) ,

6.1.0160 -
minus CORE
(n1|u1 n2|u2 -- n3|u3)

Subtract n2|u2 from n1|u1, giving the difference n3|u3.

See: [3.3.3.1](#) Address alignment.

6.1.0180 .
dot CORE
(n --)

Display n in free field format.

See: [3.2.1.2](#) Digit conversion, [3.2.1.3](#) Free-field number display.

6.1.0190 ."
dot-quote CORE
Interpretation: Interpretation semantics for this word are undefined.
Compilation: ("ccc<quote>" --)

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

Run-time: (--)

Display ccc.

See: [3.4.1](#) Parsing, [6.2.0200 .\(](#) , [A.6.1.0190 ."](#)

6.1.0230 /
slash CORE
(n1 n2 -- n3)

Divide n1 by n2, giving the single-cell quotient n3. An ambiguous condition exists if n2 is zero. If n1 and n2 differ in sign, the implementation-defined result returned will be the same as that returned by either the phrase `> R S>D R> FM/MOD SWAP DROP` or the phrase `> R S>D R> SM/REM SWAP DROP` .

See: [3.2.2.1](#) Integer division

6.1.0240 **/MOD**

slash-mod CORE

(n1 n2 -- n3 n4)

Divide n1 by n2, giving the single-cell remainder n3 and the single-cell quotient n4. An ambiguous condition exists if n2 is zero. If n1 and n2 differ in sign, the implementation-defined result returned will be the same as that returned by either the phrase `> R S>D R> FM/MOD` or the phrase `> R S>D R> SM/REM` .

See: [3.2.2.1](#) Integer division

6.1.0250 **0<**

zero-less CORE

(n -- flag)

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

6.1.0270 **0=**

zero-equals CORE

(x -- flag)

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

6.1.0290 **1+**

one-plus CORE

(n1|u1 -- n2|u2)

Add one (1) to n1|u1 giving the sum n2|u2.

6.1.0300 **1-**

one-minus CORE

(n1|u1 -- n2|u2)

Subtract one (1) from n1|u1 giving the difference n2|u2.

6.1.0310 **2!**

two-store CORE

(x1 x2 a-addr --)

Store the cell pair x1 x2 at a-addr, with x2 at a-addr and x1 at the next consecutive cell. It is equivalent to the sequence **SWAP OVER ! CELL+ ! .**

See: [3.3.3.1](#) Address alignment

6.1.0320 **2***

two-star CORE

(x1 -- x2)

x2 is the result of shifting x1 one bit toward the most-significant bit, filling the vacated least-significant bit with zero.

See: [A.6.1.0320 2*](#)

6.1.0330 **2/**

two-slash CORE

(x1 -- x2)

x2 is the result of shifting x1 one bit toward the least-significant bit, leaving the most-significant bit unchanged.

See: [A.6.1.0330 2/](#)

6.1.0350 **2@**

two-fetch CORE

(a-addr -- x1 x2)

Fetch the cell pair x1 x2 stored at a-addr. x2 is stored at a-addr and x1 at the next consecutive cell. It is equivalent to the sequence **DUP CELL+ @ SWAP @ .**

See: [3.3.3.1](#) Address alignment, [6.1.0310 2!](#) , [A.6.1.0350 2@](#)

6.1.0370 **2DROP**

two-drop CORE

(x1 x2 --)

Drop cell pair x1 x2 from the stack.

6.1.0380 **2DUP**

two-dupe CORE

(x1 x2 -- x1 x2 x1 x2)

Duplicate cell pair x1 x2.

6.1.0400 **2OVER**

two-over CORE

(x1 x2 x3 x4 -- x1 x2 x3 x4 x1 x2)

Copy cell pair x1 x2 to the top of the stack.

6.1.0430 **2SWAP**

two-swap CORE

(x1 x2 x3 x4 -- x3 x4 x1 x2)

Exchange the top two cell pairs.

6.1.0450 :

colon CORE

(C: "<spaces>name" -- colon-sys)

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

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

Initiation: (i*x -- i*x) (R: -- nest-sys)

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

name Execution: (i*x -- j*x)

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

See: [3.4](#) The Forth text interpreter, [3.4.1](#) Parsing, [3.4.5](#) Compilation, [6.1.2500](#) [, [6.1.2540](#)] , [15.6.2.0470](#) ;CODE , [A.6.1.0450](#) : , [RFI 0005](#)
Initiation semantics.

6.1.0460 ;

semicolon CORE

Interpretation: Interpretation semantics for this word are undefined.
Compilation: (C: colon-sys --)

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

```
Run-time: ( -- ) ( R: nest-sys -- )
```

Return to the calling definition specified by nest-sys.

See: [3.4](#) The Forth text interpreter, [3.4.5](#) Compilation, [A.6.1.0460](#) ;

6.1.0480 <

less-than CORE

```
( n1 n2 -- flag )
```

flag is true if and only if n1 is less than n2.

See: [6.1.2340](#) U<

6.1.0490 <#

less-number-sign CORE

```
( -- )
```

Initialize the pictured numeric output conversion process.

See: [6.1.0030](#) # , [6.1.0040](#) #> , [6.1.0050](#) #S

6.1.0530 =

equals CORE

```
( x1 x2 -- flag )
```

flag is true if and only if x1 is bit-for-bit the same as x2.

6.1.0540 >

greater-than CORE

```
( n1 n2 -- flag )
```

flag is true if and only if n1 is greater than n2.

See: [6.2.2350](#) U>

6.1.0550 >BODY

to-body CORE

```
( xt -- a-addr )
```

a-addr is the data-field address corresponding to xt. An ambiguous condition exists if xt is not for a word defined via [CREATE](#).

See: [3.3.3](#) Data space, [A.6.1.0550 >BODY](#)

6.1.0560 **>IN**

to-in CORE

(-- a-addr)

a-addr is the address of a cell containing the offset in characters from the start of the input buffer to the start of the parse area.

6.1.0570 **>NUMBER**

to-number CORE

(ud1 c-addr1 u1 -- ud2 c-addr2 u2)

ud2 is the unsigned result of converting the characters within the string specified by c-addr1 u1 into digits, using the number in [BASE](#), and adding each into ud1 after multiplying ud1 by the number in BASE. Conversion continues left-to-right until a character that is not convertible, including any + or -, is encountered or the string is entirely converted. c-addr2 is the location of the first unconverted character or the first character past the end of the string if the string was entirely converted. u2 is the number of unconverted characters in the string. An ambiguous condition exists if ud2 overflows during the conversion.

