Irreducible Complexity

eForth for Discovery

Chen-Hanson Ting

1. eForth for ARM chips

1.1 Moore's Law Marches On

Moore's Law marches on, and more and more circuits are crowded into microcontrollers. In the last 15 years, I had programmed many ARM chips, and had watched with amazement the progress of the ARM chips. My approach had always been to port an eForth system onto the chips and tried to make the best use of the chips. Here are some of the ARM chips I put eForth on.

2001: Nintendo's GameBoyAdvance had an ARM7TDMI chip in it. It had 32 KB of RAM. No flash. It had lots of external flash and RAM for games.

2004: ADuC7024 from Analog Devices had 62 KB of flash and 8 KB of RAM, and lots of IO devices, including ADC and DAC. I built a ForthStamp based on it, a really nice single chip stamp size computer.

2008: AT91SAM7x256 from Atmel. It had 64 KB of flash and 16 KB of RAM, and lots of IO devices.

A couple of years ago, I told my friends in the Silicon Valley FIG and Taiwan FIG that I had to really retire from Forth programming. I did, and worked peacefully on translating Bach's cantatas from German to Chinese, and putting Tang poems into Schubert's songs, and many other things I had neglected all the years. Then, last month, a friend in Taiwan FIG sent me this ForthDuino Board, which was used to control a laser cutting machine to make PC boards. It had footprints of IO sockets of Arduino board and MSP430 LaunchPad. It is intended to suck in all applications from Arduino and LaunchPad. I was told that the ARM chip on ForthDuino is the same one used in the STM32F4-Discovery Kit. Looking up the STM32F407 chip, I was shocked to see so much memory, and so many IO devices. 1 MB of flash and 192 KB of RAM. It is a Wow chip, and in desperate need of a good eForth system.

So. I re-open my workbench, unpacked my tools, download all necessary IDE and programming toolchains. But, the world has changed since I stopped watching. Keil is still there, but its toolchain became uVision5. STM32F4 is no longer an ARM chip. It is a Cortex M4 chip. There is no ARM in STM32F4. All that's left is a THUMB, and a really big THUMB.

The first shock was that I could not use the ARM directive in the assembler. The assembler generated lots of error messages if you do ARM. It is much happier if you use the THUMB directive. Then, the RSC instruction disappeared. Reading the ARM assembler manual carefully, I found that ARM Holdings is phasing out the ARM instruction set, and replacing it with the THUMB2 instruction set. It gave up the beautiful RISC architecture, and reverted to the ugly CISC architecture we all despised.

I missed the simple serial COM port in PC. The USB is so much harder to deal with. You don't know what's going on. You must have faith on the USB drivers given to you.

There is no simple examples to guide me, to start my exploration. The Demo project provided with STAM32F4-Discovery Kit is a huge package with 7 folders and 31 files. There is no clear entry point. I spent 3 weeks wandering around in the hardware and software maze, looking for an entry point. The great breakthrough came when I realized that I only had to set up the reset vector correctly, everything would work smoothly from that point on. Throw away all the header files, init files, device driver files. I only need one assembly file to do what I have to do.

Since STM32F4 is no longer an ARM7 chip. It is not necessary to keep the name in my eForth implementations. I planned and completed 3 versions of eForth for this chip:

STM32eforth v7.01 The eForth dictionary resides in flash memory, and executes from flash memory. It is upgraded to align with the eForth2 model, with subroutine tread model and fully optimized for performance.

STM32eforth v7.10 The eForth dictionary resides in flash memory. Flash memory is remapped to the virtual memory in Page 0. eForth executes from Page 0 memory.

STM32eforth v7.20 The eForth dictionary resides in flash memory. The dictionary is copied from flash to RAM. RAM memory is remapped to the virtual memory in Page 0. eForth executes from Page 0 memory. Applications can be easily embedded in turnkey system.

1.2 THUMB2—Death of a RISC

The ARM architecture was hailed as the prince of RISC, as the name says it all: Acorn RISC Machine. The major disadvantage of RISC is its poor coding density. A 32-bit instruction does not do much work. Lots of bits in the instruction, like the 4-bit condition field, are wasted. ARM Holdings tried very hard implementing the THUMB instruction set to complement the ARM instruction set. In the end, the THUMB is waggling the ARM, and the THUMB2 instruction set basically eased out the ARM instruction set. THUMB2 is clearly a CISC architecture. Cortex M4 core inside STM32F407 is an extremely complicated instruction set computer. Intel had proved that the RISC architecture is of no special value, and ARM Holdings concurred.

1.3 Dire Consequence of Moore's Law

A while ago, I was amazed at the 566 page reference manual of ATmega328 from Atmel, which is a lowly 8-bit microcontroller used on Arduino Uno Kit. The reference manual of STM32F407 is 1713 pages thick. How can anybody wading through this document to get a handle on this chip and all its peripheral devices?

I opened the Demo project for the STM32F407-Discovery Board on Keil's uVision5. In the Project panel I counted 7 folders with 31 files in them. Just for a Demo! It is true that the Demo does a lot of interesting things, like reading the 3-axis accelerometer and the USB connection to PC. I have great sympathy for people who gets this kit and is confronted by this huge software mess.

The dire consequence of the Moore’s Law is complexity beyond comprehension.

The only way to deal with this complexity is the Forth way. Or, put it more bluntly:

KISS Keep It Simple, Stupid!

The first thing to do is to put eForth on board. The 16 MHz high speed internal clock HSI in the chip is good enough for an USART. Forget about the fancy PLL that can push the clock to 168 MHz. We can deal with it when we really need the speed. Just get the USART going, and we can walk into the guts of the microcontroller and actually control it from inside through eForth.

What about interrupts, threads, heaps, multitasking, and preemptive task switching? All the great things this ARM/THUMB chip can do? Forget them! You will learn them in the senior year of computer engineering, if you have time to go to school. All these things can be added to eForth when you really need them.

eForth exposes all the memory and the IO registers to you. You can inspect them, and you can tinker with them. This is the way to study the peripheral devices, learn how to control them, and make use of them. Focus on one device you will use. Read that chapter in the reference manual. Inspect the status and control registers. Flip bits in the control register and see what happens. Write short commands to perform the functions you want. These functions will be called from you eventual applications.

1.4 Oddity of Thumb Transfer Instructions

In the transfer instructions, THUMB2 requires that all the target addresses must be odd. Bit 0 of the address is deposit into the T bit in the EPSR status register, indicating that it is in the Thumb state. The actual address is on the 2 byte half word boundary. Much grief was encountered when I was debugging eForth, which aligned addresses to the 4 byte word boundary. If bit 0 is not set in these addresses, the CPU may work correctly, and it may also crash. I learnt the lesson when building turnkey systems. The correct procedure is:

```
; load Lesson16.txt
; load Lesson17.txt

0 ERASE_SECTOR
‘ GUESS 1+ ‘BOOT !
TURNKEY
```

In most cases, eForth takes care of this oddity. But, when you are use addresses explicitly, make sure bit 0 in the target address is set before jumping to it.

(Note: I think this must be considered a bug in stm32eforth720. I fixed it in `@EXECUTE`. Before `@EXECUTE` jumps to the target address, it sets b0 in the address to please the M4 CPU).

1.5 eForth1 and eForth2

The original eForth Model was designed by Bill Muench in 1990. It was based on the Direct Thread Forth Model, in which the body of a high level Forth command contained a list of execution addresses, preceded by a `CALL NEST` machine code. Bill was very ambitious in that he laid down hooks for multitasking and the `CATCH-THROW` mechanisms for error handling. List of execution addresses were very easy for porting to other processors. Indeed, many people did port this model to about 30 different processors. At that time, assemblers for these processors were not easily available and very different. This simple model was very easy to be adapted to a particular assembler. In a few cases, MASM from Microsoft was used to assemble eForth for a different processor.

Getting into this century, I ported eForth to many microcontrollers at work. With good native assemblers available then, I was able to optimize eForth for performance necessary in actual products. I used the Subroutine Thread Model throughout, and realized many other advantages besides speed. Machine code can be mixed with subroutine calls. Interrupt service routines can be written in high level Forth. In all these applications, multitasking was not necessary and many user variables can be eliminated. The `CATCH-THROW` mechanism was not needed, and the error handling was greatly simplified. The cumulative result was eForth2, and earlier implementations were classified as eForth1.

eForth2 implementations were all written using native assemblers provided by microcontroller manufacturers. Forth commands which can be expressed in native machine instructions are so coded.

eForth1 is for portability.

eForth2 is for performance.

1.6 THUMB2 Instruction Set

When I started porting eForth to STM32F407-Discovery Board, I was not aware of the THUMB2 instruction set. I used sam7ef.s from the AT91SAM7x256 project and tried to assemble it under uVision5. Lots and lots of error messages. Totally confusing. The first thing I noticed was that the startup_stm32f4_xxx.s file used the THUMB directive, and I used ARM directive in sam7ef.s. Changing the THUMB directive to ASM caused more errors. Changing the ARM directive to THUMB, the assembler was much happier, but still threw lots of errors and warning at me.

Then I found that ARM Holdings changed the CPU core behind my back when I was not watching. Their chips are not ARM chips any more. They are Cortex-M4 chips with only Thumb2 instructions. Their assembler was also changed to UAL, as stated in one of its manuals:

Unified Assembler Language (UAL) is a common syntax for ARM and Thumb instructions. It supersedes earlier versions of both the ARM and Thumb assembler languages.

These are errors which I had to correct:

B<addr> becomes a 2-byte Thumb instruction. It causes following instructions to be misaligned. It has to be changed to B.W<addr>, to retain 4 byte word alignment.

Target address in Forth transfer commands must have bit 0 set.

RSC (Reverse subtract with carry) does not exist. It was used only in DENEGATE.

I had made many mistakes on my own. I had to upgrade eForth1 to eForth2, though most changes were deleting things I did not need. During the process, I really appreciated the debugger in the Keil uVision5. It allowed me to set up to 6 hardware break points freely. Watching CPU registers and IO registers in any device while single stepping the assembly code was very helpful. In the end, I was very pleased to see stm32eForth signed on and processed my command correctly.

1.7 Branch and Link

In Cortex M4, subroutine call uses the Branch and Link `BL<addr>` instruction. All high level Forth commands were assembled as tokens of BL instructions. BL instruction, as invented in the RISC architecture, assumed a return stack of 1 level, which is the link register LR. If the called subroutine had to call other subroutines, the return address in LR has to be saved on a real return stack of adequate depth

I watched the disassembled BL instructions while single stepping through the code, but could not figure out how the instructions were encoded. Only when I was testing the decompiler command SEE, I had to figure it out without a shiver of doubt. It is composed of two 16-bit THUMB instructions in the form of:

11111	Address Bits 22-11	11110	Address Bits 11-1
Byte 1	Byte 0	Byte 3	Byte 2

Very strange, indeed! But, I was able to shift the bits around and eventually get the correct address back.

1.8 First ARM Assembly Program

I was desperate to get STM32F4-Discovery to do something. I Googled ARM and Discovery tutorials, and with lots of patience I found this simple example from the website of Regina University. It contains the least amount of code to get STM32F407 to increment a register. It assembled correctly on uVision5, and the debugger lets me single step through the program. I could see that register R0 was actually incrementing. Now, I got the toolchain working.

```
; First ARM Assembly program
; Chen-Hanson Ting, 16jun14cht
; Adapted from a lab lesson at Regina University
; http://www.cs.uregina.ca/Links/class-info/301/ARM/lecture.html
; for ATM32F-Discovery kit.
; Assembled on uVision 5.10 from Keil
; Use the uVision debugger to watch the registers in ATM32F407
; First step to get used to Discovery kit and uVision
```

```
;;; Directives
PRESERVE8
THUMB
; Vector Table Mapped to Address 0 at Reset
; Linker requires __Vectors to be exported
      AREA    RESET, DATA, READONLY
      EXPORT  __Vectors
__Vectors
      DCD  0x20001000     ; stack pointer value when stack is empty
      DCD  Reset_Handler  ; reset vector
      ALIGN
; The program
; Linker requires Reset_Handler
      AREA    MYCODE, CODE, READONLY
      ENTRY
      EXPORT Reset_Handler
Reset_Handler
      MOV R0, #12
      STOP
      ADD R0, R0, #4
      B  STOP
      END   ;End of the program
```

1.9 Blinky

The next step was to get the LEDs blinking. Discovery has a Blinky demo, but it is huge. It was no fun to read the C code, all the header files and library files. How many programmers does it take to turn on a LED? Once I got the above kernel going, it was easy to add and few lines of code to turn the LEDs on and off.

The LEDs on Discovery are connected to pins PD12-15. These 4 pins on GPIOD ports must be initialized to be output pins. All IO devices require clocking, which is done through the Reset Clock Control register RCC. The program is simply:

```
; SimpleBlinky, Chen-Hanson Ting, 18jun14cht
; Simplest program to blink the LEDs on the STM32F4-Discovery kit.
; Assembled by uVision 5.10 from Keil.
; Adapted from Daniel Widyanto
; http://embeddedfreak.wordpress.com/2009/08/09/cortex-m3-blinky-in-assembly/

;;; Directives
      PRESERVE8
      THUMB
; Vector Table Mapped to Address 0 at Reset
; Linker requires __Vectors to be exported
      AREA    RESET, DATA, READONLY
      EXPORT  __Vectors
__Vectors
      DCD  0x20001000     ; stack pointer value when stack is empty
      DCD  Reset_Handler  ; reset vector
      ALIGN

; The program
```

```
; Linker requires Reset_Handler
      AREA    MYCODE, CODE, READONLY
      ENTRY
      EXPORT Reset_Handler
Reset_Handler
; Blinky program for STM32F407 - ARM Cortex-M43
; The LEDs are at these pins:
; LD3, orange, PD13
; LD4, green,  PD12
; LD3, red,    PD14
; LD3, blue,   PD15

; Declare __main() as global..Otherwise the linker won't find it
      EXPORT __main
__main
; /* Set the pins direction as output */
      LDR     R0, =set_gpio_dir
      BLX     R0
loop
      LDR     R0, =clear_leds
      BLX     R0
      LDR     R0, =delay
      BLX     R0
      LDR     R0, =set_leds
      BLX     R0
      LDR     R0, =delay
      BLX     R0
      B       loop

set_gpio_dir
; Enable clock to GPIOD
      ldr     r0, =0x40023800 ; RCC
      ldr     r1, [r0, #0x30] ; RCC_AHB1ENR
      orr     r1, #8          ; GPIODEN (1)
      str     r1, [r0, #0x30]
; Configure PD12-15 as output with push-pull
      ldr     r0, =0x40020C00 ; GPIOD
      ldr     r1, [r0, #0x00] ; GPIOx_MODER
      bic     r1, #0xFF000000 ; Mask PD12-15
      orr     r1, #0x55000000 ; output
      str     r1, [r0, #0x00]
      BX      LR
set_leds
; Set PD12-15
      ldr     r0, =0x40020C00 ; GPIOD
      ldr     r1, [r0, #0x14] ; GPIOD_ODR
      orr     r1, #0xF000 ; set PD12-15
      str     r1, [r0, #0x14]
      BX      LR
clear_leds
; Clear PD12-15
      ldr     r0, =0x40020C00 ; GPIOD
      ldr     r1, [r0, #0x14] ; GPIOD_ODR
      bic     r1, #0xF000 ;  PDclear12-15
      str     r1, [r0, #0x14]
      BX      LR
delay
```

```
; Delay about 0.3 second, with internal HSI clock at 16 MHz
      MOVW    R3, #0x0000
      MOVT    R3, #0x0004
__delay_loop
      CBZ     R3, __delay_exit
      SUB     R3, R3, #1
      B       __delay_loop
__delay_exit
      BX      LR
      ALIGN
      END
```

1.10 Hello World

eForth needs a USART to communicate with the user. I found a nice Hello World example using USART1 to send out a message:

```
; Hello World!
; Adapted from an assembly example by clive1 on STM32 Forum on www.st.com
; Chen-Hanson Ting 16jun14cht

; This is a demo program for STM32F4-Discovery Kit from STMicroelectronics.

; The STM32F407 chip is overwhelming.  The demo program Blinky provided by ST
;   is also overwhelming.  There must be a better way to get it working.
;   I am porting my Sam7eForth system on this platform.  This is another
;   step towards this goal.

; It uses USART1 port on PB6/7 to send out the "Hello World!" message.
; USART1 is configured at 115200 baud, 1 start bit, 8 data bits, 1 stop bit,
;   no parity, no flow control
; USART1 is an alternate function of the GPIOB port, pins PB6/7.
; We have to initialize the clock control register CCR, GPIOB port,
;   and USART1 port.

; Code is assembled by uVision 5.10 from Keil.  Object code is downloaded to
;   Discovery through on-board ST-Link, and debugged through uVision.

; An Arduino Uno board is used as the USART COM port.  Remove Atmega328P chip
;   from Uno board.  Connect its RX at D0 to PB7 on Discovery board, and the
;   TX at D1 to PB6 on Discovery board.  Ground together Uno and Discovery.
;   Discovery sends characters from its USART1 to Uno, and to HyperTerminal
;   on its PC host.

      AREA  RESET, CODE, READONLY
      THUMB
      EXPORT      __Vectors         ; linker needs it
      EXPORT      Reset_Handler     ; linker needs it

; Vector Table has only Reset Vector
__Vectors
      DCD   0x10000400  ; Top of hardware stack in CCM
      DCD   Reset_Handler           ; Reset Handler
      ENTRY
```

```
Reset_Handler
      BL      InitUSART1
      LDR     R0, =Hello
      BL      _OutString
      B     .
Hello DCB     "\n\015Hello World!\n\015", 0
      ALIGN

;*****************************************************************************
; Assumes system running from 16 MHz, HSI (Normal at Reset)
; USART1 PA9 TX, PA10 RX; this does not work.  Output spaces and a $.
; Try alternate USART1 PB6 TX and PB7 RX; this works.

InitUSART1      PROC
; init Reset Clock Control RCC registers
      ldr     r0, =0x40023800 ; RCC
      ldr     r1, [r0, #0x30] ; RCC_AHB1ENR
      orr     r1, #2                ; GPIOBEN (1<<1)
      str     r1, [r0, #0x30]
      ldr     r1, [r0, #0x44] ; RCC_APB2ENR
      orr     r1, #0x10       ; USART1EN (1 << 4)
      str     r1, [r0, #0x44]
; init GPIOB
      ldr     r0, =0x40020400 ; GPIOB
      ldr     r1, [r0, #0x00] ; GPIOx_MODER
      bic     r1, #0xF000     ; Mask PB6/7
      orr     r1, #0xA000     ; =AF Mode
      str     r1, [r0, #0x00]
      ldr     r1, [r0, #0x20] ; GPIOx_AFRL
      bic     r1, #0xFF000000 ; Mask PB6/7
      orr     r1, #0x77000000 ; =AF7 USART1
      str     r1, [r0, #0x20]
; init UART1
      ldr     r0, =0x40011000 ; USART1
      movw    r1, #0x0200C    ; enable USART
      strh    r1, [r0, #12]   ; +12 USART_CR1 = 0x2000
      movs    r1, #139        ; 16MHz/8.6875 (139, 0x8B) == 115200
      strh    r1, [r0, #8]    ;  +8 USART_BR
      ldr     r2, =12               ; Output 12 pound/hash symbols
iu1
      ldrh    r1, [r0, #0]    ; USART->SR
      ands    r1, #0x80       ; TXE
      beq     iu1
      mov     r1, #'#'
      strh    r1, [r0, #4]    ; USART->DR
      subs.w  r2, r2, #1      ; $1
      bne.n   iu1
      bx      lr
      ENDP ; InitUSART1

;*****************************************************************************
; Uses
;  r0 Character to output, masked
;  r1 scratch, destroyed
;  r2 scratch, destroyed
```

```
_OutChar    PROC
      ldr     r2, =0x40011000 ; USART1 F2/F4
      and     r0, #0xFF
_OutChar10
      ldrh    r1, [r2, #0]    ; USART->SR
      ands    r1, #0x80       ; TXE
      beq     _OutChar10
      strh    r0, [r2, #4]    ; USART->DR
      bx      lr
      ENDP                          ; _OutChar

;****************************************************************************
; Uses
;  r0 String to output, destroyed
;  r1,r2,r3 assumed scratch

_OutString      PROC
      push    {r4, lr}
      mov     r4, r0
_OutString10
      ldrb.w  r0, [r4], #1 ; r0 = *r4++ (BYTE)
      orrs    r0, r0
      beq     _OutString20
      bl      _OutChar
      b       _OutString10
_OutString20
      pop     {r4, pc}
      ENDP ; _OutString
      ALIGN
      END
```

1.11 HyperTerminal Setup

Stm32eforth720 uses USART1 to communication with a terminal. On STM32F407VG, USART1 can be configured to use either Pins PA9-10 or PB6-7 for communication. Since the USB on CN5 is using PA9-10 ports, I use PB6-7 for eForth. I am using a separate PC to run HyperTerminal through a USB to serial converter, which happens to be an Arduino Uno Kit. Arduino Uno Kit has a integrated USB to serial converter connecting the STmega328P chip to the host PC. It uses this USB to download programs and to communication with the 328 chip. To use its USB to serial converter, I remove the ATmega328P chip, and connect the PB6 (TX) on Discovery to D1 port on Arduino, and the PB7 (RX) on Discovery to D0 port on Arduino. A ground wire connects the ground pins on both boards.

HyperTerminal on PC is configured at 115200 baud, 1 start bit, 8 data bits, 1 stop bit, no parity, no flow control. The USART1 on STM32F407 is configured similarly. STM32F407 is clocked by its high speed internal clock HSI at 16 MHz on reset. Since this HSI is factory trimmed to 1% accuracy, it is adequate to provide reliable communication on USART1.

With the HelloWorld demo displaying “Hello, World!” on HyperTerminal, I regained my self-confidence, and proceeded to port eForth over. I used to boast that I could port eForth to a new microcontroller in 2 weeks. This time it took 5 weeks to get it working on Discovery. Am I getting too old? Or, is the world passing me by too fast?

1.12 Irreducible Complexity

STM32F407 is a very complicated chip. If you are going to program in C, the software package you are given is extremely complicated. What I am trying to do here is to reduce the complexity to the minimum, and help you to control this chip with the least amount of code.

As Lao Tze said in Tao Te Ching, Chapter 48:

> *For knowledge, add a bit a day.*
> *For wisdom, delete a bit a day.*
> *Delete until there is nothing.*
> *Then, everything can be done.*

I use only one USART device.
I use only a reset vector to get the chip starting executing code.
A Virtual Forth Machine simplifies the complicated CPU.
The complier is wrapped inside the text interpreter.
The parameter stack simplifies language syntax and list processing.
Forth is the simplest LISP processor.

I think Albert Einstein said better: “Everything should be made as simple as possible, but not simpler”.

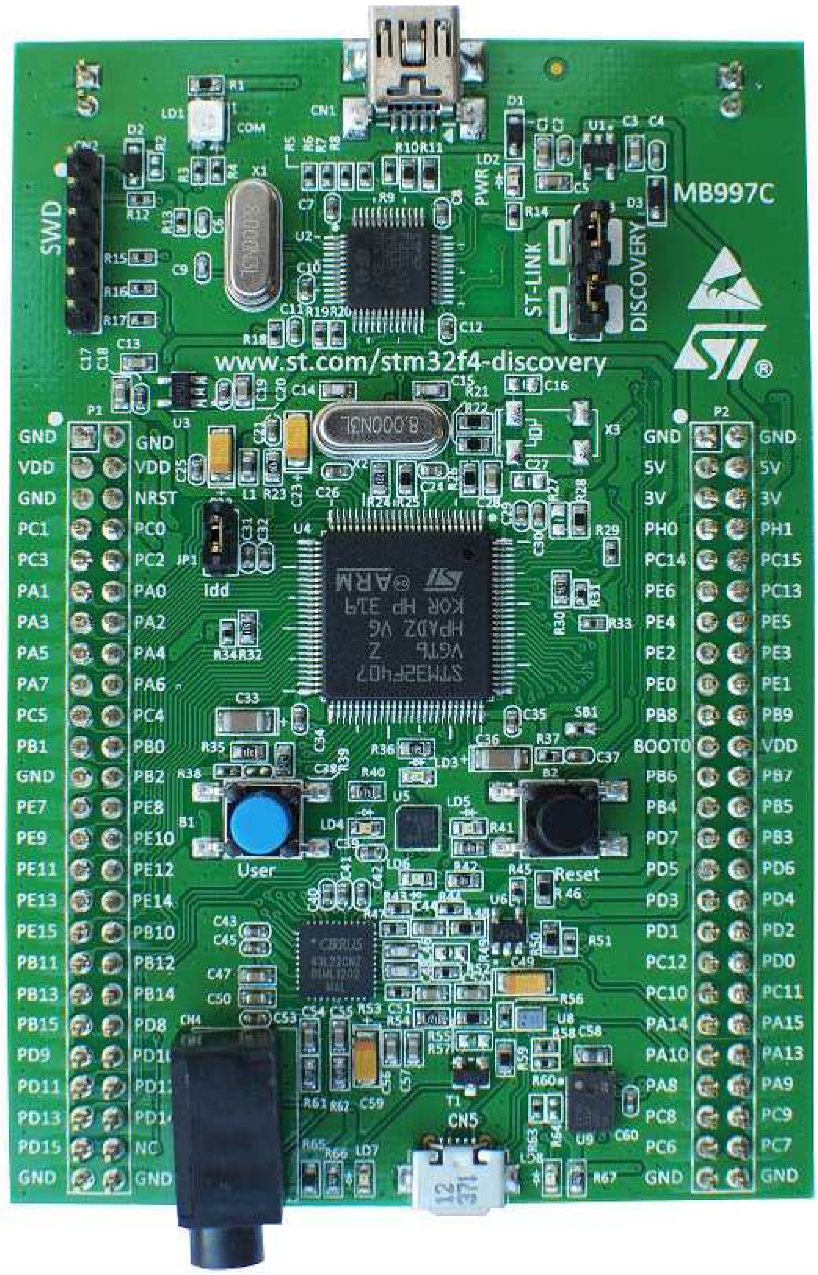
stm32eforth720 is assembled to a 8492 byte image. It seeks and. I believe, has achieved irreducible complexity. Things cannot be made any simpler.

eForth leaves the least footprint in your mind. With this understanding, you can make the STM32F407 microcontroller do what you want it to do.

