```
TEXAS INSTRUMENTS
HOME COMPUTER


READY-PRESS ANY KEY TO BEGIN


©1981   TEXAS INSTRUMENTS
```

Camel99

A Forth system for the TI-BASIC programmer

Brian Fox
brian.fox@brianfox.ca

Manual Revision 2.1
June 2021
Copyright © 2018, 2019,2020,2021

Table of Contents

Introduction

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

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

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

It is very important to understand that BASIC and Forth really take a different approach to how the computer should be presented to the human doing the programming. "Raw" Forth is closer to Assembly Language whereas BASIC protects the programmer from the hardware.

Fortunately Forth was designed to be changed. 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 lets you add things to it easily.

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 more 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 and change them to meet YOUR needs. 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

Camel Forth was created by Dr. Brad Rodriguez to be a transportable Forth system that complies with ANS/ISO Forth 94 language specification and 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 and so the name CAMEL99. Dr. Rodriguez is aware of CAMEL99 and his work in developing Camel Forth was essential in the creation of CAMEL99.

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

Brian Fox
Brian.fox@brianfox.ca
theBF@atariage.com

What you have

CAMEL99 is just less that 8K in size but it includes a Forth interpreter, a Forth compiler and the ability extend itself using source code files. The system 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. For the adventurous the cross compiler is available on GITHUB but it is not documented at this time and certainly is not useful to the new Forth student.

Program Files:

- CAMEL99
    - TI-99 "program file" (binary image) that can be loaded with the Editor/Assembler cartridge installed in the TI-99 computer.   DSK1.CAMEL99
    - CAMEL99 is a lean Forth Compiler/interpreter. In 8K bytes it has much of the ANS Forth CORE word set and it also has a good number words from the CORE Extension Word set.
    - Approximately 122 of the 315 Forth words are written in Forth Assembler for speed

Library Files/Language Extensions

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

File Format

CAMEL99 Forth assumes that all TI-99 source code files (programs in text form) are "DV80" format files which is the default format for the TI-99 file system. To open these files in BASIC you would write:

OPEN #1: "DSK1.ASM9900", INPUT, DISPLAY, VARIABLE 80, SEQUENTIAL

DV80 format is also 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

All Camel99 programs are text files called "SOURCE CODE". Source code files in TI-99 format have no extension but can have up to a 10 character filename.
On the GitHub repository there are also Forth source code files in PC format that have the extension ".FTH". The PC files should be the same as the TI-99 files. We apologize in advance for any discrepancies.

Using the PC library files

The .FTH files give you access to extra work we have done. If you use a Windows computer it is simple to:

1. Start the Classic99 Emulator
2. Select the Editor/Assembler cartridge
3. Start the Editor
4. Paste the .FTH file into the editor

5. Save the file in the TI Editor to a TI Disk of your choice

You can now use the file you saved just like any other TI file in the DSK1. library files provided.
Typically you load a file with:

       INCLUDE DSK1.MYFILE

Where the disk number is whatever disk the file is on and the file name of course is the file you want to load.

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 and is equivalent to 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 WORDS in a linked list called the dictionary.
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 and use. The DATA stack performs the same role as registers in Assembly Language
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 because you must learn all the internal details of the BASIC systems memory usage.	This is a Forth super power. Forth words can be written in Forth Assembler or machine code. You have to INCLUDE the ASSEMBLER program (DSK1.ASM9900) so that CAMEL99 can 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. You test Assembly language in Forth interactively like it was interpreted!
Colon Definition	Sub-routine, function	A word in Forth that is defined with the ':' (colon) operator. The colon turns on the compiler. It let's you make words that work 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 an identification number. That number is the "ADDRESS" of the CELL. In Forth you can specify the address in Decimal or HEX numbers and they are unsigned you you don't use non-sense negative addresses like -31876 as you see in BASIC.
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 BYTE or CHAR 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. All other features of the video chip are provided as "library" files that you load into the system when you need them.

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 DSK1.GRAFIX. The string functions that you have grown so fond of in BASIC are in the file DSK1.STRINGS. There is also a file called DSK1.INPUT that gives you something very similar to the BASIC input statement if you needed it. (You don't really)

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.

Starting CAMEL99

1. With your monitor and Peripheral Expansion box connected and turned on, insert the TI Editor/Assembler Cartridge into the TI-99 Console and turn on the console.

2. Press a key to start see the menu screen

```
     TEXAS INSTRUMENTS
       HOME COMPUTER
   PRESS
   1 FOR TI BASIC
   2 FOR EDITOR/ASSEMBLER
```
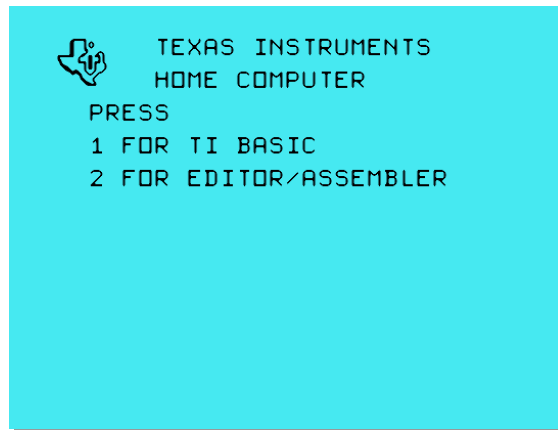
3. Select 2 For EDITOR/ASSEMBLER

4. Insert the diskette with CAMEL99 program and support files in disk drive 1. (DSK1.)

```
   * EDITOR/ASSEMBLER *

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



   ©1981   TEXAS INSTRUMENTS
```

1. Press 5 to RUN PROGRAM FILE

2. When the screen says FILE NAME?
   type DSK1.CAMEL99 on the TI-99 keyboard.

```
  * RUN PROGRAM FILE *



  FILE NAME?
  DSK1.CAMEL99
```
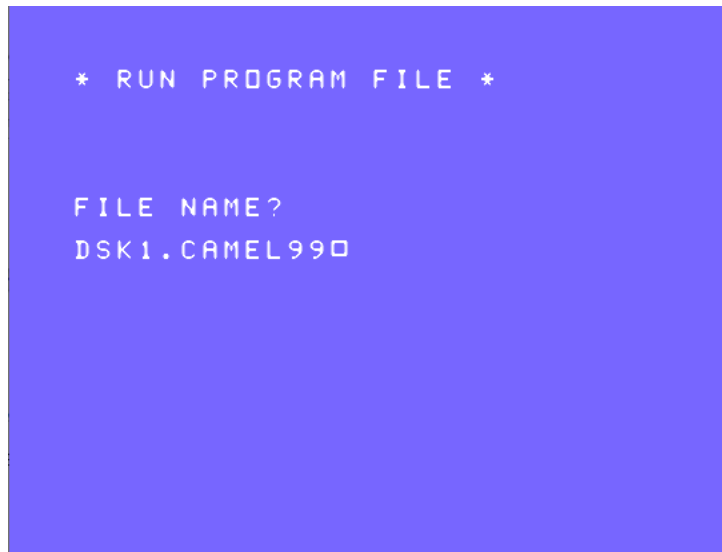
You should hear the disk drive come on. A green screen appears and loads more things and a new font and then the program screen similar to the one below should be on your screen after a few seconds.

```
      CAMEL99 Forth


  15714 Hi   RAM free
   8192 Low  RAM free
  10197 VDP  RAM free

  Ready
```
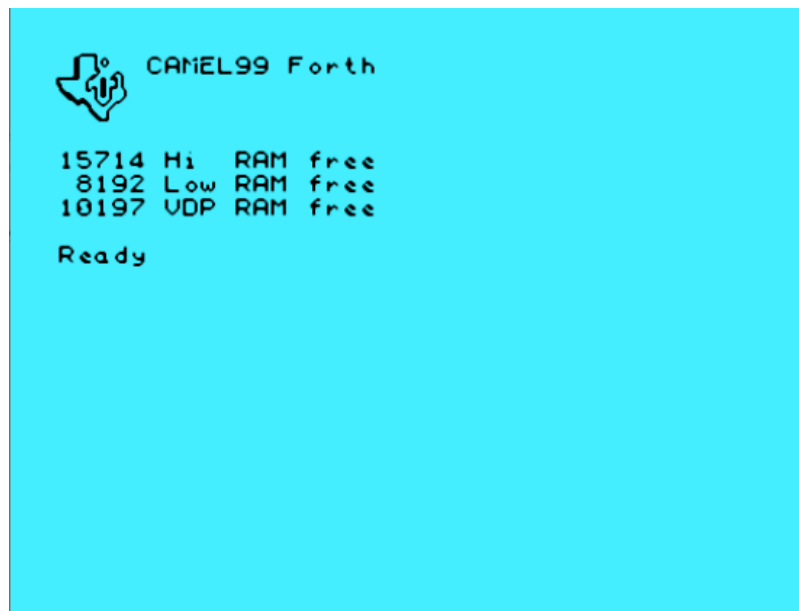
How Camel99 Forth Starts

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. When it starts Camel99 looks for a file called DSK1.START. This is like AUTOEXEC.BAT in MSDOS except that it is not written in a script language. It's just Forth. If you remove all the text or "comment out" the code in the DSK1.START file, Camel99 Forth will do nothing and just go to the Forth console.

<div align="center">

WARNING DO **NOT** EDIT DSK1.START UNTIL YOU KNOW FORTH WELL

</div>

The START File

Although you may be just learning Forth, we are going to drop you right into the language and show you the contents of the DSK1.START file. The START file is read by CAMEL99 Forth when it "boots up". This way you can make CAMEL99 Forth do almost anything by putting Forth commands in the START file.
Below is the default DSK1.START contents:

```
\ V2.1 START file loads NEEDS/FROM and then loads ANS Forth extensions

S" DSK1.SYSTEM" INCLUDED

HEX 17 7 VWTR   \ Changes the screen colours to 1 (black) on 7 (cyan)
```

The first line begins with a back-slash. This is a comment like a REM statement in BASIC. The line just describes what this file does and nothing more. Comments do NOT go into the TI-99 memory in Forth like they do in BASIC. Forth skips over them.

The next line starts a text string defined by S" and ends with a quote. The space after S" is not an error. It must be there. You will learn why later.

The string is the name of another file called DSK1.SYSTEM. This file is loaded by Forth with the command INCLUDED and gives CAMEL99 Forth some more ANS/ISO Standard Forth words and some extra words custom to CAMEL99 Forth.

The last line is changing the screen color by using the "VDP write to register" command (VWTR). It writes 1 (transparent foreground) and 7 (cyan background) to register 7 of the VDP chip in the TI-99. Congratulations! You now know some words in the Camel99 Forth programming language.

Creating Real Programs

In BASIC you make your program line by line using the internal line number editor. The BASIC LIST command lets you see your program text. You can type programs into the Forth at the console as well, but they compile into memory and then you lose the original text. For this reason in Forth, as with other compiled languages, we type our programs with an editor and save the text to a file. CAMEL99 at release time does not have an internal Editor. However there is pretty good Editor provided with the Editor Assembler package. Use this editor to create or change your programs. When you are ready, you will load your program with the INCLUDE command.

See the chapter called "THE TI EDITOR" for instructions on how to use the editor.

This section takes you through making programs in the editor, loading your programs into Forth and running your program.

The Forth Programming Pyramid

BASIC expects that you enter your programs using only the words already in the language.

You program in Forth by adding new WORDS to the existing language

Each word you add is like a sub-routine but much easier to use. You make new WORDS and you can, and should, test each word as you go. This makes your program code very solid and reliable. If you do it right you will find that you are writing your program in a tiny "language" that you created just for the job at hand. Your final project will look like the pyramid below.

ALL WORDS ARE IN THE FORTH "DICTIONARY"
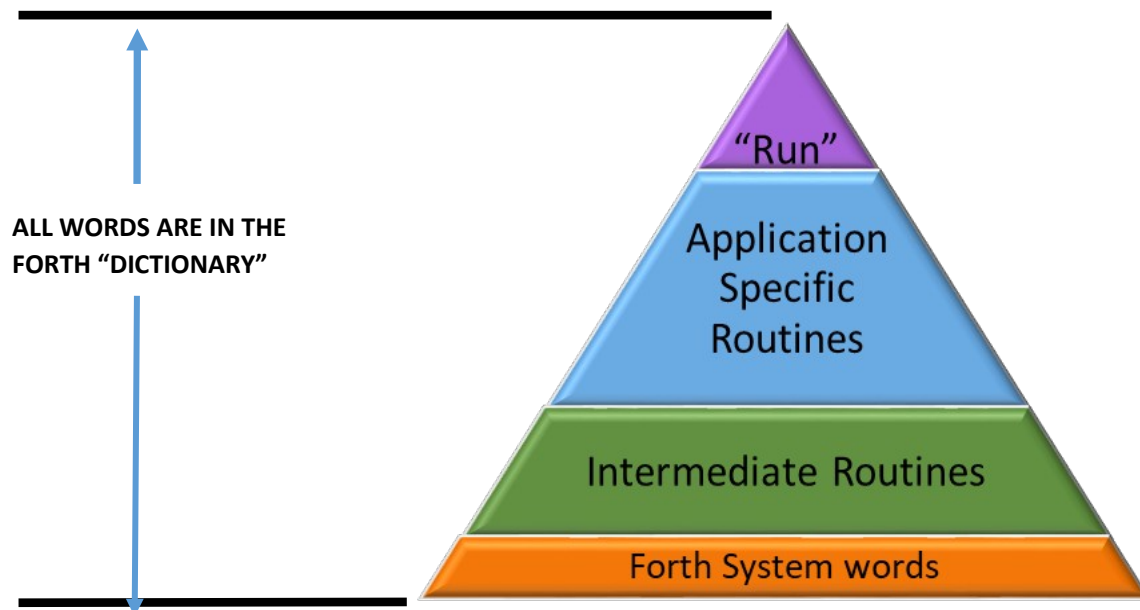
- Assembler CODE words are the foundation of the language that control the computer hardware
- The Forth language is normally written using a mix of those Assembler words and Forth words
- Your program's words are just more words added to the system. No different than the Language itself.
- All of this culminates in creating the final word that starts the entire program running which you can call anything you like. Call it RUN, and you will feel at home.

Program Process Overview

Here is how we make a program for CAMEL99 Forth:

1. Type the program text into the Editor/Assembler Editor
2. Save the program as a file on disk in the default DV80 format like Assembly language
3. Exit the editor program and start CAMEL99 Forth
4. INCLUDE the file you just saved. Example: INCLUDE DSK1.PROGRAM1 (this compiles the program)
5. If there are no errors in the compile, RUN your Forth program by typing the name you gave it in the program code

Define First/ Then USE it

For BASIC programmers this can take getting used to. We think nothing in BASIC of using a new variable name in the middle of a program. Forth can only understand a name that is in the Dictionary. If for example, you use a name for a variable that has not been defined beforehand, then Forth will HONK and show the name with a question mark. So, remember, nothing can be used that you have not explained to Forth beforehand. In the screen shot the variable X is "OK" because it was created by the line VARIABLE X but Forth asks "What the heck is Y"?

Incremental Compiling

Incremental compiling means that Forth lets you compile your program step by step.
This means you can INCLUDE a program file that is only part of your program. You can test all those words and leave them in memory. When you are happy with those words you can INCLUDE the next part of your program and test those words. This can continue until your "RUN" word is compiled that starts the whole program.

You 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.

The TI Editor

This section describes how to make a program with the Editor/Assembler Cartridge and the editor.

If you are familiar with the E/A editor you can skip this entire section*

CAMEL99 Author Note:

The TI-99 Editor is strange by today's standards in 2021 but it is a fully functional editor. These instructions from the E/A Manual explain how to use all the features of the editor.

[1]E/A Instructions

The cursor is a flashing marker that appears on the screen to indicate where your next keystroke appears. In editing, the cursor may be moved with the cursor movement keys described in Section 1 or by some of the choices in the command mode of the Editor. Before using the Editor/Assembler, be certain that all hardware is properly attached and turned on, with the Camel99 diskette in Disk Drive 1 and the Editor/Assembler module inserted in the console. If you have a TI-99/4A, it is advisable to depress the ALPHA LOCK key. After you select the module, the Editor/Assembler title appears at the top of the screen, followed by the five options as shown below.

Editor/Assembler Main Menu

To select an option, press the corresponding number key. At any time you may press <esc> to return to the previous screen or <quit> to return to the master title screen. The five Editor/Assembler options are discussed in the following sections.

```
      *  EDITOR/ASSEMBLER  *

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




      ©1981   TEXAS INSTRUMENTS
```

---

[1] This section is taken from the TI-99 Editor/Assembler Manual

Editor Menu


```
* EDITOR *
PRESS:
   1 TO LOAD
   2      EDIT
   3      SAVE
   4      PRINT
   5      PURGE
```

Editor Functions

1. TO LOAD loads an existing file into the computer's memory
2. EDIT to edit the file in memory;
3. SAVE to save a file from memory;
4. PRINT to print a file from the diskette; or
5. PURGE to delete the file in memory.

Load a File to Edit

A file on a diskette may be loaded for editing or printing. Any file stored in a fixed 80 display format or a variable 80 display format is accepted by the Loader. CAMEL99 files are always in Variable 80 Display format. (DV80) By saving your files in one of these formats, you may edit a list file or an object file, as well as a source file. However, a compressed object file cannot be edited since it contains unprintable characters.

Editing a File

The Editor allows you to load a previously existing file, to create or edit a file, to save a file that you have created or edited, to print a file, or to purge a file from the computer's memory.

From the main menu, if you press 1 for EDIT, you enter the Editor mode and the computer displays the following Menu.

Press 1 from this menu to load an existing file. If the Editor has not been loaded, the message ONE MOMENT PLEASE... is displayed on the screen briefly. Then the prompt FILE NAME? appears below the Menu.

If you have a single disk drive, remove the Editor/Assembler diskette and replace it with the diskette that contains the file. With two or three disk drives, place the program diskette in Disk Drive 2 or 3. Type the location and name of the file which you wish to edit, save, or print (such as DSK1.OLDFILE) and press <return>. (You may use the Disk Manager module to obtain a catalog of the files on your diskettes.)

The file is located and loaded into memory. The Editor Menu is then displayed and you may select another option.

Note: Each time a file is loaded, the previous file is removed from memory.

Edit

The Edit menu option loads the Editor from the Editor/Assembler diskette. The Editor allows you to create a new file or to edit a file which has been loaded with the Load option. When you enter the Editor by pressing 2 from the Editor Menu, the message "ONE MOMENT PLEASE..." is briefly displayed on the screen while the Editor is loaded from the Editor/Assembler diskette.

 (If the Editor has already been loaded, this message does not appear.)

If no file has been loaded, the Edit option clears the screen so that you may begin a new file. The cursor is positioned in the upper left corner of the screen and is followed by the end-of-file marker (*Em). Press <return> to create a new line. The Editor is now ready to accept your new input.

If a file has been loaded into memory, the Editor displays that file on the screen with the cursor at the top left, ready for you to edit it. You may leave the Editor and return to the Editor Menu by

Remember to SAVE

Note: The file in memory, whether it is a new file or an existing file, may be lost if you leave the Editor without saving it. Before returning to the Editor/Assembler Menu, be sure to save your program.

The Editor has two modes:

There is a command mode and the edit mode. You are in the edit mode when you first enter the Editor. The command mode is entered from the edit mode by pressing the <esc> key. The edit mode is re-entered automatically after you use a command in the command mode. In the edit mode, the cursor shows where your next keystroke is placed. In the command mode, the cursor is on the second line of the screen ready to accept commands. The command that you enter is effective starting from the position the cursor had when you entered the command mode.

80 Columns in 40 Column Chunks

In the edit mode, the screen is 80 columns wide with three overlapping 40 column windows available for displaying the text. You start in the left-most window with columns 1 through 40 displayed. Pressing <next-window> moves the display to the center window, with columns 21 through 60 displayed. Pressing <next-window> again moves the display to the right window with columns 41 through 80 displayed.

Pressing <next-window> at this point returns the display to the left-most window. The edit mode allows you to create, modify, and add text to program, data, and text files. When you press a key, that character is placed on the screen in the cursor position and the cursor moves one position to the right. (If the cursor is at the right margin, it moves to the first position on the next line.) In addition, the edit mode has several special keys which perform helpful edit functions.

The following table describes the special function keys that are used in the Editor.

Function KEY	Description
<return>	Enters the text into the edit buffer and places the cursor at the start of the next line. If <return> is pressed at the end of the file, a blank line is automatically inserted
<insert line>	Inserts a blank line above the line where the cursor is located.
<delete line>	Deletes the current line of text, starting at the location of the cursor. Inserts all characters typed until another function or cursor movement key is pressed. The following characters on the line are moved to the right. The insertion is effective for one line only with all characters after column 80 lost.
<Delete Char>	Deletes the character under the cursor. The following characters on the line are moved to the left.
<TAB>	Moves the cursor right to the next tab location. The tab locations are set at defaults of 1, 8, 13, 26, 31, 46, 60, and 80. To change the tab locations, use the T(AB command in the command mode. If you press <tab> from column 80, the cursor goes to position 1 on the same line.
<Next-window>	Moves the cursor position right to the next window so that you may view different portions of the text. If you press <next-window> from the right-most window, the left-most window is displayed.
<Scroll-Up>	Scrolls the screen up by 24-line segments in the edit mode. In the command mode, <roll-up> scrolls the screen by 22-line segments.
<Scroll-Down>	Scrolls the screen down by 24-line segments in the edit mode. In the command mode, <roll-down> scrolls the screen by 22-line segments.
<left-arrow> <right-arrow>	Allow cursor movement to the left or right without changing the text. When the cursor is at the left margin, pressing <left-arrow> alternately shows and removes the line numbers.
<Esc>	Invokes the command mode when in the edit mode. From the command mode, pressing <esc> returns you to the Editor Menu.
<Left-Arrow> (cursor in column 1)	The current line number may be displayed and removed by pressing the <left-arrow> key when the cursor is at the left margin. When the line numbers are displayed, the last six characters of the 80-column display cannot be viewed.

.

Editor Command Mode

The command mode, which provides additional editing features, is accessed from the edit mode by pressing <esc>. The command mode uses the first two lines of the screen for prompt lines and your input, with the remainder of the screen displaying your file. If an error is detected, the message ERROR appears in the left-hand corner of the second line. Because most of the commands use line numbers, the command mode automatically shows the line numbers. The cursor is displayed on the second line for command input.

The command mode prompt line shows the following prompts on a single line at the top of the screen. The commands are selected by pressing the first letter of the desired command. All of the edit mode function keys, except <insert-line>, <delete-line>, <up-arrow>, and <down-arrow>, are also effective in the command mode. The <esc> key returns you to the Editor Menu.

The effects of all of the commands except M(OVE, I(NSERT, C(OPY, and D(ELETE start from the position of the cursor when you entered the command mode. The commands E(DIT, A(DJUST, and H(0ME occur when you press the letter to choose those commands. The other commands require more information, and occur after that information is entered and <return> is pressed.

The following gives the command mode prompts, in the order in which they appear in the prompt line, and describes their functions.

E(DIT

 Returns you to the edit mode, with the display as it was before you entered the command mode and the cursor at its previous position.

F(IND)

Enables you to find a string. The prompt-line FIND <CNT>(<COL,COL>)/STRING/ appears on the top line of the screen. You may specify an optional count number, from 1 through 9999, and optional beginning and ending column numbers from 1 through 80. The count number specifies which occurrence of the string is to be found. If omitted, the default is 1. The two column numbers specify the columns within which the search is to be made. The column numbers must be preceded and followed by parentheses. If the column numbers are omitted, the entire line, columns 1 through 80, is searched. The string must be delimited by slashes (1). The following examples demonstrate the use of F(IND.

Example	*Result*
/HELLO/	Finds the first occurrence of HELLO.
1000/HELLO/	Finds the 1000th occurrence of HELLO.
(1,50)/ HELLO/	Finds the first occurrence of HELLO in columns 1 through 50.
1000(1,50)/HELLO/	Finds the 1000th occurrence of HELLO in columns 1 through 50.
101/ /	Finds space #101

After the string is located, the Editor leaves the command mode and returns to the edit mode. The string is displayed in line 1 with the cursor located on the first character of the string.

R(EPLACE

Replaces the given string with a new string. The prompt-line appears at the top of the screen. The count number specifies which occurrence of the string is to be found. If omitted, the default is 1. The two column numbers specify the columns within which the search is to be made. The column numbers must be preceded and followed by parentheses. If the column numbers are omitted, the entire line, columns 1 through 80, is searched. The old string and new string are entered with slashes delimiting them. After you press <return>, the replacement process begins.

If V (for verify) is specified, the prompt: REPLACE STRING (Y/N/A) is displayed followed by the string. To replace that occurrence of the string, press Y. Press N if you do not want to replace the string in that location. The next occurrence of the string is then located (if a count of more than one was specified) and the prompt is again presented. To replace all subsequent occurrences of the specified string, press A.

The following demonstrate the use of R(EPLACE.

Example   *Result*

lOOO/HELLO/GOODBYE/ Changes the first 1000 occurrences of HELLO to GOODBYE.

V,PO/HELLO---/BYE/

Presents, one at a time, the first 20 occurrences of HELLO---. You may change them to BYE by pressing Y, go on to the next one without changing that one by pressing N, or change all subsequent ones by pressing A.

M(OVE Displays the prompt-line:

MOVE START LINE, STOP LINE, AFTER LINE?

at the top of the screen. The first value you enter specifies the line number of the beginning of the section to be moved. The second value specifies the line number of the end of the section to be moved. The third value specifies the line after which you want to place the section being moved. For example, if you specify 29 as the AFTER LINE, the data is moved to line 30.

A maximum of a four-digit line number may be specified. However, if the line number is greater than the EOF marker, the line number defaults to the Eff. The ECF line number may be specified by entering E as the starting line, stopping line, or after line. Line number 0 indicates the line above line number 1.

When the move is complete, the line numbers are automatically renumbered. The following examples demonstrate the use of M(0VE.

Example   *Result*

1,51,57   Moves line 1 through 51 to a position after line 57.

Moves lines 452 through the end of the file to the beginning of the file.

S(H0W

Shows the lines starting at the line specified. The prompt-line SHOW LINE? appears. You may respond with a line number or E (to see the line at the end of the file). For example, if you enter 30, the text on line number 30 and all subsequent text is displayed beginning at the top of the screen. The cursor is located on the first character in line number 30.

C(0PY

Uses the same prompt-line and functions in the same manner as M(0VE. However, C(0PY does not delete lines; it places a copy of the designated data at the desired location.

I(NSERT (file)

Allows insertion of a file from a diskette before a specified line number. The prompt-line INSERT BEFORE LINE, FILE NAME? requires a line number (four-digit maximum) and the name of the file.
For example, 29, DSK2.OLDFILE inserts the file OLDFILE from the diskette in Disk Drive 2 to the file you are editing before line 29.

D(ELETE

Deletes the desired text. The text to be deleted is specified as in the M(OVE command. The prompt DELETE START LINE, STOP LINE? requires the entry of the beginning and ending line numbers for the deletion.

A(DJUST

Returns you to the edit mode. Changes whether numbers are shown. This allows you to see the last six columns of text or data. If the cursor is located in columns 75 through 80, you must first move it to one of the other columns before selecting A(DJUST to leave the line number mode. T(AB Modifies and sets tabs. The Editor has default tabs at columns 1, 8, 13, 26, 31, 46, 60, and 80. When you choose this command, the top line of the screen displays column numbers (123456789 123456789 ...). The second line has a T located below each of the columns where tab positions are located. Press <space> or <tab> to go to the location where a tab is desired and type T. You may remove tabs by replacing a T with a space. To adjust tabs in columns 75 through 80, tab to column 80 and backspace to the position desired. The tab settings return to the defaults when the Editor is reloaded. Note: Do not delete the tab at column 80.

H(0ME

Moves the cursor to the upper left-hand corner of the screen.

Save

After you have edited a file, it must be saved on diskette for future use. Otherwise, when you leave the Editor, the file may be lost. You save a file by pressing 3 from the Editor Menu.
The prompt VARIABLE 80 FORMAT (Y/N)? appears at the bottom of the screen.
If Y is pressed, a file is opened with a variable 80 display format, which uses less space on the diskette than a fixed 80 display format. If you press N, a fixed 80 display format is used. After a file is saved on diskette, its format cannot be changed unless the file is reloaded into memory and saved again in the new format. After the format is chosen, the prompt FILE NAME? is displayed.

If you have a single disk drive, remove the Editor/Assembler diskette and replace it with the diskette that contains the file. If you have two or three disk drives, place the program diskette in Disk Drive 2 or 3.
To save your file on a diskette, enter the device and filename. For example, DSK1.SAVEFILE saves your file on the diskette in Disk Drive 1 under the name SAVEFILE. After you save your file, be sure that the Editor/Assembler diskette is in Disk Drive 1.
You may also save your file to the RS232 by specifying RS232 as the filename. The output is then directed to the device connected to the RS232, which is normally a printer. When outputting to the RS232, you must specify a file that is in variable 80 format.

You may wish to use the print option to print a file instead of the save option. However, outputting is faster with SAVE than with the PRINT option. The TI Thermal Printer may not be used with the save option. It is only accessible from the print option.

Print

The print option allows you to print a file to the RS232 Interface, PIO, the Thermal Printer, or a diskette file. A source, list, object, or any other file in either a variable 80 display format or a fixed 80 display format can be printed. A printed compressed object file may appear somewhat confusing because it contains unprintable characters. Select the print option by pressing 4 on the Editor Menu.
The prompt: FILE NAME? appears on the screen.

If you have a single disk drive, remove the Editor/Assembler diskette and replace it with the diskette that contains the file. With two or three disk drives, place the program diskette in Disk Drive 2 or 3. Enter a filename, such as DSK1.OLDFILE. The file must be on a diskette.
After you enter the filename, the prompt DEVICE NAME? appears.

A diskette file, RS232, PIO or TP may be specified as a device name.

If the diskette is specified (to duplicate a file) the entire diskette filename, such as DSKl.PRNTFILE, must be entered. The output file is in variable 80 format, so an object file duplicated with the print option cannot be loaded by the Loader. After you specify the device, the file is printed on that device. The print option does not require that the Editor be in memory. If the Editor is in memory, the print option does not alter the text being edited, so you may continue to edit after you use the print option.
After you have printed your file, be sure that the Editor/Assembler diskette is in Disk Drive 1.

Purge

The purge option allows you to remove the file currently in memory. After you select the purge option by pressing 5 from the Editor Menu, the prompt: ARE YOU SURE (Y/N)? appears at the bottom of the screen.

If you press Y, the file is cleared from memory and is no longer accessible. If you press N, the file remains in memory and you are returned to the Editor Menu. You would normally save your file prior to purging it from memory unless you truly do not want to keep it.

```
   * EDITOR/ASSEMBLER *

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



    ©1981  TEXAS INSTRUMENTS
```

You normally only purge a file when you wish to create a new file.

Lesson 1: Forth Hello World Program (Quick Starter)

Select the Editor from Editor Assembler Menu (Press 1)

```
 * EDITOR/ASSEMBLER *

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



  ©1981   TEXAS INSTRUMENTS
```

```
\ MY FIRST FORTH PROGRAM
: RUN  CR ." Hello World!"  ;
      *EOF (VERSION 1.2)
```

Then press 2 to EDIT and type in the program on the right panel.

```
 * EDITOR *
PRESS:
  1 TO LOAD
  2     EDIT
  3     SAVE
  4     PRINT
  5     PURGE
```

Press FCNT BACK twice to get to the menu (Escape on PC keyboard)

Press 3 to save the file in VAR 80 format

Press BACK, Press 5 to RUN DSK1.CAMEL99

INCLUDE the program you just saved

Type the name of your program (RUN)

Congratulations! You just wrote, compiled and ran a program in Forth
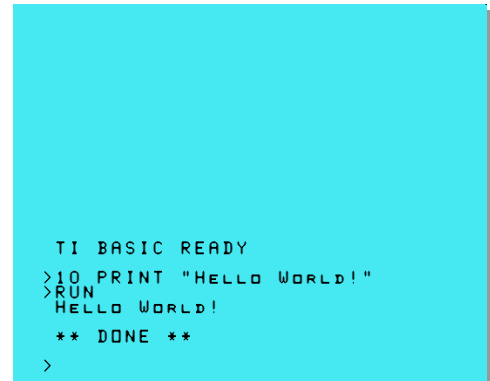
Compare Hello World In BASIC and FORTH

Consider the one-line program in BASIC in the screen image.

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.

```
 TI BASIC READY
>10 PRINT "Hello World!"
>RUN
 Hello World!
 ** DONE **
>
```
Hello world in BASIC

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 after ."
AND another thing: CAMEL99 is case sensitive. **RUN** is not the same as **Run** or **rUN.** This is done on purpose so you can create your own program names using upper or lower case and have a wider range of names. Not all Forth's do this.

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.

```
GRAPHICS 1 Mode READY ok
: RUN  CR ." Hello World!" ;    o
k
RUN
Hello World! ok
```
CAMEL99 Forth Hello World Program

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.

Loading a Library File

You might notice that the Forth screen has 40 columns. This the TEXT mode of the TMS9918 chip. TI-BASIC starts in Graphics 1 mode. Graphics mode lets you change character colors which cannot be done in TEXT mode. Can we switch CAMEL99 Forth to Graphics mode? Yes! We use a library file.