See: [3.2.1.2](#) Digit conversion

6.1.0580 **>R**

to-r CORE

Interpretation: Interpretation semantics for this word are undefined.
Execution: (x --) (R: -- x)

Move x to the return stack.

See: [3.2.3.3](#) Return stack, [6.1.2060 R>](#) , [6.1.2070 R@](#) , [6.2.0340 2>R](#) , [6.2.0410 2R>](#) , [6.2.0415 2R@](#)

6.1.0630 **?DUP**

question-dupe CORE

(x -- 0 | x x)

Duplicate x if it is non-zero.

6.1.0650 **@**

fetch CORE

```
( a-addr -- x )
```

x is the value stored at a-addr.

See: [3.3.3.1](#) Address alignment

6.1.0670 **ABORT**

CORE

```
( i*x -- ) ( R: j*x -- )
```

Empty the data stack and perform the function of [QUIT](#), which includes emptying the return stack, without displaying a message.

See: [9.6.2.0670 ABORT](#)

6.1.0680 **ABORT"**

abort-quote CORE

Interpretation: Interpretation semantics for this word are undefined.
Compilation: ("ccc<quote>" --)

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

```
Run-time: ( i*x x1 -- | i*x ) ( R: j*x -- | j*x )
```

Remove x1 from the stack. If any bit of x1 is not zero, display ccc and perform an implementation-defined abort sequence that includes the function of [ABORT](#).

See: [3.4.1](#) Parsing, [9.6.2.0680 ABORT"](#) , [A.6.1.0680 ABORT"](#)

6.1.0690 **ABS**

abs CORE

```
( n -- u )
```

u is the absolute value of n.

6.1.0695 **ACCEPT**

CORE

```
( c-addr +n1 -- +n2 )
```

Receive a string of at most +n1 characters. An ambiguous condition exists if +n1 is zero or greater than 32,767. Display graphic characters as they are received. A program that depends on the presence or absence of non-graphic characters in the string has an environmental dependency. The editing functions, if any, that the system performs in order to construct the string are implementation-defined.

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

+n2 is the length of the string stored at c-addr.

See: [A.6.1.0695 ACCEPT](#)

6.1.0705 **ALIGN**
CORE

(--)

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

See: [3.3.3](#) Data space, [3.3.3.1](#) Address alignment, [A.6.1.0705 ALIGN](#)

6.1.0706 **ALIGNED**
CORE

(addr -- a-addr)

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

See: [3.3.3.1](#) Address alignment, [6.1.0705 ALIGN](#)

6.1.0710 **ALLOT**
CORE

(n --)

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

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

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

See: [3.3.3](#) Data space

6.1.0720 **AND**
CORE

(x1 x2 -- x3)

x3 is the bit-by-bit logical **and** of x1 with x2.

6.1.0750 **BASE**

CORE

(-- a-addr)

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

6.1.0760 **BEGIN**

CORE

Interpretation: Interpretation semantics for this word are undefined.
Compilation: (C: -- dest)

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

Run-time: (--)

Continue execution.

See: [3.2.3.2](#) Control-flow stack, [6.1.2140 REPEAT](#) , [6.1.2390 UNTIL](#) , [6.1.2430 WHILE](#) , [A.6.1.0760 BEGIN](#)

6.1.0770 **BL**

b-l CORE

(-- char)

char is the character value for a space.

See: [A.6.1.0770 BL](#)

6.1.0850 **C!**

c-store CORE

(char c-addr --)

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

See: [3.3.3.1](#) Address alignment

6.1.0860 **C,**

c-comma CORE

(char --)

Reserve space for one character in the data space and store char in the space. If the data-space pointer is character aligned when C, begins execution, it will remain character aligned when C,

finishes execution. An ambiguous condition exists if the data-space pointer is not character-aligned prior to execution of C,.

See: [3.3.3](#) Data space, [3.3.3.1](#) Address alignment.

6.1.0870 **C@**

c-fetch CORE

(c-addr -- char)

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

See: [3.3.3.1](#) Address alignment

6.1.0880 **CELL+**

CORE

(a-addr1 -- a-addr2)

Add the size in address units of a cell to a-addr1, giving a-addr2.

See: [3.3.3.1](#) Address alignment, [A.6.1.0880 CELL+](#)

6.1.0890 **CELLS**

CORE

(n1 -- n2)

n2 is the size in address units of n1 cells.

See: [A.6.1.0890 CELLS](#)

6.1.0895 **CHAR**

char CORE

("<spaces>name" -- char)

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

See: [3.4.1](#) Parsing, [6.1.2520 \[CHAR\]](#) , [A.6.1.0895 CHAR](#)

6.1.0897 **CHAR+**

char-plus CORE

(c-addr1 -- c-addr2)

Add the size in address units of a character to c-addr1, giving c-addr2.

See: [3.3.3.1](#) Address alignment

6.1.0898 **CHARS**

chars CORE

(n1 -- n2)

n2 is the size in address units of n1 characters.

6.1.0950 **CONSTANT**

CORE

(x "<spaces>name" --)

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

name is referred to as a **constant**.

name Execution: (-- x)

Place x on the stack.

See: [3.4.1](#) Parsing, [A.6.1.0950](#) **CONSTANT**

6.1.0980 **COUNT**

CORE

(c-addr1 -- c-addr2 u)

Return the character string specification for the counted string stored at c-addr1. c-addr2 is the address of the first character after c-addr1. u is the contents of the character at c-addr1, which is the length in characters of the string at c-addr2.

6.1.0990 **CR**

c-r CORE

(--)

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

6.1.1000 **CREATE**

CORE

("<spaces>name" --)

Skip leading space delimiters. Parse name delimited by a space. Create a definition for name with the execution semantics defined below. If the data-space pointer is not aligned, reserve

enough data space to align it. The new data-space pointer defines name's data field. CREATE does not allocate data space in name's data field.

```
name Execution: ( -- a-addr )
```

a-addr is the address of name's data field. The execution semantics of name may be extended by using [DOES>](#).

See: [3.3.3](#) Data space, [A.6.1.1000 CREATE](#)

6.1.1170 **DECIMAL**
CORE
(--)

Set the numeric conversion radix to ten (decimal).

6.1.1200 **DEPTH**
CORE
(-- +n)

+n is the number of single-cell values contained in the data stack before +n was placed on the stack.