2. Assemble and Test STM32eForth720.s

2.1 STM32F4-Discovery Kit

To promote the commercial adoption of STM32F4 chips, STMicroelectronics provides a low-cost STM32F4-Discovery Kit. I got my first kit for $14.90 from DigiKey. The second time I placed an order, the price jumped to $20. But, it is still very cheap for its capabilities. STMicroelectronics is spending lots of money promoting these microcontrollers. It is based on an STM32F407VGT6 and includes an ST-LINK/V2 embedded debug tool interface, ST MEMS digital accelerometer, ST MEMS digital microphone, audio DAC with integrated class D speaker driver, LEDs, pushbuttons and a USB OTG micro-AB connector.



Here is a laundry list of features in STM32F4-Discovery Kit:

- STM32F407VGT6 microcontroller featuring 1 MB of Flash memory, 192 KB of RAM
- On-board ST-LINK/V2 with selection mode switch to use the kit as a standalone debugger
- Power through USB bus or from an external 5V supply voltage
- External application power supply: 3V and 5V
- LIS302DL or LIS3DSH, ST MEMS 3-axis digital accelerometer
- MP45DT02, ST MEMS audio sensor, omni-directional digital microphone
- CS43L22, audio DAC with integrated class D speaker driver
- Eight LEDs for power, accelerometer, and micro USB
- 8 LEDs for power, accelerometer, and micro USB
- Two pushbuttons (user and reset)
- USB OTG with micro-AB connector
- Extension headers for 80 IO pins
- Keil µVision5 Integrated Development Environment
- Binary code downloader through serial ports

STM32F407 microcontroller on Discovery is a very interesting and capable chip from STMicroelectronics. It integrates an 32-bit Cortex M4 core with lots of digital and analog peripheral devices. They greatly simplify control and monitoring in applications such as factory automation, network communication, and perhaps automotive control. Following is a laundry list of features in this chip:

- ARM 32-bit Cortex™-M4 CPU Core with FPU, frequency up to 168 MHz, 210 DMIPS/1.25 DMIPS/MHz (Dhrystone 2.1), and floating point and DSP instructions
- 1 MB of Flash memory, 192 KB of SRAM including 64-KB of CCM (core coupled memory) data RAM
- LCD parallel interface, 8080/6800 modes
- 3×12-bit, 2.4 MSPS A/D converters: up to 24 channels and 7.2 MSPS in triple interleaved mode
- 2×12-bit D/A converters
- 16-stream DMA controller with FIFOs and burst support
- Twelve 16-bit and two 32-bit timers up to 168 MHz, IC/OC/PWM or pulse counter and quadrature (incremental) encoder input
- Serial wire debug (SWD) & JTAG interfaces
- Up to 140 I/O ports with interrupt capability, fast I/Os up to 84 MHz, 5 V-tolerant
- 15 communication interfaces, 3 × I2C interfaces, 6 USARTs/2 UARTs, 3 SPIs, 2 × CAN interfaces, USB 2.0 full-speed device/host/OTG controller, USB 2.0 high-speed/full-speed device/host/OTG controller
- 10/100 Ethernet MAC with dedicated DMA:
- 8- to 14-bit parallel camera interface up to 54 Mbytes/s
- True random number generator
- CRC calculation unit
- 96-bit unique ID
- RTC: subsecond accuracy, hardware calendar

2.2 IDE and Assembler

STMicroelectronics wisely focuses on the chip manufacturing, and delegates software tools companies to provide assemblers and compilers to program its chips. In the STM32F4-Discovery User Manual, 4 software development toolchains are recommended:

- Embedded Workbench® for ARM (EWARM) by IAR
- Microcontroller Development Kit for ARM (MDK-ARM) by Keil
- TrueSTUDIO® by Atollic
- TASKING VX-toolset for ARM Cortex by Altium

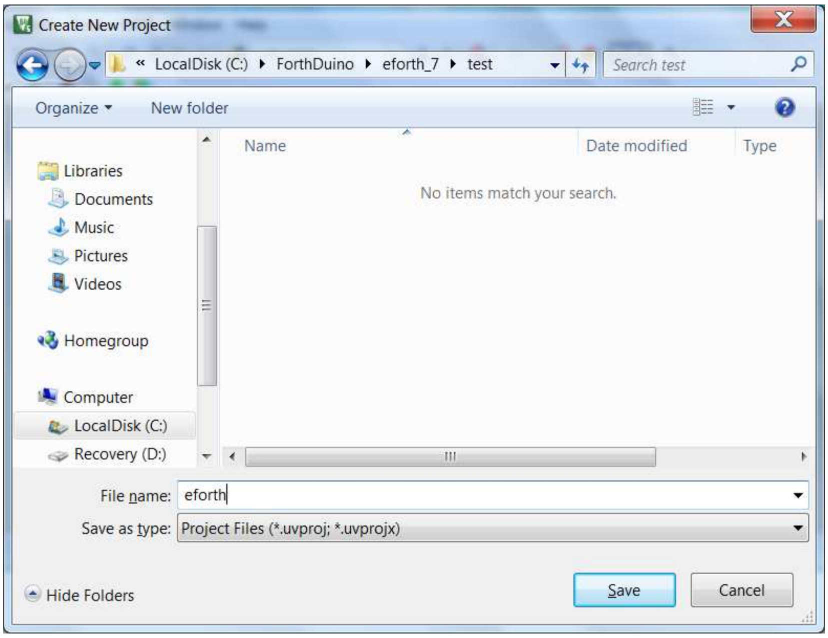
I have been using Keil's MDK-ARM Development Kit for years, and used it again for this project of eForth on Discovery. You can download a free evaluation version form its website www.keil.com. The current release is µVision5.10. The evaluation version has a size limit of 32 KB target code. This size poses no problem for eForth systems, which usually assembles to about 8 KB.

uVison5 uses the standard armasm assembler from ARM Holdings. It is now using UAL syntax.

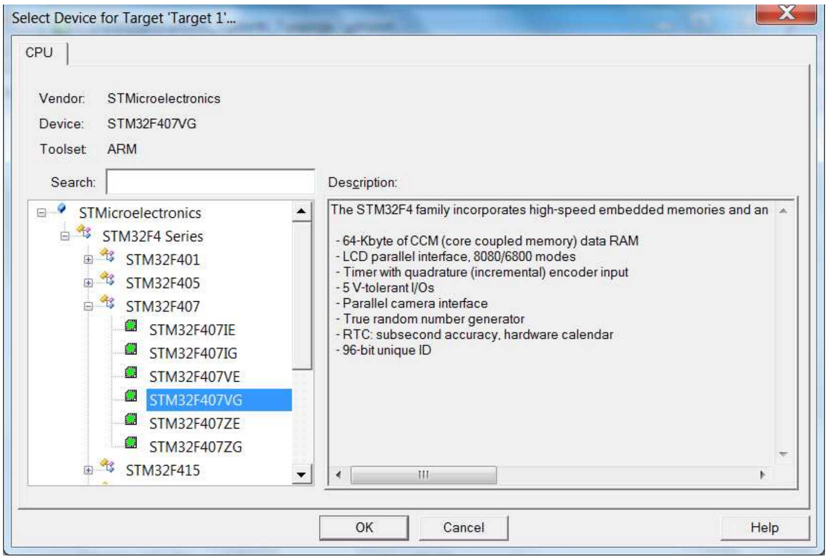
2.3 Install µVision5

After successfully install µVision5, you will see a µVision5 icon on the desk top. Double click it to start. µVision5 organizes things in workspaces and projects. Workspace is a big folder which holds many projects. A project is a smaller folder where you place your source code files for µVision5 to work on. When µVision5 is first started, it asks you to specify a workspace. The default workspace is C:\mdk\, but you can pick any folder you like.

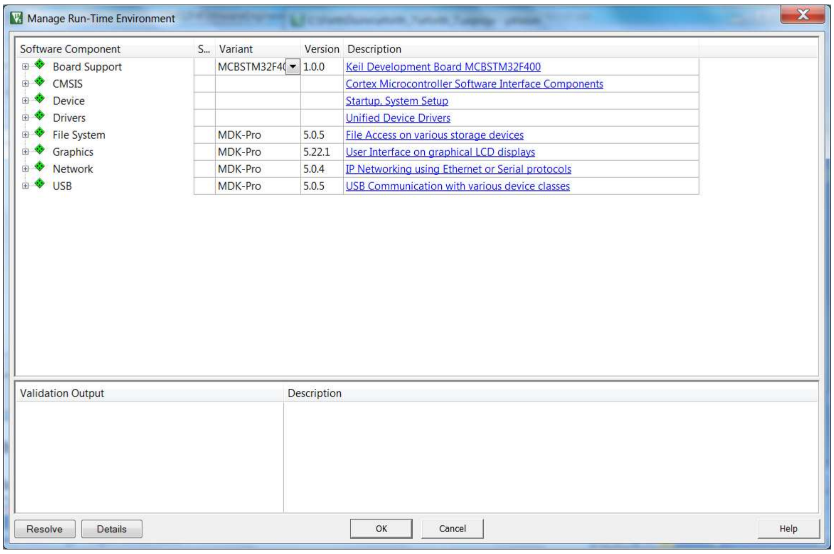
Select `Project>Create New Project`.
In the `File Selection` window, navigate to a folder you want or create a new folder.
Name new project as eforth_7 or something you like.



In the `Select Device for Target "Target 1"` window, select STM32F407VG.

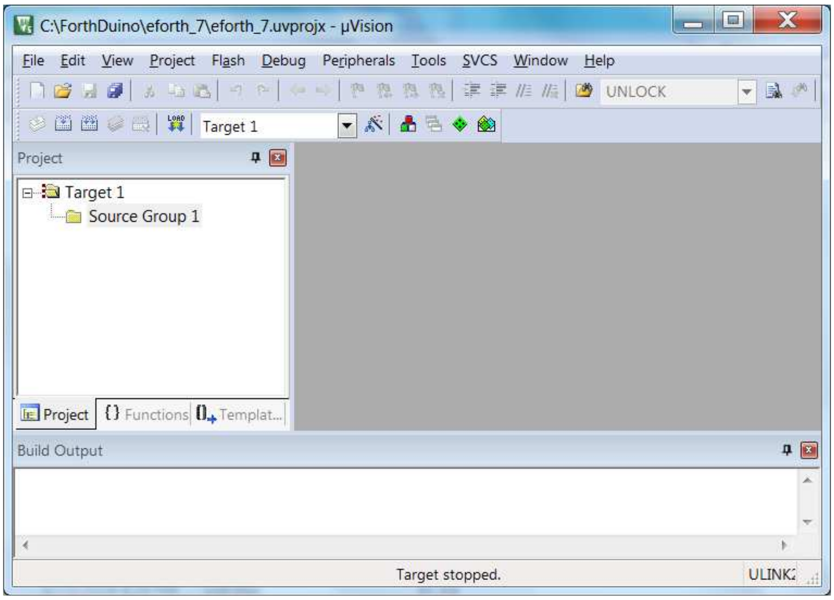


In the `Manage Run Time Environment` window, select nothing and click OK. eForth is very simple, and does not need all the supporting files and libraries usually required by a C compiler.



Copy stm32eforth720.s file into the eforth_7 project folder your created.

In Project panel, click the + box to the right of `Target 1`, to show `Source Group 1`.

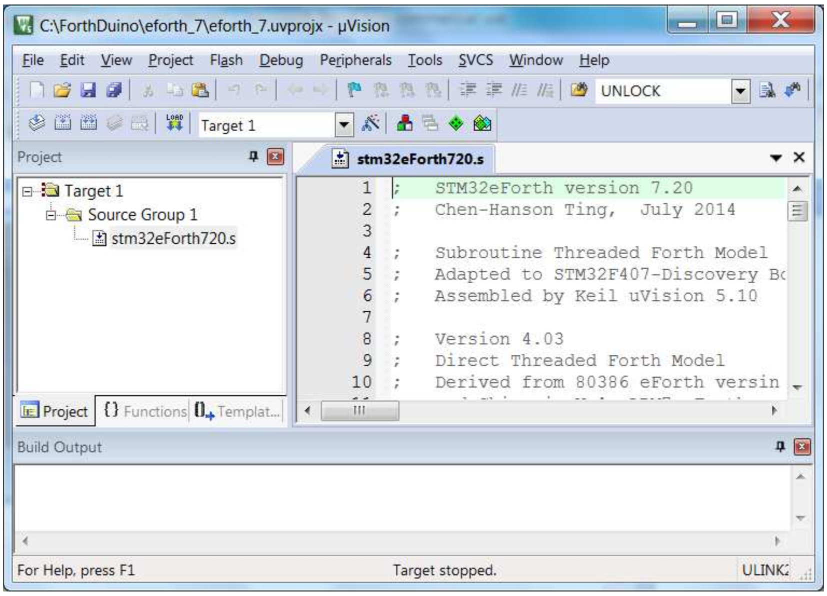


Right click `Source Group 1`, and select `Add Existing File to Group “Source Group 1”`.

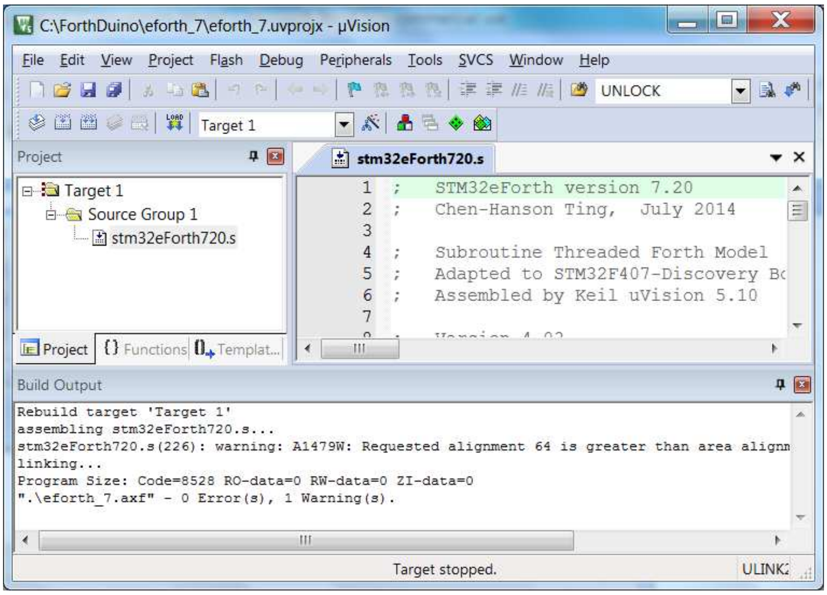
In the `Add Files to Group “Source Group1”` file selection window, select stm32eforth720.s, click `Add` box and then `Close` box.

In `Project` panel, click the + box to the right of `Source Group 1`, to show stm32eforth720.s.

Double click stm32eforth720.s, and the file is opened in the `Edit` panel.



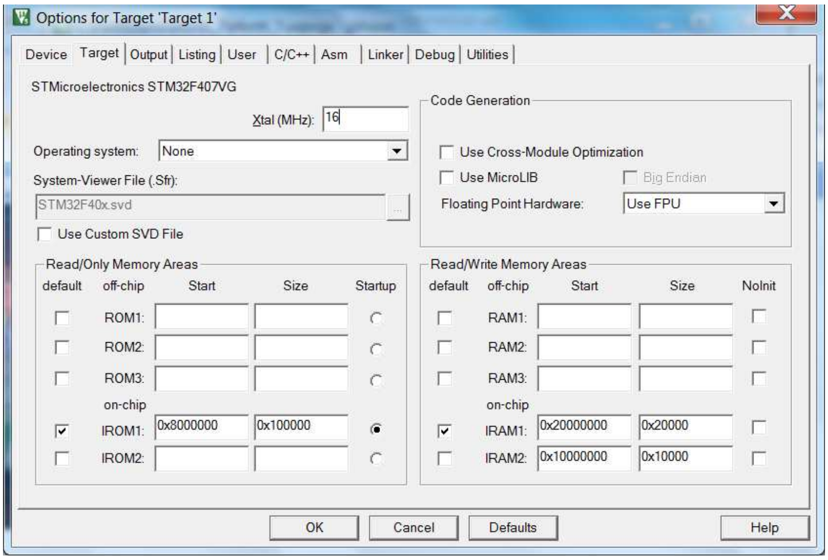
Select `Project>Rebuild all target files`, stm32eforth720.s is assembled and linked, and a stem32eforth720.axf file is created.



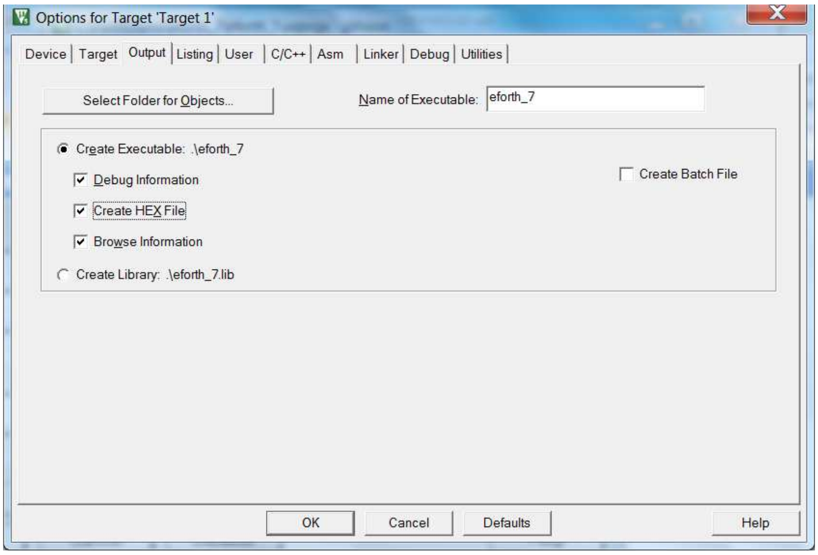
2.4 Setting up Target Environment

In order to assemble the eForth system correctly and test/debug it on Discovery, you have to be sure that the target environment is set up correctly. Pull down the `Project` menu and select `Options for Target 'Target1'`, and you will see the following window:

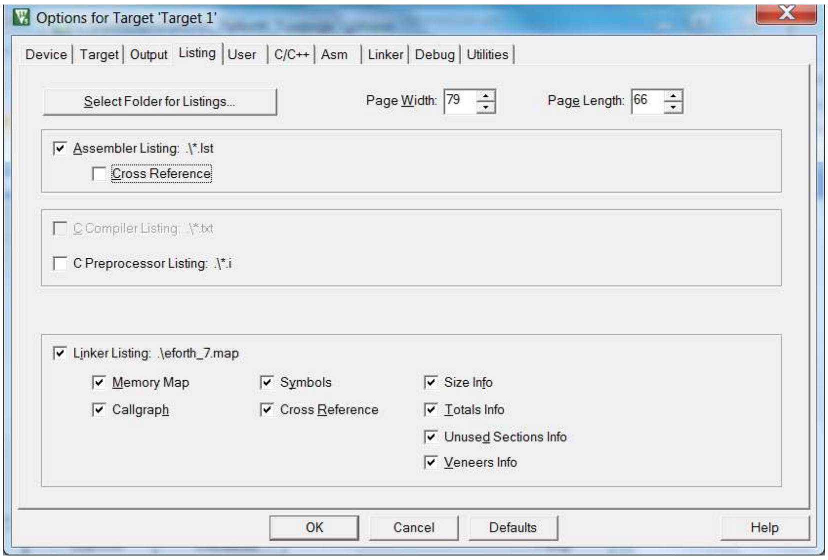
Select `Project>Options for Target "Target 1"`, or `Flash>Configure Flash Tools`.
The `Options for Target "Target1"` window appears.
Select `Target` menu, and change `Xtal frequency` to 16 MHz



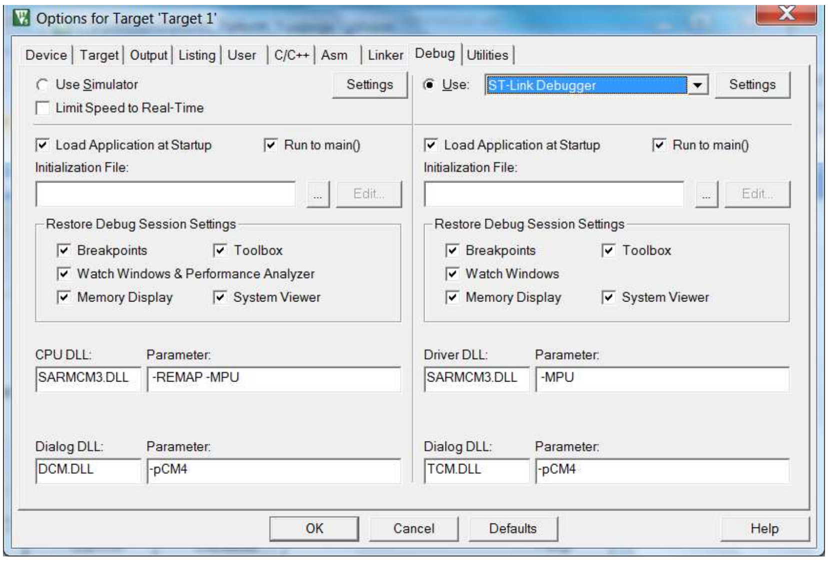
Select `Output` menu, and check `Create Hex File`.



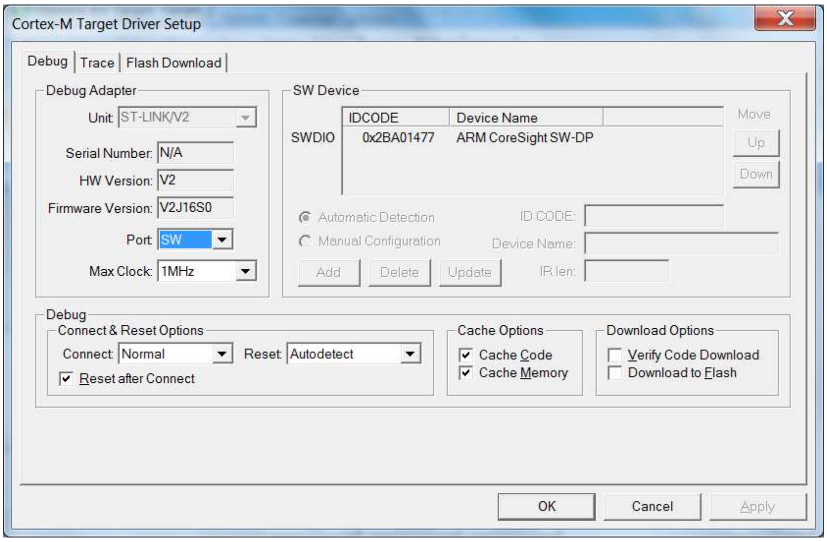
Select `Listing` menu, and uncheck `Cross Reference`.



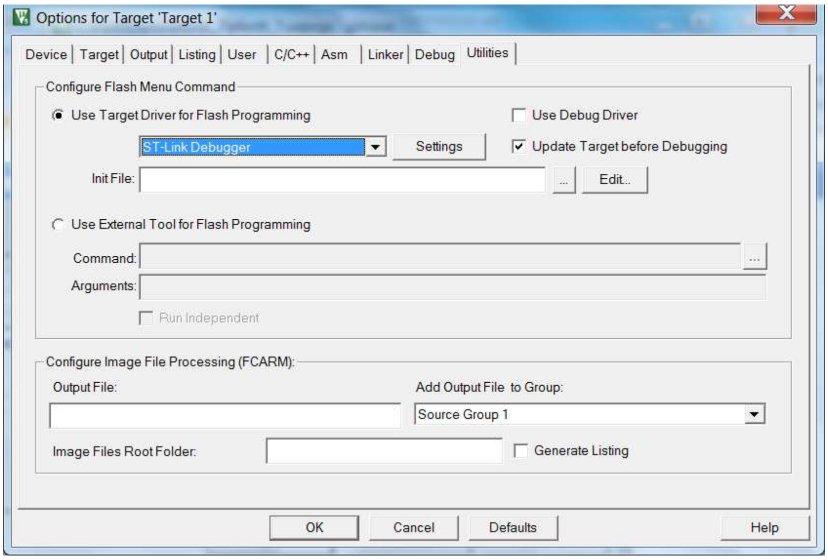
Select `Debug` menu, check `Use` debugger. In the debugger box, select `ST-Link Debugger`,



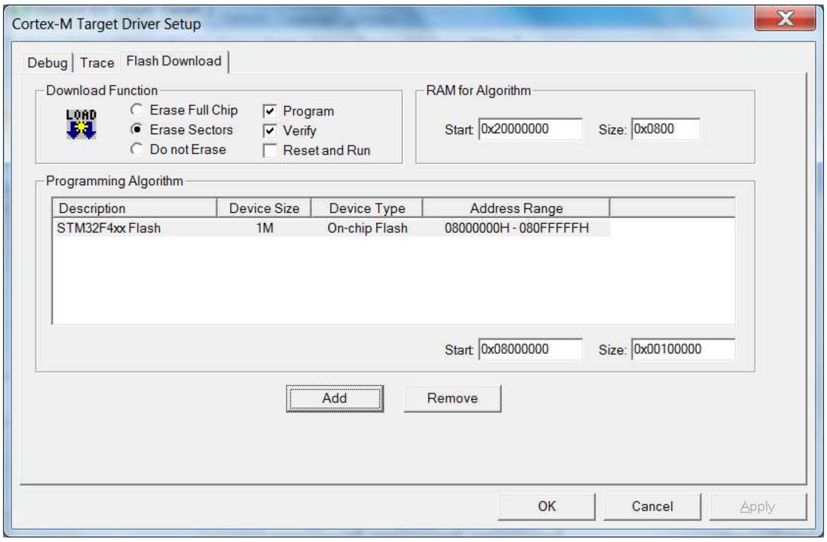
Click `Settings` box to the right of the debugger box.
In the `Cortex M Target Driver Setup` window, change `JTAG` in `Port` box to `SW`.
Click `OK`.



Select `Utilities` menu, and uncheck the box of `Use Debug Driver`.
In the device selection box under `Use Target Driver…`, select `ST-Link Debugger`.