To load the GRAPHICS mode words into the system, with Forth running, we just have to INCLUDE the file we need.

In FORTH type: **INCLUDE DSK1.GRAFIX <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 colouring.

That's it. You have loaded a library file.
Camel99 has Library files for files, sprites, strings and many things that will assist you in writing programs.
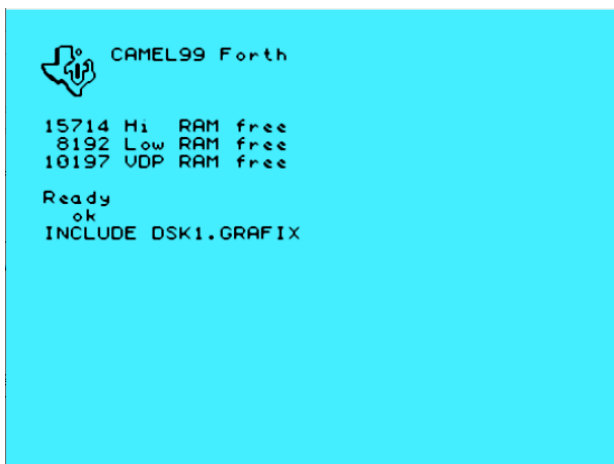
The Colon Compiler

The first thing on the line of our Hello World 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 a space and then the next characters as a string until you get to a 2nd 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" every time it is invoked.

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 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. For 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 '$' 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.

Compiling Gotchas

So now we know about the colon compiler. Forth interprets (ie: executes) everything we type or send via a file until it encounters a colon character. Everything between the colon and semi-colon is compiled, right? Well... mostly.

As mentioned the semi-colon is an immediate word. It doesn't care if the compiler is turned on it runs anyway. That's how it can turn the compiler off. There are other words that are immediate as well. The comment "word" (everything is a word in Forth) is backslash. It is also immediate. If we do this:

```
: TESTING   \            ;
```

The word testing will not be completed because everything after '\' is ignored.

There is also the bracket comment word '('.   If we write this:

```
: TESTING2  ( ignore this )   ;
```

TESTING2 will compile successfully but the words "ignore this" are skipped over because '(' is an immediate word and it just reads everything and throws it away until it finds ')'.   The new word TESTING2 will be in the dictionary but it doesn't do anything.

And one more thing...

Some other words can COMPILE words into the dictionary by themselves: (Honest)

VARIABLE X   compiles a new variable with the name X into the dictionary.
17 CONSTANT AGE   compiles 17 with the name AGE into the dictionary as a constant.
CREATE TOTO   compiles the name TOTO into the dictionary and will return its address

So these are also "compiling" words and there are a few others you will learn about.

YOU CANNOT COMPILE COMPILING WORDS INSIDE A COLON DEFINITION

This will not work:

```
: BAD-DOG    VARIABLE X   VARIABLE Y  CR ." This won't compile" ;
```

Do it this way:

```
VARIABLE X
VARIABLE Y   ( Put compiling words outside of the colon definition )

: GOOD-BOY   CR ." Now you can use X and Y if you know how to use variables. :)" ;
```

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 some text 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. Dot-quote 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, dot-quote only works while compiling a new definition. In CAMEL99 dot-quote checks the STATE of the system (a variable) and compiles if it should compile and interprets if it should interpret.
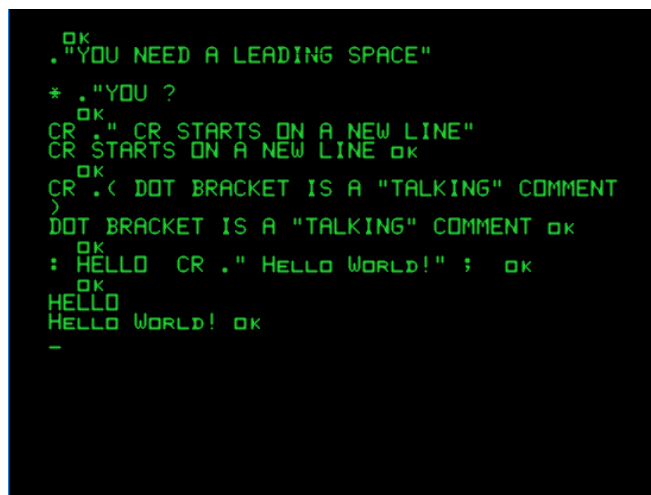
Some Dot-quote "Gotchas"

```
: BAD   ."This will abort with an error" ;

     ^^^^^
```

ALL Forth words are separated by a space. "Dot-quote" is a Forth word and so it needs that space.

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


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.

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. 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.

Errors when Compiling

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 and the text of the entire line that has a problem.

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 when you first start.

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

COLD   (this is like NEW in BASIC)

INCLUDE DSK1.BASICHLP

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

```
TOOLS          ( programmer tools, DUMP, WORDS, .S )
INPUT          ( $INPUT and #INPUT similar to BASIC)
RANDOM         ( RANDOMIZE, RND )
STRINGS        ( Strings with BASIC type functions)
GRAFIX         ( CLEAR, COLOR, SCREEN, VCHAR, HCHAR etc.)
CHARSET        ( reset all characters to default patterns)
```

After about 30 seconds of compiling all these files, the OK prompt comes back and your screen looks like this:

Type this little program in at the console:
```
: 10>  CLEAR ;
: 20>  " HELLO WORLD" PRINT ;

: RUN   10>  BEGIN  20>  ?BREAK  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 and STRINGS) 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. Forth words are smart. They know how to start themselves by name. No GOSUB, GOTO or CALL required.

And by the way….
Remember what we said about you are responsible for everything with Forth. Now notice that we had to use the word ?BREAK (in TOOLS) in our program? If we did not, the loop would never stop. If you want to break your program you have to tell it how to do it. You are the boss.

Press Function 4 to stop the demo program

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 organize 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.

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! ?BREAK 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! ?BREAK AGAIN ;
```

Smallest Version

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    ?BREAK  AGAIN ;
```

Factoring is the key to good Forth

An important part of programming Forth is "factoring". This means removing common "factors" (ie: parts that are the same) in a program and giving them a name. In BASIC terms this is like using SUB-ROUTINES as much as possible. Why? Because Forth is a bit more difficult to follow is it is one long line of code. Breaking it up into bite-sized pieces with meaningful names makes it easier to understand.

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.

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.

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

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

- **":"** defined a new word called CLEAR
- PAGE is the STANDARD Forth word to clear the screen, but cursor goes to top left.
- AT-XY positions the cursor column and row. We set it to column 0, row 23.
- That's how easy it is to add functionality to Forth.

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 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.
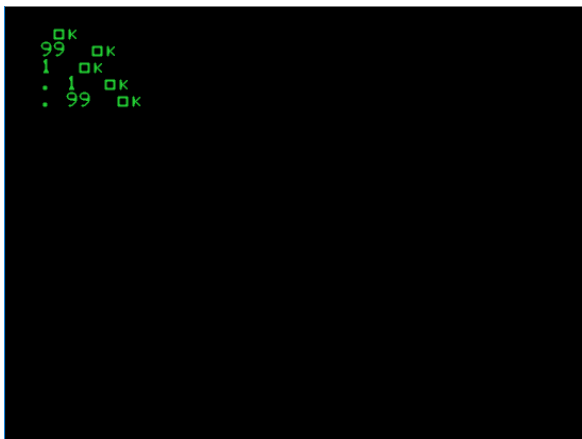
* At the console type the word DECIMAL
This makes sure that Forth is working in BASE 10 arithmetic.

    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

Adding two numbers with the word "+"

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, ie: 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 one assembler instruction on the TI-99 so it is very fast.

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

'.' Dot is more complicated but uses some clever Forth code to convert a binary number on the stack into a string of ASCII characters and then prints the string on the screen. Unlike in BASIC, you have access to all the WORDS that make "DOT" function. This means you can change the format of numbers 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 the ones in this screen on your Forth console.

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

Words not Operators

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 -32768 to 32767. There are Forth words to work with 32bit numbers and mixed 16 and 32 bit numbers. These are more advanced so typically in CAMEL99 we work with 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.

EMIT Example

For this example we will learn some new words. 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.

With BASICHLP loaded into the system try this at the CAMEL99 console:

```
DECIMAL (make sure you use base 10)
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.

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 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
- Vaddr an address in Video RAM
- caddr u – a character address and the length. (Typically a string, or byte array)
- $ - The author uses this one for a counted string. That is an address where the first byte is the length of the string
- d - a double integer. 32 bits. Requires the DSK1.DOUBLES library for CAMEL99 Forth
- f - a floating point number. Not 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.

Notice we did not need a variable because we used the stack to pass the number 100 to STARS. (Also notice we can print the entire width of the screen)

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.

DATA STACK   HOLDING PLACE (on return stack in CAMEL99)

| INDEX value |
| LIMIT value |

DO move parameters to
        temporary place (Rstack)

| INDEX value |
| LIMIT value |

        your CODE…
        your CODE…
LOOP

| INCR. INDEX |
| LIMIT |

LOOP's Action: IF   INDEX<LIMIT GOTO address
                ELSE Remove index and limit from Return stack
                THEN continue with the rest of the program
                     (It could also just go back to console)
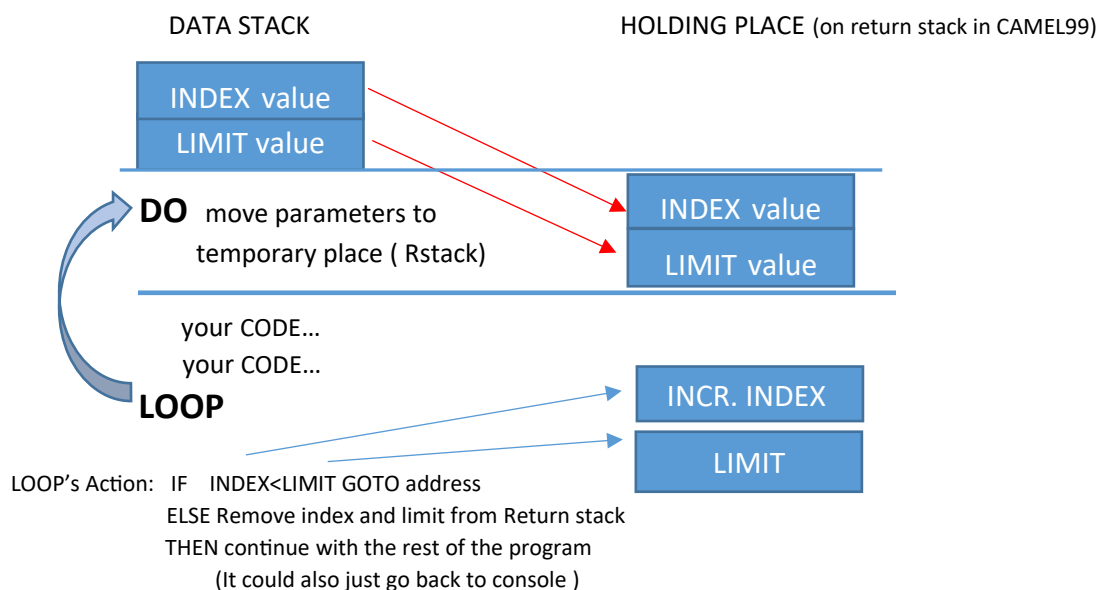
Then the code after DO runs as you would expect until the word LOOP is encountered. 'LOOP' adds 1 to the INDEX value in the *holding place.
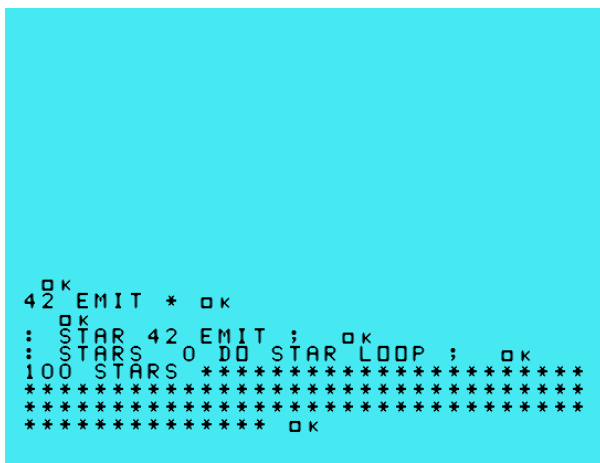
(Oh yeah and the holding place just happens to be right there on the top of the Return Stack. That's the technical reason we talked about earlier)

Then it compares the INDEX to the LIMIT value to the new value of the INDEX . If the two numbers are different, LOOP jumps back to where the code starts, 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 ⟶

The Forth equivalent is:

```
DECIMAL
:  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.

* I is NOT a variable. It is a Forth word. *

You know what all the other words mean by now.

And we run this test we see this

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, LOOP 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 65,536 LOOPs. You can try it if you don't believe it. 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. And one more thing…

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

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.

DO LOOP Source Code

The entire source code for the CAMEL99 kernel is in two files called 9900FAS3.HSF and ISOLOOPS.HSF.
The code is written in a dialect of Forth that was created just to make Forth compilers. This is called a cross-compiler. It can make your head spin to think about it but there it is for you to read. You can find the full source code for CAMEL99 Forth on GITHUB in the SRC folder:

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

To LOOP or Not to LOOP

Something that is not obvious to those new to Forth is that each time we compile a word in a Forth definition we use 1 CELL (2 bytes) of program space. And each time we compile a literal number we use 2 CELLS (4 bytes). So let's say we needed to output 5 STARS to the screen. Should we use DO LOOP?

Let's count the size of two different methods:

```
: WITHLOOP    5  0  DO  STAR  LOOP ;

\  5  is a literal, 4 bytes
\  0  is a word,    2 bytes  ( it is used so often we made it a CONSTANT word)
\  DO STAR LOOP     8 bytes  ( LOOP uses 4 bytes. [LOOP address,jump address])
\  -------------------------
\  Total          14 bytes.


: NOLOOP        STAR STAR STAR STAR STAR ;

\ STAR x 5 = 10 BYTES
\ NOLOOP is 40% smaller and it's faster because there is no jumping!
```

Don't be afraid to "unroll" small loops.
It works in Forth. You save space and it goes faster.

Lesson 5: Strings

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 ( -- )
```
Collapse is automatically invoked by PRINT and PUT. You only need it when the program first starts or if you want to clean up the string stack within your own program.

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$
```

Under the Hood of 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

(Which is also possible with BASIC)

100 PRINT SEG$(A$4,10)& SEG$(B$,12,5)
```

Technical Details about STRINGS

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 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? It can't keep growing forever. 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 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)

( -- TOP$) PRINT  ( we could PRINT the TOP$ string, which collapses the string stack)

( -- TOP$) B$ PUT ( or PUT TOP$ 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 at the top of the 8K low memory block and grows downward towards the HEAP memory.

Warning: string stack grows down, and could collide with the HEAP which grows up if your program uses excessive amounts of these spaces at the same time.

VDP Strings DSK1.VDPSTRINGS

It is also possible to create strings in VDP RAM using the library file called VDPSTRG. This library creates stack strings (addr,len) on the Forth stack but stores the counted strings in VDP RAM. It is not complete and will require customization to be fully useful however it is pretty nice to have all that RAM if you need the space.

Using VGET$ and VPLACE you can move strings back and forth from VDP RAM to CPU so feel free to pick and choose the parts that work for your application. See the code and comments for explanations.

DSK1.VTYPE

Although Forth prints to the screen quickly sometimes you want to put text on the screen a maximum speed. Two words are in the is file:

VTYPE (addr len --) takes an address and length (called a stack string) and prints it onto the screen a the current cursor position and moves the cursor forward by the length. It cannot scroll so if you write past the end of the screen it will begin writing into control tables in the VDP RAM. This is not a good thing.

AT" (col row --) Is like DISPLAY-AT but faster. It uses VTYPE, but lets you pass the column and row position to select where the text is written. It is more generally useful than VTYPE.

Warning: AT" can only be used for compiling. It cannot be used interactively.

Lesson 6: Getting Input

Standard Forth provides some very simple input words that don't align with BASIC programming practice. This can be a confusing thing for new Forth programmers.

ACCEPT (addr len -- #chars)

ACCEPT is Forth's standard keyboard input word. It is much less complicated that BASIC's INPUT. Accept needs an address to put data into, and a length on the stack, to know how many characters to "accept". The code below will let you type until it detects ENTER (return) and it will leave the number of characters you typed on the stack. The characters will be in the address called PAD.

```
PAD 80 ACCEPT
```

To print out what is in PAD you would need to do something like the following:
```
( -- #chars) PAD SWAP TYPE
```

$INPUT and #INPUT

Of course ACCEPT just gets you a string. If you wanted to get a number you need to convert the text into a number. BASIC handles this for you when you select a numeric variable. In CAMEL99 Forth we have made $INPUT and #INPUT to give you a way to do something that you might find more familiar. You only have to INCLUDE the file DSK1.INPUT to use them. Consider the code below and see what happens in the screen shot. Notice that #INPUT rejects QWERTY as an input and re-tries like TI BASIC. (And yes it also HONKS)

Try It Out
```
INCLUDE DSK1.INPUT
DECIMAL
VARIABLE X
VARIABLE A$ 32 ALLOT        \ string variables need space alloted

A$ $INPUT
X  #INPUT
```

$INPUT reads into a string variable

Type A$

Get #INPUT

#INPUT Rejects text

You can try again

Fetch & Print contents of X
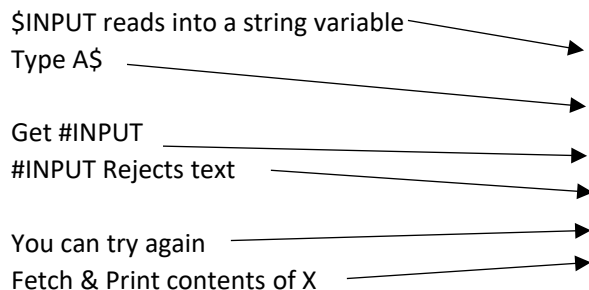
Behind the Curtain

Let's take a look at what it took to emulate BASIC's INPUT statement.

$INPUT

First we modify ACCEPT to do a few things more like BASIC. We use CR to start on a new line and print a question mark on the screen. Next we take a string address from the stack as an input parameter and DUP and add one to it. This gives us an address for ACCEPT to put your typing into. The text begins after the length byte so that is why we use 1+. When you press <enter> accept leaves the number of characters you typed on the string. That is just what we need to finish our string; a length. So we SWAP and fill in the length byte of the $addr with C!.

```
DECIMAL
: $ACCEPT ( $addr -- ) CR ." ? "  DUP  1+ 80 ACCEPT  SWAP C! ;
```

Now to make $INPUT all we need to do is add the BEEP. If you don't want a BEEP, use $ACCEPT. See how factoring the code into re-usable chunks lets us use the words in different ways. Other languages would ask you to use a control flag somewhere to turn the BEEP on and off. Forth thinking says use simple words and glue them together when you need to.

```
: $INPUT  ( $addr -- ) BEEP $ACCEPT ;  \ BEEP like TI-BASIC
```

#INPUT

To make #INPUT we need to do a lot more. TI BASIC's INPUT statement is clever.
1. First we BEEP. Then we start a WHILE loop with BEGIN and $ACCEPT the user input into a temporary memory space Forth calls PAD.
2. Then we give the string in PAD to a word called NUMBER?. NUMBER? takes the counted string and tries to convert the text to a number in the current number BASE.
3. WHILE tests the TRUE or FALSE conversion flag from ?NUMBER with 0=. While the flag=0 we HONK an error message and DROP the number off the stack and REPEAT.
4. When the flag=true we SWAP the number and the variable on the stack and store the number in the variable with ! .
5.

```
: #INPUT  ( variable -- )        \ made to look/work like TI-BASIC
        BEEP
        BEGIN
          PAD $ACCEPT          \ $ACCEPT text into temp buffer PAD
          PAD COUNT NUMBER?    \ convert string to a number. 0 means good convert
        WHILE                  \ while the conversion is bad we do this
          CR HONK ." input error "
          CR DROP
        REPEAT
        SWAP ! ;    \ store the number in the variable on the stack)
```

Lesson 7: Graphics

If you include the file DSK1.GRAFIX you will have access to words that are common to all TI-BASIC programmers. Most of the BASIC words behave has 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. [2]From the GITHUB repository see: DSK1.XBDEMOAUTO to get a feel for graphics and sprite control.

DSK1.XBDEMO re-written in Forth

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.

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 can be used like CALL CHAR() in BASIC. It works in a way that maximizes speed. Rather than taking a text string and an ASCII value, it takes [3]memory address and an ASCII value. The memory address must contain 8 bytes that are the **INTEGERS** you would see in the HEX string in a CALL CHAR() statement.

PATTERN: (n n n n -- <name>) (deprecated)
The word PATTERN: takes four integers off the stack and stores them in the Forth dictionary like a constant, but the constant has 4 CELLS of data. Notice you must break the pattern up into 4 pieces. (ie: 4 integers) By using PATTERN: your patterns get names that are easy to remember and make your program easier to read and understand. Use your PATTERNs with the CHARDEF word to change characters instantly.

PATTERN: has been deprecated in Came99 Forth because it really isn't needed. Since all we want is the address where 4 CELLS of numbers are stored, you can make a named character pattern with CREATE and comma like this:

```
CREATE TINYFACE   3C7E , DBFF , FF42 , 3C00 ,  \ don't forget the last comma!
```

---

[2] Screen image of XB Demo by Tursi, Atariage/Harmless Lion software. Re-written for Forth
[3] These memory addresses are called "pointers" in other languages but Forth keeps things simple and calls them addresses, which is what they are after all.

```
HEX
CREATE ABLOCK FFFF , FFFF , FFFF , FFFF ,

DECIMAL
ABLOCK 65 CHARDEF  \ now the letter A looks like a square block

\ ** To make your code read clearer you can also do this:

  ABLOCK CHAR A CHARDEF

\ CHAR A computes the ASCII number of 'A' for you and leaves it on the stack.
\ Right where you need it.
```

> **Why does CHARDEF use an address? Because the address contains the pattern 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.**
>
> **CHARDEF is so fast that you can create animated characters and change all 8 bytes of the character pattern in real time.**

CALLCHAR (*NEW in Version 2.6)

* In Version 2.6 we have changed CALLCHAR so the string is the first argument and the ascii value is 2ⁿᵈ. This was done because it was more efficient to code it this way. Also the new version can handle long strings so you can define four characters at once as you can do in BASIC.

This works very much like the BASIC statement. The difference is that it uses S" to create the string of hexadecimal digits. S" returns an address and a length on the stack. CALLCHAR picks up the ASCII value and the stack string, coverts the string into four 16 bit numbers and stores the numbers in the "pattern description table" (PDT) in the VDP RAM.

CALLCHAR examples:

```
DECIMAL
 S" 3844447C44444400" 65 CALLCHAR       \ this a capital letter A definition
 S" 3844447C44444400" CHAR A CALLCHAR  \ you could do this way for clarity

\ this changes characters 65,66,67,69  (A,B,C,D)
 S" FFFF0000FFFF0000FFFF0000FFFF0000FFFF0000FFFF0000FFFF0000FFFF0000" 65 CALLCHAR
```

Things to Know about CALLCHAR

- The characters in the strings will always be understood as HEX numbers.
- The ASCII value will be whatever BASE Forth is set to. It's probably less confusing to declare DECIMAL before using a CALLCHAR statement.
- CALLCHAR can be used by the interpreter to change characters when you load a source code program file, using no CPU RAM space at all. This is handy for testing patterns inside Forth.
- OR you can put CALLCHAR inside a colon definition to keep the information in the program.
  REMEMBER: the pattern strings take two times more memory than using CREATE and CHARDEF and run much slower if you want to change patterns while the program is running.

Using 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 a bit more familiar.

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 Green
6	Light Blue	14	Magenta
7	Dark Red	15	Gray
8	Cyan	16	White

COLOR (character-set foreground background --)

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

COLORS (set#1 set#2 fg bg --)

If you want to change more than one color set, use COLORS. It will change all the sets between set#1 and set#2 to values fg and bg. (foreground, background)

```
Example usage:   0 32 2 8 COLORS  ( change ALL color sets in Forth to black on cyan)
```

SET#: A New Word for Your 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 these examples"

```
    65 SET#  2 8 COLOR
CHAR P 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 much clearer to understand than CALL COLOR (5, 4, 8)

GCHAR (col row -- char)

Moves the cursor to col,row and reads the character on the screen and puts it on the stack. This function is many times faster than BASIC's version.

VROW, VCOL

Unlike BASIC you also get access to the internals of GCHAR. GCHAR reads 2 system variables at once. They are called VROW and VCOL so you can also access them directly if you need to.

```
VROW @  1+  VROW !  \ they are just Forth variables
VCOL @  2+  VCOL !  \ no hidden magic here
```

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 line in BASIC:

```
> CALL HCHAR(1,1,42,768) <enter>
```

Now INCLUDE DSK1.GRAFIX and in GRAPHICS Mode type this into the Forth console

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

This difference is shocking. This is because HCHAR in Forth, uses the word VWRITE, a word written in Forth Assembler, which 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. [4]PAD 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 permanent 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 Q CHARPAT    \ now we can use MYPATTERN to CHARDEF any character
```


SCREEN (color# --)

SCREEN works just like BASIC but without the word CALL. Remember parameters come first!

The MODE VARIABLE

The system has a variable called MODE that records the current screen mode. MODE is set to a value of 1 when your use the GRAPHICS command. Like all variables in Forth you get the screen mode value by using the "fetch" (@) command like this:


```
MODE @ .  1 OK
```

---

[4] PAD in Forth is in dictionary memory. TI-99s "SCRATCH PAD" is at address >8300 and is only 256 bytes in size

Changing Fonts   DSK1.LOADSAVE

As of version 2.2 Camel99 Forth has the ability to load fonts from disk directly into VDP RAM. Fonts can be defined as Forth code files. Examples are on DSK2 of the distribution package.

It is now possible to save the patterns in VDP RAM as binary program file using SAVE-FONT. You can load binary Font files back into VDP RAM with LOAD-FONT.

> ### LOAD-FONT can change a Font in .5 seconds!

Create a Font

If you have the time you can define the character pattern for each character of the alphabet by hand. It is far easier to use the **Magellan** program which is a character editor. Unfortunately Magellan can only export BASIC and Assembly language code so we have to do a bit of editing to make a Camel99 Font file.

Using Magellan with Forth

Use the "Export Assembler DATA" option and remove everything but the hexadecimal strings. Then using your editor modify the text to work with CALLCHAR. (it's not fun but it works)

Here is the preamble code that you should put at the top of your font file. After the preamble add the lines for each character as seen below. If you want to save your font as a binary

```
\ FONT0230  pattern source code for CAMEL99 Forth
NEEDS CALLCHAR    FROM DSK1.CALLCHAR
NEEDS SAVE-FONT   FROM DSK1.LOADSAVE

DECIMAL
 S" 0010101010001000" 33 CALLCHAR
 S" 0028282800000000" 34 CALLCHAR
 S" 00287C28287C2800" 35 CALLCHAR
 S" 0038543018543800" 36 CALLCHAR
 S" 00444C1830644400" 37 CALLCHAR
 S" 0020502054483400" 38 CALLCHAR

\     <other definitions go here>

 S" 0060101008101060" 125 CALLCHAR
 S" 0000205408000000" 126 CALLCHAR

\ Save the font as a binary file by putting this last line in the file
  S" DSK1.FONT0230" SAVE-FONT
```

DSK3.FONT0000   DSK3.FONT0004   DSK3.FONT0005

DSK3.FONT0230   DSK3.FONT0277

Lesson 8: SAVESYS

Although Camel99 compiles source pretty quickly it still takes time. The SAVESYS library lets you take the entire system from memory with everything you have INCLUDEd and save the system as a set of EA5 files that can be loaded from the Editor Assembler cartridge menu, Option 5.

To use it type: INCLUDE DSK1.SAVESYS (you will notice that SAVESYS uses LOADSAVE)

The most important part of this process is making sure that the image knows what to run after it is loaded into memory. There are some important initialization that Forth requires to get started. Fortunately this is all contained in the word WARM. Then your program must setup everything it will need. This might be allocating some memory in low ram where you can use MALLOC or defining a bunch of character patterns for example. Below is an example of how to do this with some different scenarios.

Things that are Not Saved

SAVESYS only saves the Forth dictionary. That is the memory that begins at HEX A000 and it goes up to the end of the dictionary ie: at the end of the last Forth word that you create.

SAVESYS DOES NOT SAVE THE LOW RAM

Low RAM is the 8k memory block from HEX 2000 to HEX 3FFF. This low RAM is for you to use as you see fit in your programs. It is active if you use MALLOC or SAMS memory, or FORTH block files or any other thing you want to use it for. Anything that is put there by your program is NOT saved by SAVESYS.

When to Use SAVESYS

Customize Forth for You

One use for SAVESYS is to load up the system with the things you like to have. For example you might want to work in 32 column mode all the time. You could include the GRAFIX, DIRSPRIT and AUTOMOTION libraries and save that off as a custom Forth version called GRAFORTH. Then next time you want to make a game you would load GRAFORTH and all your favourite libraries are immediately there to use.

Partial Project Saves

Forth allows something called "incremental compilation". This means you can compile a program one word or one file at a time. With SAVESYS you can compile everything for the first part of your project that you know is solid and reliable and save that as binary "image". The image can be brought back into memory very quickly. Next time you want to continue with the project you simply load that binary and everything is ready for the next additions you want to make to your project. No need to re-compile everything from the beginning.

You can think of this as a customized compiler with all the words you need for your project are pre-compiled.

Example 1: Save Forth image with Tools

Step 1: Start CAMEL99 Forth kernel from the E/A 5 menu (Run Program File)

Step 2: Type the following to build a system with handy tools in it:

```
 INCLUDE DSK1.TOOLS       \ load tools
 INCLUDE DSK1.ELAPSE      \ load the elapse timer for testing speed
 INCLUDE DSK1.SAVESYS     \ this does the saving

\ redefine COLD so you can reset the system but NOT load the start file.
: COLD
       WARM        \ Init Forth system properly (sets variables, builds stacks)
       17 7 VWTR \ set black on cyan colors
       ABORT ;   \ resets the stacks and runs QUIT, the Forth interpreter

  LOCK             \ this command remembers where the dictionary is right now.
                   \ WARM uses this info when it intializes the system


\ This line gets the Execution token of COLD with tick (') and passes it
\ to SAVESYS. When MYFORTH starts it will run COLD first.

' COLD SAVESYS DSK1.MYFORTH    \ choose a file name for yourself
```

Now run DSK1.MYFORTH at the E/A 5 menu and Forth starts with everything ready to go.

Example 2: Save Graphics Mode Forth

```
 INCLUDE DSK1.GRAFIX
 INCLUDE DSK1.DIRSPRIT
 INCLUDE DSK1.AUTOMOTION

\ redefine COLD so you can reset the system but NOT load the start file.

: COLD
       WARM        \ inits the Forth system properly
       GRAPHICS   \ set GRAPHICS 1 mode (32 columns)
       ABORT ;   \ resets the stacks and runs QUIT, the Forth interpreter

  LOCK             \ this command remembers where the dictionary is right now.
                   \ WARM uses this info when it intializes the system

INCLUDE DSK1.SAVESYS  \ this will be forgotten when this image restarts

\ This line gets the Execution token of COLD with tick (') and passes it
\ to SAVESYS. When GRAFORTH starts it will run COLD first.

' COLD SAVESYS DSK2.GRAFORTH   \ save the whole thing
```

Lesson 9: Text Mode (40 cols)

There are not too many things to know about TEXT mode. It is very handy for communication with humans but it only supports two colors at one time; foreground and background.

40 Column Colours

CAMEL99 does not have a colour word for 40 COLUMN mode because it is so simple to do it. The colours are controlled by Register 7 in the VDP chip. We use the command VWTR, which is an acronym for "Video Write to Register". The colour values used are the "machine" values from 0 to 15. BASIC uses the values 1 to 16 and we have kept that convention in CAMEL99 Graphics MODE as well.

However here we are talking to the hardware directly so colour values are 0 to 15. To change the colour of the foreground and background in TEXT to our familiar BLACK on CYAN , like BASIC, we would type:

```
HEX  17 7 VWTR
```

Too understand this, separate the hex number 17 into 1 and 7. 1 is BLACK and 7 is CYAN

The next number 7 is the VDP register number. VWTR picks up the colour values from the stack and loads them into Register 7 of the VDP chip.

There is no internal word for setting TEXT mode colours. If you wanted to make a new colour control word in Forth for TEXT mode do it like this:

```
: TINT   ( fg bg -- )  SWAP 4 LSHIFT SWAP +  7 VWTR ;
```

Explanation of TINT

It's easiest to understand this in hexadecimal numbers. Type it into the Forth console and see it for yourself. Include DSK1.TOOLS so you can use **.S** to see the stack like the screen capture.

We have two colour values on the stack. Each colour value is between 0 and 15 (0 ..F hex) so they each are only 4 bits wide. (Trust me on this if you are still learning about binary numbers)

We need to put them together into one eight bit number to feed them to VDP register 7.