6.1.1240 **DO**
CORE
Interpretation: Interpretation semantics for this word are undefined.
Compilation: (C: -- do-sys)

Place do-sys onto the control-flow stack. Append the run-time semantics given below to the current definition. The semantics are incomplete until resolved by a consumer of do-sys such as [LOOP](#).

```
Run-time: ( n1|u1 n2|u2 -- ) ( R: -- loop-sys )
```

Set up loop control parameters with index n2|u2 and limit n1|u1. An ambiguous condition exists if n1|u1 and n2|u2 are not both the same type. Anything already on the return stack becomes unavailable until the loop-control parameters are discarded.

See: [3.2.3.2](#) Control-flow stack, [6.1.0140 +LOOP](#) , [A.6.1.1240 DO](#)

6.1.1250 **DOES>**
does CORE
Interpretation: Interpretation semantics for this word are undefined.
Compilation: (C: colon-sys1 -- colon-sys2)

Append the run-time semantics below to the current definition. Whether or not the current definition is rendered findable in the dictionary by the compilation of DOES> is implementation defined. Consume colon-sys1 and produce colon-sys2. Append the initiation semantics given below to the current definition.

```
Run-time: ( -- ) ( R: nest-sys1 -- )
```

Replace the execution semantics of the most recent definition, referred to as name, with the name execution semantics given below. Return control to the calling definition specified by nest-sys1. An ambiguous condition exists if name was not defined with [CREATE](#) or a user-defined word that calls CREATE.

```
Initiation: ( i*x -- i*x a-addr ) ( R: -- nest-sys2 )
```

Save implementation-dependent information nest-sys2 about the calling definition. Place name's data field address on the stack. The stack effects i*x represent arguments to name.

```
name Execution: ( i*x -- j*x )
```

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

See: [A.6.1.1250 DOES>](#) , [RFI 0003](#) Defining words etc., [RFI 0005](#) Initiation semantics.

6.1.1260 **DROP**

CORE

```
( x -- )
```

Remove x from the stack.

6.1.1290 **DUP**

dupe CORE

```
( x -- x x )
```

Duplicate x.

6.1.1310 **ELSE**

CORE

```
Interpretation: Interpretation semantics for this word are undefined.  
Compilation: ( C: orig1 -- orig2 )
```

Put the location of a new unresolved forward reference orig2 onto the control flow stack. Append the run-time semantics given below to the current definition. The semantics will be incomplete

until orig2 is resolved (e.g., by [THEN](#)). Resolve the forward reference orig1 using the location following the appended run-time semantics.

```
Run-time: ( -- )
```

Continue execution at the location given by the resolution of orig2.

See: [6.1.1700 IF](#) , [A.6.1.1310 ELSE](#)

6.1.1320 **EMIT**

CORE

```
( x -- )
```

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

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

See: [6.1.2310 TYPE](#)

6.1.1345 **ENVIRONMENT?** NOT in CAMEL99

environment-query CORE

6.1.1360 **EVALUATE**

CORE

```
( i*x c-addr u -- j*x )
```

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

See: [7.6.1.1360 EVALUATE](#) , [A.6.1.1360 EVALUATE](#) , [RFI 0006](#) Writing to Input Buffers.

6.1.1370 **EXECUTE**

CORE

```
( i*x xt -- j*x )
```

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

See: [6.1.0070 ' , 6.1.2510 \['\]](#)

6.1.1380 **EXIT**

CORE

Interpretation: Interpretation semantics for this word are undefined.
Execution: (--) (R: nest-sys --)

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

See: [3.2.3.3](#) Return stack, [A.6.1.1380 EXIT](#)

6.1.1540 **FILL**

CORE

(c-addr u char --)

If u is greater than zero, store char in each of u consecutive characters of memory beginning at c-addr.

6.1.1550 **FIND**

CORE

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

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

See: [3.4.2](#) Finding definition names, [6.1.0070 ' , 6.1.2510 \['\]](#) , [A.6.1.1550 FIND](#) , [A.6.1.2033 POSTPONE](#) , [D.6.7](#) Immediacy.

6.1.1561 **FM/MOD** **LIBRARY FILE DSK1.FLOORED.F**

f-m-slash-mod CORE

(d1 n1 -- n2 n3)

Divide d1 by n1, giving the floored quotient n3 and the remainder n2. Input and output stack arguments are signed. An ambiguous condition exists if n1 is zero or if the quotient lies outside the range of a single-cell signed integer.

See: [3.2.2.1](#) Integer division, [6.1.2214 SM/REM](#) , [6.1.2370 UM/MOD](#) , [A.6.1.1561 FM/MOD](#)

6.1.1650 **HERE**

CORE

(-- addr)

addr is the data-space pointer.

See: [3.3.3.2](#) Contiguous regions

6.1.1670 **HOLD**

CORE

(char --)

Add char to the beginning of the pictured numeric output string. An ambiguous condition exists if HOLD executes outside of a [<# #>](#) delimited number conversion.

6.1.1680 **I**

CORE

Interpretation: Interpretation semantics for this word are undefined.

Execution: (-- n|u) (R: loop-sys -- loop-sys)

n|u is a copy of the current (innermost) loop index. An ambiguous condition exists if the loop control parameters are unavailable.

6.1.1700 **IF**

CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: -- orig)

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

Run-time: (x --)

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

See: [3.2.3.2](#) Control flow stack, [A.6.1.1700 IF](#)

6.1.1710 **IMMEDIATE**

CORE

(--)

Make the most recent definition an immediate word. An ambiguous condition exists if the most recent definition does not have a name.

See: [A.6.1.1710 IMMEDIATE](#) , [D.6.7](#) Immediacy, [RFI 0007](#) Distinction between immediacy and special compilation semantics.

6.1.1720 **INVERT**

CORE

```
( x1 -- x2 )
```

Invert all bits of x1, giving its logical inverse x2.

See: [6.1.1910 NEGATE](#) , [6.1.0270 0=](#) , [A.6.1.1720 INVERT](#)

6.1.1730 **J**

CORE

Interpretation: Interpretation semantics for this word are undefined.

Execution: (-- n|u) (R: loop-sys1 loop-sys2 -- loop-sys1 loop-sys2

)

n|u is a copy of the next-outer loop index. An ambiguous condition exists if the loop control parameters of the next-outer loop, loop-sys1, are unavailable.

See: [A.6.1.1730 J](#)

6.1.1750 **KEY**

CORE

```
( -- char )
```

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

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

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

See: [10.6.2.1305 EKEY](#) , [10.6.1.1755 KEY?](#)

6.1.1760 **LEAVE**

CORE

Interpretation: Interpretation semantics for this word are undefined.