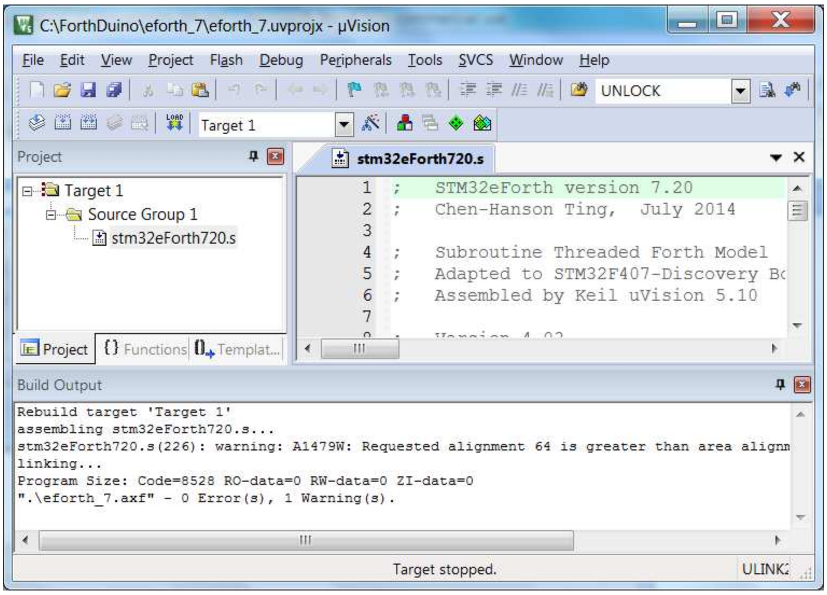


Click the `Settings` box to the right of device selection box. The `Cortex M Target Driver Setup` window opens.
Click `Add` box and add `STM32F4xx Flash to Programming Algorithm.`
Click `OK` to dismiss the `Cortex M Target Driver Setup` window.
Click `OK` to dismiss the `Options for Target “Target1”` window.



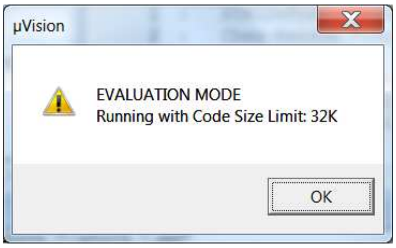
2.5 Build and Debug eForth System

To build and debug eForth System, pull down `Project` menu and select `Rebuild all target files option` and µVision5 assembles eForth file and produces an downloadable object file eforth_7.axf. The building sequence is shown in the `Output` panel at the bottom of window screen:



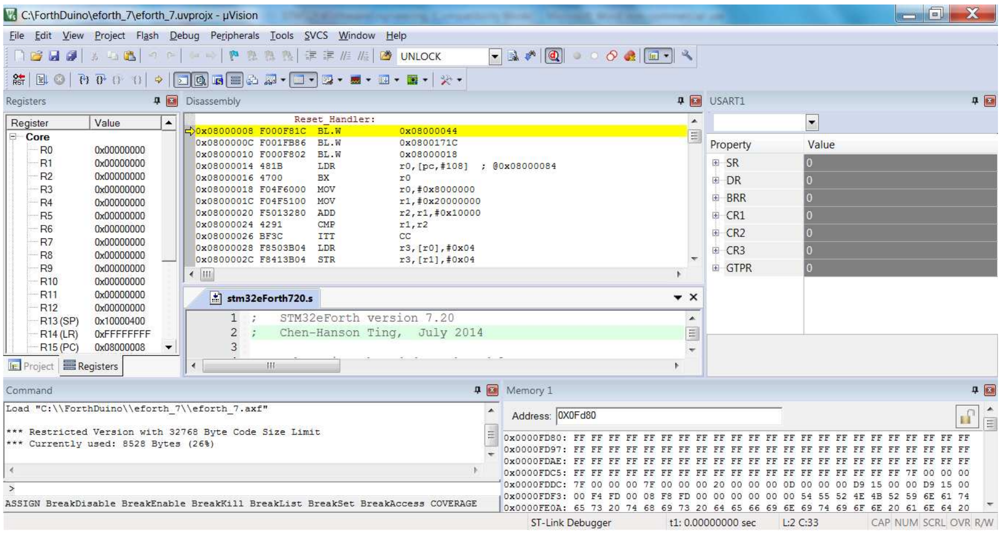
stm32eforth720.s is assembled and linked, and an eforth_7.hex file and an eforth_7.lst are also created for you to inspect.

Select `Debug>Start/Stop Debug Session`.
Dismiss the warning box `uVision warning: Evaluation Mode, Running with code size limit 32K`.



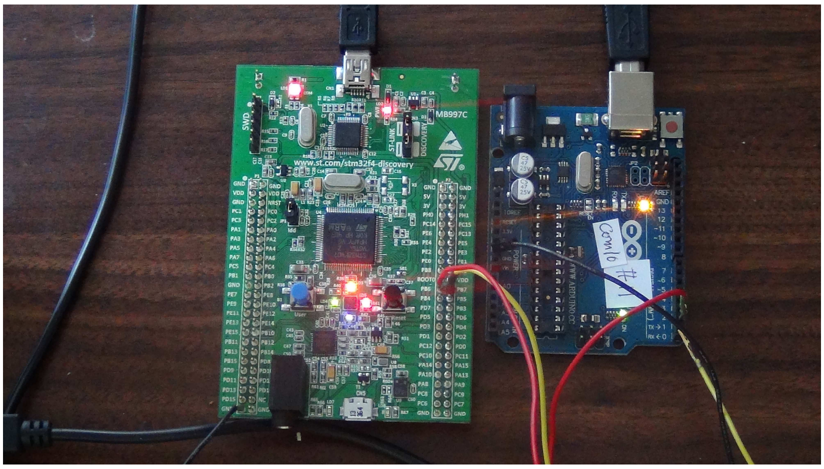
Eforth_7.axf is downloaded into the flash memory of STM32F407 chip, and the Debugger is ready to single step through the eForth code, or to run it at full speed.

Click `Debug>Run`, and stm32eforth720 signs up on HyperTerminal.



2.6 Set up HyperTerminal

Stm32eforth720 uses USART1 to communication with a terminal. On STM32F407VG, USART1 can be configured to use either Pins PA9-10 or PB6-7 for communication. Since the micro USB on CN5 is using PA9-10 pins, I use PB6-7 for eForth. I am using a separate Windows XP PC to run HyperTerminal through a USB to serial converter, which happens to be an Arduino Uno Kit. Arduino Uno Kit has a integrated USB to serial converter connecting the STmega328P chip to the host PC. To use its USB to serial converter, I remove the ATmega328P chip, and connect the PB6 (TX) on Discovery to D1 port on Arduino, the PB7 (RX) on Discovery to D0 port on Arduino. A ground wire connects the ground pins on both boards.

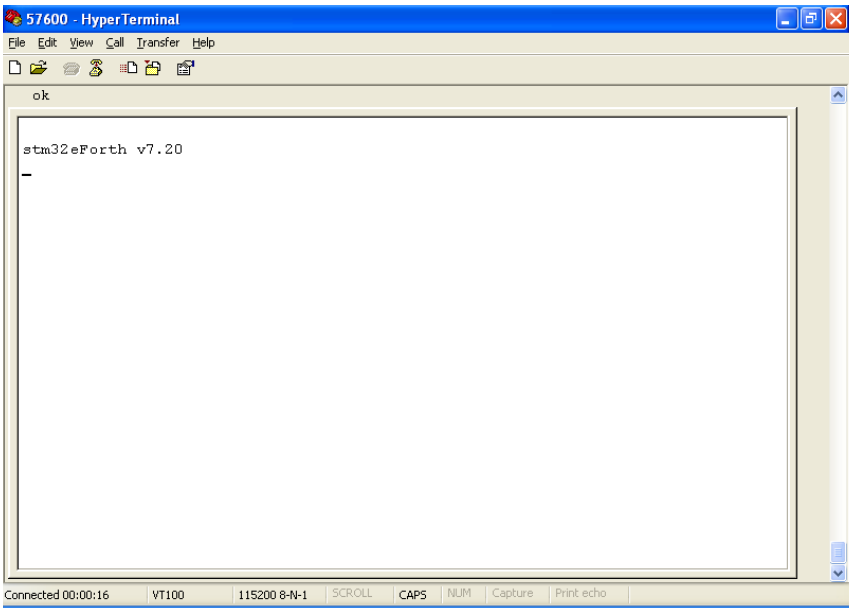


HyperTerminal on PC is configured for 115200 baud, 1 start bit, 8 data bits, 1 stop bit, no parity, no flow control. The USART1 on STM32F407 is configured similarly. STM32F407 is clocked by its high speed internal clock HSI at 16 MHz on reset. Since this HSI is factory trimmed to 1% accuracy, it is adequate to provide reliable communication on USART1.

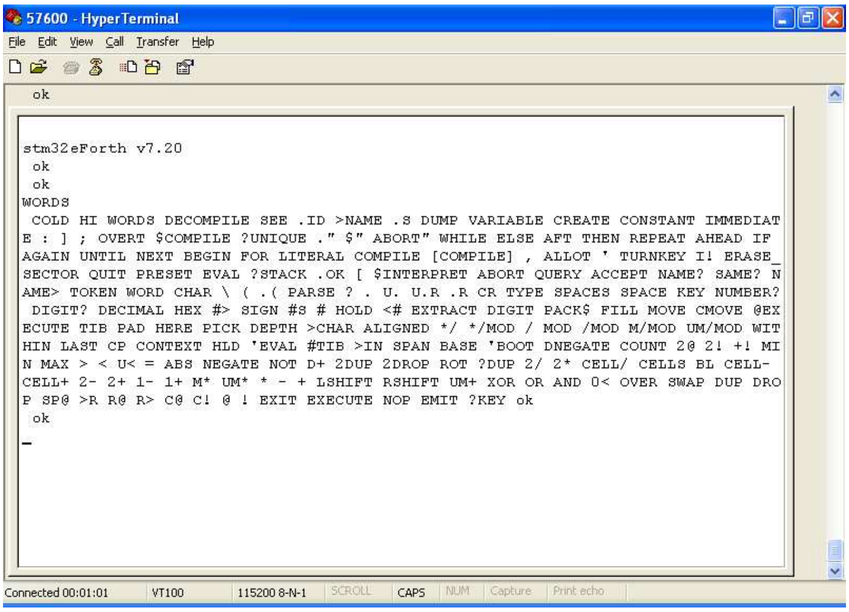
The default settings are COM1, 2400 baud, etc. You have to set the HyperTerminal to the terminal mode at 115200 baud, 8 data bits, 1 stop bit, and no parity. First put it offline by clicking the `Hang-up` icon, and pull down the `File` menu and select `Properties` option. Then you will see a property selection window. Go through the selection window and make the proper selections to get the console window like what was shown above.

Press the RESET bottom on STM32F4Discovery and HyperTerminal should display the following message:

stm32eForth v7.20



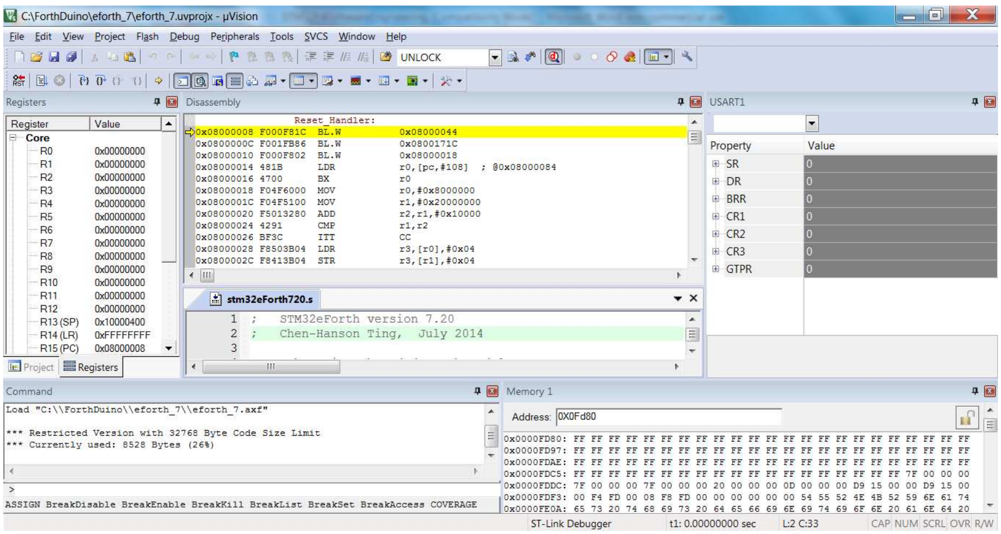
Press `Enter` key and eForth will echo `ok` messages. Type an eForth command `WORDS` followed by a `Enter`, and you will see the following console display:



HyperTerminal is thus the host environment for eForth, and you can type in Forth commands and execute them.

2.7 Return to Debugger

While eForth is running, you can stop it and return to the debugger in µVision5. Click the µVision5 window to bring it to focus. Pull down the `Debug` menu and select `Stop` and STM32F4 stops running eForth system. Now the µVision5 window changes to something like this:



Now you can inspect the Cortex M4 registers, the disassembled program, and memory contents. You can single-step through the machine instruction, set and clear break points.

2.8 Firmware Engineering

µVision5 is a very sophisticated program development environment for STM32F4. Why do you need Forth?

µVision5 is very good to develop applications. However, applications in embedded systems are only parts in a system which must be able to initialize the hardware on power up and drop into the application code correctly. It has to respond to real time stimuli and act appropriately. What Forth brings in is a complete operating system which can interact with you. To be able to interact with a human being requires a large number of commands or library routines, which are just as useful in real time applications. The system is extensible in that you can add new commands to the library by combining existing commands using very simple syntax rules. It is thus very easy to build applications which can be committed to flash memory. Using a fully debugged embedded operating system as a platform, it is easy to develop application on top of it. This is the central theme of Firmware Engineering.

There are two schools of embedded systems design. The old school is entrenched in the mentality of 8-bit microcontrollers with very limited resources, especially in ROM and RAM memories. It considers an embedded operating system unnecessary and a total waste of memory. The new school is recently liberated from memory constrains by Moore's Law, and endeavors to shoehorn entire modern operating systems like Windows CE and Linux into embedded systems. I think the truth lies somewhere in between.

Microcontroller manufacturers have struggled mightily to give us more memory and more IO devices into an SOC, System On a Chip. For embedded applications, flash memory seemed to be more important than RAM memory. This is understandable. Flash memory is cheaper and more abundant than RAM memory. Embedded applications do not need too much RAM for data storage, but they can use lots of flash to store programs. But in Forth, I can use as much RAM as possible. It is not until now that we see enough RAM memory on board so Forth can operate smoothly. STM32F4 is the first microcontroller I used that I did not feel being constrained by not having enough RAM memory.

Here we are. STM32F4-Discovery Kit is fast, big, and cheap. There are lots of projects that I had thought about but could not do because of hardware constraints. It is time to dig up these project ideas and start implementing them. 1 MB of flash, 192 KB of RAM, 80 GPIO pins, 14 counter-timers, etc., etc. Oh, boy. Where shall I start?

3. Stm32eforth720 Source Code

This chapter is a code walkthrough session. I am reading aloud the source code in the assembly file stm32eforth720.s. I will comment on the source code while reading it from the beginning to the end. At some resting pointing, where there is a big chunk of code, I will take some time to explain the intention and the implementation of the code. I hope you will be patient to walk along. In the end, I hope you will get to know this eForth system well enough to make good use of it.

3.1 A Brief History of ARM eForth

Moore's Law marches on, and more and more circuits are crowded into microcontrollers. In the last 15 years, I had programmed many ARM chips, and had amazed the progress of these chips. My approach had always been to port an eForth system onto the chips and tried to make good use of them. Here is a brief history of my eForth systems evolved with the ARM chips.

ARM7 eForth v1.10
In 2001, a then very young engineer, Mr. Chien-ja Wu in Taiwan FIG, ported the original eForth model to an ARM platform BK100PHTB from Avnet. It had a big LCD screen, and was a very impressive demo for the portability of eForth. He wrote a target compiler in Win32Forth to meta-compile eForth system.

ARM7 eForth v1.20
In 2002, I ported eForth to Nintendo's GameBoyAdvance, which was a very popular platform for game developers in Taiwan. Nintendo released very detailed information on GBA for people to build games, using flash memory cartridges. The ARM7TDMI chip in GBA had only 32KB of RAM, no flash. It had lots of external flash and RAM to host very substantial applications, besides games.

ARM7 eForth v2.01.
In 2004, I moved the eForth target compile from FPC to weForth, which later evolved into F#. At that time, I had worked on eForth2 for a while. I switched to subroutine thread model, and tried to optimize each implementation for performance. 16-bit 8086eForth also evolved into 32-bit 386eForth v4.03. That's when v.4 came to being.

ARM7 eForth v5.06
In late 2004, I started working on ADuC7024, an interesting ARM7 chip from Analog Devices. It had 62 KB of flash and 8 KB of RAM, and could thus stand alone without external memory, or any other support chip. I built a ForthStamp based on it. Business failed, because I could not handle the surface mount packages myself, and manufacturing costs killed it. Nevertheless, it was a beautiful stamp-size computer, a very small single chip computer with lots of analog capabilities, as Analog Devices was the master of ADC and DAC. I also moved the source code from Forth meta-compiler to regular assembler, using Keil's uVision3 for assembly, flash programming and debugging.

ARM7 eForth v6.03
In 2008, Dave Jaffe in Silicon Valley FIG gave me an Olimex Development Board with an AT91SAM7x256 ARM7 chip on it. It had a color LCD panel, and I used it to build a digital storage oscilloscope. The chip had 64 KB of flash and 16 KB of RAM, and lots of IO devices. Porting eForth

from the ADuC project was very easy on the same uVision3 IDE platform. It was released as Sam7ef eForth system.

STM32F4-Discovery Kit is a very nice evaluation board from STMicroelectronics. The STM32F407VG chip on it has 1 MB flash, 192 KB RAM, and a ton of peripheral devices. I ported Sam7ef.s over. Since STM32F4 is no longer an ARM chip, it is not necessary to keep the name ARM in the new eForth implementations. I planned and completed 4 versions of eForth for this chip:

STM32eforth v7.01	The eForth dictionary resides in flash memory, and executes from flash memory. It was aligned to the eForth2 model, with subroutine tread model and fully optimized for performance.
STM32eforth v7.10	The eForth dictionary resides in flash memory. Flash memory is remapped to virtual memory in Page 0. eForth executes from Page 0 memory.
STM32eforth v7.20	The eForth dictionary is still stored in flash memory. The dictionary is copied from flash to RAM. RAM memory is remapped to virtual memory in Page 0. eForth executes from Page 0 memory. Applications can be easily embedded in turnkey system.
STM32eforth v7.30	v7.20 ported to the ForthDuino Board. A thank-you gift to Taiwan FIG.

```
;*****************************************************************************
;      STM32eForth version 7.20
;      Chen-Hanson Ting,  July 2014

;      Subroutine Threaded Forth Model
;      Adapted to STM32F407-Discovery Board
;      Assembled by Keil uVision 5.10

;      Version 4.03
;      Direct Threaded Forth Model
;      Derived from 80386 eForth versin 4.02
;      and Chien-ja Wu's ARM7 eForth version 1.01

;      Subroutine thread (Branch-Link) model
;        Register assignments
;      IP          R0    ;scratch
;      SP          R1
;      RP          R2
;      UP          R3
;      WP          R4    ;scratch
;      TOS         R5
;      XP          R6    ;scratch
;      YP          R7    ;scratch
;        All Forth words are called by
;      BL.W  addr
;        All low level code words are terminaled by
;      BX    LR    (_NEXT)
;        All high level Forth words start with
;      STRFD RP!,{LR}    (_NEST)
;        All high level Forth words end with
;      LDRFD RP!,{PC}    (_UNNEST)
;        Top of data stack is cached in R5
;        USART1 at 115200 baud, 8 data bits, 1 stop bit, no parity
;      TX on PB6 and RX on PB7.
```

```
;	Version 5.02, 09oct04cht
;	fOR ADuC702x from Analog Devices

;	Version 6.01, 10apr08cht a
;	Align to at91sam7x256
;	Tested on Olimax SAM7-EX256 Board with LCD display
;	Running under uVision3 RealView from Keil

;	Version 7.01, 29jun14cht
;	Ported to STM32F407-Discovery Board, under uVision 5.10
;	Aligned to eForth 2 Model
;	Assembled to flash memory and executed therefrom.
;	Version 7.10, 30jun14cht
;	Flash memory mapped to Page 0 where codes are executed
;	Version 7.20, 02jul14cht
;	Irreducible Complexity
;	Code copied from flash to RAM, RAM mapped to Page 0.
;	TURNKEY saves current application from RAM to flash.
```

3.2 Virtual Forth Machine

3.2.1 Virtual Forth Machine on STM32F4

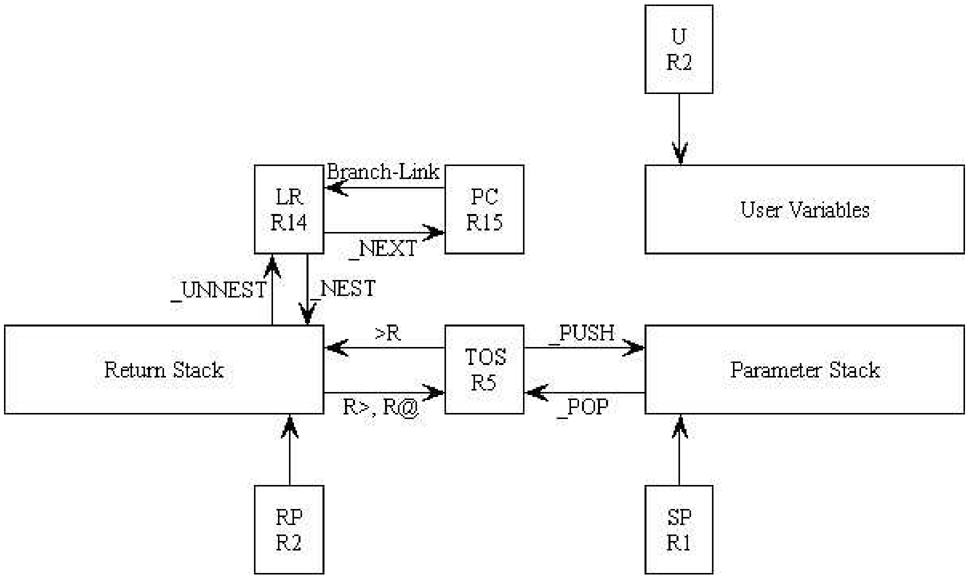
Forth is a computer model which can be implemented on any real CPU with reasonable resources. This model is generally called a Virtual Forth Machine. The components of a Virtual Forth Machine are:

- A set of Forth commands stored in memory as a dictionary.
- A text interpreter to interpret lists of Forth commands in text form.
- A compiler to compile lists of Forth commands into lists of tokens
- A CPU to traverse nested token lists and execute Forth commands.
- A return stack to traverse nested command lists.
- A parameter stack to pass parameters among commands.

The following registers are used by a Virtual Forth Machine on a Cortex M4 CPU:

Forth Register	Cortex M4 Register	Function
SP	R1	Parameter stack pointer
RP	R2	Return stack pointer
UP	R3	User area pointer
TOS	R5	Top of parameter stack
LR	R14	Link register
PC	R15	Program counter

The Virtual Forth Machine is shown schematically as in the following figure:



The text interpreter processes lists of Forth command names in text form, delimited by white spaces. The simple syntax is:

```
      <name1>  <name2>  <name3>  …  <nameN>
```

The Forth compiler converts lists of Forth command names to lists of tokens as new commands added to the dictionary. The syntax is:

```
      :  <new-name>  <name1>  <name2>  <name3>  …  <nameN>  ;
```

The text interpreter processes lists of names, the external representations of Forth commands. The Virtual Forth Machine processes lists of tokens, the internal representations of Forth commands. Forth is LISP turned inside out.

3.2.2 Reset Vector and Reset Handler

In stm32eforth720, we do not allow interrupts, do not use the interrupt stack, and do not use the heap. So, the startup code is reduced to a single reset vector, and a reset handler which initializes the Virtual Forth Machine and starts executing eForth code.

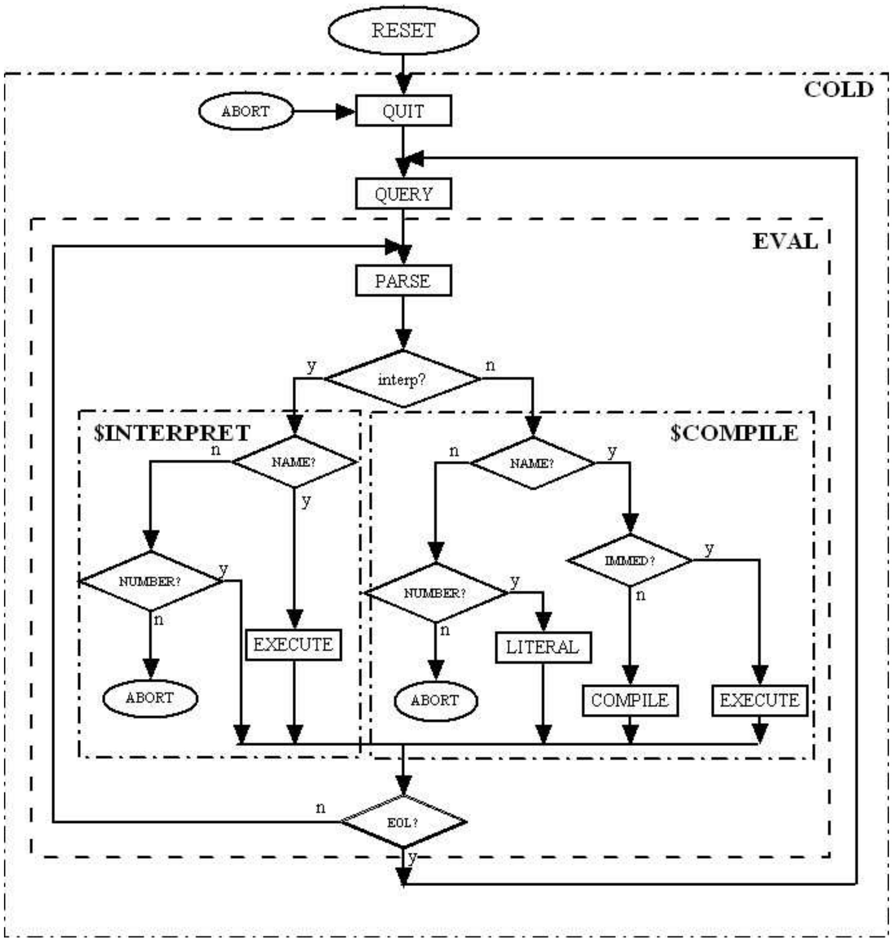
Reset_Handler	This routine is in the `Reset Vector`. When STM32F407 resets or boots up, it jumps to this routine and starts running. This is absolutely the simplest reset handler to bring up an interactive operating system. It calls up the following routines:
	`InitDevices`
	`UNLOCK`
	`REMAP`
	`COLD`

```
;*****************************************************************************
; Minimal boot-up code

      AREA        RESET, CODE, READONLY
```

```
      THUMB
      EXPORT      __Vectors         ; linker needs it
      EXPORT      Reset_Handler     ; linker needs it

; Vector Table has only Reset Vector
__Vectors   DCD   0x10000400        ; Top of hardware stack in CCM
      DCD         Reset_Handler     ; Reset Handler

      ENTRY

Reset_Handler
      BL    InitDevices       ; RCC, GPIOs, USART1
      BL    UNLOCK                  ; unlock flash memory
      BL    REMAP                   ; remap RAM to page 0
      LDR   R0,=COLD-MAPOFFSET      ; start Forth
      BX    R0
      ALIGN
```

UNLOCK	Unlock flash memory so we can write to flash. It writes two specific consecutive words into the Flash Key Registers FLASH_KEYR. UNLOCK will be discussed in the Section 3.4.10 on flash memory.
COLD	eForth cold start routine now in Page 0 of the virtual RAM memory. It is the last command at the very end of this assembly source code in Section 3.6.5.