1. SWAP the fg colour to the top of stack and shift 4 bits to the left
   a. This would make 5 into 50, or 1 into 10

2. SWAP the number again so that foreground is under background colour

3. Add the numbers together which gives us the 8 bit number needed by the VDP chip

Lesson 10: Using Sprites

The TI-99 has a nice little video chip that can support 32 moving characters called Sprites. The implementation in Extended BASIC uses the computer's internal interrupt system to move the sprites every 1/60 of a second. CAMEL99 can use that system if you include the file call AUTOMOTION.

It is also possible push the sprites around the screen manually using Forth code. This is not different than positioning a character on the screen except that the grid of possible positions is 256 columns by 191 rows. Each position is of course a pixel position.

CAMEL99 provides words that emulate the features of Extended-BASIC so you will not have difficulty there. In order to make the sprites move as fast as possible we provide some extra words beyond LOCATE that operate directly on sprite X and Y coordinates independently. We will also show you how to do some speed up tricks as well.

The simple sprite system is in file DSK1.DIRSPRIT. It means DIRECT SPRITEs and is so named because the sprites are under direct control of your program. (Not under interrupt control like Extended BASIC)

Limitation of Direct Sprite Control

Although Forth works faster than BASIC directly manipulating each individual sprite in VDP RAM is slower than writing a block of DATA to all the sprites in one shot. So you can use this direct Sprite library for games or apps that don't need really fast movement or real-time tracking of contact between multiple sprites. You are also free to move some sprite with AUTOMOTION and move others completely under Forth code control. Your choice.

How Sprites Work

When using BASIC all the details of how the language is controlling sprites is hidden from you. From a programmer's perspective it turns out that the TMS9918 chip in our TI-99s is not really difficult to understand. Here is a simple way to understand it.

Each sprite in the computer has a four bytes of memory in the VDP RAM assigned to it. You can think of these bytes as an array or a table because they are all together. This block of VDP Ram is called the **Sprite Descriptor Table** (SDT). In CAMEL99 GRAPHICS mode the SDT starts right after the video screen which is address HEX 300. (Decimal 768) The Functions of each byte are as follows:

1. Sprite X coordinate

2. Sprite y coordinate

3. Sprite character to use for its "pattern"

4. Sprite colour value

So literally all you do to control a sprite is write a number into one of those four boxes in VDP memory. The 9918 chip does all the rest. All the Sprite words do for you in CAMEL99 is try to give you nice names for the routines that do that. You can make your own if you want to.

Peek Inside the Sprite Descriptor Table

It is possible to see the sprite table directly in the debugger window of the Classic99 Simulator.

Sprite Descriptor Table address (Called SDT in Camel99 GRAPHICS 1 MODE)

1ˢᵗ Sprite Y "location"

X "location"

Character "Pattern"

Sprite color (value is 1 less than in BASIC/Camel99 Forth)


```
Classic99 Debugger                                                          ×
File  View  Debug  Make

0300: 00 09 2A 03 00 00 00 00 ..*.....      R 0   42E0      R 8   A110       Breakpoint
0308: 00 00 00 00 00 00 00 00 ........      R 1   8300      R 9   B0F0
0310: 00 00 00 00 00 00 00 00 ........      R 2   68B0      R10   838A
0318: 00 00 00 00 00 00 00 00 ........      R 3   0000      R11   A7EE
0320: 00 00 00 00 00 00 00 00 ........      R 4   02E0      R12   0000
0328: 00 00 00 00 00 00 00 00 ........      R 5   A896      R13   0000
0330: 00 00 00 00 00 00 00 00 ........      R 6   3EF4      R14   1000
0338: 00 00 00 00 00 00 00 00 ........      R 7   3FF6      R15   0000
0340: 00 00 00 00 00 00 00 00 ........
0348: 00 00 00 00 00 00 00 00 ........      UDP0  00        UDP   02E1
0350: 00 00 00 00 00 00 00 00 ........      UDP1  E0        GROM  B0B0       Add    ?   Remove
0358: 00 00 00 00 00 00 00 00 ........      UDP2  00        VDPST 44
0360: 00 00 00 00 00 00 00 00 ........      UDP3  0E          PC  838C
0368: 00 00 00 00 00 00 00 00 ........      UDP4  01          WP  8300
0370: 00 00 00 00 00 00 00 00 ........      UDP5  06          ST  8402
0378: 00 00 00 00 00 00 00 00 ........      UDP6  01        Bank  000000
0380: 10 10 10 10 10 10 10 10 ........      UDP7  07         DSR  FFFF
0388: 10 10 10 10 10 10 10 10 ........
0390: 20 20 20 20 20 20 20 20               SIT   0000
0398: 47 52 41 50 48 49 43 53 GRAPHICS      SDT   0800      SAL   0300
03A0: 20 20 20 20 20 20 20 20               PDT   0800      Size  0800
03A8: 20 20 20 20 20 20 20 20                CT   0380      Size  0020
03B0: 20 20 20 20 20 20 20 20
03B8: 20 20 20 20 20 20 20 20               VDPST:     5SP     5thSP: 4
03C0: 00 00 00 00 00 00 00 00 ........
03C8: 00 00 00 00 00 00 00 00 ........       ST : LGT AGT          OP
03D0: 00 00 00 00 00 00 00 00 ........      MASK: 2
03D8: 00 00 00 00 00 00 00 00 ........
03E0: 00 00 00 00 00 00 00 00 ........      9919 201 000 000 0
03E8: 00 00 00 00 00 00 00 00 ........      VOL   F   F   F  F
03F0: 00 00 00 00 00 00 00 00 ........
03F8: 00 00 00 00 00 00 00 00 ........       ○ Debug    ○ Disasm    ○ CPU    ⦿ UDP    ○ GROM
0400: AA 01 01 00 00 00 60 10 ......`.
                                              Edit    300        Apply   Prev   Next
```

Direct Sprite Support Words

SPRITE (char color x y spr# --)

Create a sprite as you would in BASIC. Just like BASIC the SPRITE word initializes four parameters as you can see in the stack diagram. You can use the SPRITE command to re-initialize a sprite even if the sprite has already been defined in your program.

SPR# (-- n) variable

When you create a sprite, a variable called SPR# is increased by one if the sprite number you create is greater than the current value of SPR#. In other words SPR# will always give you the number of sprites in the system. IMPORTANT: Sprites are numbered 0 to 31 (decimal)

DELALL (--)

DELALL works like BASIC in that is erases all the data in the SPRITE TABLE.

IMPORTANT difference: Your program should always invoke DELALL once before using SPRITES.

Sprite Control Words

The following words are "safe" because they detect if you are trying to make changes to a sprite that has not been defined. Use these words when you are learning how to use the system. They are slower but will have less chances of crashing your program.

POSITION (sprt# -- dx dy)

Retrieves the current X and y positions of the sprite given as the input parameter,

LOCATE   (dx dy sprt# --)

Sets the position on the screen of the sprite parameter on the stack. Like Extended BASIC

PATTERN   (char sprt# --)

Sets the character to use as the character pattern for the sprite given a parameter on the stack. Like Extended BASIC

SP.COLOR (col sprt# --)

This word is different than BASIC to avoid a name conflict with the graphics word COLOR. Otherwise it works as expected and sets the foreground color of the sprite to the color number on the stack. It will abort with an error if sprt# is not a defined sprite. (colors are like BASIC 1 ..16)

Faster than LOCATE

If you need to move a sprite at maximum Forth speed it is probably best to use these words. They have been written with no range checks and simply write numbers to the VDP Sprite Descriptor Table as fast as Forth can do so.

SP.X   (spr# -- vdp_addr)

Return the SDT address of the spr# X coordinate. Use VC@ and VC! to access these VDP variables.

SP.Y   (spr# -- vdp_addr)

Return the SDT address of the spr# Y coordinate. Use VC@ and VC! to access these VDP variables.

MAGNIFY (mag-factor --)

The MAGNIFY word does the same thing as its counterpart in Extended BASIC. The difference is that the levels go from 0 to 3 rather than 1 to 4.

Don't forget that if you want to use MAGNIFY level 2 and 3 you must define four consecutive characters. This is quite simple using PATTERN: and CHARDEF or you can use the new CALLCHAR word with a long string. The SPRITE definition must use the LAST character in the sequence of four characters. The example code below shows one way to make big sprites.

```
\ Direct sprite X.Y  control with MAGNIFY
\ big happy face sprite

NEEDS DUMP    FROM DSK1.TOOLS        \ for testing only
NEEDS SPRITE FROM DSK1.DIRSPRIT      \ this also loads GRAFIX

\ *the string must be on one line*
S" 071F3F7F7FFBFFFFFFFFFF7787F3F1F07E0F8FCFEFEDFFFFFFFFFFEF1EFEFCF8E0" CHAR t CALLCHAR

\ Ptrn     clr col   row   spr#
\ ------   --- ---   ---   ----
 CHAR w   11  150    80    0    SPRITE
 CHAR w   13   50    80    1    SPRITE
 CHAR w    8   50    80    2    SPRITE
 CHAR w   10   50    80    3    SPRITE

: RUN  ( -- )
      PAGE  2 SCREEN
      3 MAGNIFY
      BEGIN
         0 SP.X VC@ 1-  0 SP.X VC!
         0 SP.Y VC@ 1-  0 SP.Y VC!

         1 SP.X VC@ 2+  1 SP.X VC!
         1 SP.Y VC@ 2-  1 SP.Y VC!

         2 SP.X VC@ 1+  2 SP.X VC!
         2 SP.Y VC@ 1-  2 SP.Y VC!

         3 SP.X VC@ 2-  3 SP.X VC!
         3 SP.Y VC@ 1-  3 SP.Y VC!
         25 MS    ( delay controls the speed)
         ?TERMINAL
      UNTIL
      8 SCREEN ;

\ Alternate character definition code
\ Using character patterns stores the data in CPU RAM
HEX
071F 3F7F 7FFB FFFF PATTERN: TOPLEFT
FFFF F778 7F3F 1F07 PATTERN: BOTLEFT
E0F8 FCFE FEDF FFFF PATTERN: TOPRIGHT
FFFF EF1E FEFC F8E0 PATTERN: BOTRIGHT

\ CHARDEF writes the pattern data into VDP RAM
DECIMAL
TOPLEFT   CHAR t CHARDEF
BOTLEFT   CHAR u CHARDEF
TOPRIGHT  CHAR v CHARDEF
BOTRIGHT  CHAR w CHARDEF  ( the last character is the one used for the SPRITE)S
```

Moving Sprites Faster

In BASIC we only have LOCATE to place a sprite somewhere on the screen. If you only want to move one coordinate this is slow and awkward. The words SP.X and SP.Y allow you to get at the sprite addresses in VDP RAM directly and independently. These are like Forth variables except that they return to you a VDP memory address not CPU RAM. For VDP memory we use special words that start with a 'V', to fetch and store VDP RAM.

```
\ move sprite #5 to X=100

: MOVESPRITE5    100 5 SP.X VC! ;  \ put #5 at X=100 (ie: COLUMN 100)
```

This is faster than using LOCATE most of the time but inside SP.X and SP.Y a small calculation is performed each time we use them. The calculation is:

$$Sprite\# * 4 + SDT = VDP\ address\ of\ a\ sprite's\ data.$$

Explanation

This is only 2 computations and even though it is done in machine code it takes a bit of time. Wouldn't it be great if we could calculate the address of a special sprite that needs maximum control, ahead of time?

Forth gives us a couple of ways to do this. We could make CONSTANTS.

```
 6 SP.X CONSTANT #6.X      \ used to get or set sprite #6 X position
 6 SP.Y CONSTANT #6.Y      \ used to get or set sprite #6 Y position
```

```
These take up room in the dictionary which you may not want to do. There is another
way. Here is an important Forth coding trick. In other compilers this might be done by
the compiler without your knowledge. (commonly called "constant folding")

This concept is to escape from the compiler for a time, calculate the address of the
SPRITE and then compile that address in our Forth word as a literal number.
```

Examine the following code:

```
: #5.X! ( n -- ) [ 5 SP.X ] LITERAL VC! ;
```

Step by Step Explanation

```
• Define the word #5.X!  It takes n from the stack and puts in Sprite #5's X memory.
• [ turns off the compiler. We are now interpreting :-)
• 5 SP.X  computes the address of Sprite #5 and leaves it on the DATA stack
• ] turns on the compiler again
• LITERAL is an immediate word that takes a number from the data stack and compiles it
  into a definition as a literal numbere
• VC! stores a 'char' (a byte) to VDP RAM
• The entire definition becomes:  >315 VC!
  • >315 is the hex address of the sprite information for Sprite #5.
  • No computations are made! The sprite X data is changed and the sprite moves.
```

Change Sprite X & Y at Maximum Speed

If you had some sprites that required X and Y coordinate movement at faster speeds we can use V! which writes 2 bytes to VDP RAM in immediate succession. To do this we need to "fuse" together the X value and the Y value into one 16 bit number. We can use the word FUSE for that. How convenient.

It is important to understand that the Y coordinate and the X coordinate live side by side in the VDP memory. SP.Y is first in the sequence. This means that if we take a 16 bit number and write it into address SP.Y the second byte will go right into SP.X. How convenient.

Knowing this and knowing how convert a calculation into a literal number, consider the code below to study how we can write to Y and X and move a sprite to anywhere on the screen in one fast operation.

```
: #2.XY!  (  x y -- ) FUSE [ 2 SP.Y ] LITERAL  V! ;
```

The Dreaded Coincidence

We have all suffered with Extended BASIC and trying to get Sprite coincidence to be detected at the proper time. The combination of automatic movement and the slow speed of the BASIC interpreter makes this very challenging and limits sprites to slow speeds.

CAMEL99 Forth is better in this area. You are in full control of Sprite motion so you can detect coincidence whenever it makes sense in your program.

We created these coincidence words using code from the original TI-Forth but re-wrote much of it to save space and also speed it up. It works much better than BASIC but It can still be a challenge however to make coincidence work for more than a few sprites. We find that each game or application requires finding an optimal way to handle motion and coincidence. If you need more speed you could re-write some of these words with the CAMEL99 Assembler. (ASM9900) - OR - you could try using some of the performance tricks described in the PERFORMANCE chapter. (See: INLINE Code Words)

COINCALL (-- ?)
COINCALL is the fastest detector because it reads the VDP status register which is the hardware doing the detecting. It returns zero if there is no coincidence.

SP.DIST   (spr#1 spr#2 -- dist^2)
Given two sprite numbers SP.DIST will give you the distance between the sprites squared.

SP.DISTXY (x y # -- dist^2)
Given a set of pixel coordinates and one sprite number, SP.DISTXY returns the distance squared between the two sprites.

COINC   (sp#1 spr#2 tol -- ?)
COINC works like the X-BASIC version and lets you select a tolerance value to control how close sprite#1 and sprite#2 are to one another.

COINCXY (dx dy spr# tol -- ?)
COINCXY does the same thing for you as COINC but lets you control tolerance between a screen position and sprite# with a tolerance value.

Sprite Motion Code Example

```
NEEDS DUMP FROM DSK1.TOOLS
NEEDS SPRITE FROM DSK1.DIRSPRIT   \ this also loads GRAFIX


\ these routines move a sprite around the outside of the screen
\ If you paste this code into CAMEL99 V2 you can see the speed
\ differences
\ ====================================================
DECIMAL
: TEST1      \ BASIC style using LOCATE
      PAGE
\     Ptrn    clr  x  y spr#
\     ------  ---  -  - ----
    [CHAR] *  16   0  0  0    SPRITE
      1 MAGNIFY
      10 0
      DO
          239 0 DO   I   0  0 LOCATE       LOOP
          175 0 DO 239   I  0 LOCATE       LOOP
          0 239 DO   I 175  0 LOCATE  -1 +LOOP
          0 175 DO   0   I  0 LOCATE  -1 +LOOP
      LOOP ;
\ ====================================================
DECIMAL
: TEST2     \ faster method moves X & Y independantly
      PAGE
\     Ptrn    clr  x  y spr#
\     ------  ---  -  - ----
    [CHAR] $   3   0  0  1   SPRITE
      1 MAGNIFY
      10 0
      DO
          239 0 DO   I  1 SP.X  VC!      LOOP
          175 0 DO   I  1 SP.Y  VC!      LOOP
          0 239 DO   I  1 SP.X  VC! -1 +LOOP
          0 175 DO   I  1 SP.Y  VC! -1 +LOOP
      LOOP  ;
\ ====================================================
DECIMAL
: TEST3 \ ⁵fastest: pre-calculate sprite table address, compile as literal no.
      PAGE
\     Ptrn    clr  x  y spr#
\     ------  ---  -  - ----
    [CHAR] @   9   0  0  2    SPRITE
      1 MAGNIFY
      10 0
      DO
          239 0 DO   I  [ 2 SP.X ] LITERAL VC!      LOOP
          175 0 DO   I  [ 2 SP.Y ] LITERAL VC!      LOOP
          0 239 DO   I  [ 2 SP.X ] LITERAL VC!  -1 +LOOP
          0 175 DO   I  [ 2 SP.Y ] LITERAL VC!  -1 +LOOP
      LOOP  ;


\ type GO to see the speed differences

:  GO    PAGE   DELALL   TEST1 TEST2 TEST3  DELALL ;
```

---

[5] Since re-writing SP.X and SP.Y as CODE words there is only a small speed improvement using LITERAL addresses

COINCIDENCE Code Example with Forth Motion Control

This program creates double cell motion vectors manually and uses them to provide motion to two sprites. The one sprite is fast and the other is slow. When they collide if triggers a bounce routine than cause them to move with reversed vectors.

```
INCLUDE DSK1.TOOLS
INCLUDE DSK1.DIRSPRIT    \ this also loads DSK1.GRAFIX
INCLUDE DSK1.RANDOM


\ happy face in 4 pieces
HEX
071F 3F7F 7FFB FFFF PATTERN: TOPLEFT
FFFF F778 7F3F 1F07 PATTERN: BOTLEFT
E0F8 FCFE FEDF FFFF PATTERN: TOPRIGHT
FFFF EF1E FEFC F8E0 PATTERN: BOTRIGHT

DECIMAL
\ GENERATE random motion vectors
: RNDXY  ( -- dx dy ) 5 RND 2-   5  RND 2- ;

\ create some motion vectors as double variables (2 cells)
\ use 2@ and 2! on them
CREATE V0   -1 , 1 ,    \ each sprite needs its own vector
CREATE V1    1 , 0 ,

: .VECTORS
        0 0 CLRLN  V1 2@  . . ;

: NEWVECTORS
        RNDXY   V0 2!
        RNDXY   V1 2!
        .VECTORS ;

\ Usage:  V1 REVERSE
: REVERSE ( motion-vector-addr -- )
        DUP 2@ NEGATE SWAP NEGATE SWAP ROT 2! ;

: SP.POS+ ( dx dy spr# -- )  \ add dx dy sprite's position
        DUP >R  SP.Y VC@ + R@ SP.Y VC!
            R@ SP.X VC@ + R> SP.X VC! ;

: MAKE-FACE
      TOPLEFT    [CHAR] t CHARDEF
      BOTLEFT    [CHAR] u CHARDEF
      TOPRIGHT   [CHAR] v CHARDEF
      BOTRIGHT   [CHAR] w CHARDEF ;

: MAKE-SPRITES ( -- )
      \ Ptrn    clr col  row  spr#
      \ ------  --- ---  ---  ----
   \   [CHAR] w   16 150  80   0   SPRITE    \ the only white sprite
        [CHAR] w   11  20  80   1   SPRITE
        [CHAR] w    9  10  16   2   SPRITE
        [CHAR] w    9 125  16   3   SPRITE
        [CHAR] w    9 220  16   4   SPRITE

\       [CHAR] w    9  10  90   5   SPRITE
\       [CHAR] w    9 125  90   6   SPRITE
\       [CHAR] w    9 220  90   7   SPRITE

        [CHAR] w    9  10 170   8   SPRITE
        [CHAR] w    9 125 170   9   SPRITE
        [CHAR] w    9 220 170  10   SPRITE  ;
```

```
\ Speed control example.
VARIABLE SPEED

: (MOVE#1)     V1 2@  1 SP.POS+ ;

: MOVE#1   ( -- )
\ each time this word is called we
\ 1. DECREMENT the speed variable
\ 2. check if speed=0
\ 3. if speed=0 then move the sprite and reset the speed variable
          -1 SPEED +!
          SPEED @ 0=
          IF                      \ when it hits 0 move sprite
             (MOVE#1)
             8 SPEED !         \ reset the speed counter
          THEN ;

: BOUNCE ( -- )  \ bounce off each other
          V1 REVERSE
          HONK
          20 1
          DO I 1 SP.COLOR
             (MOVE#1)
             5 JIFFS
          LOOP
          11 1 SP.COLOR ;

\ 2 sprites move in random directions
\ Sprite #0 changes color if it collides
: RUN
       PAGE
       13 SCREEN
       2 MAGNIFY
       MAKE-FACE
       MAKE-SPRITES
       8 SPEED !
       BEGIN
         NEWVECTORS
         800 RND 100 + 0
         DO
           MOVE#1
           COINCALL
           IF
               BOUNCE
           THEN
         LOOP
         ?TERMINAL ABORT" HALTED"
       AGAIN ;
```

Sprite AUTOMOTION

Although it is possible to write very effective code by manually controlling the motion of each sprite one of the very fun things that exists in the TI-99/4A is code to move sprites "automagically" under interrupt control.

The interrupt is a signal that turns on every 1/60 of a second. It is needed to make the video output work correctly but it also takes control of the CPU and tests to see if it can run some code for a few milli-seconds. One of the routines it can run, if it sees the correct information in memory, is AUTOMOTION. This feature is the same one that we use when we program sprites in TI Extended Basic. Automated sprite movement is fantastic when we don't want to manage some simple sprite movements. All the code is sitting in the ROMs of your TI-99 so the overhead for your Forth program is small. It only took a few lines of Forth to enable AUTOMOTION once we understood how to setup the system.

AUTOMOTION Glossary

SMT

TI-99 has a reserved memory area in the VDP RAM called the Sprite Motion Table. (SMT) SMT in Forth returns the base address of the table in VDP RAM. It is located at HEX 780 and consists of 4 bytes for each of the 32 sprites. These bytes are used to manage auto sprite motion. For each sprite the function of the bytes is as follows:

```
Byte 1: temp variable to manage X position speed (TI calls it AUX)
Byte 2: temp variable to manage Y position speed (TI call it AUX)
Byte 3: X motion value, signed character. -127 … 128 range
BYTE 4: Y motion value, signed character. -127 … 128 range
```

AMSQ

Reserved byte at address HEX 83C2 in scratchpad ram. It controls which system interrupt routines will run. See the code above for the specific bit functions

]SMT

Given a sprite number, it computes the address of a motion record in the SPRITE Motion table.

AUTOMOTION

Sets the control bit in scratchpad (HEX 83C2) that enables the motion interrupt sets MOVING to SPR#+1.
AUTOMOTION assumes you want all SPRITES controlled by AUTOMOTION

MOVING

Tells the AUTOMOTION system the number of sprites that will be moved automatically. When we create a SPRITE it is counted as being an auto-mover by default. We can change that assumption with the MOVING command AFTER invoking AUTOMOTION

Example of how to use MOVING to keep some sprites under manual control:

1. Create five sprites as normal with the SPRITE command
2. Start AUTOMOTION with the AUTOMOTION command
3. In the code right after AUTOMOTION write 3 MOVING
4. Now only the 1st 3 SPRITEs will auto-move. The last two are under your control
   *The advantage of doing it this way is that the interrupt routine will only service the number of sprites "MOVING".   It takes CPU time away from your program to read the VDP RAM and compute where to move a sprite. By not trying to move them all we save more time for your program to run. :-)

```
Example CODE you can try
```

```
\ Create five sprites
NEEDS MOVING FROM DSK1.AUTOMOTION  (this loads grafix & dirsprit)

DECIMAL
   CHAR A   9  0  0   0 SPRITE   0 5  0 MOTION
   CHAR B  12  0 10   1 SPRITE   0 5  1 MOTION
   CHAR C   4  0 20   2 SPRITE   0 5  2 MOTION
   CHAR D  16  0 30   3 SPRITE   0 5  3 MOTION
   CHAR E  16  0 40   4 SPRITE   0 5  4 MOTION

   CLEAR
   AUTOMOTION
   3 MOVING  ( only 1ˢᵗ three SPRITES will be auto-moving)

\ you type:
\   5 MOVING   0 MOVING  1 MOVING  and watch what happens
```

INITMOTION

This word resets the entire sprite motion table in VDP RAM to zero and set MOVING to zero.

STOPMOTION

Sets the control bit in scratchpad (HEX 83C2) that disables the motion interrupt.

Automotion Code Example

```
\ AUTOMOTION Sprite Demonstration Program   BJFox July 28 2019
\ file: DSK3.COINCDEMO

NEEDS DUMP       FROM DSK1.TOOLS
NEEDS GRAPHICS   FROM DSK1.GRAFIX
NEEDS SPRITE     FROM DSK1.DIRSPRIT
NEEDS MOTION     FROM DSK1.AUTOMOTION
NEEDS RND        FROM DSK1.RANDOM
NEEDS HZ         FROM DSK1.SOUND
NEEDS .R         FROM DSK1.UDOTR


\ happy face in 4 pieces
HEX
071F 3F7F 7FFB FFFF PATTERN: TOPLEFT
FFFF F778 7F3F 1F07 PATTERN: BOTLEFT
E0F8 FCFE FEDF FFFF PATTERN: TOPRIGHT
FFFF EF1E FEFC F8E0 PATTERN: BOTRIGHT


 \ named colors for Graphics programs
: ENUM  ( 0 <text> -- n) DUP CONSTANT  1+ ;

1 ENUM TRANS       ENUM BLACK      ENUM MEDGRN
  ENUM LTGRN       ENUM DKBLU      ENUM LTBLU
  ENUM DKRED       ENUM CYAN       ENUM MEDRED
  ENUM LTRED       ENUM DKYEL      ENUM LTYEL
  ENUM DKGRN       ENUM MAGENTA    ENUM GRAY
  ENUM WHT
DROP


DECIMAL
\ GENERATE random motion vectors
: RNDXY  ( -- dx dy ) 127 RND 80 -   127 RND 80 - ;

\ create some motion vectors as double variables (2 cells)
\ use 2@ and 2! on them
CREATE V1    1 ,  0 ,

: .VECTOR   ( -- )  7 0 AT-XY V1 2@  3 .R 4 .R  ;

: NEWVECTOR
        RNDXY 2DUP 1 MOTION  V1 2!  \ change motion & record vector
        .VECTOR ;

\ Usage:  V1 REVERSE
: REVERSE ( motion-vector-addr -- )
        DUP 2@ NEGATE SWAP NEGATE SWAP ROT 2!
        V1 2@  1 MOTION ;

: MAKE-FACE
      TOPLEFT    [CHAR] t CHARDEF
      BOTLEFT    [CHAR] u CHARDEF
      TOPRIGHT   [CHAR] v CHARDEF
      BOTRIGHT   [CHAR] w CHARDEF ;
```

```
: MAKE-SPRITES ( -- )
      \ Ptrn     clr     col  row  spr#
      \ ------   ---     ---  ---  ----
      [CHAR] w  DKYEL    20   80    1 SPRITE
      [CHAR] w  DKRED    10   16    2 SPRITE
      [CHAR] w  LTGRN   125   16    3 SPRITE
      [CHAR] w  MAGENTA 220   16    4 SPRITE
      [CHAR] w  LTRED    10  170    5 SPRITE
      [CHAR] w  DKBLU   125  170    6 SPRITE
      [CHAR] w  LTBLU   220  170    7 SPRITE
;

: TINK    GEN1 2100 HZ -6 DB  40 MS MUTE ;

: BOUNCE ( -- )  \ bounce off each other
          V1 REVERSE
          DKRED 1 SP.COLOR
          TINK
          250 MS \ move back to 1/4 second
          0 0 1 MOTION            \ stop mcving
          15 1
          DO
            I 1 SP.COLOR        \ show frustration
            100 MS
          LOOP
          DKYEL 1 SP.COLOR ;      \ normal happy face color

: ?BREAK   ?TERMINAL IF
             STOPMOTION 8 SCREEN HONK
            4 19 2 1 COLORS
            ." *BREAK*"  ABORT
           THEN ;

\ Sprite #1 moves and changes color if it collides
: RUN
      PAGE ." MOTION:"
      1 SCREEN 4 19 16 1 COLORS
      2 MAGNIFY
      MAKE-FACE
      INITMOTION
      MAKE-SPRITES
      SPR# 1+ MOVING           \ flag all sprites as movers
      AUTOMOTION
      BEGIN
        NEWVECTOR
        3000 RND 300 + 0        \ stay with one vector rnd time
        DO
          COINCALL
          IF   BOUNCE LEAVE
          THEN ?BREAK
        LOOP
      AGAIN ;
```

Lesson 11: NEEDS/FROM

Camel99 Forth has a pair of words that are simple but powerful that you will see in the library and demo files, called NEEDS and FROM. These commands are unique to CAMEL99 FORTH but could be added to any standard Forth system. NEEDS and FROM work together to prevent you from double loading files so you don't waste memory space.

An example would be you write a program that uses the library DSK1.ANSFILES. You load ANSFILES and then your program. No problem. Later you want to load a file viewer program and it also needs DSK1.ANSFILES. You could load ANSFILES again but it is already in the Forth system so you don't need to re-load it. Here is what you could do to file viewer program to make sure ANSFILES is loaded when you need it to be.

```
NEEDS READ-FILE  FROM DSK1.ANSFILES
```

NEEDS looks for a word in the Forth "dictionary", a place where Forth keeps all the names of the commands. If NEEDS finds the word it returns a TRUE value. If it does not find the word it returns FALSE value.

FROM reads the true or false from NEEDS. If FROM sees false it loads the file name that follows, adding that file's words to the dictionary.

Bonus Exercises

1. At the Forth console try typing "NEEDS CODE FROM DSK1.CODE" and you will see that nothing happens! That's because Forth knows that it already has the word CODE in the dictionary.

2. Type NEEDS HCHAR FROM DSK1.GRAFIX and the system will load GRAFIX and switch to 32 column mode like TI-BASIC. Now type it again, exactly the same. What happened?

3. Type NEEDS STUFF .   (with the period on the end) You will see a zero because there is no word called STUFF in the dictionary

4. Type NEEDS FROM . (with the period). You will see -1 which means TRUE because FROM is in the dictionary.

Lesson 12: Programming Tools and Utilities DSK1.TOOLS

Forth is a very low level language. You can think of it as the Assembly Language for the Forth Virtual Machine. Since we are working at this low level, at least when we start our program, before we have made some helper words for ourselves, we need some tools to examine the status of the Forth machine. In CAMEL99 Forth we have TOOLS and TRACE to assist us.

To use the tools type: INCLUDE DSK1.TOOLS at the Forth console.

Dot-ess (.S)

The simplest tool to use can be the handiest when debugging your new Forth program. It is almost essential to have a way to examine the stack without destroying the contents. The word to do that is **.S** pronounced "dot-ess".   You can use .S any time you need to see the stack contents BUT don't want to disturb the stack.

A common method is to write your new word piece by piece at the console and examine the stack after each word is entered.

Consider the example below for the word BOUNDS. BOUNDS converts an address and a length into an end address and start address for setting up a DO LOOP.

```
\ definition for BOUNDS

: BOUNDS  ( addr len -- end start)
        OVER + SWAP ;
```

First we typed a random address and a length. Then we type .S to see the stack. We type OVER and .S to see the stack and confirm that >D000 is on the top of the stack. We type + and confirm the addition and finally type SWAP and .S to see that our program does what we expect.

DEPTH

Another word used to check the stack is called DEPTH. This word simple returns the number of items on the stack.

WORDS

If you want to see if a word is in the Forth dictionary use the word WORDS. You can stop the listing of words by pressing FNCT 4. (break)

DUMP

The DUMP command lets you see the contents of CPU memory anywhere from 0000 to FFFF. Simply type a starting address and the number of bytes you want to see. DUMP will show the raw hex data and also printable text characters are displayed on the right side.

The screen capture shows how to use DUMP to examine the contents of the Forth word TEST that is defined at the top of screen.

- We see the definition starts at address C35A
- We can see the first cell contains the address of the DOCOL routine used by all colon definitions.
- The next cell contains the address of the routine called (S") that converts a counted string into an (addr,len) stack string.
- The length byte of our string is HEX 15 (21) followed by the ASCII characters of the text string.
- The final cell contains >A032 which is the address of the Forth word EXIT which marks the end of the definition.
-
- And see the printable text characters displayed in the right-hand column. All non-printable characters appears as dots.

UNUSED

DSK1.TOOLS gives us the ANS Forth word called UNUSED which returns the number of bytes left in the dictionary.

.FREE

CAMEL99 has the word .FREE that prints the available bytes in three memory spaces, HEAP (low RAM), UPPER RAM and VDP RAM.

VDP DUMP (VDUMP)

Using the power of DEFER in Forth we can change the memory that DUMP reads and so in V2 you can do VDUMP and see the contents of VDP Ram.