Execution: (--) (R: loop-sys --)

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

See: [3.2.3.3 Return stack](#), [A.6.1.1760 LEAVE](#)

6.1.1780 **LITERAL**

CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (x --)

Append the run-time semantics given below to the current definition.

Run-time: (-- x)

Place x on the stack.

See: [A.6.1.1780 LITERAL](#)

6.1.1800 **LOOP**

CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: do-sys --)

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

Run-time: (--) (R: loop-sys1 -- | loop-sys2)

An ambiguous condition exists if the loop control parameters are unavailable. Add one to the loop index. If the loop index is then equal to the loop limit, discard the loop parameters and continue execution immediately following the loop. Otherwise continue execution at the beginning of the loop.

See: [6.1.1240 DO](#) , [6.1.1680 I](#) , [A.6.1.1800 LOOP](#)

6.1.1805 **LSHIFT**

l-shift CORE

(x1 u -- x2)

Perform a logical left shift of u bit-places on x1, giving x2. Put zeroes into the least significant bits vacated by the shift. An ambiguous condition exists if u is greater than or equal to the number of bits in a cell.

6.1.1810 **M***

m-star CORE

(n1 n2 -- d)

d is the signed product of n1 times n2.

See: [A.6.1.1810 M*](#)

6.1.1870 **MAX**

CORE

(n1 n2 -- n3)

n3 is the greater of n1 and n2.

6.1.1880 **MIN**

CORE

(n1 n2 -- n3)

n3 is the lesser of n1 and n2.

6.1.1890 **MOD**

CORE

(n1 n2 -- n3)

Divide n1 by n2, giving the single-cell remainder n3. An ambiguous condition exists if n2 is zero. If n1 and n2 differ in sign, the implementation-defined result returned will be the same as that returned by either the phrase >R S>D R> **FM/MOD DROP** or the phrase >R S>D R> **SM/REM DROP**.

See: [3.2.2.1](#) Integer division

6.1.1900 **MOVE** **LIBRARY FILE DSK1.MOVE.F**

CORE

(addr1 addr2 u --)

If u is greater than zero, copy the contents of u consecutive address units at addr1 to the u consecutive address units at addr2. After MOVE completes, the u consecutive address units at addr2 contain exactly what the u consecutive address units at addr1 contained before the move.

See: [17.6.1.0910 CMOVE](#) , [17.6.1.0920 CMOVE>](#) , [A.6.1.1900 MOVE](#)

6.1.1910 **NEGATE**

CORE

(n1 -- n2)

Negate n1, giving its arithmetic inverse n2.

See: [6.1.1720 INVERT](#) , [6.1.0270 0=](#)

6.1.1980 **OR**

CORE

(x1 x2 -- x3)

x3 is the bit-by-bit inclusive-or of x1 with x2.

6.1.1990 **OVER**

CORE

(x1 x2 -- x1 x2 x1)

Place a copy of x1 on top of the stack.

6.1.2033 **POSTPONE**

CORE

Interpretation: Interpretation semantics for this word are undefined.
Compilation: ("<spaces>name" --)

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

See: [D.6.7](#) Immediacy, [3.4.1](#) Parsing, [A.6.1.2033 POSTPONE](#) , [6.2.2530 \[COMPILE\]](#)

6.1.2050 **QUIT**

CORE

(--) (R: i*x --)

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

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

See: [3.4](#) The Forth text interpreter

6.1.2060 **R>**

r-from CORE

Interpretation: Interpretation semantics for this word are undefined.
Execution: (-- x) (R: x --)

Move x from the return stack to the data stack.

See: [3.2.3.3 Return stack](#), [6.1.0580 >R](#) , [6.1.2070 R@](#) , [6.2.0340 2>R](#) ,
[6.2.0410 2R>](#) , [6.2.0415 2R@](#)

6.1.2070 **R@**

r-fetch CORE

Interpretation: Interpretation semantics for this word are undefined.
Execution: (-- x) (R: x -- x)

Copy x from the return stack to the data stack.

See: [3.2.3.3 Return stack](#), [6.1.0580 >R](#) , [6.1.2060 R>](#) , [6.2.0340 2>R](#) ,
[6.2.0410 2R>](#) , [6.2.0415 2R@](#)

6.1.2120 **RECURSE**

CORE

Interpretation: Interpretation semantics for this word are undefined.
Compilation: (--)

Append the execution semantics of the current definition to the current definition. An ambiguous condition exists if RECURSE appears in a definition after [DOES>](#).

See: [6.1.2120 RECURSE](#) , [A.6.1.2120 RECURSE](#)

6.1.2140 **REPEAT**

CORE

Interpretation: Interpretation semantics for this word are undefined.
Compilation: (C: orig dest --)

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

Run-time: (--)

Continue execution at the location given by dest.

See: [6.1.0760 BEGIN](#) , [6.1.2430 WHILE](#) , [A.6.1.2140 REPEAT](#)

6.1.2160 **ROT**

rote CORE

```
( x1 x2 x3 -- x2 x3 x1 )
```

Rotate the top three stack entries.

6.1.2162 **RSHIFT**

r-shift CORE

```
( x1 u -- x2 )
```

Perform a logical right shift of u bit-places on x1, giving x2. Put zeroes into the most significant bits vacated by the shift. An ambiguous condition exists if u is greater than or equal to the number of bits in a cell.

6.1.2165 **S"**

s-quote CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: ("ccc<quote>" --)

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

```
Run-time: ( -- c-addr u )
```

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

See: [3.4.1](#) Parsing, [6.2.0855 C"](#) , [11.6.1.2165 S"](#) , [A.6.1.2165 S"](#)

6.1.2170 **S>D**

s-to-d CORE

```
( n -- d )
```

Convert the number n to the double-cell number d with the same numerical value.

6.1.2210 **SIGN**

CORE

```
( n -- )
```

If n is negative, add a minus sign to the beginning of the pictured numeric output string. An ambiguous condition exists if SIGN executes outside of a [<# #>](#) delimited number conversion.

s-m-slash-rem CORE

(d1 n1 -- n2 n3)

Divide d1 by n1, giving the symmetric quotient n3 and the remainder n2. Input and output stack arguments are signed. An ambiguous condition exists if n1 is zero or if the quotient lies outside the range of a single-cell signed integer.