`COLD` is the last command defined in stm32eforth720.s file. However, it is the Forth system itself, and this whole document is trying to explain it fully, following the source code. Here is a schematic drawing of the contents of `COLD`. It is enclosed in a big box, which contains a smaller box `QUIT`, which contains yet some smaller boxes. These boxes are Forth commands I will discuss later in details. There are also many diamond boxes representing branch structures. In the middle of the diagram are two boxes `$INTERGRET` and `$COMPILE`. They are the text interpreter and command compiler. This is the best graphical representation of the eForth system I can give you. You have a bird's eye view of eForth to guide you through the following discussions in minute details on how this system is constructed. You saw the forest. Later, we will see trees, flowers, and weeds. They are all essential parts of an ecosystem.



Since I am on the topic of `COLD`, I might just well show you the actual code of `COLD`. It first initializes the registers R1 as SP, R2 as RP, R3 as UP and R5 as TOS. Then it copies the user variables from 0xC0 to 0xFF00. Then it executes `HI` to send out eForth sign-on message. Finally it falls into the text interpreter loop `QUIT`. Now, Forth is running and you can communicate with it through a terminal.

```
COLD
; Initiate Forth registers
      MOVW  R3,#0xFF00        ; user area
;     MOVT  R3,#0x2000        ;
      MOV   R2,R3             ; return stack
      SUB   R1,R2,#0x100      ; data stack
      MOV   R5,#0             ; tos
      NOP
      _NEST
COLD1
      _DOLIT
      DCD   UZERO-MAPOFFSET
      _DOLIT
      DCD   UPP
      _DOLIT
      DCD   ULAST-UZERO
      BL    MOVE              ;initialize user area
      BL    PRESE             ;initialize stack and TIB
```

```
      BL    TBOOT
      BL    ATEXE              ;application boot
      BL    OVERT
      B.W   QUIT               ;start interpretation
```

3.2.3 Remap RAM memory

The primary objective in stm32eForth720 is to run in RAM, so that new command can be added to the dictionary freely. Once an application is completely debugged, the entire dictionary can then be saved into the flash memory to become a turnkey system, ready to run at power-up. The REMAP routine first copies the eForth dictionary image from flash memory to RAM memory. Then, it remaps RAM memory to Page 0, and starts eForth executing in Page 0. To remap, we simply write a 3 into the System Configuration Register SYSCFG.

Currently, stm32eforth720 uses only 64 KB of RAM memory, and only 64KB are copied from flash to RAM. It can be easily modified to use all 192 KB of available RAM.

REMAP	Copy eForth dictionary from flash memory to RAM. Then RAM memory is remapped to Page 0.

```
;****************************************************************************
; Remap eForth to execute from RAM
;
; Copy eForth from flash to RAM
REMAP
      mov   r0,#0x8000000
      mov   r1,#0x20000000
      add   r2,r1,#0x10000
REMAP1
      cmp   r1, r2
      ldrcc   r3, [r0], #4
      strcc   r3, [r1], #4
      bcc   REMAP1

; Remap RAM to page 0
      movw  R0,#0x3800         ; SYSCFG register
      movt  R0,#0x4001
      mov   R1,#3
      str   R1,[R0,#0]         ; map RAM to page 0
      bx    lr
      align
```

3.2.4 Initialize IO Devices

Stm32eforth720 uses only USART1 for communication, and GPIOD to lit up the LEDs. However, USART1 borrows pins PB6-7 for TX and RX; therefore, GPIOB has to be initialized. All three devices need to be clocked, and the Reset Clock Controller RCC must be initialized.

The USART1 on STM32F407 is configured to 115200 baud, 1 start bit, 8 data bits, 1 stop bit, no parity, no flow control. STM32F407 is clocked by a high speed internal clock HSI at 16 MHz on reset. Since this HSI is factory trimmed to 1% accuracy, it is adequate to provide reliable communication on USART1.

Just to make your head spin, STM32F4 has 9 16-bit GPIO devices, from GPIOA to GPIOI. Most of the pins in these IO devices have multiply functions. They can be configures as input pins, output pins, analog pins, or alternate function pins. UASRT1 uses PB6 pin in GPIOB port for TX and PB7 for RX. These pins are initialized for alternate function AF7 for USART1.

Wonder how 139 in USART1_BRR register sets up 115200 baud for USART1? We have a 16 MHz HSI clock. USART1 has a default divide by 16 pre-scaler, which divides HSI to 1 MHz. 1000000/115200=8.680. We have an integer part of 8, and a fractional part of 0.680. The fractional part is stored in a 4 bit field, which has 16 divisions. 0.680*16=10.88. The closest integer is 11 (0xB). Put 8 in bit 4-7, and B in bit 0-3 of the USART1_BRR register, and you have 0x8B. That's 139 in decimal. Cool?

On STM32F4-Discovery Kit, there are 4 color LEDs in the middle for an accelerometer demo. They are driven by GPIOD port, on pins PD12-15. It is nice to lit up these LEDs when eForth is running. Hence, PD12-15 are configured as output pins, and the corresponding bits in the GPIOD_ODR are set to lit up the LEDs. This is already half of a Blinky demo.

InitDevices	Initialize USART1, GPIOB and GPIOD. These are the devices we use. All devices in STM32F4 must be properly clocked. Therefore, we have to initialize the Reset and Clock Control RCC to clock USART1, GPIOB and GPIOD. GPIOB lends pins to USART1, and GPIOD drives the LEDs.

```
;*****************************************************************************
; Here are devices used by eForth
RCC   EQU   0x40023800
GPIOB EQU   0x40020400
GPIOD EQU   0x40020C00
USART1      EQU   0x40011000
; Assumes system running from 16 MHz, HSI (Normal at Reset)
; USART1 PB6 TX and PB7 RX; this works.

InitDevices
; init Reset Clock Control RCC registers
      ldr   r0, =RCC          ; RCC
      ldr   r1, [r0, #0x30]   ; RCC_AHB1ENR
      orr   r1, #0xA          ; GPIOBEN+GPIODEN
      str   r1, [r0, #0x30]
      ldr   r1, [r0, #0x44]   ; RCC_APB2ENR
      orr   r1, #0x10         ; USART1EN (1 << 4)
      str   r1, [r0, #0x44]
; init GPIOB
      ldr   r0, =GPIOB ; GPIOB
      ldr   r1, [r0, #0x00]   ; GPIOx_MODER
      orr   r1, #0xA000       ; =AF Mode
      str   r1, [r0, #0x00]
      ldr   r1, [r0, #0x20]   ; GPIOx_AFRL
      orr   r1, #0x77000000   ; =AF7 USART1
      str   r1, [r0, #0x20]
```

```
; init USART1
      ldr   r0, =USART1       ; USART1
      movw  r1, #0x0200C      ; enable USART
      strh  r1, [r0, #12]     ; +12 USART_CR1 = 0x2000
      movs  r1, #139          ; 16MHz/8.6875 (139, 0x8B) == 115200
      strh  r1, [r0, #8]      ;  +8 USART_BR
; Configure PD12-15 as output with push-pull
      ldr   r0, =GPIOD  ; GPIOD
      mov   r1, #0x55000000   ; output
      str   r1, [r0, #0x00]
      mov   r1, #0xF000       ; set PD12-15, turn on LEDs
      str   r1, [r0, #0x14]
      bx    lr
      ALIGN
      LTORG
```

3.2.5 Virtual Memory of STM32F407

STM32F407 has this memory map:

Virtual Memory	0x00000000-000FFFFF
Flash Memory	0x08000000-0807FFFF
Core Coupled Memory	0x10000000-1000FFFF
System Memory	0x1FFF0000-1FFF77FF
RAM Memory	0x20000000-2001FFFF
System and IO Devices	0x40000000-0xFFFFFFFF

1 MB of memory space from 0 to 0xFFFFF is the virtual memory, which can be mapped or aliased to Flash memory, RAM memory, or boot ROM in the System Memory. Identical code in these physical memories can assume logical addresses in the virtual memory or Page 0 memory, and can be executed as though it is in a Page 0 physical memory.

Mapping RAM memory to Page 0 is especially convenient for stm32eforth720, because new commands can be easily added in the RAM memory to extend the command dictionary. It is possible to have eForth in the flash memory and add new command to the flash memory directly. However, it requires a different set of memory store commands for the RAM memory and for the flash memory, and the system becomes more complicated than it should be.

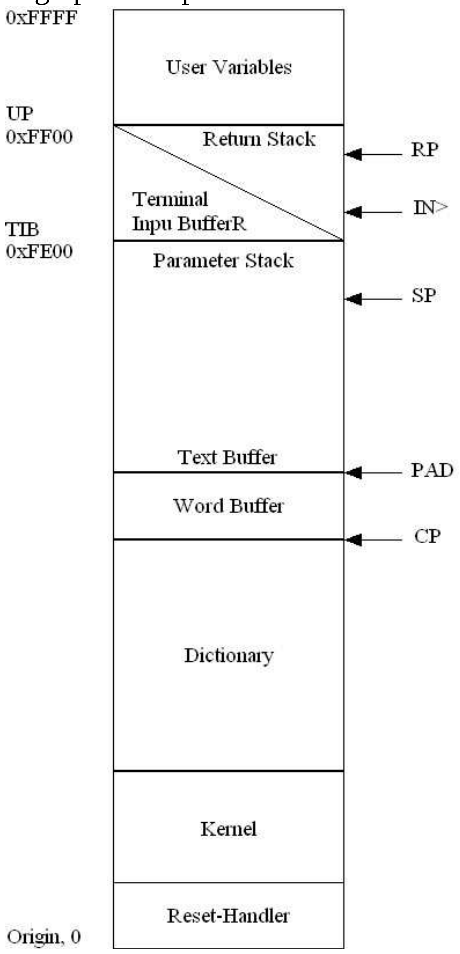
eForth dictionary is initially stored in the flash memory. Upon booting, `Reset_Handler` copies the entire dictionary from flash memory to RAM memory, re-maps RAM memory to Page 0, and executes from Page 0. The dictionary can grow at will, as new commands are added to RAM memory mapped to Page 0. When an application is complete, the entire dictionary including added commands can be saved back to the flash memory. When re-booted, the new eForth system will be activated. This way, we can develop new application interactively in RAM memory, and then save the results in flash for a final product to be released.

All the STM32F4 transfer instructions, branching and conditional branching, use PC relative addressing, and are assembled correctly for physical memory and for virtual memory. The high level branching commands in eForth use absolute addresses. So are the link field addresses which link the eForth commands as a linear list. These absolute addresses have to be corrected by a constant in MAPOFFSET. If the code is executed in the physical flash memory, MAPOFFSET is 0. If the code is executed in the virtual memory, MAPOFFSET must be 0x8000000. User variables stored in RAM must be so corrected with RAMOFFSET.

Memory allocation of eForth system inside Page 0 is as follows:

Memory allocation	Usage
0000-0007	Reset vector
0008-00BF	Reset handler and device inits
00C0-00FF	Initial user variables
0100-2127	Forth dictionary
2128-	Word buffer
2178-	PAD buffer
-FE00	Parameter stack
FE00-	TIB, terminal input buffer
-FF00	Return stack
FF00-FF3F	User variables

A graphical representation of the eForth memory map is show in the following figure:



3.2.6 Constants Used by Assembler

Constant	Value	Function
VER	7	Major release version
EXT	2	Minor extension
RAMOFFSET	0x20000000	For remapping. 0 if RAM is not remapped.
ROMOFFSET	0x08000000	For remapping. 0 if flash is not remapped.
COMPO	0x40	Lexicon compile-only bit
IMEDD	0x80	Lexicon immediate bit
BASEE	16	Default radix for number conversion
BKSPP	8	Back space ASCII character
LF	10	Line feed ASCII character
CRR	13	Carriage return ASCII character
RPP	0xFF00	Top of return stack (RP0)
TIBB	0xFE00	Terminal input buffer (TIB)
UPP	0xFF00	Start of user area (UP0)
SPP	0xFE00	Top of parameter stack (SP0)

```
;*****************************************************************************
; Version control

VER EQU     0x07  ;major release version
EXT EQU     0x20  ;minor extension

; Constants

;RAMOFFSET  EQU   0x00000000  ;absolute
;MAPOFFSET  EQU   0x00000000  ;absolute
RAMOFFSET  EQU    0x20000000  ;remap
MAPOFFSET  EQU    0x08000000  ;remap

COMPO EQU   0x040 ;lexicon compile only
IMEDD EQU   0x080 ;lexicon immediate bit
MASKK EQU   0x0FFFFFF1F ;lexicon bit mask, allowed for Chineze character

CELLL EQU   4     ;size of a cell
BASEE EQU   16    ;default radix
VOCSS EQU   8     ;depth of vocabulary stack

BKSPP EQU   8     ;backspace
LF EQU      10    ;line feed
CRR EQU     13    ;carriage return
ERR EQU     27    ;error escape
TIC EQU     39    ;tick

;; Memory allocation    0//code>--//--<sp//tib>--rp//user//
;;    0000  ;RAM memory mapped to Page 0, Reset vector
;;    0008  ;init devices
;;    00C0  ;initial system variables
;;    0100  ;Forth dictionary
;;    2150  ;top of dictionary, HERE
```

```
;;      2154  ;WORD buffer
;;      FE00  ;top of data stack
;;      FE00  ;TIB terminal input buffer
;;      FF00  ;top of return stack
;;      FF00  ;system variables
;;      8000000     ;flash, code image
;;      1000400     ;top of hardware stack for interrupts
;;      20000000    ;RAM

SPP     EQU     0x2000FE00-RAMOFFSET    ;top of data stack (SP0)
TIBB    EQU     0x2000FE00-RAMOFFSET    ;terminal input buffer (TIB)
RPP     EQU     0x2000FF00-RAMOFFSET    ;top of return stack (RP0)
UPP     EQU     0x2000FF00-RAMOFFSET    ;start of user area (UP0)
DTOP    EQU     0x2000FC00-RAMOFFSET    ;start of usable RAM area (HERE)
```

3.2.7 Assembly Macros

`_NEXT`, `_NEST` and `_UNNEST` are collectively called the 'inner interpreter' of eForth. They are the corner stones of a Virtual Forth Machine as they control the execution flow of Forth commands in the Cortex M4 system.

_NEXT	Terminate a primitive command. It is like a return from subroutine. It assembles a `BX LR` instruction, which jumps to the next command pointed to by the Link Register LR in a token list calling this primitive command. `_NEXT` thus allows the Virtual Forth Machine to exit a primitive command and resume processing the token list in a compound command which calls this primitive command.

```
;*************************************************************************
;       Assemble inline direct threaded code ending.

        MACRO
        _NEXT               ;end low level word
        BX      LR
        MEND
```

_NEST	Initiate a compound command. It pushes LR register onto the return stack, and then starts executing the following token list, as branch-link instructions, using LR to scan the token list. It assembles a single instruction `STMFD R2!,{LR}`, showing that Cortex M4 is a very efficient host for a Virtual Forth Machine.

```
        MACRO
        _NEST               ;start high level word
        STMFD R2!,{LR}
        MEND
```

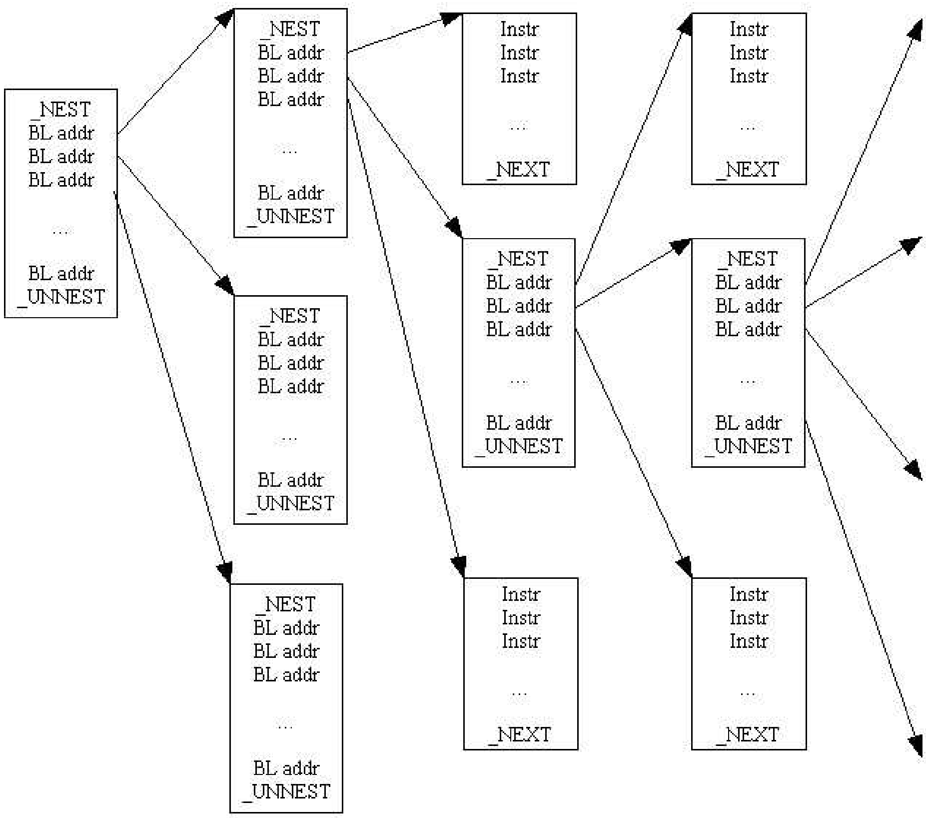
_UNNEST	Terminate a compound command. It undoes what `_NEST` accomplished. _UNNEST pops the top item on the return stack into the PC register. Consequently, execution returns to the token list which calls this compound command, briefly interrupted by calling this compound command. It assembles a single Cortex M4 instruction `LDMFD R2!,{PC}`.

```
        MACRO
```

```
_UNNEST          ;end high level word
LDMFD R2!,{PC}
MEND
```

A compound eForth commands contains a token list. Tokens are in the form of branch and link `BL<addr>` instructions in Cortex M4 CPU. Tokens may take other forms depending upon the Forth implementation. In the original eForth1 direct thread model, tokens were code field addresses of Forth commands. In the later eForth2 subroutine thread model, tokens were subroutine call instructions. In this stm32eforth720 implementation, tokens are BL instructions.

BL instructions may call other compound instructions, and the return addresses in the LR register must be nested on the return stack. At the end of a nested branch, there is always a leave of primitive command containing machine instructions. Nesting and unnesting are shown in the following figure. As shown in their macro definitions, `_NEXT`, `_NEST` and `_UNNEST` all assemble single Cortex M4 instructions, and the Virtual Forth Machine hosted on M4 is very efficient and very fast, because the calling, returning, nesting and unnesting require very little resources in either memory space of in clock cycles.



Nesting of eForth Commands

Token lists in the code field of compound commands are generally lists of BL instructions. However, other structures can be embedded in token lists. The most prevalent structure is integer literal structure, which pushes a integer value on parameter stack in run time. Numbers cannot be embedded in a token list by themselves. They have to be enclosed in a integer literal structure which begins with a `BL doLIT` instructions and ends with the integer value. The macro `_doLIT` assembles the `BL doLIT` instruction. The integer value must be assembled with a `DCD` directive.

_DOLIT	Start a integer literal structure in a compound command. It assembles a `BL doLIT` instruction to begin an integer literal structure. It is followed by the value of the integer. In run time, `doLIT` retrieves this integer and pushes it on the parameter stack.

```
MACRO
_DOLIT            ;long literals
BL    DOLIT
MEND
```

Virtual Forth Machine has a dual stack architecture and a parameter stack is used to handle numeric parameters passing among nested commands. For efficiency, the top item of the parameter stack is cached in R5 register, and the body of the stack is managed by stack pointer SP in R1. The most common stack operations are pushing R5 on the external stack, and popping the top of external stack back into R5 register. These two operations are defined as macros `_PUSH` and `_POP`. Actually they are the core of the primitive Forth stack commands `DUP` and `DROP`.

_PUSH	Push the top item on the parameter stack, which is cached in `R5` register, on the external parameter stack. It is used to implement `DUP` command, and other commands which push new data on the parameter stack.

```
MACRO
_PUSH             ;push R5 on data stack
STR   R5,[R1,#-4]!
MEND
```

_POP	Pop external parameter stack and copy the popped item into `R5` register, TOS. It is used to implement `DROP` commands, and many other commands consuming top items on the parameter stack.

```
MACRO
_POP              ;pop data stack to R5
LDR   R5,[R1],#4
MEND
```

3.2.8 User Variables

In a multitasking system, many user share one CPU and other resources in a computing system. Each user has a private memory area to store many variables necessary to run his task. The system can leave a task temporarily to serve other tasks, and return to this task continuing the unfinished work, if each task has its own copies of user variables. eForth1 was designed with multitasking in mind, and the term user variable persisted. In a single user environment, user variables can be called system variables.

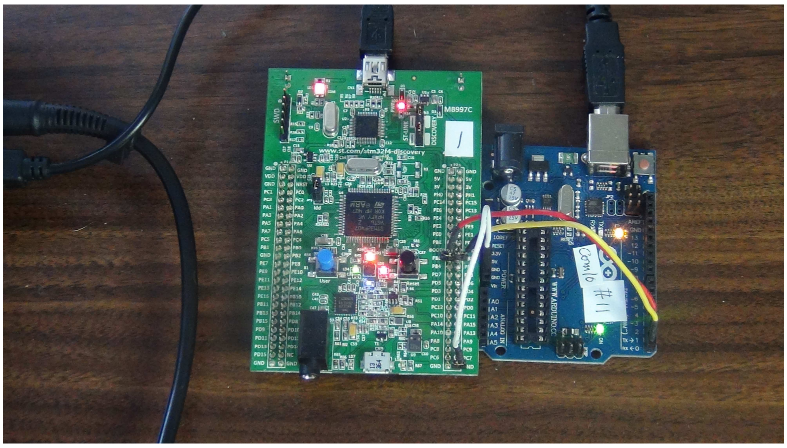
Memory location 0xC0-0xFF is allocated for a table storing initial values of user variables, which are used by eForth interpreter and compiler to perform necessary functions. This table is copied from 0xC0 to 0xFF00 when eForth enters its cold start routine `COLD`.

User Variable	Initial Value Address	Function
'BOOT	0xC4	Execution vector to start application command.
BASE	0xC8	Radix base for numeric conversion.
tmp	0xCC	Scratch pad.
SPAN	0xD0	Number of characters received by `ACCEPT`.
>IN	0xD4	Input buffer character pointer used by text interpreter.
#TIB	0xD8	Number of characters in input buffer.
'TIB	0xDC	Address of Terminal Input Buffer.
'EVAL	0xE0	Execution vector switching between `$INTERPRET` and `$COMPILE`.
HLD	0xE4	Pointer to a buffer holding next digit for numeric conversion.
CONTEXT	0xE8	Vocabulary array pointing to last name field in dictionary.
CP	0xEC	Pointer to top of dictionary, the first available flash memory location to compile new command
DP	0xF0	Pointer to the first available RAM memory location. Not used in RAM based system,
LAST	0xF4	Pointer to name field of last command in dictionary.

```
;***************************************************************************
; COLD start moves the following to USER variables.
; MUST BE IN SAME ORDER AS USER VARIABLES.

      ALIGN 64    ; align to page boundary

UZERO
      DCD   0                 ;Reserved
      DCD   HI-MAPOFFSET      ;'BOOT
      DCD   BASEE             ;BASE
      DCD   0                 ;tmp
      DCD   0                 ;SPAN
      DCD   0                 ;>IN
      DCD   0                 ;#TIB
      DCD   TIBB              ;TIB
      DCD   INTER-MAPOFFSET   ;'EVAL
      DCD   0                 ;HLD
      DCD   LASTN-MAPOFFSET   ;CONTEXT
      DCD   CTOP-MAPOFFSET    ;FLASH
      DCD   CTOP-MAPOFFSET    ;RAM
      DCD   LASTN-MAPOFFSET   ;LAST
      DCD   0,0               ;reserved
ULAST
      ALIGN
```

3.2.9 USART1 Communication

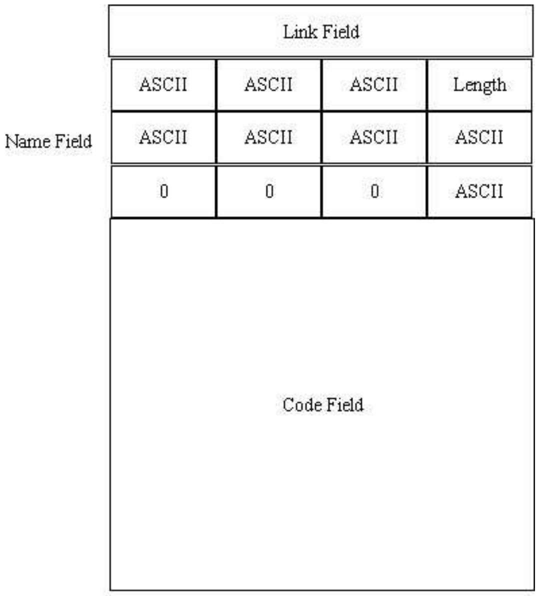
Stm32eforth720 uses USART1 to communication with a terminal. On STM32F407VG, USART1 can be configured to use either Pins PA9-10 or PB6-7 for communication. Since the micro USB port CN5 on STM32F4-Discovery Kit is using PA9-10 pins, I have to use PB6-7 for eForth. I am using a separate Windows XP PC to run HyperTerminal through a USB to serial converter, which happens to be an Arduino Uno Kit. Arduino Uno Kit has an integrated USB to serial converter connecting the STmega328P chip to a host PC. To use its USB to serial converter, I remove the ATmega328P chip, and connect the PB6 (TX) on Discovery to D1 port on Arduino, the PB7 (RX) on Discovery to D0 port on Arduino. A ground wire connects the ground pins on both boards. My Discovery-Arduino connection is show in the following picture:



`?KEY` and `EMIT` are the two primitive commands atm32eforth720 communicate with a terminal. In the `?KEY` code, notice the code fragment

```
      DCD   0
_QRX  DCB   4
      DCB   "?KEY"
      ALIGN
```

It builds an header for the command `?KEY`. All commands which are available to the user have similar headers. The names of commands are linked into a linear chain to be searched by the Forth text interpreter. There are many commands which are used to build the eForth system, but rarely or never used by users. I commented out the header of these commands to save memory space, and also hide these command from ordinary users so they will not ask too many embarrassing questions. The header has a 32-bit link field and a variable length name field, wherein the first byte contains the length of the name. The name field is zero filled to 32-bit world boundary. The code field follows the name field, as shown in the following figure:



Structure of eForth Commands

?KEY	Examine the status register USART1_SR to see if there is a valid character in the receiver. If a character is received, ?KEY reads the ASCII code of the character in data register USART1_DR and pushes it on the parameter stack. It then pushes a true flag on the top. If no character is received, it only pushes a false flag on the parameter stack.

```
;***********************************************************************
; Start of Forth dictionary
; usart1

;   ?RX      ( -- c T | F )
;     Return input character and true, or a false if no input.
      DCD   0
_QRX  DCB   4
      DCB   "?KEY"
      ALIGN
QKEY
QRX
      _PUSH
      ldr   r4, =0x40011000   ; USART1 F2/F4
      ldrh  r6, [r4, #0]      ; USART->SR
      ands  r6, #0x20         ; RXE
      BEQ   QRX1
      LDR   R5, [R4, #4]
      _PUSH
      MVNNE R5,#0
QRX1
      MOVEQ R5,#0
      _NEXT
```

EMIT	Send a character to the transmitter. It first waits on the transmitter buffer empty flag in USART1_SR register. When the transmitter is ready to

	transmit, it pops the character off the parameter stack and writes it into the transmitter data register USART1_DR. USART1 transmits the character.

```
;   TX!      ( c -- )
;     Send character c to the output device.

      DCD   _QRX-MAPOFFSET
_TXSTO      DCB   4
      DCB   "EMIT"
      ALIGN
TXSTO
EMIT
TECHO
      ldr   r4, =0x40011000   ; USART1 F2/F4
TX1   ldrh  r6, [r4, #0]      ; USART->SR
      ands  r6, #0x80         ; TXE
      beq   TX1
      strh  r5, [r4, #4]      ; USART->DR
      _POP
      _NEXT

      ALIGN
      LTORG
```

3.3 eForth Kernel

eForth kernel is a group of simple Forth commands which are necessary to build the Forth operating system, and also useful to you when you develop applications programs. Forth has two classes of commands: primitive command which contains machine instructions, and compound command which contains a token list. Simple commands are grouped together in a kernel for the convenience of discussion. After the kernel, specialized commands ar grouped together for the text interpreter, Forth compiler, and debugging tools.