SAMS CARD DUMP (SDUMP)

FILE: /LIB.ITC /SAMSDUMP.FTH

Version 2 also has a tool to allow you to view SAMS Card ram in 64K segments. To change the segment use the SEG variable.   The default value of SEG is 1. The usage is the same as DUMP but you can change the segment:

```
    HEX
     2 SEG !            \ set the segment to view
    2000 100 SDUMP   \ DUMP hex 100 bytes of memory at HEX 2000
```

Paste into CLASSIC99 emulator to see it in operation.

Smart Number Printing

When you include the tools file a feature is added to the word '.' All Decimal numbers will be printed as signed numbers but HEX and BINARY numbers will print as UN-signed. This can be very handy when de-bugging code and you want to see numbers in these different bases.

When DSK1.TOOLS is loaded into the system 'dot' becomes a DEFER word. This means you can make it do the operation of any other Forth word. By default with TOOLS, 'dot' runs a word called SMART#S which lets it understand how to print out numbers in different formats.

To go back to normal 'dot' operation the command is: FAST#S

For explanation here is the code for SMART#S and FAST#s as found in DSK1.TOOLS

```
\ From Neil Baud's Ugly Page. RIP Neil :(
\ Print HEX and Binary numbers unsigned
\ Note: Neil has been "creative" with how he used the CASE statement

DECIMAL
: <.>     . ;   \ rename regular 'dot' so we don't lose it
DEFER .         \ make a DEFER 'dot' that we can assign different actions

: (.)      ( n -- ) \ convert n to string & type it
   CASE
     BASE @
     10 OF  DUP ABS 0 <# #S ROT SIGN #>                     ENDOF
     16 OF          0 <#  BEGIN  # #   2DUP OR WHILE REPEAT #>   ENDOF
      2 OF          0 <#  BEGIN  # # # #  2DUP OR WHILE REPEAT #> ENDOF
    ( default case) 0 <# #S #>
    0 ENDCASE
    TYPE SPACE ;  \ now type the string and add a space after it

: FAST#S    ['] <.> IS . ;  \ dot becomes the old version
: SMART#S   ['] (.) IS . ;  \ dot becomes our smarter version

SMART#S      \ turn on smart numbers
```

MARKER: DSK1.MARKER

MARKER is an ANS Forth standard word. It is used to place a "marker" in the dictionary that can erase everything that was INCLUDED into the dictionary after the marker. For example if we made a marker called REMOVE and then INCLUDED a group of tools we could remove all the developer tools by just typing REMOVE.

See the screen capture for an example.
 WORD#1 and WORD#2 are visible.

```
  □K
\ MARKER DEMO   □K
: WORD#1 ." THIS IS WORD #1" ;   □K
MARKER REMOVE   □K
  □K
: WORD#2 ." THIS WAS CREATED AFTER MARKE
R" ;   □K
  □K
WORDS
WORD#2 REMOVE WORD#1 LOCK .FREE UNUSED D
UMP ?80 .ASCII ':' ',' .#### WORDS .ID ?
BREAK .S DEPTH ? MARKER ENDCODE NEXT, CO
DE [CHAR] CHAR CHARS CHAR+ CELL+ CELLS >
BODY INCLUDE PARSE-NAME ; :NONAME : \ (
COLD INIT REPEAT WHILE UNTIL AGAIN BEGIN
 ELSE THEN IF +LOOP LOOP RAKE LEAVE ?DO
DO L> >L RESOLVE BACK AHEAD INCLUDED ?FI
LE INCLD OPN REFILL EOF FSTAT NEWFILE MA
KEPAB VPLACE ?FILERR ?DEVERR ?CARDID FIL
EOP ERR@ FNAME] STAT] REC#] CHARS] RECLE
N] FBUFF] FLG] [PAB DSRNAM
*BREAK*
```

Here is what happens when we invoke the marker we created called "REMOVE".

```
DE [CHAR] CHAR CHARS CHAR+ CELL+ CELLS >
BODY INCLUDE PARSE-NAME ; :NONAME : \ (
COLD INIT REPEAT WHILE UNTIL AGAIN BEGIN
 ELSE THEN IF +LOOP LOOP RAKE LEAVE ?DO
DO L> >L RESOLVE BACK AHEAD INCLUDED ?FI
LE INCLD OPN REFILL EOF FSTAT NEWFILE MA
KEPAB VPLACE ?FILERR ?DEVERR ?CARDID FIL
EOP ERR@ FNAME] STAT] REC#] CHARS] RECLE
N] FBUFF] FLG] [PAB DSRNAM
*BREAK*
  □K
  □K
REMOVE  □K
  □K
WORDS
WORD#1 LOCK .FREE UNUSED DUMP ?80 .ASCII
 ':' ',' .#### WORDS .ID ?BREAK .S DEPTH
 ? MARKER ENDCODE NEXT, CODE [CHAR] CHAR
 CHARS CHAR+ CELL+ CELLS >BODY INCLUDE P
ARSE-NAME ; :NONAME : \ ( COLD INIT REPE
AT WHILE UNTIL AGAIN BEGIN ELSE THEN IF
+LOOP LOOP RAKE LEAVE ?DO DO L> >L
*BREAK*
```

Notice that REMOVE and WORD#2 are GONE!

Typically you can place a MARKER at the top of a program file so as you are working on it you can remove it from memory quickly before you reload the updated version.

I like to name MARKERs with a leading '/'   to mean this word "cuts" something

Examples:
     MARKER /EDITOR
     MARKER /TOOLS

DIR Utility   DSK1.DIR

One of the things that always bugged me about TI-99 BASIC was that there was no DIR command built in. I found out years later that a simple BASIC program could in fact display a disk directory. Nevertheless CAMEL99 Forth has a DIR command that you can load into the system.

Simply INCLUDE DSK1.DIR (It will test if DSK1.ANSFILE is loaded INCLUDE it if not present)

A screen shot and the main code are below. The file format is relative, hex 100 fixed, binary (internal)


```
HEX
: DIR  ( <DSK?.> )   \  needs the '.' ONLY shows file name
        BL PARSE-WORD DUP ?FILE
        RELATIVE 100 FIXED R/O BIN OPEN-FILE ?FILERR
        >R                              \ push handle onto Return stack

        PAD 50 R@ READ-LINE ?FILERR   \ read 1st record. (dev name)
        CR PAD COUNT TYPE CR            \ print device name

        LINES OFF                       \ READ-LINE increments this
        BEGIN
           PAD 50 R@ READ-LINE ?FILERR
        WHILE
           DROP                         \ don't need the byte count
           PAD 0C $.LEFT  ?CR
           ?TERMINAL                    \ check for *BREAK* key
           IF R> CLOSE-FILE             \ if detected we're done here
              2DROP
             CR CR ." *BREAK*" ABORT
           THEN 1 LINES +!
        REPEAT

        R> CLOSE-FILE
        2DROP 2DROP
        DECIMAL
        CR CR LINES @ . ." files"
        HEX ;
```

CAT Utility, DSK1.CATALOG

If you need more information on the files on a disk use the CAT utility. You might choose to keep CAT on DSK2 to save space on DSK1 for library files.

Just INCLUDE DSK2.CATALOG

This gives you the kind of display that is available in the DISK Utility cartridge. Usage is like DIR

```
CAT DSK1.   \ shows all the files on DSK1.
```

CAT will stop at the bottom of the screen and wait for you to press a key to see more catalog records. You can break out of CAT any time with FNCT 4.

```
   OK
INCLUDE DSK2.CATALOG.F
LOADING: DSK2.CATALOG.F
LOADING: DSK1.ANSFILES.F
ANSFILES  FOR CAMEL99 V2 BJF 02APR2018
..........
MAX FILES SET TO 3
LOADING: DSK1.CASE.F
86 BYTES OK
   OK
CAT DSK2._
```

```
.\DSK2\         -TYPE-    -SECT-  -B/REC-
BRAINF_K.F   TXT/VAR        6       80
CAMEL99      PROGRAM       33        0
CATALOG.F    TXT/VAR       11       80
CORE1.F      TXT/VAR       38       80
CORE2.F      TXT/VAR       63       80
DENILE       PROGRAM        5        0
DENILE.F     TXT/VAR       20       80
DUTCHFLG.F   TXT/VAR       19       80
ERRORTEST    TXT/VAR        2       80
ERRTEST.F    TXT/VAR        2       80
FACE         PROGRAM        8        0
FACE.F       TXT/VAR       30       80
GOODNILE.F   TXT/VAR       19       80
HAYESTST.F   TXT/VAR       11       80
MTASKDEM.F   TXT/VAR       10       80
MUSIC.F      TXT/VAR       24       80
PONG.F       TXT/VAR       47       80
PRIMES       TXT/VAR        7       80
RNDCOLOR     PROGRAM        3        0
RNDCOLOR.F   TXT/VAR        5       80
SANSERIF.F   TXT/VAR       17       80
SNAKE.F      TXT/VAR       35       80
PRESS ANY KEY...
```

MORE Utility DSK1.MORE

CAMEL99 Forth has small utility program called MORE (DSK1.MORE) that lets you see the contents of a text file on the screen. MORE let's you stop the screen by pressing any key or exit the viewer by pressing "ESC". To load MORE, type: INCLUDE DSK1.MORE (it will load ANSFILES if automatically for you)

We can now view the MORE file that we just compiled by typing:

MORE DSK1.MORE

While the file is scrolling on the screen, press any key to stop it.
Then press BREAK (FCNT 4) to stop displaying the file.

```
 OK
MORE DSK1.MORE.F
 \ MORE.F    SIMPLE FILE VIEWER
 \ DEFAULT IS DV80 FILES BUT YOU CAN CHAN
GE IT WITH TI-99 FILE COMMANDS

HEX
: DV80    DISPLAY 50 VARI SEQUENTIAL ;
 \ HEX 50 IS 80 BYTES/RECORD

: MORE    ( <FILENAME>)
          BL PARSE-WORD DUP ?FILE
          DV80 R/O OPEN-FILE ?FILERR
          >R                   \ PUSH H
ANDLE ONTO RETURN STACK
...

>>Esc<<
```

DSK1.SHELL

New in V2.62 is DSK1.SHELL. Include this file to load all these disk utilities at once and the CAMEL99 system as well.

COPY <PATH1> <PATH2>

Copy path1 to path2. Warning. There is no protection. You will not be asked "Are your sure?"

It will just copy over path2. BE CAREFUL.

WAITFOR <PATH>

This command is best used with Classic99 emulator. It waits for you to past a file into the emulator and saves it as a DV80 file on the TI-99 DISK. Very handy to covert PC files to TI-99 format.

'SEE' the De-Compiler   DSK1.SEE

A handy tool for looking inside the Forth system is SEE. SEE is the Forth de-compiler. It can read the contents of a Forth word and see what other words are compile inside it. It CANNOT however show you the source code of Assembly language words. The CAMEL99 de-compiler will however show you a machine code version of the word. You will have to look up the meaning of the machine code numbers if you want to know the CODE instruction names.

To load the de-compiler simply type:
```
INCLUDE DSK1.SEE
```

DSK1.SEE automatically compiles DSK1.CASE and DSK1.TOOLS for you. To decompile a word type:

```
SEE <word>
```

This is what you should see:

```
    OK
INCLUDE DSK1.SEE.F
LOADING: DSK1.SEE.F
LOADING: DSK1.CASE.F
86 BYTES
LOADING: DSK1.TOOLS.F
TOOLS for CAMEL99 FORTH V2
MARKER ? DEPTH .S ?BREAK WORDS DUMP .FRE
E
FREE MEM
HEAP    : 7808
PROGRAM: 15236
VDP     : 10199
........ OK
    OK
SEE WORDS
: WORDS
    CR O LATEST @ DUP .ID SPACE SWAP
    1+ SWAP NFA>LFA @ DUP ?BREAK O=
    ?BRANCH -18   DROP CR U. SPACE
    S" WORDS" TYPE  ;

  OK
  _
```

How SEE Works

Camel Forth is a "threaded code" system. This means that the compiler does not generate machine code, but rather creates lists of addresses. These addresses can point to other routine addresses but eventually they point to real machine code. So to decompile this kind of Forth means you have to read through each list and find the name of the word associated with the address, print the name and move ahead two bytes and read the next address. You continue doing this until you get to the address of the routine called EXIT, which is the Forth equivalent of "RETURN" or RT in assembler.

Each Forth word begins with the address of a machine code routine that is the actual "interpreter" for that kind of word. There is a special routine for variables, constants, new routine definitions (colon definitions) and one for something called USER variables that are used when tasks need their own copy of some variables. These special routines let you identify the "type" of each Forth word even though Forth is not a "typed" language. Armed with that "type" information you can figure out how to print out the info for any given Forth word.

Three Separate De-Compilers

There are three different "de-compilers" in this implementation. One for the "primitives" which are real machine code, one for data types and one for colon definitions.

CODE Words

When SEE encounter a "CODE" word it simply prints the machine code version of the word. Making a dis-assembler would be a whole other project. The de-compiled machine code can actually be type into the CAMEL Forth and used as is so that has some benefit.

DATA Words

Another de-compiler is for variables, constants and USER variables so you see what there value is.
This is a little bit frivolous because you can do that with the Forth interpreter anytime you want but it makes the thing consistent for the end user.

Colon Definitions

The final de-compiler is for "colon" definitions. (Forth words) For some language constructs SEE shows what is compiled in the Forth routine rather than re-creating the exact source code. This can be a difficult thing to understand so an explanation follows.

Under the Hood of IF/THEN and LOOPS

The IF ELSE THEN and BEGIN UNTIL words do not compile words with their same names. They actually compile little code routines called BRANCH and ?BRANCH. In fact the word BEGIN doesn't compile anything into a program. It's just a general purpose label for loops to branch back too! So BEGIN disappears when you decompile Forth code.

The branching words are always followed the BACK or AHEAD. These are magic words that compute the number of bytes the program has to jump. Positive numbers jump forward and negative numbers cause the program to jump back. So if we write a word like this:

```
: IFTEST      IF          TRUE      ELSE       FALSE   THEN ;
```

The de-compiled code looks like this:
```
: IFTEST    ?BRANCH 8    TRUE      BRANCH 4     FALSE        ;
```

This is explained as follows:
?BRANCH will jump 8 bytes forward if the top of stack is zero. This will put the program right at the word FALSE, which will run and then hit the semi-colon and return to Forth.
IF… the top of stack is NOT zero ?BRANCH will not jump and TRUE will run and then the word BRANCH will run. BRANCH is a fancy name for GOTO. BRANCH will jump 4 bytes forward, skipping FALSE and running the semi-colon which returns to Forth. In TI-XBASIC the de-compiled Forth code would be equivalent to this:
```
10 IF TOPOFSTACK=0 THEN 40
20 TOPOFSTACK=TRUE
30 GOTO 50
40 TOPOFSTACK=FALSE
50 RETURN
```

Likewise BEGIN UNTIL loops do this:

```
: TEST1
    10    BEGIN  1-  DUP 0= UNTIL ;
```

```
\ decompiled output
: TEST1
      LIT 10        1- DUP 0= ?BRANCH -8  ;
```

The word LIT tells Forth that the following number is to be read as a "literal" number. The compiler puts this in the word whenever it sees a number in your code. Then notice how ?BRANCH is followed by a negative number. This is how many bytes it must jump back to get to the beginning of the loop.

NOTICE: It only has to jump back to the **'1-'** in the code because that's where BEGIN was.

Here is a BEGIN AGAIN loop:
```
: TEST2   BEGIN   99 DUP DROP DROP   AGAIN ;

\ decompiled output
: TEST2        LIT 99 DUP DROP DROP   BRANCH -12  ;
```
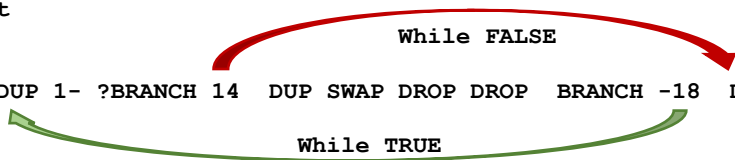

And the most complicated is the BEGIN WHILE REPEAT loop:
```
: TEST3
      10   BEGIN   DUP 1-   WHILE      DUP SWAP DROP DROP   REPEAT     DROP ;


\ decompiled output
: TEST
                                      While FALSE

    LIT 10        DUP 1- ?BRANCH 14   DUP SWAP DROP DROP   BRANCH -18   DROP ;
                                      While TRUE
```

Interpretation of TEST3
- Put literal 10 on stack
- DUP and subtract 1
- If TOP of stack=0, ?BRANCH 14 bytes forwards to DROP and semi-colon to end the loop
- If TOP of stack<>0 , do DUP SWAP DROP DROP
- BRANCH 18 bytes backwards to the first DUP

*Some Things Cannot De-compile

CAMEL99 has seven words (EXIT, ?BRANCH, BRANCH LIT, DOCOL, @,C@) that won't decompile properly because they reside in High-speed RAM and don't end with the Forth NEXT address or are part of another word. If you try it they will just print nonsense but you can stop the de-compiler with the BREAK key. (FCTN 4)

The Trace Utility   DSK1.TRACE

A handy feature in the original TI Forth was the TRACE utility. CAMEL99 has a slightly simpler version in DSK1.TRACE. Trace is a clever use of the Forth language. The TRACE utility re-defines the 'colon' compiler! Yes the keystone of the language is re-defined so that every word that you create afterwards has the code for the TRACE built into it. Here is the definition for the very curious:

```
\ re-define COLON to compile (TRACE) routine into every definition
: :     ( -- ) !CSP  HEADER (:NONAME)  COMPILE (TRACE) ;
```

Now anything that's defined/re-defined with ':' will be traceable. Inside the (TRACE) word there is a variable called TRACE. By setting this variable to TRUE or FALSE, we can control if (TRACE) runs or not.

Below is some example code. Notice how we have redefined the variables, @,!,+! and DUP and more. This is so that these words will also be re-compiled with (TRACE) built-in.

```
\ TEST CODE
 VARIABLE X
 VARIABLE Q

 : X     X ;      : Q    Q ;

 : DROP   DROP ;
 : !       !    ;
 : @       @    ;
 : +!      +!   ;
 : DUP    DUP  ;
 : 1     1 ;
 : 2     2 ;
 : 3     3 ;
 : U.    U. ;

 : WORD1  ." This is word 1" 1 2 3 DUP  X !  ;
 : WORD2  CR ." Word 2 does nothing around here!"  99 ;
 : WORD3  CR ." Don't listen to those other words!!!!"
         DROP DROP DROP DROP   X @ Q +!
         Q @ U. ;
 : TEST
        Q OFF
        4 0 DO
          WORD1 WORD2 WORD3
        LOOP ;
```

Forth has the words ON and OFF which set a variable to TRUE or FALSE. So all we need to do is type:
```
TRACE ON
\ or
TRACE OFF
```

This is how you control the TRACE utility. The following screen shots show you what happens when you trace a word with TRACE ON.

TRACE Screen Captures

```
 OK
TRACE OFF  OK
WORD1 THIS IS WORD 1 OK
```

```
 OK
TRACE ON  OK
WORD1
>WORD1_ !<-THIS IS WORD 1
>1   !<-
>2   !1 <-
>3   !1 2 <-
>DUP  !1 2 3 <-
>X   !1 2 3 3 <-
>!   !1 2 3 3 C3F4 <- OK
```

Notice how with TRACE ON, we see the DATA stack contents and the text name of each word that runs as well.

```
 OK
TRACE OFF  OK
WORD1 WORD2 WORD3 THIS IS WORD 1
WORD 2 DOES NOTHING AROUND HERE!
DON'T LISTEN TO THOSE OTHER WORDS!!!!3
OK
```

```
WORD1 WORD2 WORD3
>WORD1_ !<-THIS IS WORD 1
>1   !<-
>2   !1 <-
>3   !1 2 <-
>DUP  !1 2 3 <-
>X   !1 2 3 3 <-
>!   !1 2 3 3 C3F4 <-
>WORD2  !1 2 3 <-
WORD 2 DOES NOTHING AROUND HERE!
>WORD3  !1 2 3 99 <-
DON'T LISTEN TO THOSE OTHER WORDS!!!!
>DROP  !1 2 3 99 <-
>DROP  !1 2 3 <-
>DROP  !1 2 <-
>DROP  !1 <-
>X   !<-
>@   !C3F4 <-
>Q   !3 <-
>+!  !3 C3FE <-
>Q   !<-
>@   !C3FE <-
>U.   !6 <-6  OK
_
```

When we have TRACE ON, and type WORD1 WORD2 WORD3, we can see each part of the words activate by name and how they affect the DATA stack. This can be very handy when there is a bug you don't understand.

ANS/ISO Forth Files

CAMEL99 Implements a sub-set of the ANS Forth File words. This allows you to do things like you do in BASIC with the OPEN, CLOSE, PRINT #x: and INPUT #x: commands. The big difference to get use to here is that where BASIC uses the file identifiers #1, #2, #3 etc. the Forth file system gives you a number between 1 and the maximum number of files allowed. This number is called a "file handle" and it is completely up to you how you keep track of it. The simplest way of course is to create variables and store the handle in your variable. However sometimes you just want to drop the handle on the stack and use it from there. Alternatively you can put it on the Return Stack but you must remove it before your word ends or the program will CRASH!

Another difference from BASIC is that you decide what happens when there is an error. Each file command returns a 0 if successful and an error number if there is something wrong. Your program can decide what to do as you see fit. CAMEL99 gives you a generic ABORT on file error word called ?FILERR which is a good place to start.

By the default in the source code file called DSK1.ANSFILES the maximum number of open files is set to three (3) like Basic. (Version 2.0.xx does not allow more than 3 files)

No Cassettes Allowed

At the time of publishing the CAMEL99 file system does not support the Cassette tape file system.

File MODE Control

On top of the Forth file words there are also the TI-99 file system words that control the "file access mode". They do the same things in Forth as they do in BASIC. These words are:

- **DISPLAY**   : set file system to TEXT mode. Records are ASCII
- **INTERNAL**   : not defined in CAMEL99. See BIN below

- **SEQUENTIAL**: file records can only be read in order
- **RELATIVE**   : file records can be accessed in random order

- VARI   : set variable record length. (VARIABLE is already taken by Forth)
    - The record length PRECEDES the word VARI
- **FIXED**   : set fixed record length
    - The record length PRECEDES the word FIXED

-

These words in BASIC are in CAMEL99 but ANS Forth has other commands to do the job

- **UPDATE**   : set file system to allow reading or writing. Use R/W
- **INPUT**   : set system to allow file reading by your program. Use R/O
- **OUTPUT**   : set system to allow file writing by your program. Use W/O

This command does not have an equivalent in ANS Forth. It is available to use.

- **APPEND**   : open file and move position to end

BASIC Files vs Standard Forth

OPEN

Opening a file can be the most complicated function so we will begin there. We will OPEN a standard DV80 Text file in input mode. DV80 is an abbreviation for "DISPLAY,VARIABLE 80". Like BASIC CAMEL99 Forth use that file format as the default file format. Below are the simplest OPEN statements in both languages.

BASIC
```
10 OPEN #1:"DSK1.DEMO"
```
Forth
```
S" DSK1.DEMO" R/W OPEN-FILE ( -- fid error# )
```

They are quite comparable but notice that in Forth there is no "#1". This is because the file identifier is left sitting on the stack for your program to do something with it. Also the file-status is also on the stack for you to determine what to do with that as well. This is a little bit like using "ON ERROR" in Extended BASIC. You can make a routine that picks up the error and does anything you want.
 So OPEN-FILE gives you more control but also you have to think about more things.

Default Error Handler

The default way to handle file errors is to use the CAMEL99 word "?FILERR". This will read the error# and if it's anything but zero, the program stops and reports "File error X".   That gives you what BASIC gives you.

File Identifier Handling

What should we do about the file identifier? Well how about we put it somewhere safe like a VALUE?
A Forth VALUE is like a CONSTANT that you can change with the word 'TO'.
How about we call that VALUE "#1" just to feel at home.

Wait a minute! Can't Forth put some of this together for us and make one command? It sure can. Here is a new word you could make called OPEN .   Open will keep track of the file identifier (the handle) with VALUEs

INCLUDE DSK1.VALUES

```
0 VALUE #1    \ create some file id holders
0 VALUE #2
0 VALUE #3

: OPEN  ( string length fam -- fid) OPEN-FILE ?FILERR ;

\ Now we can open a DV80 file like this

S" DSK1.DEMO"  DISPLAY SEQUENTIAL 80 VARI R/W OPEN TO #1
```

Remember if you wanted to have an "ON ERROR" mechanism we would replace ?FILERR with our your own custom error handler word.

Making it Easier

The DSK1.ANSFILE library file contains the word DV80 which does all this for you. You can make similar Forth words for special file formats that you like to use.

```
DECIMAL
: DV80   ( -- ) UPDATE DISPLAY SEQUENTIAL 80 VARI ;
```

File Format Modifiers

Below is a comparison of the full BASIC OPEN statement with the equivalent in Forth using ?FILERR and a VALUE to hold the 'fid'. The arrows show the equivalent commands in BASIC and Forth.

BASIC

```
10 OPEN #1:"DSK1.DEMO",INPUT,DISPLAY,SEQUENTIAL,VARIABLE 80
```

FORTH

```
DECIMAL
    S" DSK1.DEMO" DISPLAY SEQUENTIAL 80 VARI R/O OPEN-FILE ?FILERR TO #1
```

BASIC File Statements	FORTH equivalent	Comment
OPEN	OPEN-FILE	Marks a file as available for the system to access. Leaves a handle and error# on stack. If error=0 all if good.
#1	File-ID (the handle)	The file handle is Left on the stack for the programmer to manage. Keep it safe. Store it on the stack or in a VARIABLE or a VALUE. The handle MUST be used to access the file or to close the file.
"DSK1.DEMO"	S" DSK1.DEMO"	Forth stack string. It is an address and length on the stack that is picked up by OPEN-FILE
INPUT	R/O	"READ/ONLY" leaves a number on the stack called the "file access mode" (FAM)
UPDATE	R/W	"READ/WRITE" sets the FAM for UPDATE mode and leave the FAM number on the stack for OPEN-FILE to pick-up
OUTPUT	W/O	"WRITE/ONLY" sets the bit in the FAM that means OUTPUT mode and leaves the FAM number on the stack for OPEN-FILE
INTERNAL	BIN	Sets file system to BINARY format.
DISPLAY	DISPLAY	Sets the bit in the FAM that means use ASCII characters in this file. (Text files) Like BASIC, text files are the default in CAMEL99 unless the BIN command is used in the file specification.
VARIABLE ###	## VARI	Sets FAM to indicate variable record length and sets the record length in the [6]PAB to ##. (## must be 1 to 255)
FIXED ###	### FIXED	Sets the record length ### bytes, fixed length format

USING BIN

It's important to understand that **BIN must FOLLOW R/O, R/W or W/O.** This is because BIN takes the file mode value on the stack and changes it so the file mode is BINARY (TI BASIC "internal") and leaves that changed file mode value on the stack. Ie: BIN is a modifier word and cannot be used alone.

---

[6] PAB = "peripheral access block". A memory structure in VDP RAM that controls the TI-99 file system code

OPEN-FILE Example #2:

Using a string variable for the file name.

BASIC

```
10 DSK$="DSK1.MYFILE"
20 OPEN #1:DSK$,INPUT,INTERNAL,RELATIVE, FIXED 100
```

FORTH Method with STRINGS library

```
INCLUDE DSK1.STRING
INCLUDE DSK1.VALUES

32 DIM DSK$        \ DIM creates DSK$ as a counted string in memory
0 VALUE #1         \ use a value to hold the file ID

" DSK1.MYFILE" DSK$ PUT      \ put some text into DSK$
DSK$ COUNT  RELATIVE 100 FIXED R/O BIN OPEN-FILE ?FILERR TO #1
```

Differences to Note:

- DSK$ is a counted string so we must use COUNT to convert it to a "stack string" for OPEN-FILE to use it.
- In the example above we are opening the file in "INPUT, INTERNAL" format as we did in BASIC, but we did not need to use the words "INPUT and "INTERNAL". They are replaced by **R/O BIN**. This means: read-only, binary. These two ANS Forth words do the same thing as using the words INPUT and INTERNAL in the BASIC version.

It is VERY important to understand that R/O leaves a number of the stack that is the "file access mode" information for OPEN-FILE to take from the stack.   R/O selects INPUT mode. BIN picks up the number from R/O and flips a bit to mark it as binary and puts the new number back on the stack.
SEE SECTION: **FORTH File Access Glossary** for full details on ANS Forth file words.

File Operation Errors

In TI-BASIC a file system error simply stops you program a drops you back into the BASIC console. In this does not happen unless you want it to happen. The way to emulate TI-BASIC is to use the word ?FILERR after each file word that returns an "ior" (input/output response) This is nothing more than the error number. An error number of zero means all is good. If ?FILERR gets a zero it does nothing. If it gets a number it ABORTs to the Forth console and says: "File error X" where is X is the error number.

What About BASIC's ON ERROR?

You can make anything happen by creating your own error handling word. For example here is the definition of ?FILERR from the ANSFILES library file:

```
: ?FILERR  ( ior -- )
    ?DUP IF
        CR
        CR ." Err# " . ."  Hndl=" LASTH @ .
        [PAB DUMP]
        FATAL ABORT" Handles reset"
    THEN ;
```

You can replace ?FILERR with anything you like even with DROP for example, which would ignore the error completely. I would advise against that, but this is Forth. You are in command of your program.

FILE INPUT

The Standard Forth word for file input is a little more complicated than we are used to in BASIC. The equivalent word in ANS Forth is READ-LINE. READ-LINE is much more primitive than INPUT #1. READ-LINE simply reads the next available record in the file and puts in in a memory location that you give it. What kind of data is in the record is not considered. That is the programmer's job to know that.

This is similar to using the BASIC statement INPUT #1: X$ when the data in the file is not a number but is a string. BASIC will halt with an error but in Forth you won't get an error for reading the wrong data. The data just goes into memory and your program has to decide if it can work with that data or not.

READ-LINE needs three input arguments: (address len handle)
1. Address is just the location of a buffer big enough to hold one record of data. The biggest it can be on the TI-99 is 256 bytes.
2. Len is the number of bytes we want to read
3. Handle is the file ID or handle that we got when we opened the file.
4.

READ-LINE returns three output arguments (len' flag err#)
1. Len' is the actual number of bytes the we read from the file
2. Flag is TRUE if there is data for us, FALSE if not
3. Err# is zero if there was not a problem, or it is a system error number

Below is an example of how we might use READ-LINE :

```
DECIMAL
VARIABLE HNDL               \ variable for the file handle (any name can be used)
CREATE BUFF  80 ALLOT   \ simple way to make some space for a record

: TYPEFILE ( addr len -- )
      R/O OPEN-FILE ?FILERR HNDL !    \ file mode is set by you later
      BEGIN
         HNDL @ EOF 0=
      WHILE
        BUFF DUP 80 HNDL @ READ-LINE ?FILERR ( -- buff len flag)
\                    ^
\   ** Notice this DUP. We make a copy for printing it later

         DROP                \ don't need the flag
         CR TYPE             \ type the BUFF on a new line
      REPEAT
      HNDL @ CLOSE-FILE ?FILERR ;


\ Usage:

 S" DSK1.CASE" DV80 TYPEFILE
```

EOF Function

Notice how we used the EOF word in a BEGIN WHILE REPEAT loop.
This is a convenient way to read or write to a file.

The EOF function takes a file handle to select the file you want to check on.
EOF returns one of the following numerical values:

- EOF = 0, if the file is not at the end.
- EOF = 1, if the file is at the logical end of file (ie: there is no more data to read)
- EOF = 2, if the file hits the physical end of file (ie: disk is full)

We test the EOF with 0= we get a TRUE or FALSE flag on the stack.
WHILE reads the FLAG and if it is TRUE, the loop will REPEAT.
When EOF <> 0 the 0= flag will return FALSE and so the loop will end.

In the last line we CLOSE-FILE with handle #1 and then test if there was an error closing the file with ?FILERR

FSTAT Function

The TI-file system also has a file status command. This is called FSTAT in CAMEL99 Forth.

```
FSTAT Bit      *Information*
---------      -------------
7  If set, the file does not exist. If reset, the file does exist.
   On some devices, such as a printer, this bit is never set since
   any file could exist.

6  If set, the file is protected against modification. If reset, the file
   is not protected.

5  Reserved for possible future use. Fixed to 0 by the current
   peripherals.

4  If set, the data type is INTERNAL. If reset, the data type is DISPLAY
   or the file is a program file.

3 If set, the file is a program file. I f reset, the file is a data file.

2  If set, the record length is VARIABLE. If reset, the record length is
   FIXED.

1  If set, the file is at the physical end of the peripheral and no more
   data can be written.

0  If set, the file is at the end of its previously created contents.
   You can still write to the file ( if it was opened in APPEND, OUTPUT, or
   UPDATE mode), but any attempt to read data from the file causes an error.
```

LINPUT: Simpler Way to Read a Record

The ANS Forth word READ-LINE is quite low level. Can't we can create a word that is more like Extended BASIC's LINPUT? Yes we can!

In your program INCLUDE DSK1.LINPUT to get it. For a BASIC programmer LINPUT makes things much more like home.