See: [3.2.2.1](#) Integer division, [6.1.1561 FM/MOD](#) , [6.1.2370 UM/MOD](#) , [A.6.1.2214 SM/REM](#)

6.1.2216 **SOURCE**

CORE

(-- c-addr u)

c-addr is the address of, and u is the number of characters in, the input buffer.

See: [A.6.1.2216 SOURCE](#) , [RFI 0006](#) Writing to Input Buffers.

6.1.2220 **SPACE**

CORE

(--)

Display one space.

6.1.2230 **SPACES**

CORE

(n --)

If n is greater than zero, display n spaces.

6.1.2250 **STATE**

CORE

(-- a-addr)

a-addr is the address of a cell containing the compilation-state flag. STATE is true when in compilation state, false otherwise. The true value in STATE is non-zero, but is otherwise implementation-defined. Only the following standard words alter the value in STATE: : ([colon](#)), ; ([semicolon](#)), [ABORT](#), [QUIT](#), [:NONAME](#), [([left-bracket](#)), and] ([right-bracket](#)).

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

See: [3.4](#) The Forth text interpreter, [15.6.2.2250 STATE](#) , [A.6.1.2250 STATE](#) , [RFI 0007](#) Distinction between *immediacy* and *special compilation semantics*.

6.1.2260 **SWAP**

CORE

(x1 x2 -- x2 x1)

Exchange the top two stack items.

6.1.2270 **THEN**

CORE

Interpretation: Interpretation semantics for this word are undefined.
Compilation: (C: orig --)

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

Run-time: (--)

Continue execution.

See: [6.1.1310 ELSE](#) , [6.1.1700 IF](#) , [A.6.1.2270 THEN](#)

6.1.2310 **TYPE**

CORE

(c-addr u --)

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

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

See: [6.1.1320 EMIT](#)

6.1.2320 **U.**

u-dot CORE

(u --)

Display u in free field format.

6.1.2340 **U<**

u-less-than CORE

(u1 u2 -- flag)

flag is true if and only if u1 is less than u2.

See: [6.1.0480 <](#)

6.1.2360 **UM***

u-m-star CORE

(u1 u2 -- ud)

Multiply u1 by u2, giving the unsigned double-cell product ud. All values and arithmetic are unsigned.

6.1.2370 **UM/MOD**

u-m-slash-mod CORE

(ud u1 -- u2 u3)

Divide ud by u1, giving the quotient u3 and the remainder u2. All values and arithmetic are unsigned. An ambiguous condition exists if u1 is zero or if the quotient lies outside the range of a single-cell unsigned integer.

See: [3.2.2.1](#) Integer division, [6.1.1561 FM/MOD](#) , [6.1.2214 SM/REM](#)

6.1.2380 **UNLOOP**

CORE

Interpretation: Interpretation semantics for this word are undefined.
Execution: (--) (R: loop-sys --)

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

See: [3.2.3.3](#) Return stack, [A.6.1.2380 UNLOOP](#)

6.1.2390 **UNTIL**

CORE

Interpretation: Interpretation semantics for this word are undefined.
Compilation: (C: dest --)

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

Run-time: (x --)

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

See: [6.1.0760 BEGIN](#) , [A.6.1.2390 UNTIL](#)

6.1.2410 **VARIABLE**

CORE

("<spaces>name" --)

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

name is referred to as a **variable**.

name Execution: (-- a-addr)

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

See: [3.4.1 Parsing](#), [A.6.1.2410 VARIABLE](#)

6.1.2430 **WHILE**

CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: dest -- orig dest)

Put the location of a new unresolved forward reference orig onto the control flow stack, under the existing dest. Append the run-time semantics given below to the current definition. The semantics are incomplete until orig and dest are resolved (e.g., by [REPEAT](#)).

Run-time: (x --)

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

See: [A.6.1.2430 WHILE](#)

6.1.2450 **WORD**

CORE

(char "<chars>ccc<char>" -- c-addr)

Skip leading delimiters. Parse characters ccc delimited by char. An ambiguous condition exists if the length of the parsed string is greater than the implementation-defined length of a counted string.

c-addr is the address of a transient region containing the parsed word as a counted string. If the parse area was empty or contained no characters other than the delimiter, the resulting string has

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

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

See: [3.3.3.6](#) Other transient regions, [3.4.1](#) Parsing, [6.2.2008 PARSE](#) , [A.6.1.2450 WORD](#)

6.1.2490 **XOR**

x-or CORE

(x1 x2 -- x3)

x3 is the bit-by-bit exclusive-or of x1 with x2.

6.1.2500 **[**

left-bracket CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: Perform the execution semantics given below.

Execution: (--)

Enter interpretation state. [is an immediate word.

See: [3.4](#) The Forth text interpreter, [3.4.5](#) Compilation, [6.1.2540 \]](#) , [A.6.1.2500 \[](#)

6.1.2510 **[']**

bracket-tick CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: ("<spaces>name" --)

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

An ambiguous condition exists if name is not found.

Run-time: (-- xt)

Place name's execution token xt on the stack. The execution token returned by the compiled phrase ['] **x** is the same value returned by ' **x** outside of compilation state.

See: [3.4.1](#) Parsing, [6.1.0070 '](#) , [A.6.1.2033 POSTPONE](#) , [A.6.1.2510 \['\]](#) , [D.6.7](#) Immediacy.

6.1.2520 [CHAR]

bracket-char CORE

Interpretation: Interpretation semantics for this word are undefined.

Compilation: ("<spaces>name" --)

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

Run-time: (-- char)

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

See: [3.4.1](#) Parsing, [6.1.0895 CHAR](#) , [A.6.1.2520 \[CHAR\]](#)

6.1.2540]

right-bracket CORE

(--)

Enter compilation state.

See: [3.4](#) The Forth text interpreter, [3.4.5](#) Compilation, [6.1.2500 \[](#) ,
[A.6.1.2540 \]](#) ,

6.2 Core extension words

CAMEL99 Forth has a large number of the Core Extension words in the KERNEL and many others available as Library source code files.

6.2.0060 **#TIB**
number-t-i-b CORE EXT
(-- a-addr)

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

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

See: [A.6.2.0060 #TIB](#)

6.2.0200 **.(**
dot-paren CORE EXT
Compilation: Perform the execution semantics given below.
Execution: ("ccc<paren>" --)

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

See: [3.4.1](#) Parsing, [6.1.0190 ."](#) , [A.6.2.0200 .\(](#)

6.2.0210 **.R** **LIBRARY FILE DSK1.UDOTR.F**
dot-r CORE EXT
(n1 n2 --)

Display n1 right aligned in a field n2 characters wide. If the number of characters required to display n1 is greater than n2, all digits are displayed with no leading spaces in a field as wide as necessary.