3.3.1 Original Primitive Commands

One of the very important features of the original eForth1 model was a very small machine dependent kernel of primitive commands. A small set of primitive commands allows eForth1 to be ported to many CPUs very conveniently. The selection of commands in this kernel is based on the criteria that they are very difficult if not impossible to synthesize from other primitive commands. From this set of primitive commands, all other Forth commands are derived. The primitive commands in the original eForth1 model are the following:

```
System interface:         ?RX, TX!, !IO
Inner interpreters:       DOLIT, DOLIST, NEXT, ?BRANCH, BRANCH, EXECUTE, EXIT
Memory access:            ! , @, C!, C@
Return stack:             RP@, RP!, R>, R@, R>
Data stack:               SP@, SP!, DROP, DUP, SWAP, OVER
Logic:                    0<, AND, OR, XOR
Arithmetic:               UM+
```

In the current STM32eForth720 implementation, I re-coded and converted as many compound commands as I can to primitive commands to improve execution speed. Since many Cortex M4 instructions match very well with many eForth compound commands, expanding the primitive commands allows us to fully utilize the Cortex M4 core.

NOP	No operation.

```
;***************************************************************************
; The kernel

;   NOP     ( -- w )
;     Push an inline literal.

      DCD   _TXSTO-MAPOFFSET
_NOP  DCB   3
      DCB   "NOP"
      ALIGN
NOP
      _NEXT
      ALIGN
```

3.3.2 Integer Literals

Integer literals are by far the most numerous data structure in compound commands other than regular branch-link tokens. Address literals are used to build control structures. String literals are used to embed text strings in compound commands.

doLIT	Push the next program word onto the parameter stack as an integer literal instead of an instruction to be executed by Cortex M4 CPU. It allows integers to be compiled as in-line literals, supplying data to the parameter stack at run time. `doLIT` is not used by itself, but rather compiled by `LITERAL` which inserts `BL doLIT` and its associated integer into the token list under construction.

```
;   doLIT   ( -- w )
;     Push an inline literal.

;     DCD   _NOP-MAPOFFSET
;_LIT DCB   COMPO+5
;     DCB   "doLIT"
;     ALIGN
DOLIT
      _PUSH
      BIC   LR,LR,#1          ; clear b0 in LR
      LDR   R5,[LR],#4        ; get literal at word boundary
      ORR   LR,LR,#1          ; aet b0 in LR
      _NEXT
      ALIGN
```

EXECUTE	Pop the code field address from the parameter stack and executes that command. This powerful command allows you to execute any command which is not a part of a branch-link instruction list. Bit b0 of the address must be set to conform to THUMB2

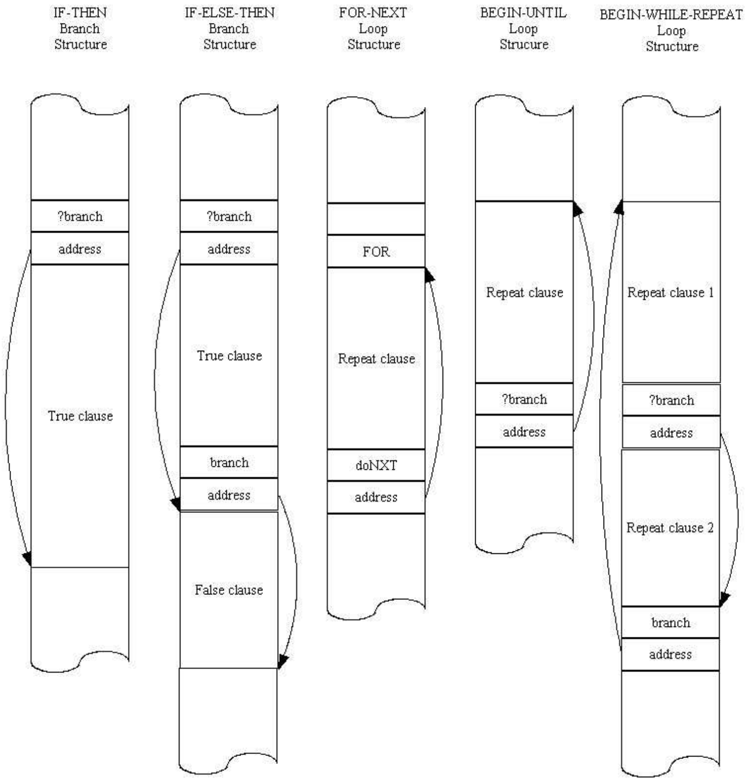
	requirement.

```
;   EXECUTE ( ca -- )
;     Execute the word at ca.

      DCD   _NOP-MAPOFFSET
_EXECU      DCB   7
      DCB   "EXECUTE"
      ALIGN
EXECU
      ORR   R4,R5,#1          ; b0=1
      _POP
      BX    R4
      ALIGN
```

3.3.3 Loop and Branch Commands

Forth uses three different types of address literals. `next`, `?branch` and `branch` are followed not by branch-link instructions but by addresses to locations in a list to be executed next. These address literals are the building blocks upon which loop structures and branch structures are constructed. An address literal is a branch command followed by a branch address which causes execution to be transferred to that address. The branch address most often points to a different location in the token list of the same compound command.

`next` is compiled by `NEXT.` `?branch` is compiled by `IF, WHILE` and `UNTIL`. `branch` is compiled by `AFT, ELSE, REPEAT` and `AGAIN`, as show in the next figure.



next	Terminate an indexed loop structures in a token list. A loop starts with `>R` which pushes a loop index on the return stack. When `next` is executed, it decrements this loop index on the return stack. If resulting index is not negative, jump back to the address in the next cell and repeat the loop. If the resulting index is negative, pop the return stack to discard the index, and exit the loop.

```
;   next    ( -- )
;     Run time code for the single index loop.
;     : next ( -- ) \ hilevel model
;      r> r> dup if 1 - >r @ >r exit then drop cell+ >r ;

;     DCD   _EXECU-MAPOFFSET
;_DONXT       DCB   COMPO+4
;     DCB   "next"
;     ALIGN
DONXT
      LDR   R4,[R2]
      MOVS  R4,R4
      BNE   NEXT1
      ADD   R2,R2,#4
      ADD   LR,LR,#4
      _NEXT
NEXT1 SUB   R4,R4,#1
      STR   R4,[R2]
      LDR   LR,[LR,#-1] ; handle b0 in LR
      ORR   LR,LR,#1
```

```
       _NEXT
```

?branch	Start a conditional branch in compound commands. In run time, if TOS is 0, branch to the address following this command; otherwise, continue the next command after the address.

```
;   ?branch ( f -- )
;     Branch if flag is zero.

;     DCD   _DONXT-MAPOFFSET
;_QBRAN      DCB   COMPO+7
;     DCB   "?branch"
;     ALIGN
QBRAN
      MOVS  R4,R5
      _POP
      BNE   QBRAN1
      LDR   LR,[LR,#-1]
      ORR LR,LR,#1
      _NEXT
QBRAN1      ADD   LR,LR,#4
      _NEXT
```

branch	Start an unconditional branch in compound commands. In run time, branch to the address following this command.

```
;   branch  ( -- )
;     Branch to an inline address.

;     DCD   _QBRAN-MAPOFFSET
;_BRAN       DCB   COMPO+6
;     DCB   "branch"
;     ALIGN
BRAN
      LDR   LR,[LR,#-1]
      ORR   LR,LR,#1
      _NEXT
      ALIGN
```

EXIT	Terminate a compound command before reaching the end. Since it is executed as a `BL EXIT` command, the return address must be popped off the return stack and then a `_NEXT` instruction is executed.

```
;   EXIT    (  -- )
;     Exit the currently executing command.

      DCD   _EXECU-MAPOFFSET
_EXIT DCB   4
      DCB   "EXIT"
      ALIGN
EXIT
      _UNNEST
```

3.3.4 Memory Commands

The 4 memory commands @, !, C@, and C! access data and code stored in memory. They access the entire memory space of STM32F4, all type of memory devices and all IO devices. Since all IO devices are mapped in memory space, their registers can be read and written at will. You can control STM32F4 chip interactively using these commands. This is the greatest advantage stm32eForth has over other operating system which severely restrict your access to memory and IO devices.

You can use @ and C@ to read flash memory. To write flash memory, we have an I! command which will be discussed in Section 3.6.5 on flash memory.

!	Store the 32-bit data `w.` the second item on parameter stack, into the address `a` on top of the parameter stack.

```
;   !    ( w a -- )
;      Pop the data stack to memory.

      DCD   _EXIT-MAPOFFSET
_STORE      DCB   1
      DCB   "!"
      ALIGN
STORE
      LDR   R4,[R1],#4
      STR   R4,[R5]
      _POP
      _NEXT
```

@	Read a 32-bit data `w` stored in the address `a` on top of the parameter stack. The address is a byte address pointing to a location in memory.

```
;   @    ( a -- w )
;      Push memory location to the data stack.

      DCD   _STORE-MAPOFFSET
_AT   DCB   1
      DCB   "@"
      ALIGN
AT
      LDR   R5,[R5]
      _NEXT
```

C!	Store an 8-bit data `c,` the second item on parameter stack, into the address `a` on top of the parameter stack.

```
;   C!        ( c b -- )
;      Pop the data stack to byte memory.

      DCD   _AT-MAPOFFSET
_CSTOR      DCB   2
      DCB   "C!"
      ALIGN
CSTOR
```

```
        LDR   R4,[R1],#4
        STRB  R4,[R5]
        _POP
        _NEXT
```

C@	Read an 8-bit data `c` stored in the address `a` on top of the parameter stack.

```
;   C@        ( b -- c )
;     Push byte memory location to the data stack.

      DCD   _CSTOR-MAPOFFSET
_CAT  DCB   2
      DCB   "C@"
      ALIGN
CAT
      LDRB  R5,[R5]
      _NEXT
```

3.3.5 Return Stack

eForth system uses the return stack for two specific purposes: to save return addresses while nest and unnest through token lists, and to store the loop index for a `FOR-NEXT` loop.

Return stack is used primarily by the Virtual Forth Machine to save return addresses to be processed later. It is also a convenient place to store data temporarily. The return stack can thus be considered as an extension of the parameter stack. However, one must be very careful in using the return stack for temporary storage. The data pushed on the return stack must be popped off before `_UNNEST` is executed. Otherwise, `_UNNEST` will get the wrong address to return to, and the system generally will crash. Since `>R` and `R>` are very dangerous to use, they are designed as `compile-only` commands and you can only use them in the compiling mode.

In setting up a loop, `FOR` compiles `>R`, which pushes a loop index from the parameter stack to the return stack. Inside the `FOR-NEXT` loop, the running index can be recalled by `R@`. `_NEXT` compiles `BL next` with an address after `FOR`. When `next` is executed, it decrements the loop index on the top of the return stack. If the index becomes negative, the loop is terminated; otherwise, `next` jumps back to the command after `FOR`. Therefore, if you have to exit a `FOR-NEXT` loop prematurely, you have to pop the loop index off the return stack first. Otherwise, you will surely crash the system because loop index is definitely not a good address to return to.

R>	Pop a number off the return stack and pushes it on the parameter stack.

```
;   R>        ( -- w )
;     Pop the return stack to the data stack.

      DCD   _CAT-MAPOFFSET
_RFROM      DCB   2
      DCB   "R>"
      ALIGN
RFROM
```

```
       _PUSH
       LDR   R5,[R2],#4
       _NEXT
       ALIGN
```

R@	Copy the top item on the return stack and pushes it on the parameter stack without disturbing the return stack

```
;   R@        ( -- w )
;     Copy top of return stack to the data stack.

       DCD   _RFROM-MAPOFFSET
_RAT   DCB   2
       DCB   "R@"
       ALIGN
RAT
       _PUSH
       LDR   R5,[R2]
       _NEXT
```

>R	Pop a number off the parameter stack and pushes it on the return stack.

```
;   >R        ( w -- )
;     Push the data stack to the return stack.

       DCD   _RAT-MAPOFFSET
_TOR   DCB   COMPO+2
       DCB   ">R"
       ALIGN
TOR
       STR   R5,[R2,#-4]!
       _POP
       _NEXT
       ALIGN
```

3.3.6 Parameter Stack

The parameter stack is the central place where all numerical data are processed, and where parameters are passed among commands. The stack items have to be arranged properly so that they can be retrieved in the Last-In-First-Out (LIFO) manner. When stack items are out of order, they can be rearranged by the stack commands DUP, SWAP, OVER and DROP. There are many other stack commands useful in manipulating stack items, but these four are considered to be the minimum set, or the classic stack operators.

SP@	Return the depth of parameter stack. It is used to determine the depth of the parameter stack, and to detect stack underflow error condition.

```
;   SP@      ( -- a )
;     Push the current data stack pointer.

       DCD   _TOR-MAPOFFSET
_SPAT DCB   3
```

```
        DCB    "SP@"
        ALIGN
SPAT
        _PUSH
        MOV    R5,R1
        _NEXT
```

DROP	Pop the parameter stack, discards the top item on it.

```
;    DROP     ( w -- )
;       Discard top stack item.

        DCD    _SPAT-MAPOFFSET
_DROP   DCB    4
        DCB    "DROP"
        ALIGN
DROP
        _POP
        _NEXT
        ALIGN
```

DUP	Duplicate the top item and pushes it on the parameter stack.

```
;    DUP       ( w -- w w )
;       Duplicate the top stack item.

        DCD    _DROP-MAPOFFSET
_DUPP   DCB    3
        DCB    "DUP"
        ALIGN
DUPP
        _PUSH
        _NEXT
        ALIGN
```

SWAP	Exchange the two top item on the parameter stack.

```
;    SWAP     ( w1 w2 -- w2 w1 )
;       Exchange top two stack items.

        DCD    _DUPP-MAPOFFSET
_SWAP   DCB    4
        DCB    "SWAP"
        ALIGN
SWAP
        LDR    R4,[R1]
        STR    R5,[R1]
        MOV    R5,R4
        _NEXT
```

OVER	Duplicates the second item and pushes it on the parameter stack.

```
;    OVER     ( w1 w2 -- w1 w2 w1 )
;       Copy second stack item to top.
```

```
       DCD    _SWAP-MAPOFFSET
_OVER DCB    4
       DCB    "OVER"
       ALIGN
OVER
       _PUSH
       LDR    R5,[R1,#4]
       _NEXT
```

3.3.7 Logic and Arithmetic Commands

The only primitive command which cares about logic is `?branch`. It tests the top item on the stack. If it is zero, `?branch` will branch to the following address. If it is not zero, `?branch` will ignore the address and execute the command after the branch address. Thus we distinguish two logic values, zero for `false` and non-zero for `true`. Numbers used this way are called logic flags which can be either `true` or `false`. Logic flags thus cause conditional branching in control structures.

0<	Examine the top item `n` on the parameter stack for its negativeness. If `n` is negative, return a -1 for true. If `n` is 0 or positive, return a 0 for false.

```
;   0<         ( n -- t )
;     Return true if n is negative.

       DCD    _OVER-MAPOFFSET
_ZLESS       DCB    2
       DCB    "0<"
       ALIGN
ZLESS
       MOV    R4,#0
       ADD    R5,R4,R5,ASR #32
       _NEXT
       ALIGN
```

AND	Pop top two items on the parameter stack and pushes their bitwise logic `AND` results on the parameter stack.

```
;   AND       ( w w -- w )
;     Bitwise AND.

       DCD    _ZLESS-MAPOFFSET
_ANDD DCB    3
       DCB    "AND"
       ALIGN
ANDD
       LDR    R4,[R1],#4
       AND    R5,R5,R4
       _NEXT
       ALIGN
```

OR	Pop top two items on the parameter stack and pushes their bitwise logic `OR` results on the parameter stack.

```
;   OR         ( w w -- w )
```

```
;     Bitwise inclusive OR.

      DCD   _ANDD-MAPOFFSET
_ORR  DCB   2
      DCB   "OR"
      ALIGN
ORR
      LDR   R4,[R1],#4
      ORR   R5,R5,R4
      _NEXT
      ALIGN
```

XOR	Pop top two items on the parameter stack and pushes their bitwise logic `exclusive OR` results on the parameter stack.

```
;   XOR      ( w w -- w )
;     Bitwise exclusive OR.

      DCD   _ORR-MAPOFFSET
_XORR DCB   3
      DCB   "XOR"
      ALIGN
XORR
      LDR   R4,[R1],#4
      EOR   R5,R5,R4
      _NEXT
      ALIGN
```

UM+	Add top two unsigned number on the parameter stack and replaces them with the unsigned sum of these two numbers and a carry on top of the sum. eForth does not have access to the carry flag in STM32F4 CPU, and `UM+` preserves the carry flag to be used in double integer arithmetic operations. In stm32eforth720, most arithmetic commands are coded in assembly and `UM+` is not used often.

```
;   UM+      ( w w -- w cy )
;     Add two numbers, return the sum and carry flag.

      DCD   _XORR-MAPOFFSET
_UPLUS      DCB   3
      DCB   "UM+"
      ALIGN
UPLUS
      LDR   R4,[R1]
      ADDS  R4,R4,R5
      MOV   R5,#0
      ADC   R5,R5,#0
      STR   R4,[R1]
      _NEXT
```

3.3.8 Extended Primitive Commands

This group of Forth commands are commonly used in writing Forth applications. In the original eForth1 Model they were coded as compound commands for portability. Here in STM32eForth720 implementations, they are coded in assembly language for performance.

RSHIFT	Pop TOS # off parameter stack, and use it as a count to shift the next item `w` right by that many bits.

```
;   RSHIFT   ( w # -- w )
;     Right shift # bits.

      DCD   _UPLUS-MAPOFFSET
_RSHIFT     DCB   6
      DCB   "RSHIFT"
      ALIGN
RSHIFT
      LDR   R4,[R1],#4
      MOV   R5,R4,ASR R5
      _NEXT
      ALIGN
```

LSHIFT	Pop TOS # off parameter stack, and use it as a count to shift the next item `w` left by that many bits.

```
;   LSHIFT   ( w # -- w )
;     Right shift # bits.

      DCD   _RSHIFT-MAPOFFSET
_LSHIFT     DCB   6
      DCB   "LSHIFT"
      ALIGN
LSHIFT
      LDR   R4,[R1],#4
      MOV   R5,R4,LSL R5
      _NEXT
      ALIGN
```

+	Add the top item on the parameter to the second item, and then pops the top item off the parameter stack.

```
;   +  ( w w -- w )
;     Add.

      DCD   _LSHIFT-MAPOFFSET
_PLUS DCB   1
      DCB   "+"
      ALIGN
PLUS
      LDR   R4,[R1],#4
      ADD   R5,R5,R4
      _NEXT
```

-	Subtract the top item on the parameter stack from the second item, and then pops the top item off the parameter stack.

```
;   -  ( w w -- w )
;     Subtract.
```

```
       DCD    _PLUS-MAPOFFSET
_SUBB  DCB    1
       DCB    "-"
       ALIGN
SUBB
       LDR    R4,[R1],#4
       RSB    R5,R5,R4
       _NEXT
       ALIGN
```

*	Multiply the top item on the parameter to the second item, and then pops the top item off the parameter stack.

```
;    *   ( w w -- w )
;      Multiply.

       DCD    _SUBB-MAPOFFSET
_STAR  DCB    1
       DCB    "*"
       ALIGN
STAR
       LDR    R4,[R1],#4
       MUL    R5,R4,R5
       _NEXT
       ALIGN
```

UM*	Unsigned multiplication. Multiply the top item on the parameter to the second item. Return unsigned double integer product.

```
;    UM*       ( w w -- ud )
;      Unsigned multiply.

       DCD    _STAR-MAPOFFSET
_UMSTA        DCB    3
       DCB    "UM*"
       ALIGN
UMSTA
       LDR    R4,[R1]
       UMULL R6,R7,R5,R4
       STR    R6,[R1]
       MOV    R5,R7
       _NEXT
```

M*	Signed multiplication. Multiply the top item on the parameter to the second item. Return signed double integer product.

```
;    M*        ( w w -- d )
;      Unsigned multiply.

       DCD    _UMSTA-MAPOFFSET
_MSTAR        DCB    2
       DCB    "M*"
       ALIGN
MSTAR
```

```
        LDR   R4,[R1]
        SMULL R6,R7,R5,R4
        STR   R6,[R1]
        MOV   R5,R7
        _NEXT
```

1+	Increment TOS by 1.

```
;   1+       ( w -- w+1 )
;     Add 1.

        DCD   _MSTAR-MAPOFFSET
_ONEP DCB   2
        DCB   "1+"
        ALIGN
ONEP
        ADD   R5,R5,#1
        _NEXT
        ALIGN
```

1-	Decrement TOS by 1.

```
;   1-       ( w -- w-1 )
;     Subtract 1.

        DCD   _ONEP-MAPOFFSET
_ONEM DCB   2
        DCB   "1-"
        ALIGN
ONEM
        SUB   R5,R5,#1
        _NEXT
        ALIGN
```

2+	Increment TOS by 2.

```
;   2+       ( w -- w+2 )
;     Add 1.

        DCD   _ONEM-MAPOFFSET
_TWOP DCB   2
        DCB   "2+"
        ALIGN
TWOP
        ADD   R5,R5,#2
        _NEXT
        ALIGN
```

2-	Decrement TOS by 2.

```
;   2-       ( w -- w-2 )
;     Subtract 2.

        DCD   _TWOP-MAPOFFSET
_TWOM DCB   2
```

```
        DCB    "2-"
        ALIGN
TWOM
        SUB    R5,R5,#2
        _NEXT
        ALIGN
```

CELL+	Increment TOS by 4.

```
;   CELL+   ( w -- w+4 )
;     Add 4.

        DCD    _TWOM-MAPOFFSET
_CELLP         DCB    5
        DCB    "CELL+"
        ALIGN
CELLP
        ADD    R5,R5,#4
        _NEXT
        ALIGN
```

CELL-	Decrement TOS by 4.

```
;   CELL-   ( w -- w-4 )
;     Subtract 4.

        DCD    _CELLP-MAPOFFSET
_CELLM         DCB    5
        DCB    "CELL-"
        ALIGN
CELLM
        SUB    R5,R5,#4
        _NEXT
        ALIGN
```

BL	Push a blank or space character (ASCII 32) on parameter stack. BL is often used in parsing out space delimited strings.

```
;   BL      ( -- 32 )
;     Blank (ASCII space).

        DCD    _CELLM-MAPOFFSET
_BLANK         DCB    2
        DCB    "BL"
        ALIGN
BLANK
        _PUSH
        MOV    R5,#32
        _NEXT
        ALIGN
```

CELLS	Multiply TOS by 4.

```
;   CELLS   ( w -- w*4 )
;     Multiply 4.
```

```
        DCD     _BLANK-MAPOFFSET
_CELLS          DCB     5
        DCB     "CELLS"
        ALIGN
CELLS
        MOV     R5,R5,LSL#2
        _NEXT
        ALIGN
```

CELL/	Divide TOS by 4.

```
;   CELL/   ( w -- w*4 )
;     Divide by 4.

        DCD     _CELLS-MAPOFFSET
_CELLSL         DCB     5
        DCB     "CELL/"
        ALIGN
CELLSL
        MOV     R5,R5,ASR#2
        _NEXT
        ALIGN
```

2*	Multiply TOS by 2.

```
;   2*      ( w -- w*2 )
;     Multiply 2.

        DCD     _CELLSL-MAPOFFSET
_TWOST          DCB     2
        DCB     "2*"
        ALIGN
TWOST
        MOV     R5,R5,LSL#1
        _NEXT
        ALIGN
```