TYPEFILE Example with LINPUT

The following program shows you how to use LINPUT and also uses the EOF function.

```
\ show the contents of a DV80 file
INCLUDE DSK1.TOOLS        \ for debugging
INCLUDE DSK1.ANSFILES     \ ANS Forth file words
INCLUDE DSK1.LINPUT

\ Print the contents of a DV80 file
DECIMAL
VARIABLE #1                    \ this variable will hold the file handle
VARIABLE A$ 80 ALLOT           \ STRING variable with 80 bytes of space

: TYPEFILE ( addr len -- )
      R/O OPEN-FILE ?FILERR #1 !
       BEGIN
         #1 @ EOF 0=
       WHILE
         A$ #1 @ LINPUT
         A$ COUNT CR TYPE
       REPEAT
       #1 @ CLOSE-FILE ?FILERR
;
\ Usage:
 S" DSK1.CASE" DV80 TYPEFILE
```

Notice

There are two things for you to notice about this code.

1. We did not load the strings library yet we were able to make a string variable with the word VARIABLE. That's because a string is just a block of memory and the first byte contains the length. Using ALLOT lets us take some dictionary memory for our string so it's no problem.

2. We used a normal variable to hold the file identifier (also called the handle). In other examples we used a VALUE. That was more for convenience than necessity. With a VARIABLE we have to use fetch and store (@ !) to get and assign a number but Forth doesn't care how a number gets onto the stack, as long it's the correct number for the next word to use!

FORTH File Access Glossary

The following file word descriptions come from: <u>http://forth-standard.org/standard/file</u>

Definition of Stack Diagram Terms (with TI-99 specifics)

c-addr   A "counted" string address in CPU RAM. (means the first byte is the length)

len   The length of a string. Maximum 255 bytes

fam   File Access Mode. A set of bits that controls how the TI-99 will open or create a file.

fid   File Identification. A [7]number that the system gives to the program to select a file

ior   Input/Output response. This is a fancy name for the error number returned after any file operation. If the 'ior'=0 there are no errors.

?   A Boolean flag. TRUE or FALSE.

OPEN-FILE (c-addr len fam -- fid ior)

Open the file named in the character string specified by *c-addr u*, with file access method indicated by *fam*. The meaning of values of *fam* is implementation defined. If the file is successfully opened, *ior* is zero, *fileid* is its identifier, and the file has been positioned to the start of the file. Otherwise, *ior* is the implementation-defined I/O result code and *fileid* is undefined.

CLOSE-FILE (fid -- ?)

Close the file identified by *fileid*. *ior* is the implementation-defined I/O result code.

READ-LINE (c-addr u1 fid -- u2 flag ior)

Read the next line from the file specified by *fileid* into memory at the address $c\text{-}addr$. At most u_1 characters are read. Up to two implementation-defined line-terminating characters may be read into memory at the end of the line, but are not included in the count u_2. The line buffer provided by $c\text{-}addr$ should be at least u_1+2 characters long. If the operation succeeded, *flag* is true and *ior* is zero. If a line terminator was received before u_1 characters were read, then u_2 is the number of characters, not including the line terminator, actually read ($0 <= u_2 <= u_1$). When $u_1 = u_2$ the line terminator has yet to be reached. If the operation is initiated when the value returned by FILE-POSITION is equal to the value returned by FILE-SIZE for the file identified by *fileid*, *flag* is false, *ior* is zero, and u_2 is zero. If *ior* is non-zero, an exception occurred during the operation and *ior* is the implementation-defined I/O result code.

An ambiguous condition exists if the operation is initiated when the value returned by FILE-POSITION is greater than the value returned by FILE-SIZE for the file identified by *fileid*, or if the requested operation attempts to read portions of the file not written. At the conclusion of the operation, FILE-POSITION returns the next file position after the last character read.

WRITE-LINE (c-addr u fileid -- ior)

Write *u* characters from *c-addr* followed by the implementation-dependent line terminator to the file identified by *fileid* starting at its current position. *ior* is the implementation-defined I/O result code.

At the conclusion of the operation, FILE-POSITION returns the next file position after the last character written to the file, and FILE-SIZE returns a value greater than or equal to the value returned by FILE-POSITION.

---

[7] This is commonly called a file handle in DOS, LINUX or UNIX

CREATE-FILE (caddr len fam -- fid ior)

Create the file named in the character string specified by *c-addr* and *u*, and open it with file access method *fam*. The meaning of values of *fam* is implementation defined. If a file with the same name already exists, recreate it as an empty file. If the file was successfully created and opened, *ior* is zero, *fileid* is its identifier, and the file has been positioned to the start of the file. Otherwise, *ior* is the implementation-defined I/O result code and *fileid* is undefined.

FILE-POSITION (fileid -- ud ior)

ud is the current file position for the file identified by *fileid*. *ior* is the implementation-defined I/O result code. *ud* is undefined if *ior* is non-zero. In the TI-99 n cannot be greater than 32,567.

REPOSITION-FILE (fid -- ior)

Reposition the file identified by *fileid* to *ud*. *ior* is the implementation-defined I/O result code. An ambiguous condition exists if the file is positioned outside the file boundaries.
At the conclusion of the operation, FILE-POSITION returns the value *ud*.

DELETE-FILE   (caddr len – ior)

Delete the file named in the character string specified by *c-addr u*. *ior* is the implementation-defined I/O result code.

File Access Mode Control

In the ANS Forth file word set, the file mode is controlled by three words which are explained below. Forth the TI-99 programmer you should know that all the conventional TI-99 words (DISPLAY, APPEND ETC…) actually set bits in a variable called FAM. (file access mode)   When you use the ANS Forth words R/W, R/O, or W/O you are simply putting the contents of FAM on the data stack for other file words to use. See file access examples for usage details.

R/W   (-- fam)

fam is the implementation-defined value for selecting the "read/write" file access method. This is equivalent to using the BASIC key word "UPDATE".

R/O   (-- fam)

fam is the implementation-defined value for selecting the "read only" file access method. This is equivalent to using the BASIC key word "INPUT".

W/O   (-- fam)

fam is the implementation-defined value for selecting the "write only" file access method. This is equivalent to using the BASIC key word "OUTPUT".

BIN   (fam -- fam')

Modify the implementation-defined file access method fam_1 to additionally select a "binary", i.e., not line oriented, file access method, giving access method *fam*. This is equivalent to using the BASIC key word "INTERNAL".   BIN must ALWAYS follow R/W, W/O or R/O.

BLOCK an Alternative File System

When Forth was first invented it was so long ago that many computers did not have a file system. Charles Moore would use the raw sectors of the disk system grouped together in 1024 byte units that he called a BLOCK. Each BLOCK had a number from 0 to the end of the disk. If there were more than one disk then the numbering just continued on to the next disk. It was up to the programmer to use these BLOCKs as they saw fit. They could contain text or binary data, programs or databases. It was up to your imagination how to use them.

The other Forth systems for TI-99 use disk blocks. TI-Forth used raw disk sectors but newer systems have taken to using TI-99 file using the FIXED format so that the file can be randomly accessed like records. By combining eight 128 byte records we get a 1K byte block.

If you INCLUDE DSK1.BLOCKS, Camel99 Forth loads up the code to give that same capability which means you can exchange data and source code with FbForth and TurboForth users. Please note that source code will rarely be usable exactly as it is found in the other systems, as they are different dialects of the Forth language but with understanding and careful editing it is possible to make the code work.

Virtual Memory

The block file system gives you something called virtual-memory. This means that the disk drive becomes an extension of the memory by using the low RAM memory in the TI-99. Each time you use the word BLOCK with a number parameter, a chunk of the disk file is brought into a memory buffer. The address of the buffer is left for you to use on the Forth DATA stack. If you ask for the same BLOCK number a second time the system checks if it already has that BLOCK in memory. If it is there the same buffer address is returned to your program. If it is not in memory it is assigned a buffer and the data flows into that buffer. If all the buffer are in use The last buffer used is overwritten by the new request. However if the data in the buffer is flagged as updated (by the UPDATE command) it is written back to the disk drive before the new block is brought into RAM. This all happens automatically with no supervision by your program.

Buffer Locations

By default the Camel99 BLOCK system has three buffers at the bottom of low RAM. This leaves you with about 5K bytes free starting at HEX 2000.

Changing No. Of Buffers

It is possible for to change the number of buffers by changing the constant #BUFFS in the DSK1.BLOCK file. This must be done before you compile DSK1.BLOCKS into you program. A safe maximum number is 6 buffers leaving just under 2K in the Low RAM for other purposes.
The minimum for #BUFFS is two.

>***Be careful using MALLOC with BLOCK. MALLOC does not check if other programs are using the LOW RAM so you could MALLOC memory right into a the block buffers if you don't pay attention.**

BLOCK File Glossary

Here is the lexicon of words you need to know to use the BLOCK file system. For full details study the code in the file, but these words will let you use the BLOCK system for most purposes.

VARIABLE HIGHBLK   sets the highest block that can be used. Default is 79. MAKE-BLOCKS also sets it.
VARIABLE BHNDL   holds the file handle used by the BLOCK file that is open.
ACTIVE   Counted text string. Holds the path for the block file that is in use

UPDATE (--)
Marks the BLOCK in use are having been changed. It will be written to disk when FLUSH is executed

BUFFER (n -- addr)
Given a number n, returns the address of the first character of the block buffer assigned to block n. The contents of the block are unspecified.

BLOCK   (block# --- addr)
Returns the address of the first character of the block buffer assigned to the disk block# in the currently active block file.

FLUSH (--)
Transfer the contents of each UPDATEd block buffer to the disk. Mark all buffers as unmodified.

EMPTY-BUFFERS (--)
Unassign all block buffers. Do not transfer the contents of any UPDATEd block buffer to the disk.

DF128
Sets the file-access-mode for the TI-99 file system to DISPLAY FIXED 128. This is the file format of the block file. You can use this word to set the file system if you needed this format.

OPEN-BLOCKS (path len --)   Usage: S" DSK1.MYBLOCKS" OPEN-BLOCKS
Given a valid path OPEN-BLOCKS attempts to open the file. The path is remembered in the string variable called ACTIVE.

CLOSE-BLOCKS (--)
Closes the active BLOCK file.

MAKE-BLOCKS (n path len --)   Usage: 45 S" DSK1.MYBLOCKS" MAKE-BLOCKS
This will attempt to create a file with 45 BLOCKS. (0 ..44) Each block is filled with the space character as it is added to the file. If the disk cannot hold 45K of data the process will abort with an error. This can take some time for a big file on floppy disk drives.

Make some Noise: DSK1.SOUND

Sound is a big part of the fun with TI-99/4A. The library file to use for sound access is called DSK1.SOUND. Including the SOUND file will give you a set of words to control the frequency, volume and duration of a sound like you have in BASIC. However unlike BASIC each part of that control is a separate word. At the end of the chapter we will show you how to combine the words to create a SOUND word similar to BASIC.

IMPORTANT Difference

The biggest difference for you to know is that SOUND in BASIC, with a negative duration, can continue in the background like a separate task after you invoke the SOUND command. This DSK1.SOUND simply delays your program for the duration of the sound. If you need background sound It is possible to build a task to play the sound with the multi-tasker or you can look and DSK1.VDPBGSND for a background sound list player. Also, because the lexicon controls each aspect of sound with separate words you can start a sound and it will keep running until you MUTE the sound channel, so it is possible to get sound running while other things are happening.

Sound Control Overview

The 9919 chip has 4 channels. The first three can generator tones and the fourth one is different and normally generates noise. For each channel there is an oscillator and an attenuator. (Audio attenuators are a volume control that reduce signal level)
The sound library uses two variables to control which sound generator it is controlling. The variables are:

```
VARIABLE OSC               \ holds the active OSC value
VARIABLE ATT               \ holds the active ATTENUATOR value
```

For the programmer there are four words that control which generator you are talking to:
```
\ sound generator selectors
: GEN1    ( -- )   OSC1  ATT1  GEN! ;
: GEN2    ( -- )   OSC2  ATT2  GEN! ;
: GEN3    ( -- )   OSC3  ATT3  GEN! ;
: GEN4    ( -- )   OSC4  ATT4  GEN! ;
```

These words use GEN! to plug the correct values into the variables OSC and ATT.

Important

The GEN1...GEN4 words above must be the first word you use in any sound control code to select the generator you want to control. Once a generator is selected it remains selected until a new Generator is chosen.

Frequency Control: HZ

The word to control the frequency of a generator is 'HZ'. (short form of Hertz the unit for Frequency)
This means you can type:
```
        DECIMAL
        GEN1 440 HZ
```

And a 440 Hertz (cycles per second) tone will be running in the TI-99. The valid ranges for HZ are the same as the BASIC sound statement, 110Hz to 44733 Hz. If you try it you will notice that the sound that there is not sound. This is because you did not set the output volume yet. See DB command next.

Volume: DB

To control the volume of a sound generator or to turn it off the command is DB.   (The short-form for Decibel, the unit of audio level). DB adjusts volume of the currently selected generator downwards from 0 DB to -28 DB. However the 9919 chip attenuates in 2 DB steps so -1 DB and -2 DB will actually be the same volume.

So new we can type
```
        DECIMAL
        GEN1 440 HZ 0 DB
```

... and the tone will run forever until you turn it off.

NOTE:

DB does not care if the values are negative or positive. -10 DB is the same volume as 10 DB. But minus values are technically correct. Allowing positive values makes it a little simpler to create volume control words. (See chord example later in the chapter)

MUTE

To turn off the selected generator use -30 DB or use the word MUTE, which does the same thing in one command.

SILENT

You can turn off all the sound generators with one command type SILENT.

Duration Control: MS

Standard Forth has a delay word called MS and it is implemented in CAMEL99 Forth. So we don't need anything special to keep a sound running for a period of time. The minimum resolution of the MS timer is 1/60 of a second. In CAMEL99 MS is also multi-tasker friendly and will let other tasks run while it is delaying.

So to make a complete sentence of Forth to make a sound type:

```
            DECIMAL GEN1 440 HZ  0 DB  200 MS  MUTE
```

Create a SOUND Command

It becomes trivial to create a SOUND command that is similar to BASIC using the HZ, DB, MS and MUTE.
```
\ Create a SOUND word from primitives HZ DB MS MUTE
: SOUND  ( dur freq att --) DB  HZ  MS MUTE ;

\ usage:

DECIMAL
GEN1 1000 440 -4 SOUND GEN2 1000 220 -2 SOUND GEN3 1000 110  0 SOUND
```

NOISE CHANNEL

There is only one command for the NOISE channel. It's called NOISE. NOISE selects Generator 4. This channel is bit controlled. The table below show the bit functions.

Bit 2 0 =Periodic Noise 1= White Noise	Bit 1	Bit 0	Output
0|1	0	0	6991 Hz
0|1	0	1	3496 Hz
0|1	1	0	1748 Hz
0|1	1	1	Hz controlled by generator 3

BASS Note Generator

CHORDS

Remember that this is not a multi-tasking sound word so if you wanted to make different note sounds on the other generators at the same time the previous SOUND word would not work. But no need to panic. We have all the words to do the job. To make a "chord" of three notes you would to do something like this:

```
\ Frequencies for the notes were taken from page III-7
\ of the TI-99 User Reference Guide

INCLUDE DSK1.SOUND

\ CAMEL99 is case sensitive so we can use lower case letters
\ in our word names which lets these chords look the way
\ we normally see them in print.

DECIMAL
: Am \ runs forever or until frequencies change
      GEN1 110 HZ  0 DB
      GEN2 131 HZ  0 DB
      GEN3 165 HZ  0 DB ;

: Dm \ runs forever or until frequencies change
      GEN1 147 HZ  0 DB
      GEN2 175 HZ  0 DB
      GEN3 220 HZ  0 DB ;

: E7 \ runs forever or until frequencies change
      GEN1 147 HZ  0 DB
      GEN2 165 HZ  0 DB
      GEN3 208 HZ  0 DB ;

: Am2  \ runs forever or until frequencies change
      GEN1 131 HZ  0 DB
      GEN2 165 HZ  0 DB
      GEN3 220 HZ  0 DB ;

: WHOLE-NOTE    \ puts a little space between each chord
      950 MS
      SILENT 50 MS ;

: FADE3  \ just for fun let's make a fade out ☺
31 0      \ from 0 DB to 30
DO
    GEN1 I DB
    GEN2 I DB
    GEN3 I DB
    100 MS
LOOP ;


: PROGRESSION
      Am WHOLE-NOTE
      Dm WHOLE-NOTE
      E7 WHOLE-NOTE
      Am2 1500 MS
      FADE3 ;
```

Envelope Control

Real sounds have something called an envelope. This describes how the sound starts, how long it remains on and how it finishes. These three elements of a sound envelope are called attack, sustain and decay.

In BASIC it is impossible to rapidly control the volume of sound statements because the BASIC interpreter is just too slow. In Forth we can run the sound chip attenuators fast enough to create sound effects that are more interesting than just simple tones that go on and off. Try these sound effects in your CAMEL99 Forth system

```
NEEDS DB FROM DSK1.SOUND

DECIMAL
: DECAY   ( ms -- )
   31 0
   DO
     DUP MS       \ delay to the value on stack
     I DB         \ adjust the volume.
   2 +LOOP        \ next level is 2 db more
   MUTE
   DROP ;

: PLINK   ( freq -- )
\  attack      freq  volume   sustain    decay
\  ------      ----  ------   -------   --------
  ( instant)   HZ    0 DB     20 MS     16 DECAY ;

: PING    ( freq -- )
\  attack      freq  volume   sustain    decay
\  ------      ----  ------   -------   --------

  ( instant)   HZ    0 DB     15 MS     60 DECAY ;

\ After compiling the code above try these commands in Forth

DECIMAL
2000 PLINK

1000 PING
```

Envelope Exercise

 It is not to difficult to pass 3 parameters to a word called ENVELOPE that controls the shape of a sound. You have DECAY as an example of an attenuator that goes down in volume. Can you create an upwards volume control in the same way? Try using both to create a word: ENVELOPE (attack sustain decay --)

Techie Stuff: How Forth Controls the TMS9919 Sound Chip

Sound in the TI-99 is controlled by the TMS9919 chip. There are plenty of explanations on the details of the chip on the WEB so we won't get too low level here. We will simply explain what you need to know about how the library file controls the chip.

The Note Generators

To produce a sound the 9919 chip starts with a high frequency clock running at 111,860 Hz (vibrations per second). To get a specific note out of it you must program divider circuits in the chip to divide the clock by the correct amount to get your note. The formula is:

Clock/desired frequency = bytecode

There are two challenges with this for CAMEL99. First 111,860 is bigger than a 16 bit integer. This means we need to use a 32 bit integer. We create that in the code and call it f(clk). Then we need to divide the 32bit integer by a 16bit frequency we want to output. We can do this in Forth using the "unsigned mixed slash MOD" operator call UM/MOD.

The second problem is that the bytes needed for the sound chip are not in the order that comes out of the equation. For example if the HEX bytes computed are "ABC", they need to be re-ordered to "CAB" We do this with a fast CODE word called [8]>FCODE.

The final word that does the entire process is called HZ>CODE and it looks like this:

```
: HZ>CODE ( freq -- fcode ) f(clk) ROT UM/MOD NIP >FCODE ;
```

With the "FCODE" calculated we add the upper 4 bits that select the sound generator to use (the chip has three generators) with "OSC @ OR". Then we split it into two bytes and send each byte to the sound chip with the words SPLIT and SND!

Here is how this final piece of the puzzle looks in the code.

```
: (HZ)    ( f -- code)  HZ>CODE  OSC @ OR ;

: HZ      ( f -- )  (HZ)  SPLIT SND! SND! ;
```

Notice we use an intermediate word (HZ). The reason for that is if we wanted to record musical notes as "FCODE" bytes we could do that by using (HZ) to pre-calculate the FCODE for a group of notes and store the fcode as data. Later this would save a lot of CPU time when we play the fcodes. TI Sound lists do this.

Example of pre-calculated FCODE
```
DECIMAL
GEN1 440 (HZ) CONSTANT ANOTE  \ ANOTE includes which generator to use

\ now A3 has the correct code to make 440Hz, pre-calculated
\ You only need to SPLIT it and send the 2 bytes.
\ this would play the note with no need to calculate the 9919 Fcode

: PLAYCODE ( fcode -- ) SPLIT SND! SND! ;

   ANOTE PLAYCODE
```

---

[8] FarmerPotato on the Atariage forum gave us a four instruction version of this routine that is very fast.

Background Sound Using Interrupts

The Ti-99 software has the ability to run a "sound controller" as it is called in the Editor/Assembler Manual. The sound controller is a program that runs every 1/60 of a second. It happens this way because the Video chip "interrupts" the CPU every 1/60 of a second. During that time your program stops and other programs take over. If things are set up correctly one of those programs that runs is the "sound controller". This gives the illusion that the sound is running at the same time as your program. In reality the TI-99 CPU is just switching back and forth from your program to the sound controller very quickly. But the effect is still very impressive.

The instructions for using it in the E/A manual are:

"*Three addresses in CPU RAM are associated with processing sound information.*
   1. *You place a pointer to the sound table in VDP RAM at address >83CC.*
   2. *You place >01 at address >83CE to start the processing of the sound generator.*
      This address is used by the interrupt routine as a count-down timer during the execution of sound.
   3. *The least-significant bit of the byte at address >83FD (which is the least-significant byte*
      of GPL Workspace Register 14) must be set to indicate that the sound table is in VDP
      RAM."

Don't Panic.

You don't need to know all that stuff to use the sound controller in Camel99 Forth.

All you need to do is:

   1. Load the ISRSOUND file with the command: INCLUDE DSK1.ISRSOUND
   2. Make a sound table for your program
   3. Play the sound tables with the command: MYSOUND ISRPLAY

What is a Sound Table?

A sound table is a list of bytes that are in a special format so that is read by the sound controller program. The ISRSOUND file gives you a way to "borrow" sound table code from Assembly Language programs without requiring much change. Below is an example of the sound table for a Chime sound as written on page 322 of the Editor Assembler Manual: (Here the BYTE directive puts the DATA in CPU RAM)

```
CDATA    BYTE    >05,>9F,>BF,>DF,>FF,>E3,1
         BYTE    >09,>8E,>01,>A4,>02,>C5,>01,>90,>B6,>D3,6
         BYTE    >03,>91,>B7,>D4,5
         BYTE    >03,>92,>B8,>D5,4
         BYTE    >05,>A7,>04,>93,>B0,>D6,5
         BYTE    >03,>94,>B1,>D7,6
         BYTE    >03,>95,>B2,>D8,7
         BYTE    >05,>CA,>02,>96,>B3,>D0,6
         BYTE    >03,>97,>B4,>D1,5
         BYTE    >03,>98,>B5,>D2,4
         BYTE    >05,>85,>03,>90,>B6,>D3,5
         BYTE    >03,>91,>B7,>D4,6
         BYTE    >03,>92,>B8,>D5,7
         BYTE    >05,>A4,>02,>93,>B0,>D6,6
         BYTE    >03,>94,>B1,>D7,5
         BYTE    >03,>95,>B2,>D8,4
         BYTE    >05,>C5,>01,>96,>B3,>D0,5
```

Here is the same list written in CAMEL99 Code but it puts the DATA in VDP RAM:

```
HEX
VCREATE CHIME
    VBYTE 05,9F,8F,DF,FF,E3,1,
    VBYTE 09,8E,01,A4,02,C5,01,90,B6,D3,6
    VBYTE 03,91,B7,D4,5
    VBYTE 03,92,B8,D5,4
    VBYTE 05,A7,04,93,B0,D6,5
    VBYTE 03,94,B1,D7,6
    VBYTE 03,95,B2,D8,7
    VBYTE 05,CA,02,96,B3,D0,6
    VBYTE 03,97,B4,D1,5
    VBYTE 03,98,B5,D2,4
    VBYTE 05,85,03,90,B6,D3,5
    VBYTE 03,91,B7,D4,6
    VBYTE 03,92,B8,D5,7
    VBYTE 05,A4,02,93,B0,D6,6
    VBYTE 03,94,B1,D7,5
    VBYTE 03,95,B2,D8,4
    VBYTE 05,C5,01,96,B3,D9,5
    VBYTE 03,94,B1,D7,5
    VBYTE 03,95,B2,D8,F
    VBYTE 03,9F,BF,DF,0
/VEND
```

In Forth we don't use the '>" symbol for HEX. We specified HEX at the top.

- VCREATE makes the label name in the dictionary and will give you the VDP address of the sound table when you use the name in your program. The address of course will go onto the Data stack.
- VBYTE reads the numbers, comma-delimited, and places them in VDP RAM directly.
- /VEND just puts a zero on the end of the list that tells the sound controller to stop.

Technical Details of a Sound table

For a tone, the information consists of the sound generator value (1,2 or 3) and data for frequency and attenuation. For a noise, the information consists of a noise source and attenuation. Tones can be specified, singly or in combination, for generator 1, 2, or 3. Noise can be specified by a noise generator.

When you are generating tones, the first byte in the sound list is the number of bytes to be loaded into the sound processor. Following that are the bytes to be loaded.
- A generator and frequency can be specified by two bytes.
- A generator and attenuation can be specified by one byte.
- A generator and noise can be specified by one byte.

After you give all generator, tone, noise, and attenuation bytes, you give a duration time.

Simple Example Sound Table Data (BEEP sound)

```
HEX
VBYTE 3,80,5,91,2A
VBYTE 1,9F,0
```
Line 1:
- 3 numbers of sound data bytes in this string
- 80 turns on sound generator 1
- 5 first byte to set the frequency
- 91 second byte to set the frequency
- 2A delay time in 1/60 seconds. (Decimal value=42)

Line 2:
- 1 number of sound data bytes in this string
- 9F set attenuator #1 to 15, turns off the sound output
- 0 end of the sound table (no delay value needed since table is ended)

How Do I know the Correct Values for Frequencies and Attenuation?

The simplest way is to study and use the DSK1.SOUND file in the CAMEL99 system.
The word (HZ) will take a frequency and return the sound bytes need to make that sound. The bytes are combined together in one integer so for sound tables you need to SPLIT them apart with the word SPLIT.

Example to see single tone values typed at the Forth console:

```
DECIMAL  440 (HZ)  SPLIT HEX .  .  8E  F  ok
```

This shows that 8E is the first byte in the table followed by 0F.

To get a volume byte use this:
```
0 (DB)  .   90  ok
```

The first nibble, 9, is for generator 1, The 0 is the actual volume. 0 .. F
For generator 2 use B0
For generator 3 use D0
For generator 4 use F0   (noise generator)

TI-99 Memory Spaces

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.

CPU RAM

High CPU RAM

This is the memory from HEX A000..FFFF. The CPU can it 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 High CPU RAM from address HEX A000 to FFFF for the dictionary.

Low CPU RAM

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

The 256 bytes of the low 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.

SuperCart CPU RAM

In the Classic99 emulator there is another 8K block of RAM that is seldom mentioned. It is also included in the Editor/Assembler SuperCart for the real TI-99. It's a great resource giving the TI-99 at total of 40Kbytes of CPURAM

VDP Memory

VDP Memory is 16K bytes that is connected to the VDP graphics processor chip. It is used for Console screen display, the character patterns, sprites and character colours. VDP memory is accessed via "memory mapped ports". These are special memory addresses that allow communication with the VDP chip. (TI BASIC uses VDP memory for everything if you do not have an expansion memory card.) Forth provides the words VC@, VC!, V@, V!, VREAD, VWRITE and VFILL to access this memory.

Super AMS Memory (SAMS)

The SAMS card gives us a 1Mbyte or 4Mbyte memory space but it is only visible to the computer in 4K blocks. Out of the box CAMEL99 accesses this memory in the memory space starting at HEX 3000 with the DSK1.SAMS library. If you can spare all 8K of LOW RAM DSK1.SBLOCKS is a better option.

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

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) They are 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 address leaving nothing on the stack

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

Memory Block Operations

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

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

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.

FILL (addr len char --) Fill memory at addr with char for len bytes

VDP Memory Words

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

[10]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

[11]VWRITE (CPU-addr VDP-addr n --) Transfer n bytes from CPU address to VDP-addr (CODE word)

[12]VREAD (VDP-addr CPU-addr n --)   Transfer n bytes from VDP-addr to CPU address (CODE word)

  VFILL (VDP-addr len char --) Fill VDP memory at VDP-addr with char for len bytes.

---

[9] V@ and V! have no equivalent in Editor/Assembler but work with Forth integers perfectly

[10] VC@ is called VSBR in Editor/Assembler manual

[11] VWRITE is called VMBW in E/A manual

[12] VREAD is called VMBR in E/A manual

HEAP Memory Control

Most of the time in Forth we use the "dictionary" memory to hold our program data. VARIABLE, CONSTANT, CREATE and ALLOT all put our data in the dictionary. Sometimes we want some data space temporarily that is separate from the Forth dictionary. The lower 8K memory block in the TI-99 can be put to good use for these times. We call this area in memory a HEAP. Below are some words to help manage that HEAP.

Dynamic Memory Allocation/De-Allocation

When we allow a program to change the size and usage of pieces of memory while the program is running this is called "dynamic memory allocation". Forth

MALLOC/MFREE

CAMEL99 gives you an ultra-simple method to manage the HEAP with just two words. MALLOC and MFREE. INCLUDE the library file called DSK1. MALLOC to use these words.

These two words let you control the HEAP very much the same way as Forth controls the dictionary space. Below is the implementation CODE for this simple system:

```
VARIABLE H  \ pointer to Heap in TI-99 low memory (init to >2000)

: MALLOC     ( n -- addr ) H @  SWAP H +! ;    \ allocate heap and return pointer
: MFREE      ( n -- )       NEGATE H +! ;       \ free n bytes of heap memory
```

MALLOC Usage

The system variable H is being used as a "pointer" meaning it points to the next available low memory address. When CAMEL99 starts it sets H to HEX 2000 (Decimal 8,192) ie: the beginning of low memory.

If we put the statement   HEX 100 MALLOC in our program we will be given the address of 100 bytes of low RAM that we are free to use any way we want and the variable H is moved forward by HEX 100 bytes.

Here is a simple code example:
```
: TEST
    100 MALLOC >R                      \ buffer address is on the return stack
    S" a temporary string"  R@ PLACE   \ PLACE the string at the address on RSTACK
    R@ 2 /STRING                       \ Cut off 1st 2 characters of our copy
    R> COUNT TYPE                      \ use COUNT and TYPE display it
    100 MFREE                          \ MFREE it.
                                       \ H now = >2000 again
;
```

Remember it is your responsibility to MFREE the memory you use when you are finished with it. Alternatively if you know you will use a MALLOC memory piece for the entire program you can leave it allocated. In that case assign it to a CONSTANT or a VALUE so you can get it whenever you need to like this:

```
HEX
    2 MALLOC CONSTANT INTEGER1 \ This is simple variable but in HEAP memory
  100 MALLOC VALUE A$          \ use a value and A$ can be switched later
\ Use:
   99 INTEGER1 !    S" I'm a heap string!" A$ PLACE
```

ANS/ISO Forth ALLOCATE,FREE,RESIZE

Standard Forth specifies words to dynamically allocate memory. There is a file in the package in the PC text files, called DSK1.ALLOCATE2.FTH that gives you these standard words. On a TI-99 the code uses 812 bytes of precious dictionary space and since we only have a small HEAP (8K) it hardly seems worth it.

But if you need to try Forth code that uses the standard words we have dumb-ed down the features so that you can use them in the PC text file called ALLOCATE.FTH in the LIB.ITC folder of the CAMEL99 Forth ZIP FILE.

The limitation is that if you for example allocate memory A B and C and then you want to FREE A or B but leave C unchanged, the HEAP will not be able to re-use A and B. It is lost to the HEAP. But it gives you the ability to RESIZE the last allocation safely and you can also see the SIZE of each allocation with the word SIZE.

ALLOCATE Glossary
 HEAP,   (n --)
Compile 'n' into heap memory and reduce HEAP size by 2 bytes.

ALLOCATE (n -- addr ?)
Allocate 'n' bytes in the heap, return an address and error flag. 0 means all good.

FREE   (addr -- ?)
Free up the memory at 'addr'. *Warning* FREE removes everything above it as well.

RESIZE   (n addr -- addr ?)
Resize 'addr' to 'n' bytes. This destroys the previous version and makes a new allocation so do not use it very much. It is here for compatibility.

 SIZE   (addr -- n)
Return the size of HEAP memory item at 'addr'.

Below is an example of code using the library file **ALLOCATE.FTH**

```
\ define the some values to hold the memory addresses
 0 VALUE X
 0 VALUE Y

 : START-PROGRAM
    50 ALLOCATE -> X    ( -> gives protection if ALLOCATE throws an error )
    50 ALLOCATE ?ERR TO Y   ( you could also do this)
    50 ALLOCATE DROP TO X   ( no protection. Take your chances)

\    ... PROGRAM continues from here
\   X,Y,Z can be used just like they were created with CREATE and ALLOT

\   end the program by freeing memory in reverse order they were created.
    Z FREE
    Y FREE
    X FREE
```

HEAP Memory Uses and Final Caution

Although having some simple DYNAMIC (changeable) memory allocation seems cool, it can make your programs more complicated and sometimes less reliable. Forth has traditionally use "STATIC" memory allocations when used in mission critical applications because it is dead simple and therefore reliable.