See: [A.6.2.0210 .R](#)

6.2.0260 **0<>** **NOT IN THE SYSTEM**
zero-not-equals CORE EXT
(x -- flag)

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

6.2.0280 **0>**
zero-greater CORE EXT
(n -- flag)

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

6.2.0340 **2>R**

two-to-r CORE EXT

Interpretation: Interpretation semantics for this word are undefined.
Execution: (x1 x2 --) (R: -- x1 x2)

Transfer cell pair x1 x2 to the return stack. Semantically equivalent to **SWAP >R >R** .

See: [3.2.3.3 Return stack](#), [6.1.0580 >R](#) , [6.1.2060 R>](#) , [6.1.2070 R@](#) , [6.2.0410 2R>](#) , [6.2.0415 2R@](#) , [A.6.2.0340 2>R](#)

6.2.0410 **2R>**

two-r-from CORE EXT

Interpretation: Interpretation semantics for this word are undefined.
Execution: (-- x1 x2) (R: x1 x2 --)

Transfer cell pair x1 x2 from the return stack. Semantically equivalent to **R> R> SWAP** .

See: [3.2.3.3 Return stack](#), [6.1.0580 >R](#) , [6.1.2060 R>](#) , [6.1.2070 R@](#) , [6.2.0340 2>R](#) , [6.2.0415 2R@](#) , [A.6.2.0410 2R>](#)

6.2.0415 **2R@**

NOT IN THE SYSTEM

two-r-fetch CORE EXT

Interpretation: Interpretation semantics for this word are undefined.
Execution: (-- x1 x2) (R: x1 x2 -- x1 x2)

Copy cell pair x1 x2 from the return stack. Semantically equivalent to **R> R> 2DUP >R >R SWAP** .

See: [3.2.3.3 Return stack](#), [6.1.0580 >R](#) , [6.1.2060 R>](#) , [6.1.2070 R@](#) , [6.2.0340 2>R](#) , [6.2.0410 2R>](#)

6.2.0455 **:NONAME**

colon-no-name CORE EXT

(C: -- colon-sys) (S: -- xt)

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

The execution semantics of xt will be determined by the words compiled into the body of the definition. This definition can be executed later by using xt [EXECUTE](#).

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

Initiation: (i*x -- i*x) (R: -- nest-sys)

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

```
xt Execution: (  $i*x$  --  $j*x$  )
```

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

See: [A.6.2.0455 :NONAME](#) , [3.2.3.2](#) Control-flow stack.

6.2.0500 <>

not-equals CORE EXT

```
( x1 x2 -- flag )
```

flag is true if and only if x1 is not bit-for-bit the same as x2.

6.2.0620 ?DO

question-do CORE EXT

Interpretation: Interpretation semantics for this word are undefined.
Compilation: (C: -- do-sys)

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

```
Run-time: ( n1|u1 n2|u2 -- ) ( R: -- | loop-sys )
```

If $n1|u1$ is equal to $n2|u2$, continue execution at the location given by the consumer of do-sys. Otherwise set up loop control parameters with index $n2|u2$ and limit $n1|u1$ and continue executing immediately following ?DO. Anything already on the return stack becomes unavailable until the loop control parameters are discarded. An ambiguous condition exists if $n1|u1$ and $n2|u2$ are not both of the same type.

See: [3.2.3.2](#) Control-flow stack, [6.1.0140 +LOOP](#) , [6.1.1240 DO](#) , [6.1.1680 I](#) , [6.1.1760 LEAVE](#) , [6.1.2380 UNLOOP](#) , [A.6.2.0620 ?DO](#)

6.2.0700 **AGAIN**

CORE EXT

Interpretation: Interpretation semantics for this word are undefined.
Compilation: (C: dest --)

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

```
Run-time: ( -- )
```

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

See: [6.1.0760 BEGIN](#) , [A.6.2.0700 AGAIN](#)

6.2.0855 **C"** **LIBRARY FILE STRINGS.F (Renamed ")**

c-quote CORE EXT

Interpretation: Interpretation semantics for this word are undefined.
Compilation: ("ccc<quote>" --)

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

Run-time: (-- c-addr)

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

See: [3.4.1 Parsing](#), [6.1.2165 S"](#) , [11.6.1.2165 S"](#) , [A.6.2.0855 C"](#)

6.2.0873 **CASE** **LOAD FROM FILE DSK1.CASE.F**

CORE EXT

Interpretation: Interpretation semantics for this word are undefined.
Compilation: (C: -- case-sys)

Mark the start of the CASE ... [OF](#) ... [ENDOF](#) ... [ENDCASE](#) structure. Append the run-time semantics given below to the current definition.

Run-time: (--)

Continue execution.

See: [A.6.2.0873 CASE](#)

6.2.0945 **COMPILE,** **LOAD FROM FILE DSK1.COMPILE.F**

compile-comma CORE EXT

Interpretation: Interpretation semantics for this word are undefined.
Execution: (xt --)

Append the execution semantics of the definition represented by xt to the execution semantics of the current definition.

CAMEL99: This functionally the same a comma ‘,’ in CAMEL99 Forth

See: [A.6.2.0945 COMPILE,](#)

~~6.2.0970 CONVERT OBSOLETE. NOT IN CAMEL99~~

~~CORE EXT~~

~~(ud1 e-addr1 -- ud2 e-addr2)~~

~~ud2 is the result of converting the characters within the text beginning at the first character after e-addr1 into digits, using the number in [BASE](#), and adding each digit to ud1 after multiplying ud1 by the number in BASE. Conversion continues until a character that is not convertible is encountered. e-addr2 is the location of the first unconverted character. An ambiguous condition exists if ud2 overflows.~~

Note: ~~This word is obsolescent and is included as a concession to existing implementations. Its function is superseded by [6.1.0570](#) NUMBER.~~

See: [3.2.1.2](#) Digit conversion, ~~[A.6.2.0970 CONVERT](#)~~

6.2.1342 **ENDCASE** LOAD FROM FILE DSK1.CASE.F

end-case CORE EXT

Interpretation: Interpretation semantics for this word are undefined.
Compilation: (C: case-sys --)

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

Run-time: (x --)

Discard the case selector x and continue execution.