2/	Divide TOS by 2.

```
;   2/      ( w -- w/2 )
;     Divide by 2.

        DCD     _TWOST-MAPOFFSET
_TWOSL          DCB     2
        DCB     "2/"
        ALIGN
TWOSL
        MOV     R5,R5,ASR#1
        _NEXT
        ALIGN
```

?DUP	Duplicate the top item on the parameter stack if it is non-zero.

```
;   ?DUP    ( w -- w w | 0 )
```

```
;     Conditional duplicate.

      DCD   _TWOSL-MAPOFFSET
_QDUP DCB   4
      DCB   "?DUP"
      ALIGN
QDUP
      MOVS  R4,R5
      STRNE R5,[R1,#-4]!
      _NEXT
      ALIGN
```

ROT	Rotate the top three items on the parameter stack. The third item `w1` is pulled out to the top. The second item w2 is pushed down to the third item, and the top item `w3` is pushed down to be the second item.

```
;   ROT     ( w1 w2 w3 -- w2 w3 w1 )
;     Rotate top 3 items.

      DCD   _QDUP-MAPOFFSET
_ROT  DCB   3
      DCB   "ROT"
      ALIGN
ROT
      LDR   R4,[R1]
      STR   R5,[R1]
      LDR   R5,[R1,#4]
      STR   R4,[R1,#4]
      _NEXT
      ALIGN
```

2DROP	Discard the top two items on the parameter stack.

```
;   2DROP   ( w1 w2 -- )
;     Drop top 2 items.

      DCD   _ROT-MAPOFFSET
_DDROP      DCB   5
      DCB   "2DROP"
      ALIGN
DDROP
      _POP
      _POP
      _NEXT
      ALIGN
```

2DUP	Duplicate the top two items on the parameter stack.

```
;   2DUP    ( w1 w2 -- w1 w2 w1 w2 )
;     Duplicate top 2 items.

      DCD   _DDROP-MAPOFFSET
_DDUP DCB   4
      DCB   "2DUP"
      ALIGN
```

```
DDUP
      LDR   R4,[R1]
      STR   R5,[R1,#-4]!
      STR   R4,[R1,#-4]!
      _NEXT
```

D+	Add two double integers and return a double integer sum.

```
;   D+      ( d1 d2 -- d3 )
;     Add top 2 double numbers.

      DCD   _DDUP-MAPOFFSET
_DPLUS      DCB   2
      DCB   "D+"
      ALIGN
DPLUS
      LDR   R4,[R1],#4
      LDR   R6,[R1],#4
      LDR   R7,[R1]
      ADDS  R4,R4,R7
      STR   R4,[R1]
      ADC   R5,R5,R6
      _NEXT
```

NOT	Invert each individual bit in the top item on the parameter stack. It is often called 1's complement operation.

```
;   NOT      ( w -- !w )
;     1"s complement.

      DCD   _DPLUS-MAPOFFSET
_INVER      DCB   3
      DCB   "NOT"
      ALIGN
INVER
      MVN   R5,R5
      _NEXT
      ALIGN
```

NEGATE	Negate the top item on the parameter stack. It is often called 2's complement operation.

```
;   NEGATE  ( w -- -w )
;     2's complement.

      DCD   _INVER-MAPOFFSET
_NEGAT      DCB   6
      DCB   "NEGATE"
      ALIGN
NEGAT
      RSB   R5,R5,#0
      _NEXT
      ALIGN
```

ABS	Replace the top item on the parameter stack with its absolute value.

```
;   ABS      ( w -- |w| )
;     Absolute.

      DCD   _NEGAT-MAPOFFSET
_ABSS DCB   3
      DCB   "ABS"
      ALIGN
ABSS
      TST   R5,#0x80000000
      RSBNE   R5,R5,#0
      _NEXT
      ALIGN
```

=	Compare top two items on the parameter stack. If they are equal, replace these two items with a true flag; otherwise, replace them with a false flag.

```
;   =  ( w w -- t )
;     Equal?

      DCD   _ABSS-MAPOFFSET
_EQUAL      DCB   1
      DCB   "="
      ALIGN
EQUAL
      LDR   R4,[R1],#4
      CMPS  R5,R4
      MVNEQ R5,#0
      MOVNE R5,#0
      _NEXT
```

U<	Compare two unsigned numbers on the top of the parameter stack. If the top item is less than the second item in unsigned comparison, replace these two items with a true flag; otherwise, replace them with a false flag. .

```
;   U<       ( w w -- t )
;     Unsigned equal?

      DCD   _EQUAL-MAPOFFSET
_ULESS      DCB   2
      DCB   "U<"
      ALIGN
ULESS
      LDR   R4,[R1],#4
      CMPS  R4,R5
      MVNCC R5,#0
      MOVCS R5,#0
      _NEXT
```

<	Compare two signed numbers on the top of the parameter stack. If the top item is less than the second item in signed comparison, replace these two items with a true flag; otherwise, replace them with a false flag.

```
;   < ( w w -- t )
;     Less?
```

```
        DCD    _ULESS-MAPOFFSET
_LESS DCB    1
      DCB    "<"
      ALIGN
LESS
      LDR    R4,[R1],#4
      CMPS   R4,R5
      MVNLT  R5,#0
      MOVGE  R5,#0
      _NEXT
```

>	Compare two signed numbers on the top of the parameter stack. If the top item is greater than the second item in signed comparison, replace these two items with a true flag; otherwise, replace them with a false flag.

```
;   > ( w w -- t )
;     greater?

      DCD    _LESS-MAPOFFSET
_GREAT       DCB    1
      DCB    ">"
      ALIGN
GREAT
      LDR    R4,[R1],#4
      CMPS   R4,R5
      MVNGT  R5,#0
      MOVLE  R5,#0
      _NEXT
```

MAX	Retain the larger of the top two items on the parameter stack. Both numbers are assumed to be signed integers.

```
;   MAX      ( w w -- max )
;     Leave maximum.

      DCD    _GREAT-MAPOFFSET
_MAX  DCB    3
      DCB    "MAX"
      ALIGN
MAX
      LDR    R4,[R1],#4
      CMPS   R4,R5
      MOVGT  R5,R4
      _NEXT
```

MIN	Retain the smaller of the top two items on the parameter stack. Both numbers are assumed to be signed integers.

```
;   MIN      ( w w -- min )
;     Leave minimum.

      DCD    _MAX-MAPOFFSET
_MIN  DCB    3
      DCB    "MIN"
      ALIGN
```

```
MIN
      LDR   R4,[R1],#4
      CMPS  R4,R5
      MOVLT R5,R4
      _NEXT
```

+!	Add the second item on the parameter stack `w` to the cell addressed by `a`, the top item on the stack.

```
;   +!       ( w a -- )
;     Add to memory.

      DCD   _MIN-MAPOFFSET
_PSTOR      DCB   2
      DCB   "+!"
      ALIGN
PSTOR
      LDR   R4,[R1],#4
      LDR   R6,[R5]
      ADD   R6,R6,R4
      STR   R6,[R5]
      _POP
      _NEXT
```

2!	Store a double integer `d` into memory at `addr`.

```
;   2!       ( d addr -- )
;     Store double number.

      DCD   _PSTOR-MAPOFFSET
_DSTOR      DCB   2
      DCB   "2!"
      ALIGN
DSTOR
      LDR   R4,[R1],#4
      LDR   R6,[R1],#4
      STR   R4,[R5],#4
      STR   R6,[R5]
      _POP
      _NEXT
```

2@	Fetch a double integer `d` from memory at `addr`.

```
;   2@       ( addr -- d )
;     Fetch double number.

      DCD   _DSTOR-MAPOFFSET
_DAT  DCB   2
      DCB   "2@"
      ALIGN
DAT
      LDR   R4,[R5,#4]
      STR   R4,[R1,#-4]!
      LDR   R5,[R5]
      _NEXT
```

```
        ALIGN
```

COUNT	Fetch one byte `c` from memory pointed to by the address `b` on the top of the parameter stack. This address is incremented by 1, and the byte just read is pushed on the stack. `COUNT` is designed to get the count byte at the beginning of a counted string, and returns the address of the first byte in the string and the length of this string. However, it is often used in a loop to read consecutive bytes in a byte array.

```
;   COUNT   ( b -- b+1 c )
;     Fetch length of string.

      DCD   _DAT-MAPOFFSET
_COUNT      DCB   5
      DCB   "COUNT"
      ALIGN
COUNT
      LDRB  R4,[R5],#1
      _PUSH
      MOV   R5,R4
      _NEXT
```

DNEGATE	Negate the top two items on the parameter stack, as a 64-bit double integer.

```
;   DNEGATE ( d -- -d )
;     Negate double number.

      DCD   _COUNT-MAPOFFSET
_DNEGA      DCB   7
      DCB   "DNEGATE"
      ALIGN
DNEGA
      LDR   R4,[R1]
      SUB   R8,R8,R8
      SUBS  R4,R6,R4
      SBC   R5,R6,R5
      STR   R4,[R1]
      _NEXT
```

doVAR	Fetch the address in LR register after the `BL doVAR` instruction and pushes it on the parameter stack. `BL doVAR` instruction and the value after it form the code field of all variable commands. The address in LR has the lowest bit b0 set as a THUMB2 instruction. This bit must be cleared to be a correct address.

```
;***************************************************************************
; System and user variables

;   doVAR   ( -- a )
;     Run time routine for VARIABLE and CREATE.

;     DCD   _DNEGA-MAPOFFSET
;_DOVAR     DCB  COMPO+5
;     DCB   "doVAR"
;     ALIGN
```

```
DOVAR
      _PUSH
      SUB   R5,LR,#1          ; CLEAR B0
      _UNNEST
      ALIGN
```

doCON	Fetch a value stored after the `BL doCON` instruction, as pointed to by LR register, and pushes it on the parameter stack. `BL doCON` instruction and the value after it form the code field of all constant commands.

```
;   doCON   ( -- a )
;     Run time r outine for CONSTANT.

;     DCD   _DOVAR-MAPOFFSET
;_DOCON     DCB  COMPO+5
;     DCB   "doCON"
;     ALIGN
DOCON
      _PUSH
      LDR   R5,[LR,#-1] ; clear b0
      _UNNEST
```

3.3.9 User Variables Commands

In stm32eForth720, all user variables used by the system are merged together and are sometimes called system variables. They are stored in a memory array starting from location 0xFF00. They are initialized by copying a table of initial values starting at 0xC0. They are variables and memory area pointers eForth needs to manage the interpreter and compiler.

The CPU register R3 is used to point to this user variable array, allowing easy and fast access to these user variables.

Variable	Address	Function
'BOOT	FF04	Execution vector to start application command.
BASE	FF08	Radix base for numeric conversion.
tmp	FF0C	Scratch pad.
SPAN	FF10	Number of characters received by `EXPECT`.
>IN	FF14	Input buffer character pointer used by text interpreter.
#TIB	FF18	Number of characters in input buffer.
'TIB	FF1C	Address of Terminal Input Buffer.
'EVAL	FF20	Execution vector switching between `$INTERPRET` and `$COMPILE`.
HLD	FF24	Pointer to a buffer holding next digit for numeric conversion.
CONTEXT	FF28	Vocabulary array pointing to last name fields of dictionary.
CP	FF2C	Pointer to top of dictionary, the first available flash memory location to compile new command
DP	FF30	Pointer to the first available RAM memory location.

LAST	FF34	Pointer to name field of last command in dictionary.

```
;   'BOOT    ( -- a )
;     Applicarion.

      DCD   _DNEGA-MAPOFFSET
_TBOOT      DCB   5
      DCB   "'BOOT"
      ALIGN
TBOOT
      _PUSH
      ADD   R5,R3,#4
      _NEXT
      ALIGN

;   BASE    ( -- a )
;     Storage of the radix base for numeric I/O.

      DCD   _TBOOT-MAPOFFSET
_BASE DCB   4
      DCB   "BASE"
      ALIGN
BASE
      _PUSH
      ADD   R5,R3,#8
      _NEXT
      ALIGN

;   tmp      ( -- a )
;     A temporary storage location used in parse and find.

;     DCD   _BASE-MAPOFFSET
;_TEMP      DCB   COMPO+3
;     DCB   "tmp"
;     ALIGN
TEMP
      _PUSH
      ADD   R5,R3,#12
      _NEXT
      ALIGN

;   SPAN    ( -- a )
;     Hold character count received by EXPECT.

      DCD   _BASE-MAPOFFSET
_SPAN DCB   4
      DCB   "SPAN"
      ALIGN
SPAN
      _PUSH
      ADD   R5,R3,#16
      _NEXT
      ALIGN

;   >IN      ( -- a )
;     Hold the character pointer while parsing input stream.
```

```
	DCD	_SPAN-MAPOFFSET
_INN	DCB	3
	DCB	">IN"
	ALIGN
INN
	_PUSH
	ADD	R5,R3,#20
	_NEXT
	ALIGN

;	#TIB	( -- a )
;	Hold the current count and address of the terminal input buffer.

	DCD	_INN-MAPOFFSET
_NTIB DCB	4
	DCB	"#TIB"
	ALIGN
NTIB
	_PUSH
	ADD	R5,R3,#24
	_NEXT
	ALIGN

;	'EVAL	( -- a )
;	Execution vector of EVAL.

	DCD	_NTIB-MAPOFFSET
_TEVAL		DCB	5
	DCB	"'EVAL"
	ALIGN
TEVAL
	_PUSH
	ADD	R5,R3,#32
	_NEXT
	ALIGN

;	HLD	 ( -- a )
;	Hold a pointer in building a numeric output string.

	DCD	_TEVAL-MAPOFFSET
_HLD	DCB	3
	DCB	"HLD"
	ALIGN
HLD
	_PUSH
	ADD	R5,R3,#36
	_NEXT
	ALIGN

;	CONTEXT ( -- a )
;	A area to specify vocabulary search order.

	DCD	_HLD-MAPOFFSET
_CNTXT		DCB	7
	DCB	"CONTEXT"
	ALIGN
```

```
CNTXT
CRRNT
      _PUSH
      ADD   R5,R3,#40
      _NEXT
      ALIGN

;   CP      ( -- a )
;     Point to top name in vocabulary.

      DCD   _CNTXT-MAPOFFSET
_CP   DCB   2
      DCB   "CP"
      ALIGN
CPP
      _PUSH
      ADD   R5,R3,#44
      _NEXT
      ALIGN

;   LAST    ( -- a )
;     Point to the last name in the name dictionary.

      DCD   _CP-MAPOFFSET
_LAST DCB   4
      DCB   "LAST"
      ALIGN
LAST
      _PUSH
      ADD   R5,R3,#52
      _NEXT
      ALIGN
```

3.3.10 Common Functions

These commands are coded as compound command. They contain logic structures which are difficult to express in assembly code.

WITHIN	Check whether the third item `u` on the parameter stack is within the range as specified by the top two numbers on the parameter stack. The range is inclusive as to the lower limit `ul` and exclusive to the upper limit `uh`. If the third item is within range, a true flag is returned on the parameter stack, replacing all three items. Otherwise, a false flag is returned. All numbers are assumed to be signed integers.

```
;***********************************************************************
; Common functions

;   WITHIN  ( u ul uh -- t )
;     Return true if u is within the range of ul and uh.

      DCD   _LAST-MAPOFFSET
_WITHI      DCB   6
      DCB   "WITHIN"
      ALIGN
```

```
WITHI
      _NEST
      BL    OVER
      BL    SUBB
      BL    TOR
      BL    SUBB
      BL    RFROM
      BL    ULESS
      _UNNEST
```

UM/MOD	Divide an unsigned double integer `udl-udh` by an unsigned single integer `u`. It returns an unsigned remainder `ur` and an unsigned quotient `uq` on the parameter stack. Division is carried out similar to long hand division.

```
; Divide

;   UM/MOD  ( udl udh u -- ur uq )
;     Unsigned divide of a double by a single. Return mod and quotient.

      DCD   _WITHI-MAPOFFSET
_UMMOD      DCB   6
      DCB   "UM/MOD"
      ALIGN
UMMOD
      MOV   R7,#1
      LDR   R4,[R1],#4
      LDR   R6,[R1]
UMMOD0      ADDS  R6,R6,R6
      ADCS  R4,R4,R4
      BCC   UMMOD1
      SUB   R4,R4,R5
      ADD   R6,R6,#1
      B UMMOD2
UMMOD1      SUBS  R4,R4,R5
      ADDCS R6,R6,#1
      BCS   UMMOD2
      ADD   R4,R4,R5
UMMOD2      ADDS  R7,R7,R7
      BCC   UMMOD0
      MOV   R5,R6
      STR   R4,[R1]
      _NEXT
      ALIGN
```

M/MOD	Divide a signed double integer `d` by a signed single integer `n`. It returns signed remainder `r` and signed quotient `q` on the parameter stack. The signed division is floored towards negative infinity.

```
;   M/MOD   ( d n -- r q )
;     Signed floored divide of double by single. Return mod and quotient.

      DCD   _UMMOD-MAPOFFSET
_MSMOD      DCB  5
      DCB   "M/MOD"
      ALIGN
```

```
MSMOD
      _NEST
      BL    DUPP
      BL    ZLESS
      BL    DUPP
      BL    TOR
      BL    QBRAN
      DCD   MMOD1-MAPOFFSET
      BL    NEGAT
      BL    TOR
      BL    DNEGA
      BL    RFROM
MMOD1   BL  TOR
      BL    DUPP
      BL    ZLESS
      BL    QBRAN
      DCD   MMOD2-MAPOFFSET
      BL    RAT
      BL    PLUS
MMOD2   BL  RFROM
      BL    UMMOD
      BL    RFROM
      BL    QBRAN
      DCD   MMOD3-MAPOFFSET
      BL    SWAP
      BL    NEGAT
      BL    SWAP
MMOD3
      _UNNEST
```

/MOD	Divide a signed single integer by a signed integer. It replaces these two items with signed remainder and quotient.

```
;   /MOD    ( n n -- r q )
;     Signed divide. Return mod and quotient.

      DCD   _MSMOD-MAPOFFSET
_SLMOD      DCB   4
      DCB   "/MOD"
      ALIGN
SLMOD
      _NEST
      BL    OVER
      BL    ZLESS
      BL    SWAP
      BL    MSMOD
      _UNNEST
```

MOD	Divide a signed single integer by a signed integer. It replaces these two items with a signed remainder.

```
;   MOD      ( n n -- r )
;     Signed divide. Return mod only.

      DCD   _SLMOD-MAPOFFSET
_MODD DCB  3
```

```
        DCB   "MOD"
        ALIGN
MODD
        _NEST
        BL    SLMOD
        BL    DROP
        _UNNEST
```

/	Divide a signed single integer by a signed integer. It replaces these two items with a signed quotient.

```
;   /    ( n n -- q )
;     Signed divide. Return quotient only.

        DCD   _MODD-MAPOFFSET
_SLASH        DCB  1
        DCB   "/"
        ALIGN
SLASH
        _NEST
        BL    SLMOD
        BL    SWAP
        BL    DROP
        _UNNEST
```

3.3.11 Scaling, Multiply-Divide

*/MOD	Multiply the signed integers `n1` and `n2`, and then divides the double integer product by `n3`. It in fact is scaling `n1` by `n2/n3`. It returns both the remainder and the quotient. The intermediate product is kept as double integer, and scaling has minimal round off error. This scaling operation allows high precision integer arithmetic operations equivalent to floating point operations.

```
;   */MOD   ( n1 n2 n3 -- r q )
;     Multiply n1 and n2, then divide by n3. Return mod and quotient.

        DCD   _SLASH-MAPOFFSET
_SSMOD        DCB  5
        DCB   "*/MOD"
        ALIGN
SSMOD
        _NEST
        BL    TOR
        BL    MSTAR
        BL    RFROM
        BL    MSMOD
        _UNNEST
```

*/	Multiply the signed integers `n1` and `n2`, and then divides the double integer product by `n3`. It returns only the quotient. Scaling `n1` by `n2/n3`.

```
;   */        ( n1 n2 n3 -- q )
;     Multiply n1 by n2, then divide by n3. Return quotient only.
```

```
      DCD   _SSMOD-MAPOFFSET
_STASL      DCB  2
      DCB   "*/"
      ALIGN
STASL
      _NEST
      BL    SSMOD
      BL    SWAP
      BL    DROP
      _UNNEST
```

3.3.12 Miscellaneous Commands

ALIGNED	Modify the byte address on top of the parameter stack so that it points to the next 32-bit word boundary.

```
;***************************************************************************
; Miscellaneous

;   ALIGNED ( b -- a )
;     Align address to the cell boundary.

      DCD   _STASL-MAPOFFSET
_ALGND      DCB   7
      DCB   "ALIGNED"
      ALIGN
ALGND
      ADD   R5,R5,#3
      MVN   R4,#3
      AND   R5,R5,R4
      _NEXT
      ALIGN
```

>CHAR	Convert a non-printable character to a harmless underscore character(ASCII 95). As stm32eForth is designed to communicate with a terminal through a serial I/O device, it is important that stm32eForth will not emit control characters to the host and thereby causes unexpected behavior on the terminal. >CHAR thus filters the characters before they are sent out by EMIT.

```
;   >CHAR   ( c -- c )
;     Filter non-printing characters.

      DCD   _ALGND-MAPOFFSET
_TCHAR      DCB  5
      DCB   ">CHAR"
      ALIGN
TCHAR
      _NEST
      _DOLIT
      DCD   0x7F
      BL    ANDD
      BL    DUPP  ;mask msb
      BL    BLANK
      _DOLIT
      DCD   127
```

```
        BL    WITHI ;check for printable
        BL    INVER
        BL    QBRAN
        DCD   TCHA1-MAPOFFSET
        BL    DROP
        _DOLIT
        DCD   '_'   ;replace non-printables
TCHA1
          _UNNEST
```

DEPTH	Push the number of items currently on the parameter stack to the top of the stack.

```
;   DEPTH   ( -- n )
;     Return the depth of the data stack.

        DCD   _TCHAR-MAPOFFSET
_DEPTH        DCB  5
        DCB   "DEPTH"
        ALIGN
DEPTH
        _PUSH
        MOVW  R5,#0XFE00
;       MOVT  R5,#0X2000
        SUB   R5,R5,R1
        ASR   R5,R5,#2
        SUB   R5,R5,#1
        _NEXT
        ALIGN
```

PICK	Pop the number `+n` off the parameter stack and replaces it with the n'th item on the parameter stack. The number `+n` is 0-based; i.e., the top item is number 0, the next item is number 1, etc. Therefore, `0 PICK` is equivalent to `DUP`, and `1 PICK` is equivalent to `OVER`.

```
;   PICK    ( ... +n -- ... w )
;     Copy the nth stack item to tos.

        DCD   _DEPTH-MAPOFFSET
_PICK DCB  4
        DCB   "PICK"
        ALIGN
PICK
        _NEST
        BL    ONEP
        BL    CELLS
        BL    SPAT
        BL    PLUS
        BL    AT
        _UNNEST
```

3.3.13 Memory Array Commands

A memory array is generally specified by its starting address and its length in bytes. In a count string, the first byte is a count byte, specifying the number of bytes in the following string. String literals in compound commands and the name strings in the headers of command records are all represented by count strings. Following commands are useful in accessing memory arrays used by eForth.

HERE	Push the address of the first free memory above the eForth dictionary. The text interpreter stores at `HERE` a string parsed out of the Terminal Input Buffer and then searches the dictionary for a command with this name. The compiler builds a header at `HERE` for a new command. It is generally referred to as the word buffer in Forth terminology.

```
;**************************************************************************
; Memory access

;   HERE    ( -- a )
;     Return the top of the code dictionary.

      DCD   _PICK-MAPOFFSET
_HERE DCB  4
      DCB   "HERE"
      ALIGN
HERE
      _NEST
      BL    CPP
      BL    AT
      _UNNEST
```

PAD	Push on the parameter stack the address of the text buffer where numbers to be output are constructed and text strings are stored temporarily. It is 80 bytes above `HERE`, and floats above the dictionary. It is always available to store things temporarily. It moves when you defined a new command. The area below `PAD` is used to build numeric strings in ASCII characters for output to the terminal. A numeric string is built backwards from `PAD`, the least significant digit is laid down first, and the area below `PAD` is often referred to as number buffer.

```
;   PAD      ( -- a )
;     Return the address of a temporary buffer.

      DCD   _HERE-MAPOFFSET
_PAD  DCB  3
      DCB   "PAD"
      ALIGN
PAD
      _NEST
      BL    HERE
      ADD   R5,R5,#80
      _UNNEST
```

TIB	Push the address of the Terminal Input Buffer on the parameter stack. Terminal Input Buffer stores a line of text from the serial I/O input device. Forth text interpreter then processes or interprets this line of text. In stm32eforth720, `TIB` starts at 0xFE00, at the top of the 64 KB RAM space. It grows up from 0xFE00, and the return stack grows down from 0xFF00. They generaaly do not bother each other.

```
;   TIB      ( -- a )
```

```
;     Return the address of the terminal input buffer.

      DCD   _PAD-MAPOFFSET
_TIB  DCB  3
      DCB   "TIB"
      ALIGN
TIB
      _PUSH
      MOVW  R5,#0xFE00
      _NEXT
      ALIGN
```

@EXECUTE	Fetch a code field address of a command which is stored in the address `a` on the top of the parameter stack, and jumps to it to execute this command. It is used extensively to execute vectored commands stored in memory. The behavior of a vectored command can be changed dynamically at the run time.

```
;   @EXECUTE      ( a -- )
;     Execute vector stored in address a.

      DCD   _TIB-MAPOFFSET
_ATEXE      DCB   8
      DCB   "@EXECUTE"
      ALIGN
ATEXE
      MOVS  R4,R5
      _POP
      LDR   R4,[R4]
      BXNE  R4
      _NEXT
      ALIGN
```

CMOVE	Copy a byte array from one location to another in memory. The top three item on the parameter stack are the source address `b1`, the destination address `b2`, and the number of bytes to be copied `u`.

```
;   CMOVE   ( b1 b2 u -- )
;     Copy u bytes from b1 to b2.

      DCD   _ATEXE-MAPOFFSET
_CMOVE      DCB   5
      DCB   "CMOVE"
      ALIGN
CMOVE
      LDR   R6,[R1],#4
      LDR   R7,[R1],#4
      B CMOV1
CMOV0 LDRB  R4,[R7],#1
      STRB  R4,[R6],#1
CMOV1 MOVS  R5,R5
      BEQ   CMOV2
      SUB   R5,R5,#1
      B CMOV0
CMOV2
      _POP
```

```
        _NEXT
        ALIGN
```

MOVE	Copy a word array from one location to another in memory. The top three item on the parameter stack are the source address `a1`, the destination address `a2`, and the number of bytes to be copied `u`. Addresses are on word boundary, and number of bytes moved must be divisible by 4.

```
;    MOVE    ( a1 a2 u -- )
;       Copy u words from a1 to a2.

        DCD    _CMOVE-MAPOFFSET
_MOVE   DCB    4
        DCB    "MOVE"
        ALIGN
MOVE    AND    R5,R5,#-4
        LDR    R6,[R1],#4
        LDR    R7,[R1],#4
        B MOVE1
MOVE0   LDR    R4,[R7],#4
        STR    R4,[R6],#4
MOVE1   MOVS   R5,R5
        BEQ    MOVE2
        SUB    R5,R5,#4
        B MOVE0
MOVE2
        _POP
        _NEXT
        ALIGN
```

FILL	Fill a memory array with the same byte. The top three items on the parameter stack are the address of the array `b`, the length of the array in bytes `u`, and the byte value to be filled into this array `c`.

```
;    FILL    ( b u c -- )
;       Fill u bytes of character c to area beginning at b.

        DCD    _MOVE-MAPOFFSET
_FILL   DCB    4
        DCB    "FILL"
        ALIGN
FILL
        LDR    R6,[R1],#4
        LDR    R7,[R1],#4
FILL0   B FILL1
        MOV    R5,R5
FILL1   STRB   R5,[R7],#1
        MOVS   R6,R6
        BEQ    FILL2
        SUB    R6,R6,#1
        B FILL0
FILL2
        _POP
        _NEXT
```

PACK$	Pack a byte string at address b of length u to form a counted string at cell address a. Null filled to cell boundary. This is how a name field is constructed.

```
;   PACK$   ( b u a -- a )
;     Build a counted string with u characters from b. Null fill.

      DCD   _FILL-MAPOFFSET
_PACKS      DCB  5
      DCB   "PACK$$"
      ALIGN
PACKS
      _NEST
      BL    ALGND
      BL    DUPP
      BL    TOR                 ;strings only on cell boundary
      BL    OVER
      BL    PLUS
      BL    ONEP
      _DOLIT
      DCD   0xFFFFFFFC
      BL    ANDD                ;count mod cell
      _DOLIT
      DCD   0
      BL    SWAP
      BL    STORE               ;null fill cell
      BL    RAT
      BL    DDUP
      BL    CSTOR
      BL    ONEP                ;save count
      BL    SWAP
      BL    CMOVE
      BL    RFROM
      _UNNEST                   ;move string
```

3.4 Text Interpreter

The text interpreter is actually the Forth operating system itself. It performs these tasks:

Step 1. Accept one line of text from the terminal.
Step 2. Parse out a space delimited name string.
Step 3. Search the dictionary for a command of this name.
Step 4. If it is a command, execute it. Go to Step 8.
Step 5. If it is not a command, convert it to a number.
Step 6. If it is a number, push it on parameter stack. Go to Step 8.
Step 7. If it is not a number, abort. Go back to step 1.
Step 8. If the text line is not exhausted, go back to step 2.
Step 9. If the text line is exhausted, go back to Step 1.