There is an application for dynamic memory in Forth when it is handy. If you save the entire Forth system to make a binary program you can keep the program smaller by putting variables and arrays in the HEAP. This because the space they use will not be in the dictionary when you save the system. (SEE: DSK1.SAVESYS)

In this case you must declare all your memory spaces as VALUEs in the program. Then when your program starts you must MALLOC or ALLOCATE some memory for each VALUE before continuing on.

Examples:

```
\ First we make the names of the items which we will "allocate" to the HEAP.
0 VALUE MYARRAY
0 VALUE X
0 VALUE Y
0 VALUE TITLE$

DECIMAL
: MAIN
      400 MALLOC TO MYARRAY      \  MYARRAY can now hold 400 bytes
        2 MALLOC TO X            \ X is an integer. Needs 2 bytes
        2 MALLOC TO Y            \ Y is an integer. Needs 2 bytes
       32 MALLOC TO TITLE$       \ TITLE$ 31 bytes of text (1st Byte is the length)

        etc...


\ Alternative if you use ALLOCATE

DECIMAL
: MAIN
         400 ALLOCATE -> MYARRAY
           2 ALLOCATE -> X
           2 ALLOCATE -> Y
          32 ALLOCATE -> TITLE$
        etc..
```

When multi-tasking you can imagine that if two different tasks used MALLOC they could each move the H pointer variable to get a memory block without knowing that the other has done it.

The VARIABLE H is a master VARIABLE. There is only one heap that all tasks can share. This can make some real problems if not used wisely. ☺ Fortunately Forth uses a cooperative multi-tasker so you have direct control over when a task gives up control but it is still possible to really mess up shared memory.

HEAP Option 1: LOCK the HEAP while in use

It is simple to prevent another task from using the heap while a different Task is using it. You just let one task HOG the machine until it's done with the HEAP. This is done by **not** using the keyword PAUSE until you have used the heap and executed MFREE. This way you use the HEAP and give back the memory only when you are finished using it.

```
\ Example of locking memory in a TASK
: PROGRAM1
    S" DSK2.DATAFILE" R/W OPEN-FILE ?FILERR TO #1
    BEGIN
\ LOCKED…
        100 MALLOC >R              \ PROGRAM1 has complete control now
        R@ #1 READ-FILE-RECORD     \ read stuff into buffer
           TOUPPER                 \ convert to upper case (example)
        R@ #1 WRITE-FILE-RECORD    \ write it back to disk
        100 MFREE                  \ release the HEAP memory
\ UNLOCKED…
        PAUSE                      \ we're done. give control to another task

    AGAIN ;

' PROGRAM1 TASK2 ASSIGN
```

Option 2: Separate HEAPS

If you plan to have a task use a separate heap, you MUST set the start address of the heap to be different than the main Forth program. You can't do it with the GLOBAL variable H but… you can create a variable that is local to any task. They are called USER variables in Forth. There are are system USER variables already used so start yours at HEX 48 to be safe.

You define a USER variable n your program like this:

```
HEX 48 USER LH    \ local heap pointer
```

The global HEAP starts at HEX 2000 and it ends at HEX 3F00. (8192 to 15872 DECIMAL)
What we need to do is steal a part of the global heap for our task and give that memory to the local heap.
From inside another TASK you can assign heap that TASK's LH variable like this:

```
3200 LH ! \ local heap pointer now starts HEX 3200 and can use up to 3FFF
```

Now define the following code in your multi-tasking program outside of the task to use a local heap.
```
: LMALLOC ( n -- addr ) LH @  SWAP LH +! ;  \ allocate local heap and return pointer
: LMFREE  ( n -- ) NEGATE LH +! ;           \ free n bytes of local heap memory
```

```
\ Example of LOCAL HEAP usage
: PROGRAM2
    HEX 3200 LH !
    S" DSK2.DATAFILE" R/W OPEN-FILE ?FILERR TO #1
    BEGIN
      100 LMALLOC >R              \ allocate from my local heap
      R@ #1 READ-FILE-RECORD      \ read stuff into buffer
      PAUSE                       \ file Ops are slow, run other tasks

      TOUPPER                     \ now convert to upper case
      PAUSE                       \ give control to another task
      R@ #1 WRITE-FILE-RECORD     \ write it back to disk
      PAUSE                       \ give time to other tasks

      100 LMFREE                  \ release the local HEAP memory
    AGAIN ;

` PROGRAM2 TASK3 ASSIGN
```

Notice how simple it is to do an operation and then give control to other tasks.

Pre-emptive vs Cooperative Multi-tasking

If you have the CPU interrupt your tasks and switch to another task with a timer this is called a PRE-EMPTIVE Multi-tasker. Its stops the running task whether it is ready or not. This can make life complicated if your program is sharing things like files and variables with other tasks.

Camel99 Forth does not do this. Camell99 uses a cooperative multi-tasker. This means each task has complete control of the CPU until it gives it away with the word PAUSE. The dis-advantage here is that you must use PAUSE in your task routines or nobody else can run! Typically place PAUSE before an I/O operation.

In the example above, with preemptive multi-tasking you would need to create a way to protect critical parts of the code with special commands and controls around the critical areas of code. Writing to a file record or reading a file record is a critical area because another task could change the record while you are in the middle of editing it. Modern operating systems, typically require using protection and they have ways to do it but the solutions are always more complicated than using PAUSE.

For our little TI-99 the cooperative approach is small and simple. It does not interfere in any way with the existing system interrupts and it does not prevent you from adding interrupt driven features to your program where they are really needed.

Super AMS Memory: DSK1.SAMSFTH

The SAMS card is a [13]1M memory system that can page extended memory into "pages" in the TI-99 memory map. CAMEL99 has code that lets you access multiple 64k segments via a 4K "window" The window is located at HEX 3000, the upper part of the HEAP. The key word is "PAGED" . Given an address PAGED decides if that memory is already sitting in the window or not. If not it, it flips the correct range of a 64k segment into the 4k window. Using PAGED, you can create words to access SAMS memory by byte or word or as a 4K block. The file DSK1.SAMS does the same thing but is written in machine code for maximum speed.

SAMS Terms
SAMS page: A 4K chunk of SAMS memory.
Window   : The memory location where SAMS pages appear in the TI-99 memory.

SAMS Glossary

Primitive words support the system
```
'R12      ( -- addr) User variable. Returns the address of R12 in any Forth workspace
SAMSCARD ( -- ) select SAMS card as the active card in Peripheral Expansion Box (PEB)
DMEM      ( -- addr) Constant the is the memory "window" location. (HEX 3000)
SEG       ( -- addr) Variable that holds current 64K segment in use.
BANK#     ( -- addr) Variable that holds the current SAMS bank mapped into the window.
0SBO      ( -- ) Code word turns ON  bit zero at the CRU address in 'R12.
0SBZ      ( -- ) Code word turns OFF bit zero at the CRU address in 'R12.
1SBO      ( -- ) Code word turns ON  bit one at the CRU address in 'R12.
1SBZ      ( -- ) Code word turns OFF bit one at the CRU address in 'R12.
```

Programmers Words
```
SAMS-ON  ( -- ) Enable the page mapper circuit so memory will appear in window.
SAMS-OFF ( -- ) Disable mapper. SAMS card will become a 32K expansion RAM card.

SEGMENT ( 1..F -- )
      Set the current 64K segment used by PAGED (0 is not allowed)

DMAP    ( bank# -- )
      "data map"  maps in a  page at DMEM window. You can use this to manually pull in
      SAMS pages for your own application. BANK# is stored in the variable BANK#

PAGED   ( virtual-addr – real-addr)
      When given an address in the range of >0000 to >FFFF PAGED maps in the correct
      SAMS page for the address in the SEGMENT that is selected. It returns the
      address in the window space where you can access that virtual address. Once you
      have the real-address you can use normal Forth memory fetch and story words.
      BANK# is stored when PAGED is used if you need it.

SAMSINI  ( -- )
      Initialize the SAMS card to the default memory pages so it appears to the
      computer to be a normal 32k Expansion RAM card.
```

Options:
A machine code version of the SAMS library called DSK1.SAMS provides a faster version of PAGED.
DMAP is not available in that library.

---

[13] Camel99 Forth has only been tested with the 1MByte SAMS card

SAMS Test Code

Timings were made with DSK1.SAMS. (machine code version) The speed to access bytes and cells is approximately 78% slower compared to normal CPU fetch and store.

*Timings below used DSK1.SAMS (machine code version)
You can see that just erasing 960K Bytes of memory takes 18 seconds!

```
INCLUDE DSK1.TOOLS
INCLUDE DSK1.ELAPSE
INCLUDE DSK1.SAMS


HEX
7FFF CONSTANT 32K
FFFF CONSTANT 64K
1000 CONSTANT 4K

1 SEGMENT  \ select the segment we will use. (Valid segments are 1 to 15)

: ERASE    0 FILL ;
: BLANKS  BL FILL ;

\ ************************************************************
\ make some SAMS fetch and store operators using PAGED
\ virtual address range can be anywhere in 1 segment: >0000 to >FFFF

: C!SAMS  ( char virt-addr -- ) PAGED ( real-addr) C! ;
: C@SAMS  ( virt-addr - char)   PAGED ( real-addr) C@ ;

: !SAMS   ( n virt-addr -- )    PAGED ( real-addr) ! ;
: @SAMS   ( virt-addr -- n)     PAGED ( real-addr) @ ;
\ ************************************************************

\ How to fill one 64K segment of SAMS memory at maximum speed, 4K at a time
: 64KERASE    64K 0 DO I PAGED 4K ERASE   4K +LOOP ;
: 64KBLANKS   64K 0 DO I PAGED 4K BLANKS  4K +LOOP ; ( 1.5 secs)


\ Erase 940,000 bytes of SAMS memory (Segments 1 to 15)
DECIMAL
: ERASEALL    16 1 DO  I SEG ! 64KERASE   LOOP ;  ( 18.6 secs)


\ 64k single byte writes to SAMS RAM
: 64KBYTES    64K 0 DO  I    I C!SAMS    LOOP ; ( 27 secs)


\ 32k word writes to SAMS memory
: 32KWORDS    64K 0 DO  I  I !SAMS   2 +LOOP ; ( 15.3 secs)
\ Normal RAM
: 32KRAM      64K 0 DO  I  3000 !    2 +LOOP ; (  8.6 secs)

HEX
: SEE32K      64K 0 DO  I @SAMS . ?BREAK  2 +LOOP ;
```

SAMS Memory as a BLOCK

In Version 2.6 a new method to manage Super AMS memory was developed using the concept of the Forth BLOCK. Normally the word BLOCK is used to manage a large disk file but the idea works well with SAMS memory.

Explanation

The SAMS memory board is 1MByte of RAM but the TI-99 cannot create an address number big enough for 1,000,000 bytes because it is a 16 bit computer. The solution is to expose the SAMS memory card in 4Kbyte chunks and replace a 4K piece of regular memory in the TI-99 with a 4K chunk of SAMS memory. This a not ideal but it is how it works. We have enhanced this idea by using two 4K areas in Low RAM. You can think of these like buffers that hold SAMS card information.

The Forth word BLOCK in the SBLOCKS library file takes a single number argument. This number is the specific 4K memory piece in the SAMS card that you want. The valid range of this number is 0 to 255 for a 1Mbyte card however SAMS blocks 0 to 15 are the expansion RAM memory so it is best to avoid them.

*WE RECOMMEND NOT TO USE SAMS BLOCKS BELOW 16 (hex 10) *

BLOCK will return the address where is has put that SAMS memory on the DATA stack. When you ask for a 2nd SAMS BLOCK, the piece will go in the 2nd buffer. The buffers will remain the same until you request a different BLOCK. The last used SAMS block stays in it's buffer.

The big advantage of this system is that you can transfer from SAMS to SAMS because you always have two buffers to work with. The previous PAGED system uses only 1 buffer in LOW RAM so copying from sams block A to SAMS block B requires copying A into another buffer, selecting B and copying buffer to B.

Example SAMS BLOCK Code

```
INCLUDE DSK1.TOOLS
INCLUDE DSK1.ELAPSE
INCLUDE DSK1.SBLOCKS

HEX
7FFF CONSTANT 32K
FFFF CONSTANT 64K
1000 CONSTANT 4K

: ERASE    0 FILL ;

: ERASE-BLOCKS   100 10 DO  I BLOCK 4K ERASE  LOOP ;  ( 18.6 seconds)

\ 1st safe virtual address is 0000 1 (65536)

1 CONSTANT SEG
: VIRT>REAL  ( virt-addr -- real-addr) SEG 4K UM/MOD BLOCK + ; (34 seconds)

: 64KTEST   64K 0 DO  I VIRT>REAL  DROP LOOP ;
```

CAMEL99 Memory Map

CAMEL99 Terminal Input Buffer >FF00

Data stack >FF80
Return Stack >FE80
DSR workspace >FEFF

CAMEL99 Free Dictionary
>C000 (15K)

VDP RAM
16K

CAMEL99 KERNEL >A000...BFFF (8K)

CAMEL99 Workspace >8300
+ High-speed routines

Read PORT

Write PORT

Cartridge Slot >6000

Device ROM >4000
(Expansion Cards: DISK, RS232, etc.)

CAMEL99 HEAP - or - SAMS window

4k SAMS
Page 0

· · ·

4K SAMS
Page 255

CAMEL99 HEAP >2000...>2FFF

TI-99 ROM 0...>1000 (2K)

Stealing Dictionary 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 from 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'. (dictionary pointer)

If you fetch the contents of DP you can see the end address like this:

```
DP @ U.
```

(You must use U. to print the unsigned value because the number is bigger than 32767)

If you need to know where the next available dictionary memory is located Forth says it is right **HERE**.

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

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

The word HERE will give you the dictionary address on the DATA stack.

There is the word ALLOT which lets us add or subtract from the value in DP and therefore you can move HERE forward or backward as you wish.

```
 100 ALLOT  \ move HERE forward 100 bytes
-100 ALLOT  \ move HERE back to where is was before.
```

Also there is nothing preventing us in Forth from changing 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... as long as we remember where it was in the first place.

Where could we keep it?. On the data stack of course.

```
INCLUDE DSK1.GRAFIX     \ we need to be in Graphics mode

\ get a copy of the dictionary address on the DATA stack with HERE
HERE ( -- addr) \ This addr sits on the stack. Will use it later…

\ Now compile patterns into unused dictionary space
        0103 , 0303 , 0303 , 0303 ,  \ each pattern is 8 bytes
        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 , \ We just used 8 x 9 = 72 bytes

\ Write the data from HERE on stack, to VDP memory starting at the location in
\ the pattern description table ( ]PDT ) for character HEX 11.  We calculate how
\ many bytes index into the VDP pattern description table like this:

\    8 bytes per character x 9 characters

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

\ restore the value of the dictionary pointer (DP)
\ back to HERE that we left on stack earlier. Memory is now restored!

( -- addr) DP !


\ word to print the 9 character TI logo on a new line
: TI.LOGO ( -- )
        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


\ clear the screen and print the logo

PAGE TI.LOGO

\ DISCLAIMER:
\ No CPU memory was permanently used in the making of the this LOGO  :-)
```
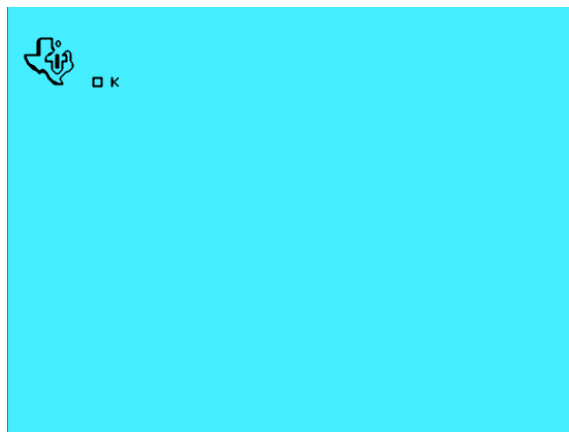
Performance Enhancements

As we have seen Forth operates like everything is a sub-routine. In many languages this would cause a big overhead but Forth was built to minimize sub-routine overhead. One way it does this is by using separate stacks for data and return addresses. Even so there are times when you need every bit of speed you can get. This can mean that calling a routine in a time critical loop makes things too slow.

Text Macros

The solution to removing some calling overhead is to use a feature that became part of Forth in the ANS Forth 94 Standard. The name of the feature is the "Text Macro" and it combines a literal text string and the word EVALUATE. Consider the following code:

```
INCLUDE DSK1.ELAPSE      \ elapsed timer utility

\ Calculate the address of an array at index 'n'
: []  ( ndx addr – addr[n])  SWAP CELLS + ;

\ store n in array at ndx
: []! ( n ndx addr -- ) [] ! ;

\ fetch n from array at ndx
: []@ ( ndx addr – n)   [] @ ;

DECIMAL
2000 CONSTANT SIZE

CREATE Q  SIZE CELLS ALLOT    \ make some space called Q

: FILL-Q    SIZE 0 DO  I  I Q []!  LOOP ;
: SEE-Q     SIZE 0 DO    I Q []@ . LOOP ;
: EMPTY-Q   SIZE 0 DO    I Q [] OFF  LOOP ;
```
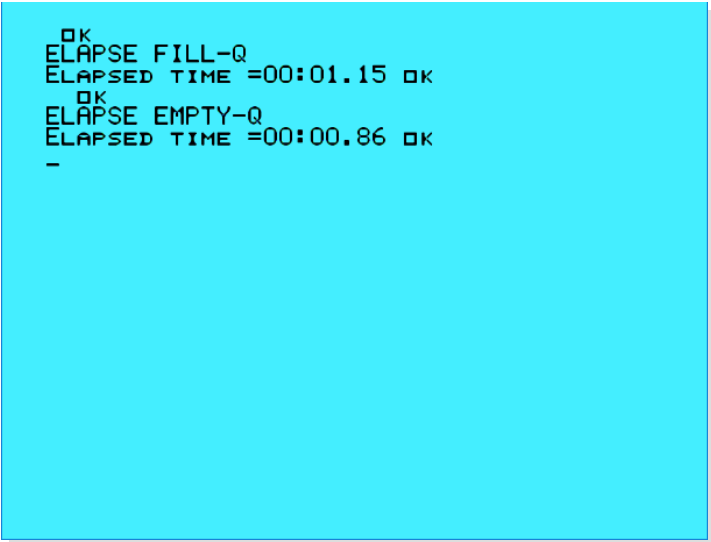
This code will work as expected and you can fill the array, see-the-array and empty the array. If we run this we get the following results:

Now let's replace the array index calculator with a TEXT Macro like this:

```
: []    S" SWAP CELLS +" EVALUATE ;  IMMEDIATE
```

What's the difference?

Notice that now [] is an IMMEDIATE word. This means that the text string will be interpreted while we are compiling the definition of []!   The effect of this is that the compiler will compile SWAP CELLS + as separate words into the routine []! .   This therefore means that there will no longer be a sub-routine call to [], but rather a call to each of the separate words. In other words we have compiled the definition of [] "inline". Let's see what it does to the speed of our loops.

```
  □K
\ DEM□ [] AS TEXT MACRO   □K
   □K
ELAPSE FILL-Q
ELAPSED TIME =00:01.00 □K
ELAPSE EMPTY-Q
ELAPSED TIME =00:00.71 □K
```

We get a 15% speed improvement in the first case and a 21% improvement in the second case. The downside of using text macros is that every time we use [] we consume 3 CELLS (6 bytes) of space rather than only 1 CELL with a colon definition. So use TEXT macros wisely and you can get some improvements in your code speed with very little effort.

INLINE CODE WORDS

CAMEL99 Forth is implemented as something called an "indirect threaded code". This is a very clever way to make a language that fits a lot of stuff in a small space. The secret to this code density is that every routine is identified by just one address. The secret to making it run quickly is creating a little routine to read those addresses and do the code associated with them as fast as possible. On the TMS9900 that little routine is only three instructions of assembly language so it's pretty fast but there is still a price penalty that can range from 2.5 times to as much as ten times in the worst cases. It turns out that ITC Forth spends about 50% of the time running those three instructions that read the address lists. The three instructions are called the [14]inner interpreter and is given the name NEXT.

At the bottom of every Forth system are a pile of little routines written in assembler (CODE words) that do all the real work. They are simple things that read the contents of a memory address (@) or add two numbers together. (+) The code is all there and it is normally just called by the inner interpreter. Each routine ends with a call back to the inner interpreter to read the NEXT address.

What if we could use all that code and eliminate the inner interpreter between each CODE word?

In the library file called DSK1.INLINE, CAMEL99 has the word INLINE[to do just that. It reads a string of Forth CODE words (written in assembly language) and puts the code together in the HEAP memory starting at HEX 2000. INLINE[remembers where it compiled the code and puts that address in your new word definition.

You could say that INLINE[makes a "headless" CODE word for you.   It's "headless" because there is no name in the dictionary for the word that INLINE[creates.

How it works
- INLINE[copies the code from the Forth kernel into low RAM at the address in the variable H.
- INLINE[remembers that address and compiles the address into your colon word definition.
- This takes up more memory space, but is outside of the dictionary.
- When used wisely it can make parts of a program run two times faster!

Below is an example of how we could use INLINE[to improve the speed of our previous example but not write one instruction of Assembly language. Let's see how our new programs perform.

---

[14] This is NOT the "outer" text interpreter that we communicate with from keyboard, but an internal routine that reads addresses and runs machine code

Performance with INLINE[ ] CODE Words

As we can see our little programs now run 40% faster because the core routines inside the loop are much faster. Even faster than using text macros

```
INCLUDE DSK1.ELAPSE  \ elapsed timer utility
INCLUDE DSK1.INLINE

DECIMAL
2000 CONSTANT SIZE

CREATE Q  SIZE CELLS ALLOT    \ make some space called Q

: []  ( index addr -- addr')  INLINE[ SWAP 2* + ] ;
: []! ( n index addr -- )     INLINE[ SWAP 2* + ! ] ;
: []@ ( index addr - n)       INLINE[ SWAP 2* + @ ] ;

: FILL-Q    SIZE 0 DO  I  I Q []!   LOOP ;
: SEE-Q     SIZE 0 DO     I Q []@ . LOOP ;
: EMPTY-Q   SIZE 0 DO     I Q [] OFF  LOOP ;
```

```
  ok
ELAPSE FILL-Q
Elapsed time =0:00.78 ok
  ok
ELAPSE EMPTY-Q
Elapsed time =0:00.70 ok
_
```

Can we make this code even faster?

Yes! The INLINE[command knows how to inline constants variables and literal number as well. We can include the address of the array inside of the optimization. Let's make a specialized set of words to compute the address of an array item and fetch , store or erase the contents.

```
: ]Q@  ( index - n ) INLINE[ 2* Q + ! ] ;
: ]Q!  ( index - n ) INLINE[ 2* Q + ! ] ;
: ]Q=0 ( index - n ) INLINE[ 2* Q + OFF ]
;

: FILL-Q    SIZE 0 DO  I  I ]Q!   LOOP ;
: SEE-Q     SIZE 0 DO     I ]Q@ . LOOP ;
: EMPTY-Q   SIZE 0 DO     I ]Q=0  LOOP ;
```

```
  ok
ELAPSE FILL-Q
Elapsed time =0:00.66 ok
  ok
ELAPSE EMPTY-Q
Elapsed time =0:00.53 ok
  _
```
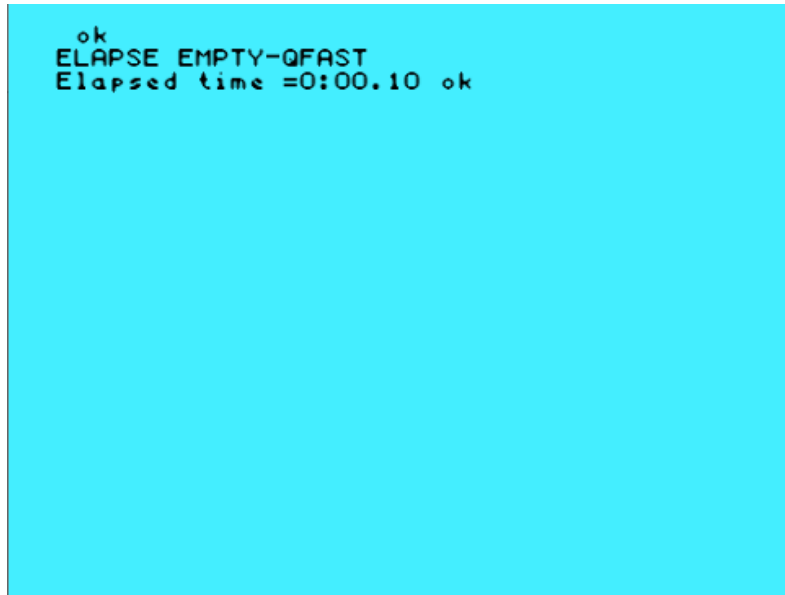
Here is how that method performed ->

Assembly Language is Faster

So we can see that we have the flexibility to code in various ways to get performance improvements. It should be understood however that in a flat out "apples to apples" competition Assembler wins.

Here is how we SHOULD do EMPTY-Q using the Forth word FILL which is written in Assembly language.

```
: EMPTY-QFAST ( -- )  Q SIZE CELLS  0 FILL ;
```


Using FILL this routine is over 10X faster than our original Forth version!

Code Macros

Another way to make faster arrays is in the pc file CODEMACROS.FTH

This file will give you tiny snippets of machine code in the form of "macros". These code macros will write themselves into a CODE word that you write so that you don't need to write the code yourself. At the moment they are mostly useful for creating fast methods to fetch and store items in an integer array or character array. Lets do the same thing as before but use code macros.

Example:

```
INCLUDE DSK1.ELAPSE     \ elapsed timer utility

DECIMAL
2000 CONSTANT SIZE

CREATE Q  SIZE CELLS ALLOT    \ make some space called Q

: MACRO        CODE ;
: ;MACRO       NEXT, ENDCODE  ;
HEX

\ stack primitives
: DUP,   ( n -- n n) 0646 , C584 , ;  \ Also used as: TOS PUSH,
: DROP, ( n --)       C136 , ;  \ Also used as:  TOS POP,
: 2*,    ( n -- n') 0A14 , ;  \ TOS  1 SLA,

\ Very FAST integer array words
: ()@,  ( addr -- ) 2*,  C124 , ( addr) , ;        \ addr(TOS) TOS MOV
: ()!,  ( addr -- ) 2*,  C936 , ( addr) ,  DROP, ; \ *SP+ ARRAY (TOS) MOV,

 MACRO ]Q@ ( ndx -- n)    Q ()@,   ;MACRO
 MACRO ]Q! ( ndx -- n)    Q ()!,   ;MACRO


: FILL-Q    SIZE 0 DO  I  I ]Q!   LOOP ;
: SEE-Q     SIZE 0 DO     I ]Q@ . LOOP ;
```

```
  ok
ELAPSE FILL-Q
Elapsed time =0:00.46 ok
```

Multi-Tasking   DSK1.MTASK99

CAMEL99 can be 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 do 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 trained dogs 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 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, INIT-MULTI runs at the end of the file for convenience. If you make a stand alone program you MUST run it when your program starts.

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   ([15]PID uvar -- addr')

Returns the addr of a user-variable in the Process Identifier (PID) user area.

Example: TASK3 TFLAG LOCAL @   fetches the value of TFLAG in TASK3's memory space.
         TASK1 TFLAG LOCAL @   fetches a different variable's contents from TASK1.

SINGLE   (--)

Turns off the MULTI-TASKER. Single changes the action of PAUSE to the action of NEXT. This disables all MULTI-tasking because NEXT is just the Forth inner interpreter. It goes to the NEXT Forth word to run In the program.

MULTI   (--)

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

---

[15] PID means Process Identifier. In this system the PID is the "workspace address".

SLEEP   (PID --)

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

WAKE   (PID --)

Wake up a task. This is done by setting the local TFLAG variable to TRUE. (-1)

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 task that calls FORK into the workspace (PID) of a new task and then modifies it so it is ready to run as a separate task with its own workspace, stacks 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.

The Missing Multi-task Word

If you make a background task that does some work and then should go to sleep it is not obvious that if a task puts **itself** to sleep then the multi-tasker will never switch to another task. This is because each task in the system must "cooperate" and pass control to the next task. There is no central "scheduler" routine.

The solution is in the example below. The word STOPTASK not only puts **myself** to sleep but then passes control to the next task with PAUSE.

```
: STOPTASK    ( -- ) MYSELF SLEEP  PAUSE ;

\ this is only an example. You will have to make your own print queue
: PRINTJOB  ( -- )
      HEGIN
        FILE-QUEUE?
      WHILE
        PRINTQ
      REPEAT
      STOPTASK ;
```

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?

```
              USER0  (Forth console)
              R13=TASK3 WKSP
              R14=TSTAT
              R15=TASK3 STS


  TASK 3                              TASK 1
  R13=TASK2 WKSP      Round-robin     R13=USER0 WKSP
  R14=TSTAT          Multi-tasker     R14=TSTAT
  R15=TASK2 STS                       R15=USER0 STS


              TASK 2
              R13=TASK1 WKSP
              R14=TSTAT
              R15=TASK1 STS
```
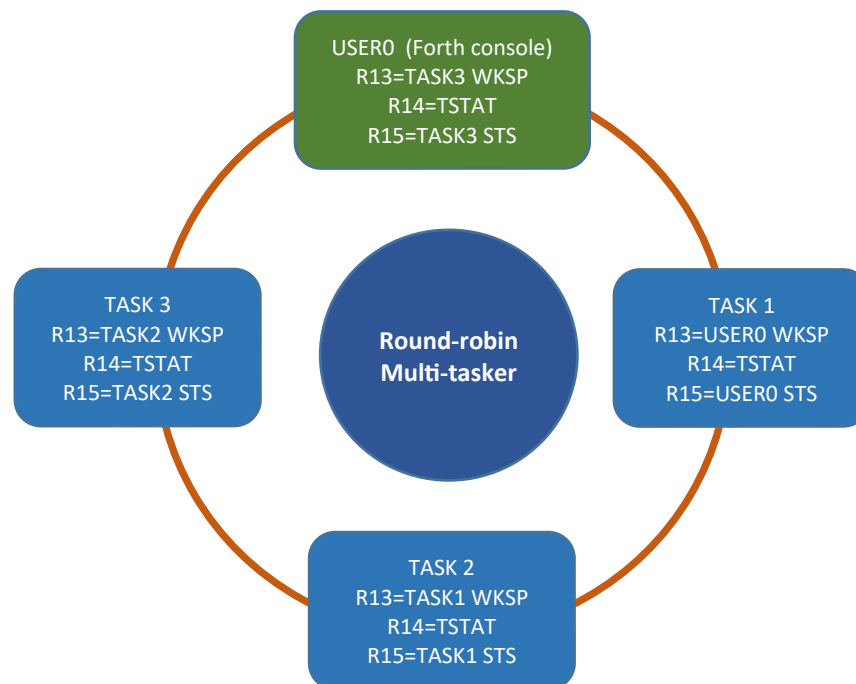
But TMS9900 created a problem. The RTWP instruction will change context immediately given an address and program counter in R13 and R14. This is different than conventional round robin where YIELD remains in a loop testing each task in the linked list, only leaving the loop when a task is awake. (tflag<>0)

SOLUTION: *Divide YIELD into 2 parts*

Part1: YIELD

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

Part2: TSTAT

- PRE-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.
DECIMAL
 CODE YIELD   ( -- )
             BEGIN,                  \ we're in the current task
                RTWP,                \ one instruction switches context
  1: _TSTAT       R1  STWP,          \ 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

R13, R14 and R15 have been stolen by this MULTI-TASKER. Once the multi-tasker is initialized R13, R14 and R15 are used to jump to the next task in the round robin task queue. **You cannot use these registers for any other purpose if you are using the multi-tasker unless you save them and restore them when your code completes**

One way is to use RPUSH and RPOP to push the critical Forth registers onto the Return stack and then POP them back when your routine completes.

Alternatively you could BLWP to your Assembly code which keeps R13, R14 and R15 safe in the TASK's own workspace.

Using BLWP and a different workspace you can safely run a sub-program from a TASK, but it MUST return to the TASK that called it.

YOU CANNOT run PAUSE inside a SUB-PROGRAM

CAMEL99 MULTI-TASKING USER AREA

SEE SECTION: PAD RAM Usage Table for details of the USER AREA

NOTE:

The CPU registers are not declared as named USER VARIABLES in the KERNEL but you can create them if you need them like this. The tick (') character is Forth naming convention for "give me the address of"

```
0 USER 'R0     \ Remember each USER variable number goes up by two (2)
2 USER 'R1
4 USER 'R2
etc…
```

MORE User Variables Please

The Forth console task is called USER0. It exists in the SCRATCHPAD RAM and >8300. This means that USER0 is limited by the size of the SCRATCHPAD chip in the console. This limits the number of USER VARIABLES USER0 can have.