See: [A.6.2.1342 ENDCASE](#)

6.2.1343 **ENDOF** LOAD FROM FILE DSK1.CASE.F

end-of CORE EXT

Interpretation: Interpretation semantics for this word are undefined.
Compilation: (C: case-sys1 of-sys -- case-sys2)

Mark the end of the [OF](#) ... ENDOF part of the [CASE](#) structure. The next location for a transfer of control resolves the reference given by of-sys. Append the run-time semantics given below to the current definition. Replace case-sys1 with case-sys2 on the control-flow stack, to be resolved by [ENDCASE](#).

Run-time: (--)

Continue execution at the location specified by the consumer of case-sys2.

See: [A.6.2.1343 ENDOF](#)

6.2.1350 **ERASE** **LIBRARY FILE DSK1.ERASE.F**
CORE EXT
 (addr u --)

If u is greater than zero, clear all bits in each of u consecutive address units of memory beginning at addr .

~~6.2.1390 **EXPECT**~~
~~CORE EXT~~
~~(c-addr +n ---)~~

~~Receive a string of at most +n characters. Display graphic characters as they are received. A program that depends on the presence or absence of non-graphic characters in the string has an environmental dependency. The editing functions, if any, that the system performs in order to construct the string of characters are implementation-defined.~~

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

~~Store the string at c-addr and its length in [SPAN](#).~~

Note: This word is obsolescent and is included as a concession to existing implementations. Its function is superseded by [6.1.0695 ACCEPT](#).

See: [A.6.2.1390 EXPECT](#)

6.2.1485 **FALSE**
CORE EXT
 (-- false)

Return a false flag.

See: [3.1.3.1](#) Flags

6.2.1660 **HEX**
CORE EXT
 (--)

Set contents of [BASE](#) to sixteen.

6.2.1850 **MARKER** **LIBRARY FILE DSK1.MARKER.F, also in DSK1.TOOLS.F**
CORE EXT
 ("<spaces>name" --)

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

```
name Execution: ( -- )
```

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

See: [3.4.1](#) Parsing, [15.6.2.1580 FORGET](#) , [A.6.2.1850 MARKER](#)

6.2.1930 **NIP**

CORE EXT

```
( x1 x2 -- x2 )
```

Drop the first item below the top of stack.

6.2.1950 **OF** **LOAD FROM FILE DSK1.CASE.F**

CORE EXT

Interpretation: Interpretation semantics for this word are undefined.
Compilation: (C: -- of-sys)

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

```
Run-time: ( x1 x2 --     | x1 )
```

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

See: [6.2.0873 CASE](#) , [6.2.1342 ENDCASE](#) , [A.6.2.1950 OF](#)

6.2.2000 **PAD**

CORE EXT

```
( -- c-addr )
```

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

CAMEL99: PAD is memory HEX 26 bytes past the HERE (end of Forth dictionary)

See: [3.3.3.6](#) Other transient regions, [A.6.2.2000 PAD](#)

6.2.2008 **PARSE**

CORE EXT

```
( char "ccc<char>" -- c-addr u )
```

Parse ccc delimited by the delimiter char.

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

See: [3.4.1 Parsing](#), [A.6.2.2008 PARSE](#)

6.2.2030 **PICK**

CORE EXT

```
( xu ... x1 x0 u -- xu ... x1 x0 xu )
```

Remove u. Copy the xu to the top of the stack. An ambiguous condition exists if there are less than u+2 items on the stack before PICK is executed.

See: [A.6.2.2030 PICK](#)

~~6.2.2040 **QUERY**~~

~~CORE EXT~~

~~(---)~~

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

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

~~See: [A.6.2.2040 QUERY](#), [RFI 0006](#).~~

6.2.2125 **REFILL** **LIBRARY FILE DSK1.ANSFILES.F**

CORE EXT

```
( -- flag )
```

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

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

When the input source is a string from [EVALUATE](#), return false and perform no other action.

See: [7.6.2.2125 REFILL](#) , [11.6.2.2125 REFILL](#) , [A.6.2.2125 REFILL](#)

6.2.2148 **RESTORE-INPUT** NOT IN CAMEL99

CORE EXT

```
( xn ... x1 n -- flag )
```

Attempt to restore the input source specification to the state described by x1 through xn. flag is true if the input source specification cannot be so restored.

An ambiguous condition exists if the input source represented by the arguments is not the same as the current input source.

See: [A.6.2.2182 SAVE-INPUT](#)

6.2.2150 **ROLL**

CORE EXT

```
( xu xu-1 ... x0 u -- xu-1 ... x0 xu )
```

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

See: [A.6.2.2150 ROLL](#)

6.2.2182 **SAVE-INPUT** NOT IN CAMEL99

CORE EXT

```
( -- xn ... x1 n )
```

x1 through xn describe the current state of the input source specification for later use by [RESTORE-INPUT](#).

See: [A.6.2.2182 SAVE-INPUT](#)

6.2.2218 **SOURCE-ID**

source-i-d CORE EXT

```
( -- 0 | -1 )
```

Identifies the input source as follows:

SOURCE-ID	Input source
-1	String (via EVALUATE)
0	User input device (TI-99 Console)
1..8	File handle in CAMEL99

See: [11.6.1.2218 SOURCE-ID](#)

~~6.2.2240 **SPAN**~~

~~CORE EXT~~

~~(-- a-addr)~~

~~a-addr is the address of a cell containing the count of characters stored by the last execution of [EXPECT](#).~~

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

6.2.2290 **TIB**

t-i-b CORE EXT

(-- c-addr)

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

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

See: [A.6.2.2290 TIB](#) , [RFI 0006](#).

6.2.2295 **TO** **LIBRARY FILE DSK1.VALUES.F**

CORE EXT

Interpretation: (x "<spaces>name" --)

Skip leading spaces and parse name delimited by a space. Store x in name. An ambiguous condition exists if name was not defined by [VALUE](#).

Compilation: ("<spaces>name" --)

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

Run-time: (x --)

Store x in name.

Note: An ambiguous condition exists if either [POSTPONE](#) or [\[COMPILE\]](#) is applied to **TO**.

See: [13.6.1.2295 TO](#) , [A.6.2.2295 TO](#)

6.2.2298 **TRUE**

CORE EXT

(-- true)

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

See: [3.1.3.1](#) Flags, [A.6.2.2298 TRUE](#)

6.2.2300 **TUCK**

CORE EXT

(x1 x2 -- x2 x1 x2)

Copy the first (top) stack item below the second stack item.

6.2.2330 **U.R** **LIBRARY FILE DSK1.UDOTR.F**

u-dot-r CORE EXT

(u n --)

Display u right aligned in a field n characters wide. If the number of characters required to display u is greater than n, all digits are displayed with no leading spaces in a field as wide as necessary.