It looks very complicated. Yes, it is complicated and we will discuss all the supporting commands leading to the text interpreter. But, it is an operating system! Have you ever read the source code of an operating system? Very few people did. Very few people wrote operating systems. Here I will show you how to write this Forth operating system. We will do parsing, command searching, number conversion, terminal input, terminal output, command execution, and everything else that's necessary.

Need to see a flow chart? You had seen it already. It was in the figure on COLD I showed you in Section 3.2.2 on the reset handler. It was not a flow chart you used to see, but it is a flow chart nonetheless. It not only shows the text interpreter. It also shows the Forth compiler as well.

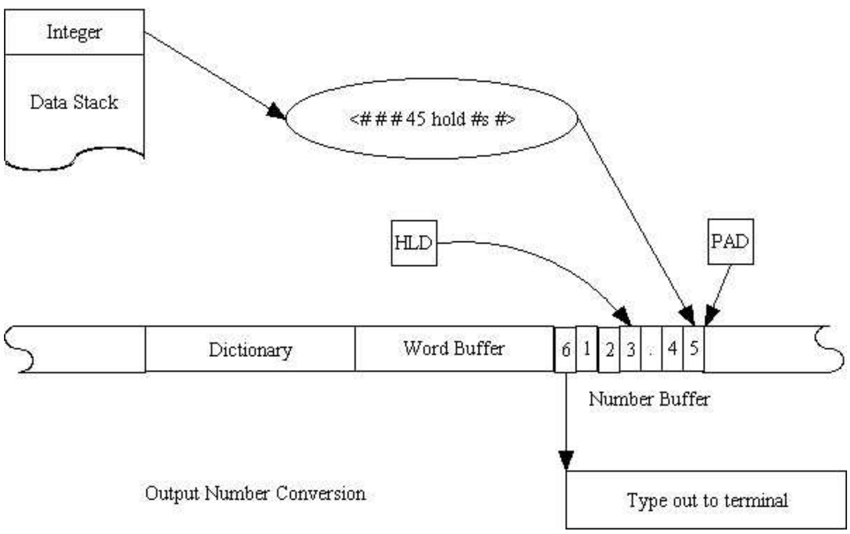
3.4.1 Numeric Output

Forth is interesting in its special capabilities in handling numbers across a man-machine interface. It recognizes that machines and humans prefer very different representations of numbers. Machines prefer binary representation, but humans prefer decimal Arabic representation. However, depending on circumstances, you may want numbers to be represented in other radices, like hexadecimal, octal, and sometimes binary.

Forth solves this problem of internal (machine) versus external (human) number representations by insisting that all numbers are represented in binary form in CPU and memory. Only when numbers are imported or exported for human consumption are they converted to external ASCII representation. The radix of the external representation is stored in user variable BASE. You can select any reasonable radix in BASE, up to perhaps 72, limited by available printable characters in the ASCII character set.

The output number string is built below the PAD buffer in memory. The least significant digit is extracted from the integer on the top of the parameter stack by dividing it by the current radix in BASE. The digit thus extracted is added to the output string backwards from PAD to the low memory. The conversion is terminated when the integer is divided to zero. The address and length of the number string are made available by #> for output.

An output number conversion is initiated by <# and terminated by #>. Between them, # converts one digit at a time, #S converts all the digits, while HOLD and SIGN inserts special characters into the string under construction. This set of commands is very versatile and can handle many different output formats. The following figure shows how a number on parameter stack is converted to an output string.



DIGIT	Convert an integer digit `u` to the corresponding ASCII character `c`.

```
;***************************************************************************
; Numeric output, single precision

;   DIGIT   ( u -- c )
;     Convert digit u to a character.

      DCD   _PACKS-MAPOFFSET
_DIGIT      DCB  5
      DCB   "DIGIT"
      ALIGN
DIGIT
      _NEST
      _DOLIT
      DCD   9
      BL    OVER
      BL    LESS
      AND   R5,R5,#7
      BL    PLUS
      ADD   R5,R5,#'0'
      _UNNEST
```

EXTRACT	Extract the least significant digit from a number `n` on the top of the parameter stack. `n` is divided by the radix `base` and the extracted digit is converted to its ASCII character `c` which is pushed on the top of new `n`.

```
;   EXTRACT ( n base -- n c )
;     Extract the least significant digit from n.

      DCD   _DIGIT-MAPOFFSET
_EXTRC      DCB  7
      DCB   "EXTRACT"
      ALIGN
EXTRC
      _NEST
      _DOLIT
      DCD   0
      BL    SWAP
      BL    UMMOD
      BL    SWAP
      BL    DIGIT
      _UNNEST
```

<#	Initiate the output number conversion process by storing `PAD` buffer address into user variable `HLD`, which points to a location the next numeric digit will be stored.

```
;   <#        ( -- )
;     Initiate the numeric output process.

      DCD   _EXTRC-MAPOFFSET
_BDIGS      DCB  2
      DCB   "<#"
```

```
      ALIGN
BDIGS
      _NEST
      BL    PAD
      BL    HLD
      BL    STORE
      _UNNEST
```

HOLD	Append an ASCII character `c` whose code is on the top of the parameter stack, to the numeric out put string at `HLD`. `HLD` is decremented to receive the next digit.

```
;   HOLD    ( c -- )
;     Insert a character into the numeric output string.

      DCD   _BDIGS-MAPOFFSET
_HOLD DCB  4
      DCB   "HOLD"
      ALIGN
HOLD
      _NEST
      BL    HLD
      BL    AT
      BL    ONEM
      BL    DUPP
      BL    HLD
      BL    STORE
      BL    CSTOR
      _UNNEST
```

#	Extract one digit from integer `u` on the top of the parameter stack, according to radix in user variable `BASE`, and append it to output numeric string.

```
;   #    ( u -- u )
;     Extract one digit from u and append the digit to output string.

      DCD   _HOLD-MAPOFFSET
_DIG  DCB  1
      DCB   "#"
      ALIGN
DIG
      _NEST
      BL    BASE
      BL    AT
      BL    EXTRC
      BL    HOLD
      _UNNEST
```

#S	Extract all digits in `u` to output string until the integer `u` on the top of the parameter stack is divided to 0.

```
;   #S       ( u -- 0 )
;     Convert u until all digits are added to the output string.

      DCD   _DIG-MAPOFFSET
_DIGS DCB  2
```

```
      DCB   "#S"
      ALIGN
DIGS
      _NEST
DIGS1   BL  DIG
      BL    DUPP
      BL    QBRAN
      DCD   DIGS2-MAPOFFSET
      B     DIGS1
DIGS2
        _UNNEST
      ALIGN
```

SIGN	Insert a - sign into the numeric output string if the integer on the top of the parameter stack is negative.

```
;   SIGN    ( n -- )
;     Add a minus sign to the numeric output string.

      DCD   _DIGS-MAPOFFSET
_SIGN DCB  4
      DCB   "SIGN"
      ALIGN
SIGN
      _NEST
      BL    ZLESS
      BL    QBRAN
      DCD   SIGN1-MAPOFFSET
      _DOLIT
      DCD   '-'
      BL    HOLD
SIGN1
        _UNNEST
```

#>	Terminate the numeric conversion and pushes the address b and length of output numeric string u on the parameter stack.

```
;   #>        ( w -- b u )
;     Prepare the output string to be TYPE'd.

      DCD   _SIGN-MAPOFFSET
_EDIGS      DCB  2
      DCB   "#>"
      ALIGN
EDIGS
      _NEST
      BL    DROP
      BL    HLD
      BL    AT
      BL    PAD
      BL    OVER
      BL    SUBB
      _UNNEST
```

str	Convert a signed integer n on the top of the parameter stack to a numeric

	output string at address b with u digits.

```
;   str      ( n -- b u )
;     Convert a signed integer to a numeric string.

;     DCD   _EDIGS-MAPOFFSET
;_STRR       DCB  3
;     DCB   "str"
;     ALIGN
STRR
      _NEST
      BL    DUPP
      BL    TOR
      BL    ABSS
      BL    BDIGS
      BL    DIGS
      BL    RFROM
      BL    SIGN
      BL    EDIGS
      _UNNEST
```

HEX	Set numeric conversion radix to 16 for hexadecimal conversions.

```
;   HEX      ( -- )
;     Use radix 16 as base for numeric conversions.

      DCD   _EDIGS-MAPOFFSET
_HEX  DCB  3
      DCB   "HEX"
      ALIGN
HEX
      _NEST
      _DOLIT
      DCD   16
      BL    BASE
      BL    STORE
      _UNNEST
```

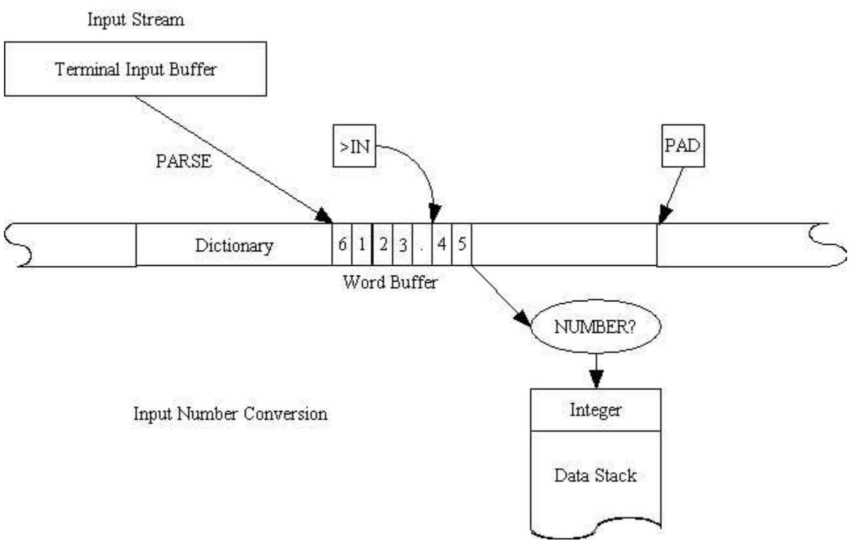
DECIMAL	Set numeric conversion radix to 10 for decimal conversions.

```
;   DECIMAL ( -- )
;     Use radix 10 as base for numeric conversions.

      DCD   _HEX-MAPOFFSET
_DECIM       DCB  7
      DCB   "DECIMAL"
      ALIGN
DECIM
      _NEST
      _DOLIT
      DCD   10
      BL    BASE
      BL    STORE
      _UNNEST
```

3.4.2 Numeric Input

The stm32eForth text interpreter must handle numbers input to the system. It parses commands out of the input stream and executes them in sequence. When the text interpreter encounters a string which is not the name of a command in the dictionary, it assumes that the string must be a number and attempts to convert the ASCII string to a number according to the current radix. When the text interpreter succeeds in converting the string to a number, the number is pushed on the parameter stack for future use, if the text interpreter is in the interpreting mode. If it is in the compiling mode, the text interpreter will compile the number to the dictionary as an integer literal so that when the command under construction is later executed, the integer value will be pushed on the parameter stack.

The following figure show how a number string is converted to a number and pushed on the parameter stack.



If the text interpreter fails to convert the string to a number, this is an error condition which will cause the text interpreter to `ABORT`, post an error message to you, and then wait for your next line of commands.

DIGIT?	Convert an ASCII numeric digit `c` on the top of the parameter stack to its numeric value `u` according to current radix `base`. If conversion is successful, push a true flag above `u`. If not successful, return `c` and a false flag.

```
;***************************************************************************
; Numeric input, single precision

;   DIGIT?  ( c base -- u t )
;     Convert a character to its numeric value. A flag indicates success.

      DCD   _DECIM-MAPOFFSET
_DIGTQ      DCB  6
      DCB   "DIGIT?"
      ALIGN
DIGTQ
```

```
        _NEST
        BL     TOR
        _DOLIT
        DCD    '0'
        BL     SUBB
        _DOLIT
        DCD    9
        BL     OVER
        BL     LESS
        BL     QBRAN
        DCD    DGTQ1-MAPOFFSET
        _DOLIT
        DCD    7
        BL     SUBB
        BL     DUPP
        _DOLIT
        DCD    10
        BL     LESS
        BL     ORR
DGTQ1   BL     DUPP
        BL     RFROM
        BL     ULESS
        _UNNEST
```

NUMBER?	Convert a count string of ASCII numeric digits at location `a` to an integer. If first character is a $, convert in hexadecimal; otherwise, convert using radix in `BASE`. If first character is a -, negate converted integer. If an illegal character is encountered, the address of string `a` and a false flag `F` are pushed on the parameter stack. Successful conversion pushes integer value and a `true` flag on the parameter stack. `NUMBER?` is very complicated because it has to handle many different characters in the input numeric string. It also has to detect the error condition when it encounters an illegal numeric digit. .

```
;    NUMBER? ( a -- n T | a F )
;       Convert a numberDCB to integer. Push a flag on tos.

        DCD    _DIGTQ-MAPOFFSET
_NUMBQ         DCB  7
        DCB    "NUMBER?"
        ALIGN
NUMBQ
        _NEST
        BL     BASE
        BL     AT
        BL     TOR
        _DOLIT
        DCD    0
        BL     OVER
        BL     COUNT
        BL     OVER
        BL     CAT
        _DOLIT
        DCD    '_'
        BL     EQUAL
        BL     QBRAN
```

```
        DCD   NUMQ1-MAPOFFSET
        BL    HEX
        BL    SWAP
        BL    ONEP
        BL    SWAP
        BL    ONEM
NUMQ1     BL  OVER
        BL    CAT
        _DOLIT
        DCD   '-'
        BL    EQUAL
        BL    TOR
        BL    SWAP
        BL    RAT
        BL    SUBB
        BL    SWAP
        BL    RAT
        BL    PLUS
        BL    QDUP
        BL    QBRAN
        DCD   NUMQ6-MAPOFFSET
        BL    ONEM
        BL    TOR
NUMQ2     BL  DUPP
        BL    TOR
        BL    CAT
        BL    BASE
        BL    AT
        BL    DIGTQ
        BL    QBRAN
        DCD   NUMQ4-MAPOFFSET
        BL    SWAP
        BL    BASE
        BL    AT
        BL    STAR
        BL    PLUS
        BL    RFROM
        BL    ONEP
        BL    DONXT
        DCD   NUMQ2-MAPOFFSET
        BL    RAT
        BL    SWAP
        BL    DROP
        BL    QBRAN
        DCD   NUMQ3-MAPOFFSET
        BL    NEGAT
NUMQ3     BL  SWAP
        B.W   NUMQ5
NUMQ4     BL  RFROM
        BL    RFROM
        BL    DDROP
        BL    DDROP
        _DOLIT
        DCD   0
NUMQ5     BL  DUPP
NUMQ6     BL  RFROM
        BL    DDROP
```

```
	BL	RFROM
	BL	BASE
	BL	STORE
	_UNNEST
```

3.4.3 Terminal Output

KEY	Execute ?KEY continually until a valid character is received and the character c is returned.

```
;***************************************************************************
; Basic I/O

;   KEY      ( -- c )
;     Wait for and return an input character.

	DCD	_NUMBQ-MAPOFFSET
_KEY	DCB	3
	DCB	"KEY"
	ALIGN
KEY
	_NEST
KEY1	   BL QRX
	BL	QBRAN
	DCD	KEY1-MAPOFFSET
	_UNNEST
```

SPACE	Output a blank (space) character, ASCII 32.

```
;   SPACE   ( -- )
;     Send the blank character to the output device.

	DCD	_KEY-MAPOFFSET
_SPACE		DCB	5
	DCB	"SPACE"
	ALIGN
SPACE
	_NEST
	BL	BLANK
	BL	EMIT
	_UNNEST
```

SPACES	Output +n blank (space) characters.

```
;   SPACES  ( +n -- )
;     Send n spaces to the output device.

	DCD	_SPACE-MAPOFFSET
_SPACS		DCB	6
	DCB	"SPACES"
	ALIGN
SPACS
	_NEST
	_DOLIT
	DCD	0
	BL	MAX
```

```
       BL    TOR
       B.W   CHAR2
CHAR1  BL    SPACE
CHAR2  BL    DONXT
       DCD   CHAR1-MAPOFFSET
       _UNNEST
```

TYPE	Output u characters from a string at memory location a. The second item on the parameter stack b is the address of the string array, and the length in bytes u is on the top of the parameter stack. TYPE is safe, because all non-printable characters are converted to a harmless underscore character.

```
;   TYPE    ( b u -- )
;     Output u characters from b.

       DCD   _SPACS-MAPOFFSET
_TYPEE       DCB   4
       DCB   "TYPE"
       ALIGN
TYPEE
       _NEST
       BL  TOR
       B.W   TYPE2
TYPE1  BL  COUNT
       BL    TCHAR
       BL    EMIT
TYPE2  BL  DONXT
       DCD   TYPE1-MAPOFFSET
       BL    DROP
       _UNNEST
```

CR	Output a carriage-return and a line-feed, ASCII 13 and 10.

```
;   CR        ( -- )
;     Output a carriage return and a line feed.

       DCD   _TYPEE-MAPOFFSET
_CR    DCB  2
       DCB   "CR"
       ALIGN
CR
       _NEST
       _DOLIT
       DCD   CRR
       BL    EMIT
       _DOLIT
       DCD   LF
       BL    EMIT
       _UNNEST
```

3.4.4 String Literals

String literals are data structures compiled in compound command, in-line with other tokens, literal structures, and control structures. A string literal must start with a string token which knows how to handle the following string at run time. Here are two examples of string literals:

```
: xxx    ...   $" A compiled string"  ...   ;
: yyy    ...   ." An output string"  ...   ;
```

In compound command `xxx`, `$"` is an immediate command which compiles the following string as a string literal preceded by a special token `$"|`. When `$"|` is executed at run time, it returns the address of this string on the parameter stack. In `yyy`, `."` compiles a string literal preceded by another token `."|`, which prints the compiled string to the output device at run time.

do$	Push the address of a string literal on the parameter stack. It is called by a string token like `$"|` or `."|`, which precede their respective strings in memory. Therefore, the second item on the return stack points to this string. This address is pushed on the parameter stack. This second item on the return stack must be modified so that it will point to the next token after the string literal. This way. the token after the string literal will be executed, skipping over the string literal. Both `$"|` and `."|` use this command `do$`, which retrieve the address `a` of the counted string.

```
;   do_$    ( -- a )
;     Return the address of a compiled string.

;     DCD   _CR-MAPOFFSET
;_DOSTR      DCB  COMPO+3
;     DCB   "do$$"
;     ALIGN
DOSTR
      _NEST
      BL    RFROM
      BL    RFROM              ; b0 set
      BL    ONEM               ; clear b0
      BL    DUPP
      BL    COUNT              ; get addr-1 count
      BL    PLUS
      BL    ALGND              ; end of string
      BL    ONEP               ; restore b0
      BL    TOR                ; address after string
      BL    SWAP               ; count tugged
      BL    TOR
      _UNNEST
```

$"|	Push the address `a` of the following string on the parameter stack, and then executes the token immediately following the string.

```
;   $"|     ( -- a )
;     Run time routine compiled by _". Return address of a compiled string.

;     DCD   _DOSTR-MAPOFFSET
;_STRQP      DCB  COMPO+3
;     DCB   "$$""|"
;     ALIGN
```

```
STRQP
      _NEST
      BL    DOSTR
      _UNNEST                  ;force a call to dostr
```

.$	Print a string at address a.

```
;   .$      ( a -- )
;     Run time routine of ." . Output a compiled string.

;     DCD   _STRQP-MAPOFFSET
;_DOTST     DCB  COMPO+2
;     DCB   ".$$"
;     ALIGN
DOTST
      _NEST
      BL    COUNT
      BL    TYPEE
      _UNNEST
```

."|	Print the following string, and then executes the token immediately following the string.

```
;   ."|     ( -- )
;     Run time routine of ." . Output a compiled string.

;     DCD   _DOTST-MAPOFFSET
;_DOTQP     DCB  COMPO+3
;     DCB   ".""|"
;     ALIGN
DOTQP
      _NEST
      BL    DOSTR
      BL    DOTST
      _UNNEST
```

.R	Print a signed integer n , the second item on the parameter stack, right-justified in a field of +n characters. +n is on the top of the parameter stack.

```
;   .R        ( n +n -- )
;     Display an integer in a field of n columns, right justified.

      DCD   _CR-MAPOFFSET
_DOTR DCB  2
      DCB   ".R"
      ALIGN
DOTR
      _NEST
      BL    TOR
      BL    STRR
      BL    RFROM
      BL    OVER
      BL    SUBB
      BL    SPACS
      BL    TYPEE
```

```
        _UNNEST
```

U.R	Print an unsigned integer u right-justified in a field of +n characters.

```
;   U.R      ( u +n -- )
;     Display an unsigned integer in n column, right justified.

        DCD   _DOTR-MAPOFFSET
_UDOTR       DCB  3
        DCB   "U.R"
        ALIGN
UDOTR
        _NEST
        BL    TOR
        BL    BDIGS
        BL    DIGS
        BL    EDIGS
        BL    RFROM
        BL    OVER
        BL    SUBB
        BL    SPACS
        BL    TYPEE
        _UNNEST
```

U.	Print an unsigned integer u in free format, followed by a space.

```
;   U.        ( u -- )
;     Display an unsigned integer in free format.

        DCD   _UDOTR-MAPOFFSET
_UDOT DCB  2
        DCB   "U."
        ALIGN
UDOT
        _NEST
        BL    BDIGS
        BL    DIGS
        BL    EDIGS
        BL    SPACE
        BL    TYPEE
        _UNNEST
```

.	Print a signed integer n in free format, followed by a space.

```
;   .    ( w -- )
;     Display an integer in free format, preceeded by a space.

        DCD   _UDOT-MAPOFFSET
_DOT  DCB  1
        DCB   "."
        ALIGN
DOT
        _NEST
        BL    BASE
        BL    AT
        _DOLIT
```

```
        DCD   10
        BL    XORR              ;?decimal
        BL    QBRAN
        DCD   DOT1-MAPOFFSET
        BL    UDOT
        _UNNEST                 ;no,display unsigned
DOT1       BL STRR
        BL    SPACE
        BL    TYPEE
        _UNNEST                 ;yes, display signed
```

?	Print signed integer stored in memory `a` on the top of the parameter stack, in free format followed by a space.

```
;   ?    ( a -- )
;     Display the contents in a memory cell.

        DCD   _DOT-MAPOFFSET
_QUEST        DCB  1
        DCB   "?"
        ALIGN
QUEST
        _NEST
        BL    AT
        BL    DOT
        _UNNEST
```

3.4.5 Parsing

Parsing is always considered a very advanced topic in computer science. However, because Forth uses very simple syntax rules, parsing is easy. Forth input stream consists of a list of ASCII names separated by spaces and other white space characters like tabs, carriage returns, and line feeds. The text interpreter scans the input stream, parses out names, search tokens in the dictionary, and executes them in sequence. After a name is parsed out of the input stream, the text interpreter will 'interpret' it; i.e., execute it if it is a valid command, compile it if the text interpreter is in the compiling mode, and convert it to a number if the name is not a Forth command.

The case where the delimiting character is a space (ASCII 32) is special, because this is when the text interpreter is parsing for valid names. It thus must skip over leading space characters. When `parse` is used to compile string literals, it will use a double quote character (ASCII 34) as the delimiting character. It the delimiting character is not space, `parse` starts scanning immediately, looking for the designated delimiting character.