The last USER variable for the Forth console task that you can safely use is HEX 6E AND THERE ARE SOME THAT YOU CANNOT USE. See:

New TASKS that you create can have **more** USER variables. You need to change the value of USIZE in MTASK99 to make more space, or create a new constant for the big user spaces. FORK always puts the local stacks at the end of the user area so everything will expand with that small change. Then define more USER variables with higher numbers.

```
\ Example
HEX
300 MALLOC CONSTANT BIGTASK  \ EXTRA large user variable space

\ define user variable for the BIGTASK
70 USER X  \ now we can make more task specific user variables
72 USER Y
73 USER Z
\ ETC…
1F0 USER ANOTHERVARIABLE
1FE USER LASTVARIABLE

\ BUT REMEMBER: IF THE FORTH CONSOLE TRIES TO USE THESE HIGH NUMBER USER VARIABLES
\ IT WILL BE REACHING INTO GPL SCRATCHPAD MEMORY AND IT WILL CRASH THE SYSTEM
```

Example Multi-tasking Code

You can type this example into the editor and then INCLUDE the file to make a simple multi-tasking program.

```
NEEDS MULTI FROM DSK1.MTASK99

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

\ You can examine the variables with this word
: SCOPE ( variable -- ) BEGIN   PAUSE DUP @ .  KEY? UNTIL DROP ;

\ Type these commands at the Forth console
\ MULTI        ( enable task switcher)
\ TASK1 WAKE
\ TASK2 WAKE

\ X SCOPE   ( press any key to stop scoping)
\ Y SCOPE

\ TASK2 SLEEP
\ Y SCOPE    ( it will have stopped changing. Cool! :-)
```

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 principle "How do you eat an elephant?"   Answer: Piece by piece.

Consider this traditional Assembly language work flow:
1. Write your program in the editor, save as SOURCE file
2. Assemble (means translate from text) the program with the Assembler, giving OBJECT file.
3. *Link the OBJECT file with the LINKER giving a BINARY file
4. Load the BINARY file into RAM to run it
5. If it fails Goto 1
   -
*TI-99 joins step 3 and 4 with a special "LINKING LOADER" program to reduce a step.

Now consider how you make an Assembly language program in Forth"
1. Start CAMEL99 Forth
2. Load the Assembler (INCLUDE DSK1.ASM9900)
3. Write a short CODE word in Assembly language at the console.
4. Test the CODE word by typing its name and examine the stack output
5. if it fails Goto 3

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

\ "two-plus-store" adds 2 to the contents at the address in the Top of stack.
\ 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.

```
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. Notice the code says *TOS. This is called "indirect addressing". It tells the CPU to increment the value that is at the address contained in the register. More on that later.

INCT, is the Assembler name for the "INCREMENT BY TWO" instruction.   All the CPU instructions in ASM9900 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. The removes 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 is the instruction that NEXT, 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?
```

Addressing Modes

This manual cannot be a comprehensive treatise on Assembly language but we provide some concepts here to help you get started. Consult the Editor Assembler manual for more details or visit Artariage.com

Direct Addressing

The simplest way to use CPU register is to take the contents of one register and move it to another register.

```
R0 R1 MOV,
```

Indirect Addressing

Sometimes we have a memory address in a register and we want to get the value that is in that address location, into a register. This is when we use "Indirect Addressing". Indirect addressing in Assembly language can be confusing for the newcomer to Assembly language. It might be helpful to understand it as being similar to 'fetch' in Forth.

For example, if we created a VARIABLE X in Forth, X returns to us the address in memory where it keeps it's value stored. The fetch operator '@' gets the value and puts it on the DATA stack. Below is how that could work in Assembly language

```
VARIABLE X          \ make a normal Forth variable
R0 X LI,            \ load immediate, that address of X into a CPU register
R0 **  R1 MOV,      \ move the contents of the address in R0 to R1
```

After the MOV instruction R1 will contain the value stored in the variable X.
(There is an even easier way to do this. See: "symbolic addressing")

Example: increment a variable by two

Not only can you get the contents of a memory address with indirect addressing you can also change the contents. This part of the power of the TMS9900 CPU. The TMS9900 CPU has an instruction INCT that increments a register by two. If we use indirect addressing we can increment not the register value itself but the value ... in the address ... that is in the register. It's indirect.

Below we see code that gives us a new word that can add 2 to any Forth variable .

```
INCLUDE DSK1.TOOLS
INCLUDE DSK1.ASM9900

VARIABLE X

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

Below is a screen image of how we can test 2+!

```
   ok
    ok
VARIABLE X   ok
    ok
CODE 2+! ( addr -- ) *TOS INCT,    NEXT,
 ENDCODE   ok
    ok
X .S ¦C17C   ok
X ? 00   ok
    ok
X 2+!   ok
X ? 02   ok
X 2+!   ok
X ? 04   ok
```

Explanation of the Screen image and the code for 2+!

- We created the Variable X and assembled the CODE word 2+!
- Invoked X which puts the address of X on the Forth DATA Stack
- The top of the DATA stack is kept in R4 (renamed TOS)
- The value of X is 0 when we use ? (from DSK1.TOOLS)
- We looked at the address X (C17C) by using .S to see the DATA stack contents
- Use 2+! on X
    - 2+! used indirect addressing on the TOS register to add 2 to the variable X
- Inspect X with ? And see it now contains 2
- Use 2+! again
- Confirm that X now contains 4

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. Below we show you the code that creates the new names as constants and the different addressing modes as colon definition macros. This code is in DSK1.ASM9900

Also notice that we have made extra names that make the different addressing modes look like TI Assembler notation. This is a bit of "syntax sugar" that make code look nicer when you are working with the Forth system registers. You can still use the ordinary register addressing modes if you prefer them.

```
\ R0..R3 are available for your programs.
\ R5 can be used inside your CODE words but will be erased when you return to Forth
\ R8 can be used inside your CODE words but will be erased when you return to Forth

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

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

\ R7 is renamed to RP and is used as the return stack pointer
R7 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 erased when you return to Forth
R8 CONSTANT W
: (W)        W ()  ;
: *W         W **  ;
: *W+        W *+  ;

\ R9 renamed to IP is the Forth machine's "Interpreter Pointer"
R9 CONSTANT IP
: (IP)       IP () ;
: *IP        IP ** ;
: *IP+       IP *+ ;

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

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

\ R12 is used for CRU addresses only in CAMEL99.  See CRU! CRU@ in DSK1.CRU2

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

Forth 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 Assemble 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.

Source 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.

```
\  ASM9900    Usage                 * TI Syntax Equivalent
\  -----------------              * --------------------
\   src.    dst.                    * label     src.  dst.
\   ----    ----                    * -----     ----  ----
   R13 **    R2  MOV,                         MOV *R13, R2
   R13 *+    R2  MOV,                         MOV *R13+,R2
```

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

```
VARIABLE X                       X       DATA 0
VARIABLE Y                       Y       DATA 0

   X @@  R12  ADD,                       A    X@,R12
   X @@  Y @@ MOV,                       MOV   X@,Y@

HEX
CREATE ARRAY  100 CELLS ALLOT     ARRAY  BSS >100*2

ARRAY R13 () R2 MOV,                     MOV @ARRAY(R13),R2
```

*We don't use the "@" sign in the Forth Assembler when we use index addressing mode but if you really wanted to see it you could make a macro at the top of your code.

```
: @(R13)      R13 () ;
```

 Now you can write the code like this

```
  ARRAY @(R13) R2 MOV,
```

Assembly Language Macros

A macro is a group of instructions that can be used in your program with one name. This is a common feature in advanced assemblers but can be complicated to implement. An interesting feature of the Forth Assembler is that <u>we can</u> and <u>we do</u> use the colon definition to create "macros". Below are some the macro-instructions that are used in CAMEL99 Forth.

Example 1:

The Forth address interpreter is a three instruction routine that must run at the end of every code word. To save space we keep the address of this routine in Register 10 and branch indirectly to the address. This way, we use only 1 CELL (2 bytes) each time we call NEXT. Here is how we create the macro that does the job.

```
: NEXT,       *R10  B,  ;
```

Notice that all it took was to wrap the instruction inside a colon definition.

Example 2:

The TI-99 Assembler has something they call a "pseudo-instruction" "RT" to return from a sub-routine. Since the branch and link instruction (BL) saves the return address in R11, all we need do to return is branch indirectly through the address in R11. We create this "pseudo-instruction" with a macro.

```
: RT,         *R11 B, ;
```

Example 3:

Since the Forth virtual machine has two stacks it is handy to have single instructions that push and pop things from these stacks. The CAMEL99 Assembler provides PUSH, POP,   RPUSH and RPOP, for these purposes. Here are the definitions of those macros.

```
: PUSH,     ( src -- )  SP DECT,  *SP  MOV, ;
: POP,      ( dst -- ) *SP+       SWAP  MOV, ;
: RPUSH,    ( src -- )  RP DECT,  *RP  MOV, ;
: RPOP,     ( dst -- ) *RP+       SWAP  MOV, ;
```

Example 4:   Macros that use Macros

The 9900 CPU cannot "nest" sub-routines. (make a sub-routine call another sub-routine) but you can nest sub-routines with a CALL macro.   The CALL, macro uses RPUSH and RPOP macros to save and restore R11 on the Forth return stack. Very convenient.
You use CALL, exactly the same as you would use BL,
Return from CALL is done with the pseudo instruction RT, just like when you use BL,

```
: CALL,  ( address -- )
         R11 RPUSH,         \ save R11 on forth return stack
    ( addr)  BL,            \ branch & link, which saves the PC in R11
         R11 RPOP, ;
```

We have to end the macro with R11 RPOP, after BL, so that when we return from the Branch and link, R11 is restored to the original value that we pushed onto the return stack.

CALL Macro Usage Example

```
CREATE WMODE  ( Vaddr – Vaddr)      \ sub-routine to set VDP write mode
      0 LIMI,
      R1 STWP,                      \ R1 has our workspace (ie: 8300)
      9 (R1) 8C02 @@ MOVB,          \ write odd byte from R4, the TOS register
      TOS 4000 ORI,
      TOS 8C02 @@ MOVB,
      RT,                           \ return from sub-routine


CREATE VC! ( char vaddr --)         \ sub-routine to write byte to VDP address
      WMODE @@ CALL,                \ inside a sub-routine, we *MUST* CALL
      TOS POP,
      9 (R1) VDPWD @@ MOVB,         \ Odd byte R4, write to screen
      TOS POP,                      \ refill TOS
      2 LIMI,
      RT,


CODE MYWORD
```

Register Usage in CAMEL99 Forth

It is important that you understand which registers are free to use and which registers are used by the Forth System. Registers that must not be used unless you save them, are highlighted in grey.

Registers that must not be used in multi-tasking programs, are highlighted in pink.

Register Table

The highlighted registers in the table cannot be used by your CODE words unless you save them first and restore them when your CODE word is finished. Generally not necessary since you have 6 free registers.

Register Name	Alias	Use
R0		general purpose register
R1		general purpose register
R2		general purpose register
R3		general purpose register
R4	TOS	Top of DATA stack cache register
R5		Temp for NEXT, overflow for '*' and '/', general purpose register
R6	SP	Forth DATA stack pointer
R7	RP	Forth Return stack pointer
R8	W	Forth "working register", free for use in your code words
R9	IP	Forth Instruction pointer
R10		Contains the address of Forth's NEXT routine (>838A)
R11		9900 sub-routine return register
R12		9900 CRU address register (I/O devices use this. Disk, RS232)
R13		Multi-tasker LINK to next task
R14		Multi-tasker Program counter
R15		Multi-tasker task Status register

Of course you are free to make a new workspace in memory with the TMS9900 and BLWP (branch and load workspace pointer) to the new workspace giving you 13 free registers each time.

Proper Use of the TOS Register

CAMEL99 keeps the top item of the DATA stack, called TOS, 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 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,
         ENDCODE
```

Condition 3

This situation is one where you need a cell on the stack to hold your output data. 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,
         ENDCODE
```

Condition 4

The final example is one where you have consumed the data in the TOS register and you need to refill TOS
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. Use the POP, macro for this purpose.

```
CODE ON  ( addr -- )
         *TOS SETO,       \ set all bits at the address in TOS to one
         TOS POP,         \ Refill the TOS from data stack with POP,
         NEXT,
         ENDCODE
```

Accessing Forth Data in Assembler

A side-effect of the Forth's simple approach to memory is that it is easy to use Forth's data from the assembler. If we consider the Forth VARIABLE, it does nothing more than give us the address in memory where an integer is held. This is the same way that TI Assembler looks at memory.

Below is a table showing how to understand Forth data structures and their equivalents in conventional Assembler.

Forth	TI Assembler	Description
VARIABLE <LABEL>	<LABEL> DATA 0	Variable is like a label and the DATA directive together
CREATE <LABEL>	<LABEL> DATA 0	CREATE the same as a variable but not init to zero
CREATE <LABEL> 40 ALLOT	<LABEL> BSS 40	Same as a label and the BSS directive
CONSTANT	<LABEL> EQU	Use Forth Constants as you would use EQU
CREATE STUFF BYTE 1,2,3	<LABEL> BYTE 1,2,3	A custom BYTE creator has been made in Camel99 *(INCLUDE DSK1.DATABYTE)*
CREATE MORESTUFF DATA 1,2,3	<LABEL> DATA 1,2,3	A custom DATA creator has been made in Camel99 Forth. *(INCLUDE DSK1.DATABYTE)*

CREATE A$ S" Hello" S,	<LABEL> BYTE 5
	         TEXT 'Hello'
	         EVEN.
		S" makes a string literal, Leaves address, length on data stack.
		S, compiles a string (address, length) into memory at HERE

ASM Data Examples

```
\ fast fetch from a variable
VARIABLE X
CODE X@  ( -- n)
     TOS PUSH,
     X @@ TOS MOV,     \ @@ is symbolic addressing
     NEXT,
ENDCODE

\ fast store to a variable
CODE X! ( n -- )
     TOS X @@ MOV,
     TOS POP,
     NEXT,
ENDCODE

\ fastest variable to variable movement
CODE := ( addr addr -- )
     *SP+ R0 MOV,
     R0 ** *TOS MOV,
     TOS POP,
     NEXT,
ENDCODE

\ Fast access to an array in CPU RAM
100 CONSTANT SIZE
CREATE ARRAY    SIZE CELLS ALLOT

\ fetch contents of any cell in ARRAY
CODE ARRAY@  ( i -- array[i]@)
     TOS  1 SLA,              \ shift R1 1 bit left (multiply by 2)
     ARRAY (TOS) TOS MOV,     \ fetch contents of ARRAY(TOS) to TOS
     NEXT,
ENDCODE
```

```
\ store 'n' on stack to any cell in ARRAY
CODE ARRAY!  ( n index --)
     TOS  1 SLA,              \ shift TOS 1 bit left (mult. By 2)
    *SP+ ARRAY (TOS) MOV,     \ POP 2nd stack item into address ARRAY(TOS)
     TOS POP,                 \ refill TOS register
     NEXT,
ENDCODE
```

```
\ Example: building strings for use with assembler, get the length byte

CREATE A$ S" The rain in spain falls mainly on the plain" S,

CODE LENGTH ( $ -- n) \ get the length of a counted string on the stack
     *TOS TOS MOVB,      \ the $ address is in TOS, get first byte
      TOS  8 SRL,        \ SHIFT the byte to the other side of the register
      NEXT,              \ return to Forth
      ENDCODE
```

Note:

LENGTH is shown here as an example. In FACT it is the definition of the Forth word C@ (character fetch)

```
\ CREATE array of data constants

CREATE MYCONST  0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 ,

CODE []@ ( index array – n )
     *SP  *SP ADD     \ multiply index by 2 ( we are indexing cells)
     *SP+ TOS ADD,    \ add index to array address
     *TOS TOS MOV,    \ fetch the contents of the address to TOS
      NEXT,
      ENDCODE

\ usage:
\ 6 MYCONST []@ . 6 ok




\ Using DATA and BYTE directives for large data blocks
\ It has been written to work similarly to TI Assembler's directives

INCLUDE DSK1.DATABYTE

\ Crash sound list from TI E/A Manual page 324

HEX
CREATE CRASH
       BYTE 03,9F,E4,F2,05
       BYTE 02,E4,F2,10
       BYTE 02,E4,F4,8
       BYTE 02,E4,F6,6
       BYTE 02,E4,F8,4
       BYTE 02,E4,FA,2
       BYTE 01,FF,0
```

Structured Branching and Looping

One of the cleverest things about the traditional Forth Assembler is the use of branching and looping that looks very much like regular Forth. In the conventional **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.

```
CODE TEST1                   \ structured Forth assembler code
      BEGIN,
          R0 1000 ADDI,
          R0 R1 CMP,
          GTE IF,
                R3 CLR,
          ENDIF,
          R1 R0 SUB,
      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.

Countdown Loop in Assembler using WHILE,

```
DECIMAL
VARIABLE X               \ we can use a Forth variable as the loop counter
CODE COUNTER
      X @@ 100 LI,       \ set the value of X to 100
      BEGIN,             \ start of loop
        X @@ DEC,        \ reduce X by 1 (decrement instruction)
      NE WHILE,          \ while x <> 0
        <your code>      \ do stuff…
        <your code>
        <your code>
      REPEAT,            \ jump back to BEGIN,
```

Countdown Loop in Assembler using UNTIL,

```
CODE COUNTDOWN
      R1 100 LI,         \ using a register as a loop counter is fastest
      BEGIN,
        <your code>      \ do stuff…
        <your code>
        R1 DEC           \ decrement the loop counter
      NE UNTIL,          \ loop until R1 <> 0
      NEXT,
ENDCODE
```

Pass the count value from Forth to Assembly Language

```
CODE FASTLOOPS ( n --)   \ Pass the loop value as a parameter from Forth
\ n is already in R4, the TOS register
      BEGIN,
        <your code>      \ do stuff…
        <your code>
        TOS DEC          \ decrement the loop counter
      NE UNTIL,          \ loop until R1 <> 0
      TOS POP,           \ we don't need n now, so refill TOS with item on DATA stack
      NEXT,
ENDCODE
```

ASM9900 Comparisons

Below is the list of accepted comparisons that you can used with ASM9900 for IF and BEGIN loops. Beside them is what they actually do in your code. This table is taken directly out of the code of ASM9900.

```
01 CONSTANT GTE      \ JLT to ENDIF,
02 CONSTANT HI       \ JLE to ENDIF,
03 CONSTANT NE       \ JEQ to ENDIF,
04 CONSTANT LO       \ JHE to ENDIF,
05 CONSTANT LTE      \ JGT to ENDIF,
06 CONSTANT EQ       \ JNE to ENDIF,
07 CONSTANT OC       \ JNC to ENDIF,
08 CONSTANT NC       \ JOC to ENDIF,
09 CONSTANT OO       \ JNO to ENDIF,
0A CONSTANT HE       \ JLO to ENDIF,
0B CONSTANT LE       \ JH  to ENDIF,
0C CONSTANT NP       \ JOP to ENDIF,
```

Labels for Branching and Looping

Most of the time is best to stick with structured branches and loops. Although the computer doesn't care how we make it jump around later when you have to understand how your code works, structured programming is far easier to re-learn.

There are special times however when you might want to jump out of loop because of an error condition. When you need this, structured programming can get messy when a simple "GOTO" is all you really need. In 2021 we added numbered labels to ASM9900 so you have them when you need them.

Just include the file: DSK1.ASMLABELS

Below the same structured code example from above has been re-written with numbered labels:

```
CODE TEST2                  \ un-structured Forth Assembler
1 $:      R0 1000 ADDI,     \ begin
          R0 R1 CMP,
          2 $ JLT,           \ **GTE if
          R3 CLR,
2 $:      R1 R0 SUB,
          1 $ JMP,          \ again
ENDCODE
```

** Notice that "GTE IF," in the structured assembler, is really a JLT instruction, ie: the opposite logic

When you use labels you use the following TMS9900 jump instructions **not the comparison tokens** seen before.

```
JMP, jump unconditional        JLT,  jump less than
JLE, jump low or equal         JEQ,  jump if eq
JHE, jump high or equal        JGT,  jump greater than
JNE, jump not equal            JNC,  jump if no carry
JOC, jump on carry             JNO,  jump if no overflow
 JL, jump if low               JH,   jump if high
JOP, jump on parity
```

Using TMS9900 Sub-routines

Sometimes you want to use "native" sub-routines that are provided by the TMS9900 CPU. These are called with the "branch and link" instruction, (BL) and are the equivalent to GOSUB in BASIC. It's not quite as convenient as GOSUB in BASIC because BL, can only remember how to return from one GOSUB! If you use the BL instruction while you are already in sub-routine, your program cannot find its way back!

Here are things to remember if you want to make sub-routines in Forth Assembler:

1. When you use the BL instruction, the CPU stores the current program address in R11. (Literally it takes the internal program counter register (PC) and puts that value in R11.)
2. After BL executes, R11 holds the "return address" that your program must go back to. You must branch back to the address in R11 to return from a sub-routine. The CAMEL99 Assembler gives you the macro-instruction called RT, which does exactly that. (RT, is an alias for "R11 B,")
3. You CANNOT call a sub-routine while inside another sub-routine.
   (Unless you save R11 somewhere else temporarily. Read on for a Forth solution)
4. Your sub-routines need a "label" to branch to in Assembler. That's easy in Forth. We can use "CREATE" which puts a name in the dictionary and when the name is invoked gives the address after the text name. If you assemble some code

Below is a simple example of creating and calling a 9900 sub-routine.

Labels for BL, B, Instructions

If you want to branch to code or to a 9900 sub-routine you need to a way to tell the program where the sub-routine is. We can do that with the word CREATE. CREATE makes a name in the Forth dictionary and when we invoke that name we get back the address immediately after the name. If we assemble code after we CREATE a name that is the address where our code starts. That's exactly what we need!

Below is a simple example.

```
CREATE SQUARES      \ Name our sub-routine with CREATE
      R0 R0 MPY,  \ Square R0 result goes to R1
      R3 R3 MPY,  \ Square R3 result goes to R4, the TOS register
      RT,         \ return to calling program

CODE SUMSQUARES ( n1 n2 -- n3)
      *SP+ R0 MOV,         \ pop n1 into R0
      TOS  R3 MOV,         \ move n2 to R3
      SQUARES @@ BL,       \ Branch & Link to SQUARES
      R1 R4 ADD,           \ sum the squares, result is in R4 (which is top of stack)
      NEXT,
ENDCODE
```

If you don't like using the word CREATE for sub-routines, DSK1.ASMLABELS also has the L: operator which creates a dictionary word that gives the program address. L: is just an alias name for CREATE. You can make it yourself if you just want to use L: without loading the numbered labels file.

```
    : L:    CREATE ;
```

Multi-level Branch and Link

If you need to call a sub-routine from another sub-routine on the 9900 CPU you must figure out a way to save R11 on your own after the first branch and link is used. The can be simple if your program has a free register. Consider the example code below.

* Notice that in forth we must declare the sub-routines BEFORE we can use them*

```
CREATE SUM3
     R1 R0 ADD,
     R2 R0 ADD,
     R3 R0 ADD,
     RT,                  \ R11 has the way back to MULT3 so just RT,

CREATE MULT
     R11   R1 MOV,        \ SAVE R11 in R1
     SUM3  @@ BL,         \ call the SUM3 routine
     R0    R4 MPY,        \ we return from SUM3 to this instruction
     R1    R11 MOV,       \ R1 has the way to MAIN_PROGRAM, put it in R11
     RT,                  \ return to main program

CODE MAIN_PROGRAM
     R1 111 LI,           \ sum these 3 parameters
     R2 222 LI,
     R3 333 LI,
     R4   3 LI,           \ multiplier value goes into R4
     MULT @@ BL,          \ call the mult routine

     Etc…

     END-CODE
```

Remember this is only good for two levels of calling a sub-routine. If you needed three levels you would need another spare register and keep track of it manually. Couldn't we automate this? Absolutely!

The CALL Macro

If you wanted to write a program entirely in Forth Assembler you can do it but you might find it very handy to be able to call little nested sub-routines the way Forth does. We can do that with a macro that puts all the instructions together for you "automagically". We use the Forth return stack to save R11 every time we call a sub-routine and then restore the value of R11 when we come back.

Adding this Forth word to your system allows you to 'CALL' a sub-routine from a sub-routine until the entire return stack is full.

```
: CALL,  ( dst -- )
     R11 RPUSH,       \ save R11 on forth return stack
    ( addr)  BL,      \ branch & link saves the PC in R11
     R11 RPOP, ;      \ R11 RPOP, is laid down by CALL,
                      \ We have to lay it in the code after BL so
                      \ when we return from the Branch&link, R11 is
                      \ restored to the original value from the rstack
```

Example using CALL

```
CREATE SUM3
      R1 R0 ADD,
      R2 R0 ADD,
      R3 R0 ADD,
      RT,                     \ R11 has the way back to MULT3 so just RT,


CREATE MULT
      SUM3 @@ CALL,       \ call SUM3 with SYMBOLIC addressing, while in sub MULT
      R0    R4 MPY,       \ we return from SUM3 to this instruction
      RT,                 \ return to main program

CODE MAIN_PROGRAM
     R1 111 LI,           \ sum these 3 parameters
     R2 222 LI,
     R3 333 LI,
     R4   3 LI,           \ multiplier value goes into R4
     MULT @@ CALL,        \ call MULT with symbolic addressing
     Etc…

     END-CODE
```

Why not use BLWP?

 TI's BLWP instruction uses 32 cycles to call (symbolic addressing) and RTWP takes 14 cycles. This is slower than BL/RT but a bit faster than CALL,/RT. However even though it gives you a new register set it consumes another 32 bytes for a new workspace *AND* you have to create a 4 byte [16]vector data structure for every sub-program you call. In total it means you need to add 34 bytes of space for your program to use a BLWP sub-program. But remember, it adds eight bytes to your program every time you use the CALL macro.

Another dis-advantage with using BLWP is that you have reach back into the previous workspace or to the Forth stack to get parameters from the previous program into your sub-program. You are not in the Forth workspace anymore so some special code is needed to find the data in the Forth workspace. With BL, or CALL, you can see Forth's top of stack register, the data stack pointer and the return stack pointer right in your current workspace.

All that to say there is no free lunch. Only you can decide if your program should use BL, CALL, or BLWP. But if you need to use BLWP we have made that easier for you as well.

See the next chapter…

---

[16] A "vector" is a workspace and code-address put into memory side by side. BLWP typically needs this memory structure to be used to get to a new sub-program

Using BLWP in Forth

CAMEL99 Forth has a library word called BLWP that takes a vector address from the stack and does the BLWP instruction to the new workspace. /LIB.ITC/BLWP.FTH

*Don't confuse the Forth word BLWP with the assembler instruction called BLWP, (with the comma).

BLWP is typically used to call something referred to as a SUB-PROGRAM versus a sub-routine. The difference is a SUB-PROGRAM actually leaves the calling program and begins running in a completely different set of registers or "workspace" in TI speak. This is almost like running on a new CPU. This can be a very powerful feature when you need a lot of registers for a specific calculation. BLWP out of the Forth workspace, do the intensive math and come back to Forth. It's a nice feature of the 9900 CPU.

Normally in ASM9900 we would do something like this to call a SUB-PROGRAM:

```
INCLUDE DSK1.TOOLS
INCLUDE DSK1.ASM9900
INCLUDE DSK1.BLWP

DECIMAL
CREATE MYWORKSPACE   16 CELLS ALLOT   \ make space for 16 registers

\ Notice: We don't need "CODE" We just need the name's address. Therefore CREATE
CREATE SUBPROG-SUM10  \ this program sums 10 numbers into R0 very fast!
     R1 R0 ADD,
     R2 R0 ADD,
     R3 R0 ADD,
     R4 R0 ADD,
     R5 R0 ADD,
     R6 R0 ADD,
     R7 R0 ADD,
     R8 R0 ADD,
     R9 R0 ADD,
     R10 R0 ADD,
     RTWP,

\              workspace address    address of the code
\              ----------------     -------------------
CREATE MYVECTOR  MYWORKSPACE ,       SUBPROG-SUM10 ,
```

Things to understand here:

1. A BLWP vector is a memory structure with two addresses in it. The workspace address and the code-address.
2. We only need CREATE and COMMA to build a BLWP vector in Forth
3. Unlike in TI Assembler, the vector MUST be created AFTER the code. This is because Forth only understands words that are already defined.
4. Create makes the name MYVECTOR in the dictionary
5. The address of MYWORSPACE is comma-compiled into memory
6. The address of the SUBPROG-SUM10 is comma-compiled into memory.
7. That's it!
   •

* ALL sub-programs must end with the RTWP, instruction *

Running a SUB-PROGRAM from Forth

The Forth word BLWP can be used right at the console interpreter like any other Forth word… as long as you make a correct BLWP vector and the code ends with the RTWP instruction. So to run our sub-program example above, all we need to do is :

```
MYVECTOR BLWP ok
```

Getting data from a Sub-Program

So where is the sum that we so quickly calculated into R0 of our sub-program? Well… remember we jumped out of the Forth workspace completely to do the calculation. The convenient thing is that a 9900 workspace is just normal memory so we can read the external sub-program registers with @,! C@ and C! . It just like using Forth VARIABLES!

Since R0 is the first address in the workspace if we type MYWORKSPACE @ . we see the results of the summing. But how do we load our numbers to sum into the external sub-program registers?
Exactly as if the registers were a Forth array of cells.

Consider this code:
```
MYWORKSPACE DUP
          CONSTANT XR0
CELL+ DUP CONSTANT XR1
CELL+ DUP CONSTANT XR2
CELL+ DUP CONSTANT XR3
CELL+ DUP CONSTANT XR4
CELL+ DUP CONSTANT XR5
CELL+ DUP CONSTANT XR6
CELL+ DUP CONSTANT XR7
CELL+ DUP CONSTANT XR8
CELL+ DUP CONSTANT XR9
CELL+ DUP CONSTANT XR10
DROP
```

With this code we created a set of "variables" (ie addresses) that are the "external" registers of our sub-program. Now we can erase our 10 EXTERNAL registers and load the new registers:

```
DECIMAL
: CLRXREGS      MYWORKSPACE  10 CELLS 0 FILL ;

\ Now load up the workspace to do a sum:

CLRXREGS
1 XR1  !  2 XR2 !   3 XR3 !  4 XR4 !  5 XR5  !
6 XR6 !   7 XR7 !   8 XR8 !  9 XR  ! 10 XR10 !

MYVECTOR BLWP   \ run the sub-program from the interpreter!

XR0 ?           \ see our answer
```

Of course normally part of your program would be feeding numbers into the XR1..XR10 registers all the time and when you needed that fast total you would BLWP to your sub-program and pull the answer out of XR0.

We can view the workspace with dump to access the workspace like a variable

Get Parameters from Forth

A common trick used on the 9900 let's us get data from the "caller's" workspace and bring it into the registers in the SUB-PROGRAM'S workspace. This is simple because when we BLWP to our SUB-PROGRAM the address of the caller's workspace is sitting in R13 of the SUB-PROGRAM workspace. If we use 9900 Indexed addressing mode (ie: address+Register) the CPU will do this for us. Consider the following code example:

```
\ Macro to make is easy to access Forth's top of stack
: [TOS]      8 R13 () ;    \ 8 byte+R13 is the address of R4 in CAMEL99 Forth

HEX
CREATE SUBPGRM1 ( -- ) \ subpgram adds Forth TOS to HEX 99
      [TOS] R1 MOV,      \ move the content of Forth R4 to R1 in the new workspace
      your code...
      ...
      R7 [TOS] MOV,      \ put the answer back in Forth's top of stack register
      RTWP,              \ return to Forth!