6.2.2350 **U>**

u-greater-than CORE EXT

(u1 u2 -- flag)

flag is true if and only if u1 is greater than u2.

See: [6.1.0540 >](#)

6.2.2395 **UNUSED** **LIBRARY FILE DSK1.TOOLS.F**

CORE EXT

(-- u)

u is the amount of space remaining in the region addressed by [HERE](#) , in address units.

6.2.2405 **VALUE** **LIBRARY FILE DSK1.VALUES.F**

CORE EXT

(x "<spaces>name" --)

Skip leading space delimiters. Parse name delimited by a space. Create a definition for name with the execution semantics defined below, with an initial value equal to x.

name is referred to as a **value**.

```
name Execution: ( -- x )
```

Place x on the stack. The value of x is that given when name was created, until the phrase **x TO name** is executed, causing a new value of x to be associated with name.

See: [3.4.1 Parsing](#), [A.6.2.2405 VALUE](#) , [6.2.2295 TO](#)

6.2.2440 **WITHIN**

CORE EXT

```
( n1|u1 n2|u2 n3|u3 -- flag )
```

Perform a comparison of a test value n1|u1 with a lower limit n2|u2 and an upper limit n3|u3, returning true if either (n2|u2 < n3|u3 and (n2|u2 <= n1|u1 and n1|u1 < n3|u3)) or (n2|u2 > n3|u3 and (n2|u2 <= n1|u1 or n1|u1 < n3|u3)) is true, returning false otherwise. An ambiguous condition exists if n1|u1, n2|u2, and n3|u3 are not all the same type.

See: [A.6.2.2440 WITHIN](#)

6.2.2530 **[COMPILE]** **REPLACED BY POSTPONE**

bracket-compile CORE EXT

Intrepretation: Interpretation semantics for this word are undefined.

Compilation: ("<spaces>name" --)

Skip leading space delimiters. Parse name delimited by a space. Find name. If name has other than default compilation semantics, append them to the current definition; otherwise append the execution semantics of name. An ambiguous condition exists if name is not found.

See: [3.4.1 Parsing](#), [6.1.2033 POSTPONE](#) , [A.6.2.2530 \[COMPILE\]](#)

6.2.2535 ****

backslash CORE EXT

Compilation: Perform the execution semantics given below.

Execution: ("ccc<eol>"--)

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

See: [7.6.2.2535 \](#) , [A.6.2.2535 \](#)

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 as a 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 created 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 DATA stack (to hold data, 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 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.

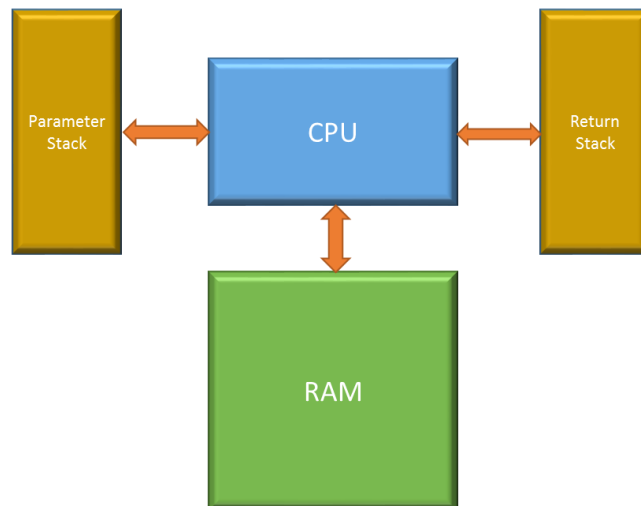
Forth is much older than Java but performs a similar function. In the writings about Chuck Moore, it is told that when IBM created the 360 mainframe, they struggled to get an operating system working built for it. (See the Mythical Man Month, by Fred Brooks (ISBN 0-201-00650-2) to see how many years 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.

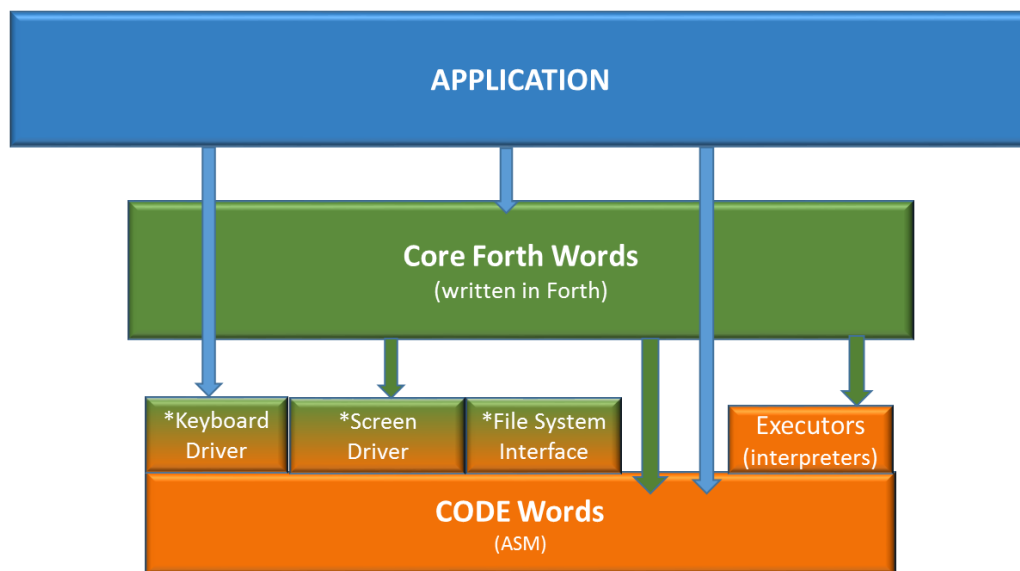
In typical Forth style however you are free to use either stack as your imagination sees fit. For example if you need temporary storage in the middle of a Forth word, you are free to push data onto the return stack as long as you clean it up before you end the word which is when it has to return from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

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. Assembler **Optionally** loaded into the Forth system so it can Assemble code
3. Editor For creating source code text files

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.

Alternative Usage (Future for CAMEL99)

Save the Binary image of the system so it can be loaded and started like an Assembly Language program

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
: JMYNUMS      ( INDEX - ADDR) CELLS  MYNUM
S + ;         OK
OK
OK
3 JMYNUMS @ . 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.