\ Example accessing parameters on Forth's data stack
DECIMAL
: [SP]    12 R13 () ;    \ get Forth's data stack pointer register

HEX
CREATE  SUBPRGRM2
        [SP] R10 MOV,      \ R10 has the address of 2nd item on the Forth data stack
        R10 **  R2 MOV,  \ get the 2nd item into local R2
      2 R10 ()  R3 MOV,  \ get the 3rd item into local R3
      4 R10 ()  R4 MOV,  \ get the 4th item into local R4
       ...
       your code
       ...
       R4 [TOS] MOV,      \ put the result in TOS
       R3 2 R10 () MOV,  \ put another result back into Forth's stack
       RTWP,
```

Sub-program with Built-in VECTOR

You can save creating a separate word for the vector and the code routine by putting them together. Consider the code example below:

```
        DECIMAL
CREATE WRKSPC3  16 CELLS ALLOT

CREATE SUBPROG4
     WRKSPC3 ,  HERE CELL+ ,
     R1 R0 ADD,  \ HERE CELL+ above, points to this line
     R2 R0 ADD,
     R3 R0 ADD,
     R4 R0 ADD,
     RTWP,
```

We can see that we made a workspace. Then in the first line of our code we created a vector with comma but what is "HERE CELL+" about? HERE is the current address of the dictionary. We took that address added one cell to it and compiled it with comma. In other words "HERE CELL+" points to the next line of the code, where our first ADD instruction begins. So our vector is created by compiling the address 1 cell ahead of HERE.   Now we can call SUBPROG4 with BLWP directly! No need for a separate program name in the dictionary.

But Remember...

 If you incorporate the vector in the SUB-PROGRAM all the registers will begin 2 CELLS (ie: 4 bytes) **after** the vector. So in the example above we could define the addresses of the external registers like this:

```
SUBPROG4 2 CELLS + CONSTANT XR0
        XR0 CELL+  CONSTANT XR1
        XR1 CELL+  CONSTANT XR2

        ETC…
```

```
      □K
DROP  □K
DECIMAL  □K
: CLRXREGS     MYWORKSPACE  10 CELLS 0 F
ILL ;   □K
   □K
   □K
   □K
   □K
\ NOW LOAD UP THE WORKSPACE TO DO A SUM:
   □K
CLRXREGS  □K
1 XR1  !  2 XR2 !   3 XR3 ! 4 XR4 !   5
 XR5  !  □K
6 XR6 !   7 XR7 !   8 XR8 ! 9 XR9 ! 10
 XR10 !  □K
MYWORKSPACE 10 CELLS DUMP
C37E: 0000 0001 0002 0003 ........
C386: 0004 0005 0006 0007 ........
C38E: 0008 0009 000A 0000 ........
   □K
MYVECTOR BLWP  □K
XR0 ? 55  □K
```

The PROG: Directive

To simplify setting up everything for the programmer who needs to use a workspace sub-program we have created a directive called PROG: (SEE /LIB.ITC/PROG.FTH)

PROG: only needs you to allot some space for your workspace. After that you write your sub-program like a regular CODE word, but you give PROG: the workspace you want to use, remembering to end ALL SUB-PROGRAMS with RTWP. The PROG: directive does all the rest like this:

```
DECIMAL
CREATE WKSP     16 CELLS ALLOT  \ create a workspace

WKSP PROG: SUM-PROG  \ prog: COMPILES the workspace into SUM-PROG
     R1 R0 ADD,
     R2 R0 ADD,
     R3 R0 ADD,
     R4 R0 ADD,
     R5 R0 ADD,
     R6 R0 ADD,
     R7 R0 ADD,
     R8 R0 ADD,
     R9 R0 ADD,
     R10 R0 ADD,
     RTWP,
 ;PROG
```

Now the word SUM-PROG knows which workspace to use and how to BLWP itself!
We can now use our sub-program like a normal Forth word. See the screen capture below.

This SUB-PROGRAM would add 10 numbers faster than using other methods.

How Prog: Works

PROG: uses a magic part of Forth where we can specify what happens at compile time, when the word is created, but then also we can set what code will run when the word is invoked by Forth. Below is the complete definition of PROG:

```
: PROG: ( wksp -- )
    CREATE                      \ compile-time code is here
      ( wksp) ,  HERE CELL+ ,  \ make the vector
      !CSP                      \ record stack position (error checking)

    ;CODE *W BLWP,              \ W register holds the Data field address
                                \ we just run BWLP to the address in W
        NEXT,                   \ return to Forth
        ENDCODE


: ;PROG  ( -- ) ?CSP  ;  \ check stack position for junk left on it.
```

Notice that PROG: uses our integrated VECTOR trick with HERE CELL+ but it takes a workspace value from the stack at compile time and compiles it into vector created by PROG:

When we run a word created with PROG: the stuff that comes after ;CODE is what executes. Inside the Forth virtual machine there is a working register called "W". It contains the address of the data we "created". So in this case it is the beginning of the vector.

By running the BLWP instruction on the address in W, we run the code at our vector.
You are forgiven if you need to read this again. It's a little confusing the first time you encounter it.

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 muscial 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 a similar program in CAMEL99 Forth with BASIC line numbers in comments and extra comments for explanation.

(Comments don't compile into your program memory in Forth so you can use them liberally)

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

```
\ Random Color Dots
INCLUDE DSK1.RANDOM
INCLUDE DSK1.SOUND
INCLUDE DSK1.CHARSET
INCLUDE DSK1.GRAFIX

DECIMAL
\ rather than use variables we make words with the same names
\ that calculate the numbers we need and leave them on the stack
: Y   ( -- n ) 1001 RND 110 + ;  \ 170 (does not calc. musical notes.)
: CHR ( -- n )   79 RND 32 + ;   \ 180
: ROW ( -- n )   24 RND   ;      \ 190
: COL ( -- n )   32 RND   ;      \ 200

\ create a SOUND word from low level sound chip words
: SOUND  ( dur freq att --) DB  HZ  MS MUTE ;

: RUN ( -- )  \ we could call this whatever we want. RUN is fine
     RANDOMIZE               \ 110
     CLEAR                   \ 120
     19 4 DO                 \ 130
        I I I COLOR          \ 140
     LOOP                    \ 150
     BEGIN
       GEN1 125 Y -2 SOUND   \ 210 Sound Generator #1. Controls program speed
       COL ROW CHR 1 HCHAR   \ 220
      ?TERMINAL              \ check for the break key is up to you in Forth
     UNTIL                   \ 230  (GOTO BEGIN)
\ BASIC assumes you always want to restore things. Forth let's you choose.
     8 SCREEN                \ restore screen like BASIC does
     4 19 2 1 COLORS         \ change color on all char sets 4..19
     CHARSET ;               \ restore default TI-99 character shapes
```

Guess the Number in Forth

```
\ Demonstrate Forth style. Create words to make the program.
INCLUDE DSK1.INPUT
INCLUDE DSK1.RANDOM
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"
        0= ;              \ if it's NOT 'Y' return true

: 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$)
50 NEXT L
60 DATA 010207091F247F92,8040E090F824FE49,FF92FF24FF92FF49,
   AA55448920024801,000217357CFC44AA,0008081C2A081414
70 CALL CHAR(104,"0083C7AEFBEFBDF7")
80 CALL CHAR(105,"00078F5DF7DF7BEF")
90 CALL CHAR(106,"000E1FBAEFBFF6DF")
100 CALL CHAR(107,"001C3E75DF7FEDBF")
110 CALL CHAR(108,"00387CEABFFEDB7F")
120 CALL CHAR(109,"0070F8D57FFDB7FE")
130 CALL CHAR(110,"00E0F1ABFEFB6FFD")
140 CALL CHAR(111,"00C1E357FDF7DEFB")
150 CALL COLOR(10,6,5)
160 X=13
170 C=1
180 PRINT TAB(X+1);"AB"
190 C$=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.

NEVER write a Forth program like this, as one giant word definition **It is for education only.**

It uses strings like BASIC, but they are not needed. See the next example for a Forth "style" version.

```
\ load the language extensions to imitate BASIC a little
INCLUDE DSK1.GRAFIX
INCLUDE DSK1.STRINGS
INCLUDE DSK1.CALLCHAR
INCLUDE DSK1.CHARSET


\ ALL variables must be defined first
VARIABLE X
VARIABLE C
VARIABLE T
VARIABLE Y
\ Define and erase string variables.
32 DIM Q$   Q$ =""   \ (Not standard Forth)
32 DIM B$   B$ =""   \ BTW Strings can have any name. Imagine that...
32 DIM C$   C$ =""

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

DECIMAL                              \ Interpret DECIMAL numbers
: RUN   \ Forth doesn't have RUN so we make one
        CLEAR                              \ 10 CALL CLEAR
        4 SCREEN                           \ BASIC does when running.
        S" 010207091F247F92" 65 CALLCHAR \ 20 FOR L=65 TO 70
        S" 8040E090F824FE49" 66 CALLCHAR \ 30 READ Q$
        S" FF92FF24FF92FF49" 67 CALLCHAR \ 40 CALL CHAR(L,Q$)
        S" AA55448920024801" 68 CALLCHAR \ ...
        S" 000217357CFC44AA" 69 CALLCHAR \ ...
        S" 0008081C2A081414" 70 CALLCHAR \ 50 NEXT L

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

        S" 0083C7AEFBEFBDF7" 104 CALLCHAR \ 70 CALL CHAR(104,"0083C7AEFBEFBDF7")
        S" 00078F5DF7DF7BEF" 105 CALLCHAR \ 80 CALL CHAR(105,"00078F5DF7DF7BEF")
        S" 000E1FBAEFBFF6DF" 106 CALLCHAR \ 90 CALL CHAR(106,"000E1FBAEFBFF6DF")
        S" 001C3E75DF7FEDBF" 107 CALLCHAR \ 100 CALL CHAR(107,"001C3E75DF7FEDBF")
        S" 00387CEABFFEDB7F" 108 CALLCHAR \ 110 CALL CHAR(108,"00387CEABFFEDB7F")
        S" 0070F8D57FFDB7FE" 109 CALLCHAR \ 120 CALL CHAR(109,"0070F8D57FFDB7FE")
        S" 00E0F1ABFEFB6FFD" 110 CALLCHAR \ 130 CALL CHAR(110,"00E0F1ABFEFB6FFD")
        S" 00C1E357FDF7DEFB" 111 CALLCHAR \ 140 CALL CHAR(111,"00C1E357FDF7DEFB")

( Notice Forth has different character set numbers)
        13 6 5 COLOR                        \ 150 CALL COLOR[10,6,5]
 \ PYRAMID
        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)
\ 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                           \ Forth is too Fast so we delay
           KEY?
        UNTIL                           \ 400 GOTO 320
        8 SCREEN                \ BASIC does this automatically
        CHARSET                 \ Forth must be told.
        COLLAPSE                \ collapse the string stack
;
```

Forth Style Version of Denile

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 other down-side of using variables in Forth is that your code can only be use 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. Since '.' Prints a number it is common to start other print words with '.'
   b. 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 Concept

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
\ Orignal program by RETROSPECT, Atariage.com
INCLUDE DSK1.GRAFIX
\ INCLUDE DSK1.STRINGS   \ We don't need strings for this version.
INCLUDE DSK1.CHARSET


\ Character DATA patterns can be 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 FBEF 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 and writes to VDP RAM
\ We didn't have to but we used the word [CHAR] to make the code read easier.
\ [CHAR] is an IMMEDIATE WORD used in compiling.
\ It PARSES a Character from the input and compiles the ASCII number into
\ our code

        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 concatentation 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 regular BRICK


\ print n*2 BRICKs. If n=0 ?DO will do nothing.
\ we multiply 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 with slope on ends
      LOOP
      CR .SAND                         \ print some sand
      CR                               \ make room for the river Nile
      CR
      CR .SAND ;                       \ and print more 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  \ yes! So drop it and replace with 104
            THEN ;       \ then return

\ Forth Style:
\ Create custom variable incrementers, that limit themselves!
: 1+@   ( variable -- n ) DUP @ 1+ LIMITER DUP ROT ! ;
: 2+@   ( variable -- n ) DUP @ 2+ LIMITER DUP ROT ! ;

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 RIVER loop is so simple. Compare to previous version.
: RIVER   ( -- )
            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
          RIVER                  \ this loops until key pressed

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

APPENDIX

Background Information

Library files [17]Available with CAMEL99 Forth

The best way to know what is in the library files is to read them. The comments should explain what is going on and in most cases there is also example code that you can use to try the features of the library yourself.

File Name	Description
3RD4TH	Fast move 3rd or 4th item to top of stack
80COL	F18 card only. Switch to 80 column mode
9902C	RS232 driver written in Forth
9902CODE	RS232 driver written in Forth Assembler (faster)
9902D	RS232 driver written in Forth with Assembler assistance
9902PRIM	RS232 driver translated to machine code. No assembler needed
ANSFILES	Standard ISO Forth file words
ASM9900	TMS9900 Forth Assembler
AUTOBAUD	RS232 automatic baud set experiment (not working)
AUTOMOTION	Automatic sprite motion like Extended BASIC
BASICHLP	Load file to include files needed for BASIC programmers
BFBLOCKS	Empty block file for use with DSK1.BLOCKS
BGSOUND	Play sound lists from CPU RAM in the background (multi-tasking)
BLOCKS	Create and Open Forth block file. (Like FbForth,TurboForth)
BLWP	Call subprograms that you write, from Forth
BOOLEAN	Create and manage arrays of bits. Very compact
BREAK	A word to emulate BASIC Fnct 4 key
BUFFER	Forth 2012 word to create a buffer
CALLCHAR	Interpreting word to convert a string to char pattern
CAMEL99	The CAMEL99 Forth Kernel. (8K bytes)
CASE	ANS/ISO Forth case statement
CATALOG	implements CAT to list the contents of a disk drive
CELLS	Shows implementation of Forth word CELLS (Now in DSK1.SYSTEM)
CHAR	Show implementaiton of CHARS. (Now is DSK1.SYSTEM)
CHARSET	Like X-BASIC. Restores default characters. Show GROM access
CLOSEALL	CLOSEs all open files. Includes DSK1.ANSFILES
CODE	Shows implementation of CODE, ENDCODE. (Now in DSK1.SYSTEM)
CODEMACROS	Tools to make fast data access macros and arrays
CONDCOMP	Implements conditional compilation words
COOLSPRITE	Demo program. Translation of BASIC to Forth example
CORE1	Hayes Forth compliance test program for Core words.
CRU, CRU2	Two flavours of CRU word set. CRU is smaller.
DATABYTE	Implements DATA and BYTE directives like Assmbly Language
DEFER	Implements DEFER and IS for deferred word creation
DEV	Loads all files needed for program development
DIR	Load DIR to see a listing of the disk directory file names
DIRSPRIT	Implements "direct" Sprite control. (no automotion)
DOTLINES	.LINES can print the number of lines compiled in a file
DOUBLE	Double integer wordset for 32bit math
EASYFILE	Simplified file access words
ELAPSE	ELAPSE times the runtime of code up to 9 minutes
ENUM	Provides enumatorated CONSTANTs

---

[17] New library files are added to the CAMEL99-V2 GITHUB repository so check the LIB.ITC folder for more tools and language extensions

```
TARGLOOPS          Implements Forth Loops and IF/ELSE/THEN. *TEACHING FILE*
TIRAND             TI-FORTH random number generator if you need it
TOOLS              Programmers tools DUMP, .S WORDS etc.
TRACE              TRACE utility adapted from TI-Forth
TRAILING           Strip trailing spaces from a string. Also –LEADING & TRIM
TRIG               Trig functions using lookup table
TTY1               Terminal on RS232 code. (no tested)

UDOTR              "Un-signed dot R", prints numbers right-justified

VALUES             VALUES are like constants that can be changed
VBYTEQ             Implement a circular byte queue in VDP RAM
VDPBGSND           Built sound lists in VDP RAM with Multi-tasking player
VDPDOTQ             VDP dot quote: builds text strings VDP RAM to save space
VDPEXTRA           Machine code routines used by TTY Forth
VDPEXTRAS          Same as VDPextra but with Assembly source code
VDPMEM             VDP memory manager word
VDPSAVE            Generic LOAD and SAVE VDP ram at high speed
VDPSOUND           VDP sound list creation with play. NON-Multi-tasking
VDPSTRNG           VDP memory based string library. Saves CPU RAM use
VDPTYPE            Faster type to text on screen displays
VECTORIO           Example of Vectoring I/O in Forth (untested)
VT100              VT100 terminal control for RS232 TTY Forth
```

SCRATCHPAD RAM Usage

CAMEL99 makes special use of the small 256 byte RAM chip (PAD) that resides on the 16 bit buss. In this Forth system there are three separate uses of the "PAD" RAM:

1. Workspace: CPU registers 0 to 15
2. User Variables: A table of variables that are duplicated every time you create a new task.
3. CODE routines: This 16 bit memory is two times faster than expansion ram so putting some of the internal parts of Forth here make the system run about 20% faster.

User Area Description

User Variables were created in Forth to support multitasking. They create what is typically called a USER AREA, a block of memory that is unique to each Forth task running on the machine. In a typical Forth system the USER AREA may contain all the system variables needed by the Forth interpreter and the compiler, the I/O system and the DATA and RETURN stacks as well. Most Forth implementations must either allocate a register or create a memory location to be the USER Pointer (UP) that points to the USER AREA in memory, of the currently running task. The 9900 lets us do things differently…

No UP Register Required with the 9900

CAMEL99 Forth takes the 9900 "Workspace" concept one step further. Using to the TMS9900 Architecture, CAMEL99 has the CPU registers in the Workspace, as per normal, but then it also adds the USER VARIABLES immediately following the registers. The 9900 internal Workspace register (WP) already points to a form of "user area" for the CPU registers.

CAMEL99 simply expands the workspace to be 110 bytes rather than 32 bytes. The first 32 bytes are registers and the rest are USER VARIABLES. We are using the internal CPU Workspace pointer as a replacement for the normal "User Pointer" (UP) register in a conventional Forth implementation.

This means there is one less thing to change when we switch from task to task. In fact we context switch in CAMEL99 with one instruction: RTWP

In practical programmer terms this means that the USER VARIABLE list starts at >20 rather than 0. No big deal. At runtime to access a USER VARIABLE we run three instructions:

\ The 'W' register points to the DATA field of the running Forth word

```
CODE: DOUSER ( -- addr)
          TOS PUSH,      \ make some space in the TOS register
          TOS STWP,      \ store workspace register WP in TOS
        *W TOS ADD,      \ add the offset stored in the USER variable
              NEXT,
          END-CODE
```

One Minor Caveat

The only problem we had to navigate was there are 3 CELLS right in the middle of our variables that are used by the Device Service Routines.
Not a problem!   We simply NEVER declare USER VARIABLES >54, >56 and >58. Problem solved.

SCRATCHPAD RAM Usage Table

http://www.unige.ch/medecine/nouspikel/ti99/padram.htm .

"The first part of the scratch-pad is used mainly by TI-Basic and Extended Basic. If a program does not require any of these languages, these bytes are free for use."

Registers are orange, user variables are highlighted in GREEN, GREY is TI-99 O/S locations, RED is CAMEL99 code.

Address	NAME or MNEMONIC	Use	DATA Type
>8300	'R0	Scratch register, also used by video routines	CPU Register
>8302	'R1	Scratch register, also used by video routines	CPU Register
>8304	'R2	Scratch register, also used by video routines	CPU Register
>8306	'R3	Scratch register, also used by video routines	CPU Register
>8308	TOS	Top of data stack cache register	CPU Register
>830A	'R5	Temp used by thread interpreter	CPU Register
>830C	SP	Forth DATA Stack pointer	CPU Register
>830E	RP	Forth Return Stack pointer	CPU Register
>8310	W	Forth Working register	CPU Register
>8312	IP	Forth IP register	CPU Register
>8314	NEXT	Forth Address of NEXT	CPU Register
>8316	'R11	Forth sub-routine link	CPU Register
>8318	'R12	Forth CRU base address	CPU Register
>831A	'R13	Multi-tasker Next Task Workspace	CPU Register
>831C	'R14	Multi-tasker Next Task PC	CPU Register
>831E	'R15	Multi-tasker Next Task Status Reg.	CPU Register
>8320	TFLAG	TASK Flag Awake/Sleep	User VAR 20
>8322	JOB	TASK Forth word that will run	User VAR 22
>8324	DP	Forth Dictionary pointer	User VAR 24
>8326	HP	Forth HOLD pointer for number conversion	User VAR 26
>8328	CSP	Check Stack Position. Compile time error checks	User VAR 28
>832A	BASE	Forth number conversion RADIX	User VAR 2A
>832C	>IN	Forth Interpreter Pointer	User VAR 2C
>832E	C/L	TASK Characters per line variable	User VAR 2E
>8330	OUT	Counter for no. chars output since last CR	User VAR 30
>8332	VROW	Video COL	User VAR 32
>8334	VCOL	Video ROW	User VAR 34
>8336	'KEY	Vector to receive a char	User VAR 36
>8338	'EMIT	Vector to transmit a char	User VAR 38

Address	NAME or MNEMONIC	Use	DATA Type
>833A	LP	Leave stack pointer. DO LOOP support	User VAR 3A
>833C	SOURCE-ID	Interpreter source identifier 0=console, -1 EVALUATE, 1…8 File handle	
>833E	'SOURCE	Interpreted string address (>833E, 8340 are a 2VARIABLE)	User VAR 3E
>8340	---	Interpreted string length	CELL
>8342		Free user variable	User VAR 42
>8344		Free user variable	User VAR 44
>8346		Free user variable	User VAR 46
>8348		Free user variable	User VAR 48
>834A		Free user variable	User VAR 4A
>834C		Free user variable	User VAR 4C
>834E		Free user variable	User VAR 4E
>8350		Free user variable	User VAR 50
>8352		Free user variable	User VAR 50
>8354		(PROTECTED)	Byte
>8355	DSRSIZ	O/S Length of DSR name (PROTECTED)	Byte
>8356	DSRNAM	O/S Pointer to DSR name for LINK (PROTECTED)	CELL
>8358		(PROTECTED)	CELL
>835A		Free user variable	User VAR 5A
>835C		Free user variable	User VAR 5C
>835E		Free user variable	User VAR 5E
>8360		Free user variable	User VAR 60
>8362		Free user variable	User VAR 62
>8364		Free user variable	User VAR 64

"The second part of the scratch-pad memory is heavily used by the GPL interpreter and the console ROM routines."

Address	NAME or MNEMONIC	Use	DATA Type
>8366		Free user variable	User VAR 66
>8368		Free user variable	User VAR 68
>836A		Free user variable	User VAR 6A
>836C		Free user variable	User VAR 6C
>836E		Free user variable	User VAR 6E
>8370	VDPEND	Highest free VDP memory address	CELL
>8372	Dstack	GPL data stack pointer	Byte
>8373	Rstack	GPL return stack pointer	Byte
>8374	MODE	Keyboard scanning mode	Byte

>8375	KVAL	Variable read by KEY if a key press is detected by KEY?	Byte
>8376	JOYY	Joystick vertical value (4, 0, >FC)	Byte
>8377	JOYX	Joystick horizontal value (4, 0, >FC)	Byte
>8378	SEED	Random number, found after RND	Byte
>8379	TIMER	VDP interrupt timer	Byte
>837A			Byte
>837B	VDPSTS	Copy of VDP status byte	Byte
>837C	GPLSTS	GPL status byte	Byte
>837D			
>837E	VPG	Active VDP page address	CELL
		--- HI-SPEED CODE WORDS ---	
>8388	_EXIT	Forth EXIT code	CODE
>838A	_NEXT	Forth NEXT code	CODE
>8390	_?BRANCH	Forth conditional branch code	CODE
>8396	_BRANCH	Forth un-conditional branch code	CODE
>8393	_ENTER	Forth Enter (DOCOL) code	CODE
>83A6	_LIT	Forth DoLIT code	CODE
>83AE	_@	Forth Fetch code	CODE
>8EB2	_DROP	Forth DROP code	CODE
>83B8		FREE CELL	
>83BA		FREE CELL	
>83BC		FREE CELL	
>83BE	VPG	Forth VDP screen page variable. Defaults value is >0000	DATA
>83C0	RND#	Interrupt routine workspace (32 bytes) Random number seed	CELL
>83C2	AMSQ	ISR disabling flags: >80 All, >40 Motion, >20 Sound, >10 Quit key	CELL
>83C4	ISR	Interrupt Service Routine hook: routine to be executed	CELL
>83C6	KUNIT#	Default keyboard value. Stored from last KSCAN	BYTE
>83CC	SNDTAB	Address of the sound table	CELL
>83CE	SNDSIZ	Sound bytes to play (>0100)	BYTE?
>83D0		Used by console DSR	CELL
>83D2		Used by console DSR	CELL
>83D4	VDPR1	Copy of VDP register 1, used by KSCAN	BYTE
>83D6	CLRSC	Screen timeout counter: INCREMENTED by 2, clears when 0	CELL
>83D8	.	Return address saved by SCAN.	
>83DA to >83DF	.	Used for RTWP (workspace, pc and status).	
>83E0	GPLWS R0	GPL interpreter workspace	CPU Register
>83FA	GBASE	GPL R13, GROM port in use (normally >9800)	CPU Register

>83FC	SPEED GPL R14 (byte)	Speed value, added to TIMER	BYTE
>83FD	FLAGS GPL R14 (byte)	>20 cassette operations, >10 cassette verify, >08 16K VDP mem >02 multicolor mode, >01 sound table in VDP mem	BYTE
>83FE	VDPWA GPL R15	VDP write address port (>8C02).	CPU Register

Interpreter Internals

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: a space delimited word from input
  Is it in the dictionary of WORDS?
  IFYES:
        ARE WE COMPILING?
         IFYES: compile the command
         ELSE: EXECUTE the command
        ENDIF:
  ELSE:
        IS IT A NUMBER?
        IFYES:
                Are we compiling?
                IFYES: compile the number as a literal into program
                ELSE: Put the number onto the stack
                ENDIF:
        ELSE: Don't know what this is.
               Print error message, reset stacks
        ENDIF:

AGAIN: (goto begin)
```

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

Essential Elements of Forth

The Forth language was create by Charles Moore, 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 to write and debug, 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 in software that suited his needs.

Here is the list of things Chuck needed in a computer:
1. Memory (RAM)
2. Disk storage (originally 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 experience with Java. The Java virtual machine is a single stack computer architecture but it is a really a program. It allows the Java language to be portable to many different hardware platforms, because all you need to do is re-write the Java VM for the new hardware and Java programs can run on it.

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

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

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

Forth Virtual Machine Illustrations

Hardware View

```
┌─────────────┐         ┌─────────────┐         ┌─────────────┐
│  Parameter  │ <────>  │             │ <────>  │   Return    │
│   Stack     │         │     CPU     │         │   Stack     │
│             │         │             │         │             │
└─────────────┘         └─────────────┘         └─────────────┘
                              ↕
                        ┌─────────────┐
                        │             │
                        │     RAM     │
                        │             │
                        └─────────────┘
```
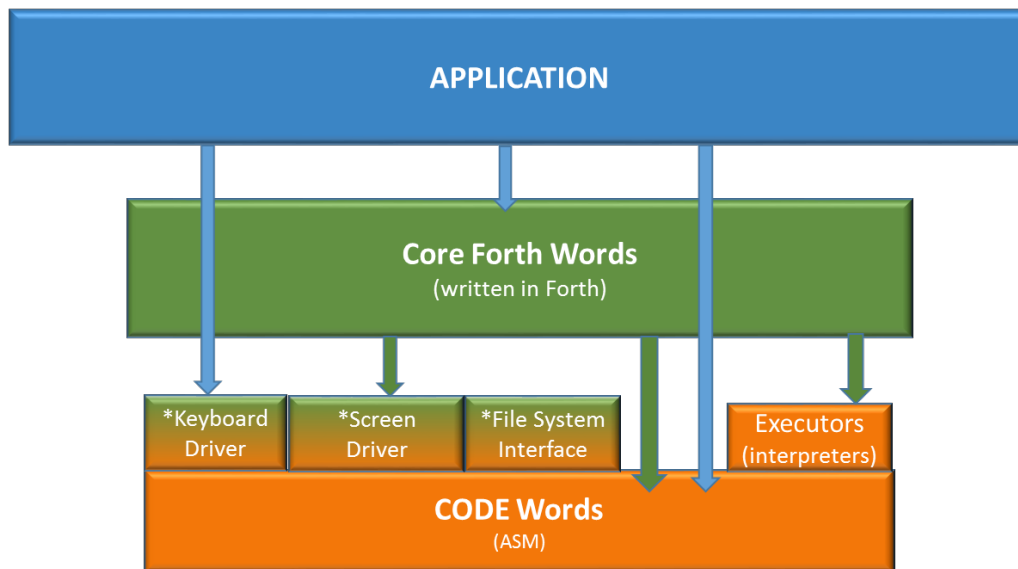
Software Layers

```
┌──────────────────────────────────────────────────────────────┐
│                        APPLICATION                             │
└──────────────────────────────────────────────────────────────┘
         │                                    │
         ▼                                    ▼
┌──────────────────────────────────────────────────────────────┐
│                    Core Forth Words                            │
│                    (written in Forth)                          │
└──────────────────────────────────────────────────────────────┘
     │          │          │              │          │
     ▼          ▼          ▼              ▼          ▼
┌─────────┬──────────┬───────────┐   ┌──────────────┐
│*Keyboard│ *Screen  │*File System│  │   Executors  │
│  Driver │  Driver  │ Interface  │  │ (interpreters)│
└─────────┴──────────┴───────────┘  └──────────────┘
┌──────────────────────────────────────────────────────────────┐
│                      CODE Words                                │
│                        (ASM)                                   │
└──────────────────────────────────────────────────────────────┘
```
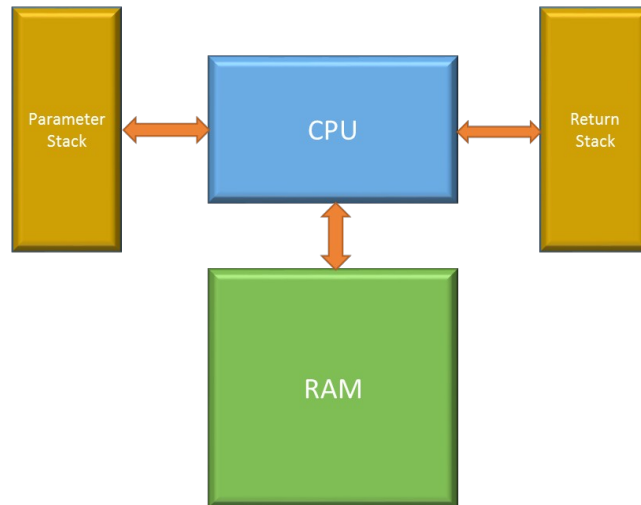
* Written in ASM or Forth or both

- 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.
    - (Constants, variables, colon words, CREATE DOES>)

Program Development Comparisons

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'

-

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 executable 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 new disk file into the Forth system
3. Test this small part 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 files 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 a 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. Put it on the data stack
```

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 [18]'+!" (Pronounced "plus-store") Plus-store takes a number and a variable as inputs and adds the number to contents of the address. 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:

---

[18] Plus-store is very much like '+=' for those familiar with 'C'.

```
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 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  ;  (Forth 2012 Standard word)
```

Now it's all done with one word!

```
50 BUFFER: ABUFFER
```

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 as comma.

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

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
  - (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
```

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 the thing to remember is 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.

Understanding "MORE" Step by Step

The MORE utility in the system is written using the low-level READ-LINE function. We show it here as an example of how to use READ-LINE and also how to keep the file handle on the Return stack for multiple uses. Below is the source code for the MORE utility with line numbers and a detailed explanation below.
NOTE:

- We set the number base to DECIMAL first so our record size of 80 will be correct.

Program listing for MORE

```
DECIMAL
NEEDS READ-LINE FROM DSK1.ANSFILES

\ : DV80   DISPLAY INPUT  80 VARI SEQUENTIAL ; ( defined in DSK1.ANSFILES)

DV80  ( set file access mode to DV80 right now)
: MORE    ( <filename>)
   1. BL PARSE-WORD DUP ?FILE ( -- $addr len)
   2. R/O OPEN-FILE ?FILERR
   3. >R
   4. BEGIN
        a. PAD DUP 80 R@ READ-LINE ?FILERR  ( -- PAD #bytes flag)
   5. WHILE
        a. CR TYPE
        b. KEY?
        c. IF CR ." ..."
        d. KEY 0F =            \ test for escape key
        e. IF R> CLOSE-FILE    \ if detected we're done here
            i.   2DROP
           ii.   CR CR ." >>Esc<<" ABORT
          iii.   THEN
        f. THEN
   6. REPEAT
   7. R> CLOSE-FILE
   8. 2DROP DROP ;
```

First we select DV80 files. This is the default Text file format of the TI-99. The new Forth word is called DV80. We invoke it right away to set the default file access mode for MORE.

1. Next we use PARSE-WORD to get a file name string. PARSE-WORD reads Forth's input stream, normally from the keyboard, skips all leading spaces and collects the characters up to the delimiter character which in this case is BL (blank/also called space) and returns a "stack string". A stack string consists of an address and a length sitting on the data stack. We DUP the length and test if the filename is empty with ?FILE. After ?FILE runs, there is a "stack string" left behind on the DATA stack.
2. We have already set the file access details with the word DV80 so all we need to do is use the word R/O, which means READ/ONLY, to put the file mode on the stack. This puts all three needed parameters on the stack for OPEN-FILE. OPEN-FILE calls the Device Service Routine in the Disk Controller Card. Then OPEN-FILE returns a file "handle" and an error code to the DATA stack.   The error-code is picked up by ?FILERR which prints an error message and the error code if there is a problem.

3. The file handle is simply a number that identifies this open file. We need it to do anything to this file so we push it onto the return stack so we can get it anytime we need it.

4. 4. Now we start a loop

a. We perform a READ-LINE of 80 bytes into the temporary memory space that Forth calls PAD. We made a [19]copy of PAD with DUP because we are going to need it later. Notice that we get a copy of that file handle we saved with R@ and feed it to READ-LINE. This is how READ-LINE knows which file to read. We test if there is an error with ?FILERR. READ-LINE also leaves a TRUE/FALSE flag on the stack and the number of bytes that it read up to the maximum we specified. (in this case 80 bytes is the max)

5. "WHILE" tests the flag on the stack from READ-LINE. WHILE the flag is true, we do a carriage return and use TYPE to print the address and length that are on the stack from READ-LINE
   a. Next we test if a key was pressed with KEY?
   b. IF a key was pressed we go to a new line and print "…".
   c. Then we wait for a KEY
   d. If the KEY is <esc>
   e. Pull the file handle off the return stack and do CLOSE-FILE.
      i. Drop both outputs from CLOSE-FILE ,
      ii. Drop down 2 lines and print >>ESC<<< and ABORT to the Forth console.
      iii. THEN continue the program (ends IF statement)
   f. THEN continue the program (ends IF statement)

6. If a key is not press we REPEAT the loop starting at BEGIN.

7. When READ-LINE gives us a false flag we jump to REPEAT, pull the handle and close the file.

8. Then clean up the three items left on the DATA stack.

---

[19] That's the Forth way to do this rather than calling PAD again later, because PAD is slower that DUP