`parse`	The elementary command to parse text. From the input stream, which starts at `b1` and is of `u1` characters long, it parses out the first text string delimited by character `c`. It returns the address `b2` and length `u2` of the string just parsed out and the difference `n` between `b1` and `b2`. Leading spaces are skipped over if space is the delimiting character.

```
;***************************************************************************
; Parsing
```

```
;	parse	( b u c -- b u delta ; string> )
;	ScanDCB delimited by c. Return found string and its offset.

;	DCD	_QUEST-MAPOFFSET
;_PARS		DCB	5
;	DCB	"parse"
;	ALIGN
PARS
	_NEST
	BL	TEMP
	BL	STORE
	BL	OVER
	BL	TOR
	BL	DUPP
	BL	QBRAN
	DCD	PARS8-MAPOFFSET
	BL	ONEM
	BL	TEMP
	BL	AT
	BL	BLANK
	BL	EQUAL
	BL	QBRAN
	DCD	PARS3-MAPOFFSET
	BL	TOR
PARS1	BL	BLANK
	BL	OVER
	BL	CAT			;skip leading blanks
	BL	SUBB
	BL	ZLESS
	BL	INVER
	BL	QBRAN
	DCD	PARS2-MAPOFFSET
	BL	ONEP
	BL	DONXT
	DCD	PARS1-MAPOFFSET
	BL	RFROM
	BL	DROP
	_DOLIT
	DCD	0
	BL	DUPP
	_UNNEST
PARS2	BL	RFROM
PARS3	BL	OVER
	BL	SWAP
	BL	TOR
PARS4	BL	TEMP
	BL	AT
	BL	OVER
	BL	CAT
	BL	SUBB			;scan for delimiter
	BL	TEMP
	BL	AT
	BL	BLANK
	BL	EQUAL
	BL	QBRAN
	DCD	PARS5-MAPOFFSET
	BL	ZLESS
```

```
PARS5   BL  QBRAN
      DCD   PARS6-MAPOFFSET
      BL    ONEP
      BL    DONXT
      DCD   PARS4-MAPOFFSET
      BL    DUPP
      BL    TOR
      B     PARS7
PARS6   BL  RFROM
      BL    DROP
      BL    DUPP
      BL    ONEP
      BL    TOR
PARS7   BL  OVER
      BL    SUBB
      BL    RFROM
      BL    RFROM
      BL    SUBB
      _UNNEST
PARS8   BL  OVER
      BL    RFROM
      BL    SUBB
      _UNNEST
      ALIGN
```

PARSE	Scan the input stream in the Terminal Input Buffer from where `>IN` points to, until the end of the buffer, for a string delimited by character `c`. It returns the address and length of the string parsed out. `PARSE` calls `parse` to do the dirty work. `PARSE` is used to implement many specialized parsing commands to perform different parsing operations.

```
;   PARSE   ( c -- b u ; string> )
;     Scan input stream and return counted string delimited by c.

      DCD   _QUEST-MAPOFFSET
_PARSE      DCB  5
      DCB   "PARSE"
      ALIGN
PARSE
      _NEST
      BL    TOR
      BL    TIB
      BL    INN
      BL    AT
      BL    PLUS                ;current input buffer pointer
      BL    NTIB
      BL    AT
      BL    INN
      BL    AT
      BL    SUBB                ;remaining count
      BL    RFROM
      BL    PARS
      BL    INN
      BL    PSTOR
      _UNNEST
```

.(	Print the following string till the next) character. It is used to output text to the serial output device.

```
;   .(        ( -- )
;     Output following string up to next ) .

      DCD   _PARSE-MAPOFFSET
_DOTPR      DCB  IMEDD+2
      DCB   ".("
      ALIGN
DOTPR
      _NEST
      _DOLIT
      DCD   ')'
      BL    PARSE
      BL    TYPEE
      _UNNEST
```

(	Discard the following string till the next) character. It is used to place comments in source code.

```
;   (    ( -- )
;     Ignore following string up to next ) . A comment.

      DCD   _DOTPR-MAPOFFSET
_PAREN      DCB  IMEDD+1
      DCB   "("
      ALIGN
PAREN _NEST
      _DOLIT
      DCD   ')'
      BL    PARSE
      BL    DDROP
      _UNNEST
```

\	Discard all characters till end of a line. It is used to insert comment lines in source code.

```
;   \    ( -- )
;     Ignore following text till the end of line.

      DCD   _PAREN-MAPOFFSET
_BKSLA      DCB  IMEDD+1
      DCB   "\\"
      ALIGN
BKSLA
      _NEST
      BL    NTIB
      BL    AT
      BL    INN
      BL    STORE
      _UNNEST
```

CHAR	Parse the next string out but returns only the first character in this string. It gets an ASCII character from the input stream.

```
;   CHAR    ( -- c )
;     Parse next word and return its first character.

      DCD   _BKSLA-MAPOFFSET
_CHAR DCB  4
      DCB   "CHAR"
      ALIGN
CHAR
      _NEST
      BL    BLANK
      BL    PARSE
      BL    DROP
      BL    CAT
      _UNNEST
```

WORD	Parse out the next string delimited by the ASCII character `c`. It then copies this string as a counted string to the word buffer on top of the dictionary and returns its address `a`. The length of the string is limited to 255 characters. It is used to parse text strings in general.

```
;   WORD    ( c -- a ; string> )
;     Parse a word from input stream and copy it to code dictionary.

      DCD   _CHAR-MAPOFFSET
_WORDD      DCB  4
      DCB   "WORD"
      ALIGN
WORDD
      _NEST
      BL    PARSE
      BL    HERE
      BL    CELLP
      BL    PACKS
      _UNNEST
```

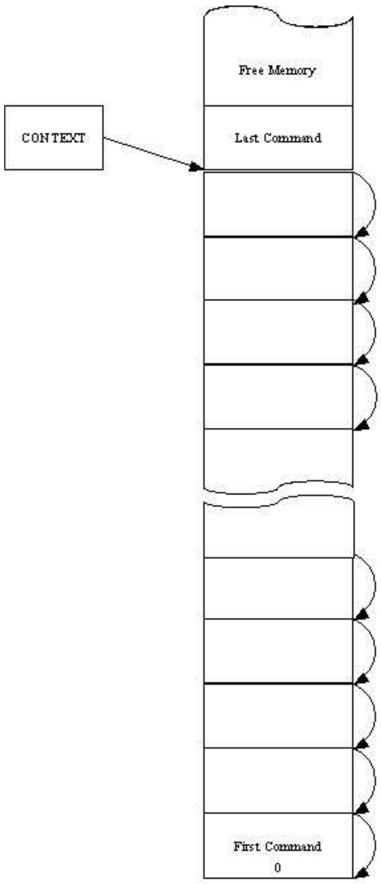
TOKEN	Parse out the next string delimited by space characters. It then copies this string as a counted string to the word buffer on top of dictionary and returns its address `a`. The length of the string is limited to 31 characters. It is used to parse out names of command tokens for interpretation and compilation.

```
;   TOKEN   ( -- a ; string> )
;     Parse a word from input stream and copy it to name dictionary.

      DCD   _WORDD-MAPOFFSET
_TOKEN      DCB  5
      DCB   "TOKEN"
      ALIGN
TOKEN
      _NEST
      BL    BLANK
      BL    WORDD
      _UNNEST
```

3.4.6 Dictionary Search

In eForth, command records are linearly linked into a dictionary. A command record contains three fields: a link field holding the name field address of the previous command record, a name field holding the name as a counted string, and a code field holding executable code and data. A dictionary search follows the linked list of records to find a command with a matching name. It returns the name field address and the code field address, if a match is found.

The link field of the first command record in dictionary contains a 0, indicating it is the end of the linked list. A user variable `CONTEXT` holds an address pointing to the name field of the last command record. The dictionary search starts at `CONTEXT` and terminates at the first matched name, or at the first command record. The linking of records in dictionary is show in the following figure:



From `CONTEXT`, we can locate the name field of the last command record in the dictionary. If this name does not match the search string to be searched, we can find the link field of this record, which is 4 bytes less than the name field address. From this link field, we can locate the name field of the prior command record. Compare the name with the search string. And so forth. We will either find a command or reach the end of the linked list.

NAME>	Convert a name field address `na` in a command record to the code field address `ca` of this command record. Code field address is the name field address plus length of name plus

	one, and aligned to the next cell boundary.

```
;***************************************************************************
; Dictionary search

;   NAME>   ( na -- ca )
;     Return a code address given a name address.

      DCD   _TOKEN-MAPOFFSET
_NAMET      DCB  5
      DCB   "NAME>"
      ALIGN
NAMET
      _NEST
      BL    COUNT
      _DOLIT
      DCD   0x1F
      BL    ANDD
      BL    PLUS
      BL    ALGND
      _UNNEST
```

SAME?	Compare two strings at addresses a and b for u words. It returns a 0 if two strings are equal. It returns a positive integer if a string is greater than b string. It returns a negative integer if a string is less than b string.

```
;   SAME?   ( a a u -- a a f \ -0+ )
;     Compare u cells in two strings. Return 0 if identical.

      DCD   _NAMET-MAPOFFSET
_SAMEQ      DCB  5
      DCB   "SAME?"
      ALIGN
SAMEQ
      _NEST
      BL    TOR
      B.W   SAME2
SAME1   BL  OVER
      BL    RAT
      BL    CELLS
      BL    PLUS
      BL    AT                  ;32/16 mix-up
      BL    OVER
      BL    RAT
      BL    CELLS
      BL    PLUS
      BL    AT                  ;32/16 mix-up
      BL    SUBB
      BL    QDUP
      BL    QBRAN
      DCD   SAME2-MAPOFFSET
      BL    RFROM
      BL    DROP
      _UNNEST                   ;strings not equal
SAME2   BL  DONXT
```

```
        DCD    SAME1-MAPOFFSET
        _DOLIT
        DCD    0
        _UNNEST                  ;strings equal
```

find	Assume that a count string is at memory address `a`, and the name field address of the last command record is in address `va`. If the string matches the name of a command, both the code field address `ca` and the name field address `na` of the command record are returned. If the string is not a valid command, the original string address and a false flag are returned. `find` runs the dictionary search very quickly because it first compares the length byte and the first 3 characters in the name field as a 32 bit integer. In most cases of mismatch, this comparison would fail and the next record can be reached through the link field. If the first 4 characters match, then SAME? is invoked to compare the rest of the name field, one cell at a time. Since both the target text string and the name field are null filled to the cell boundary, the comparison can be performed quickly across the entire name field without worrying about the word boundaries.

```
;   find    ( a na -- ca na | a F )
;      Search a vocabulary for a string. Return ca and na if succeeded.

;      DCD    _SAMEQ-MAPOFFSET
;_FIND        DCB   4
;      DCB    "find"
;      ALIGN
FIND
        _NEST
        BL     SWAP              ; na a
        BL     DUPP              ; na a a
        BL     CAT               ; na a count
        BL     CELLSL            ; na a count/4
        BL     TEMP
        BL     STORE             ; na a
        BL     DUPP              ; na a a
        BL     AT                ; na a word1
        BL     TOR               ; na a
        BL     CELLP             ; na a+4
        BL     SWAP              ; a+4 na
FIND1
        BL     DUPP              ; a+4 na na
        BL     QBRAN
        DCD    FIND6-MAPOFFSET   ; end of vocabulary
        BL     DUPP              ; a+4 na na
        BL     AT                ; a+4 na name1
        _DOLIT
        DCD    MASKK
        BL     ANDD
        BL     RAT               ; a+4 na name1 word1
        BL     XORR              ; a+4 na ?
        BL     QBRAN
        DCD    FIND2-MAPOFFSET
        BL     CELLM             ; a+4 la
        BL     AT                ; a+4 next_na
        B.w    FIND1             ; try next word
FIND2
```

```
      BL    CELLP             ; a+4 na+4
      BL    TEMP
      BL    AT                ; a+4 na+4 count/4
      BL    SAMEQ             ; a+4 na+4 ?
FIND3
      B.w   FIND4
FIND6
      BL    RFROM             ; a+4 0 name1 -- , no match
      BL    DROP              ; a+4 0
      BL    SWAP              ; 0 a+4
      BL    CELLM             ; 0 a
      BL    SWAP              ; a 0
      _UNNEST                 ; return without a match
FIND4
      BL    QBRAN             ; a+4 na+4
      DCD   FIND5-MAPOFFSET   ; found a match
      BL    CELLM             ; a+4 na
      BL    CELLM             ; a+4 la
      BL    AT                ; a+4 next_na
      B.w   FIND1             ; compare next name
FIND5
      BL    RFROM             ; a+4 na+4 count/4
      BL    DROP              ; a+4 na+4
      BL    SWAP              ; na+4 a+4
      BL    DROP              ; na+4
      BL    CELLM             ; na
      BL    DUPP              ; na na
      BL    NAMET             ; na ca
      BL    SWAP              ; ca na
      _UNNEST                 ; return with a match
      ALIGN
```

NAME?	Search the dictionary starting at CONTEXT for a name string at address `a`. Return the code field address `ca` and name field address `na` if a matching command is found. Otherwise, return the original string address `a` and a false flag.

```
;   NAME?   ( a -- ca na | a F )
;     Search all context vocabularies for a string.

      DCD   _SAMEQ-MAPOFFSET
_NAMEQ      DCB  5
      DCB   "NAME?"
      ALIGN
NAMEQ
      _NEST
      BL    CNTXT
      BL    AT
      BL    FIND
      _UNNEST
```

3.4.7 Terminal Input

The text interpreter interprets a line of text received from an terminal and stored it in the Terminal Input Buffer. To process characters received from the terminal, we need special commands to deal with backspace character and carriage return. On top of stack, three special parameters are referenced in many commands: `bot` is the Beginning Of the Input Buffer, `eot` is the End Of the Input Buffer, and `cur` points to the current character in the input buffer.

^H	Process back-space character (ASCII 8). It erases the last character previously entered, and decrement the character pointer `cur`. If `cur=bot`, do nothing because you cannot backup beyond beginning of input buffer.

```
;***************************************************************************
; Terminal input

;   ^H        ( bot eot cur -- bot eot cur )
;     Backup the cursor by one character.

;     DCD   _NAMEQ-MAPOFFSET
;_BKSP      DCB  2
;     DCB   "^H"
;     ALIGN
BKSP
      _NEST
      BL    TOR
      BL    OVER
      BL    RFROM
      BL    SWAP
      BL    OVER
      BL    XORR
      BL    QBRAN
      DCD   BACK1-MAPOFFSET
      _DOLIT
      DCD   BKSPP
      BL    TECHO
;     BL    ATEXE
      BL    ONEM
      BL    BLANK
      BL    TECHO
;     BL    ATEXE
      _DOLIT
      DCD   BKSPP
      BL    TECHO
;     BL    ATEXE
BACK1
        _UNNEST
```

TAP	Output character `c` to terminal, store `c` in `cur`, and increment the character pointer `cur`, which points to the current character in the input buffer. `bot` and `eot` are pointers pointing to the beginning and end of the input buffer.

```
;   TAP      ( bot eot cur c -- bot eot cur )
;     Accept and echo the key stroke and bump the cursor.

;     DCD   _BKSP-MAPOFFSET
;_TAP DCB  3
;     DCB   "TAP"
;     ALIGN
TAP
```

```
        _NEST
        BL     DUPP
        BL     TECHO
;       BL     ATEXE
        BL     OVER
        BL     CSTOR
        BL     ONEP
        _UNNEST
```

kTAP	Process character `c`. `bot` is pointing at the beginning of the input buffer, and `eot` is pointing at the end. `cur` points to the current character in the input buffer. The character `c` is normally stored at `cur`, which is then incremented by 1. If `c` is a carriage-return (ASCII 13), echo a space and make `eot=cur`., thus terminating the input process If `c` is a back-space (ASCII 8), erase the last character and decrement `cur`.

```
;   kTAP    ( bot eot cur c -- bot eot cur )
;       Process a key stroke, CR or backspace.

;       DCD    _TAP-MAPOFFSET
;_KTAP         DCB   4
;       DCB    "kTAP"
;       ALIGN
KTAP
TTAP
        _NEST
        BL     DUPP
        _DOLIT
        DCD    CRR
        BL     XORR
        BL     QBRAN
        DCD    KTAP2-MAPOFFSET
        _DOLIT
        DCD    BKSPP
        BL     XORR
        BL     QBRAN
        DCD    KTAP1-MAPOFFSET
        BL     BLANK
        BL     TAP
        _UNNEST
        DCD    0                 ;patch
KTAP1     BL   BKSP
        _UNNEST
KTAP2     BL   DROP
        BL     SWAP
        BL     DROP
        BL     DUPP
        _UNNEST
```

ACCEPT	Accept `u` characters into an input buffer starting at address `b`, or until a carriage return (ASCII 13) is encountered. The value of `u` returned is the actual number of characters received.

```
;   ACCEPT  ( b u -- b u )
;       Accept characters to input buffer. Return with actual count.
```

```
      DCD   _NAMEQ-MAPOFFSET
_ACCEP      DCB  6
      DCB   "ACCEPT"
      ALIGN
ACCEP
      _NEST
      BL    OVER
      BL    PLUS
      BL    OVER
ACCP1   BL  DDUP
      BL    XORR
      BL    QBRAN
      DCD   ACCP4-MAPOFFSET
      BL    KEY
      BL    DUPP
      BL    BLANK
      _DOLIT
      DCD   127
      BL    WITHI
      BL    QBRAN
      DCD   ACCP2-MAPOFFSET
      BL    TAP
      B     ACCP3
ACCP2   BL  KTAP
;     BL    ATEXE
ACCP3
      B     ACCP1
ACCP4   BL  DROP
      BL    OVER
      BL    SUBB
      _UNNEST
```

QUERY	Accept up to 80 characters from the input device to the Terminal Input Buffer. It also prepares the Terminal Input Buffer for parsing by setting `#TIB` to the length of the input text stream, and clearing `>IN` so it points to the beginning of the Terminal Input Buffer.

```
;   QUERY   ( -- )
;     Accept input stream to terminal input buffer.

      DCD   _ACCEP-MAPOFFSET
_QUERY      DCB  5
      DCB   "QUERY"
      ALIGN
QUERY
      _NEST
      BL    TIB
      _DOLIT
      DCD   80
      BL    ACCEP
      BL    NTIB
      BL    STORE
      BL    DROP
      _DOLIT
      DCD   0
      BL    INN
```

```
        BL    STORE
        _UNNEST
```

3.4.8 Error Handling

When the text interpreter encounters a string which is not a name and not a number, it prints out this string followed by a ? mark as an error message. Then the text interpreter starts over. Stacks are cleared and then jump to `QUIT`.

ABORT	Print the string in memory located at address `a`, followed by a `?` mark and aborts. 'Abort' means clearing the parameter stack and the return stack, and returns to the text interpreter loop `QUIT`.

```
;***************************************************************************
; Error handling

;   ABORT   ( a -- )
;     Reset data stack and jump to QUIT.

        DCD   _QUERY-MAPOFFSET
_ABORT      DCB  5
        DCB   "ABORT"
        ALIGN
ABORT
        _NEST
        BL    SPACE
        BL    COUNT
        BL    TYPEE
        _DOLIT
        DCD   0X3F
        BL    EMIT
        BL    CR
        BL    PRESE
        B.W   QUIT
        ALIGN
```

abort"	It is compiled with an error message in a compound command. When `abort"` is executed in run time, it examines the top item on the parameter stack. It the flag is true, print out the following error message and `QUIT`; otherwise, skip over the error message and continue executing the next command.

```
;   _abort" ( f -- )
;     Run time routine of ABORT" . Abort with a message.

;     DCD   _ABORT-MAPOFFSET
;_ABORQ     DCB  COMPO+6
;     DCB   "abort\""
;     ALIGN
ABORQ
        _NEST
        BL    QBRAN
        DCD   ABOR1-MAPOFFSET   ;text flag
        BL    DOSTR
```

```
       BL     COUNT
       BL     TYPEE
       BL     CR
       B.W    QUIT
ABOR1    BL   DOSTR
       BL     DROP
       _UNNEST                 ;drop error
```

3.4.9 String Interpreter

Text interpreter in Forth is like a conventional operating system of a computer. It is the primary interface you use to get the computer to do work. Since Forth uses very simple syntax rule--commands are separated by spaces, the text interpreter is also very simple. It accepts a line of text from the terminal, parses out a name delimited by spaces, locates the name in the dictionary and then executes it. The process is repeated until the input text is exhausted. Then the text interpreter waits for another line of text and interprets it again. This cycle repeats until you are exhausted and turns off the computer.

In eForth, the text interpreter is coded in the command `QUIT`. `QUIT` contains an infinite loop which repeats the `QUERY`-`EVAL` command pair. `QUERY` accepts a line of text from the input terminal. `EVAL` interprets the text one name at a time till the end of the text line. `EVAL` uses the command whose address is in user variable ‘EVAL to process the name string. `‘EVAL` contains either `$INTERPRET` or `$COMPILE`, which executes or compiles the name, respectively.

$INTERPRET	Execute a command whose name string is stored at address `a` on the parameter stack. If the string is not a valid command, convert it to a number. Failing the numeric conversion, execute `ABORT` and return to `QUIT`.

```
;**************************************************************************
; The text interpreter

;   $INTERPRET  ( a -- )
;     Interpret a word. If failed, try to convert it to an integer.

       DCD    _ABORT-MAPOFFSET
_INTER        DCB  10
       DCB    "$$INTERPRET"
       ALIGN
INTER
       _NEST
       BL     NAMEQ
       BL     QDUP  ;?defined
       BL     QBRAN
       DCD    INTE1-MAPOFFSET
       BL     AT
       _DOLIT
       DCD    COMPO
       BL     ANDD                 ;?compile only lexicon bits
       BL     ABORQ
       DCB    13
       DCB    " compile only"
       ALIGN
       BL     EXECU
```

```
      _UNNEST                  ;execute defined word
INTE1   BL  NUMBQ
      BL    QBRAN
      DCD   INTE2-MAPOFFSET
      _UNNEST
INTE2   B.W ABORT ;error
```

[	Activate the text interpreter by storing the code field address of `$INTERPRET` into the variable `'EVAL`, which is executed in `EVAL` while the text interpreter is in the interpretive mode.

```
;   [    ( -- )
;     Start the text interpreter.

      DCD   _INTER-MAPOFFSET
_LBRAC      DCB  IMEDD+1
      DCB   "["
      ALIGN
LBRAC
      _NEST
      _DOLIT
      DCD   INTER-MAPOFFSET
      BL    TEVAL
      BL    STORE
      _UNNEST
```

.OK	Print the familiar `ok>` prompting message after executing to the end of a line. The message `ok>` is printed only when the text interpreter is in the interpretive mode. While compiling, the prompt is suppressed.

```
;   .OK      ( -- )
;     Display "ok" only while interpreting.

      DCD   _LBRAC-MAPOFFSET
_DOTOK      DCB  3
      DCB   ".OK"
      ALIGN
DOTOK
      _NEST
      _DOLIT
      DCD   INTER-MAPOFFSET
      BL    TEVAL
      BL    AT
      BL    EQUAL
      BL    QBRAN
      DCD   DOTO1-MAPOFFSET
      BL    DOTQP
      DCB   3
      DCB   " ok"
      ALIGN
DOTO1   BL  CR
      _UNNEST
```

?STACK	Check for stack underflow. Abort, resetting the parameter stack pointer, if the stack depth

	is negative.

```
;   ?STACK  ( -- )
;     Abort if the data stack underflows.

      DCD   _DOTOK-MAPOFFSET
_QSTAC      DCB  6
      DCB   "?STACK"
      ALIGN
QSTAC
      _NEST
      BL    DEPTH
      BL    ZLESS ;check only for underflow
      BL    ABORQ
      DCB   10
      DCB   " underflow"
      ALIGN
      _UNNEST
```

EVAL	It is contained in the text interpreter loop QUIT. It parses tokens from the input stream and invokes whatever command in `'EVAL` to process the token, either execute it with `$INTERPRET` or compile it with `$COMPILE`.

```
;   EVAL    ( -- )
;     Interpret the input stream.

      DCD   _QSTAC-MAPOFFSET
_EVAL DCB  4
      DCB   "EVAL"
      ALIGN
EVAL
      _NEST
EVAL1   BL  TOKEN
      BL    DUPP
      BL    CAT   ;?input stream empty
      BL    QBRAN
      DCD   EVAL2-MAPOFFSET
      BL    TEVAL
      BL    ATEXE
      BL    QSTAC ;evaluate input, check stack
      B.W   EVAL1
EVAL2   BL  DROP
      BL    DOTOK
      _UNNEST     ;prompt
      ALIGN
```

PRESET	Reset the parameter stack pointer to clear the parameter stack.

```
;   PRESET  ( -- )
;     Reset data stack pointer and the terminal input buffer.

      DCD   _EVAL-MAPOFFSET
_PRESE      DCB  6
      DCB   "PRESET"
      ALIGN
```

```
PRESE
      _NEST
      MOVW  R1,#0XFE00        ; init SP
;     MOVT  R1,#0X2000
      MOVW  R0,#0             ; init TOS
      _UNNEST
```

QUIT	It is the operating system, the text interpreter, or a shell, of the stm32eForth system. It is an infinite loop eForth will never get out. It uses QUERY to accept a line of commands from the input terminal and then lets EVAL to parse out tokens and execute them. After a line is processed, it displays an ok> message and wait for the next line of commands. When an error occurred during execution, it prints the string which caused the error as an error message. After the error is reported, it re-initializes the system by clearing the stacks and comes back to receive the next line of commands. Because the behavior of EVAL can be changed by storing either $INTERPRET or $COMPILE into 'EVAL, QUIT exhibits the dual nature of a text interpreter and a compiler.

```
;   QUIT    ( -- )
;     Reset return stack pointer and start text interpreter.

      DCD   _PRESE-MAPOFFSET
_QUIT DCB  4
      DCB   "QUIT"
      ALIGN
QUIT
      _NEST
      MOVW  R2,#0XFF00
;     MOVT  R2,#0X2000
QUIT1   BL  LBRAC             ;start interpretation
QUIT2   BL  QUERY             ;get input
      BL    EVAL
      BL    BRAN
      DCD   QUIT2-MAPOFFSET   ;continue till error
```

3.4.10 Flash Memory

STM32F407VG on Discovery Kit has only 1 MB of flash. That's plenty as far as eForth is concerned. eForth core occupies about 8-10 KB, but it can use lots of memory for applications. Flash memory is generally difficult to use, and you have to be very careful so that your system will not inadvertently mess up the flash memory and cause the system to crash.

When programming in eForth, you add new command to the dictionary. New commands can generally be added to the flash memory directly, if the area is erased properly. Once a command is added, you cannot modify it. You cannot erase a command individually, because flash memory must be erased in whole sectors. One particular problem in eForth is that you cannot change a command so that it becomes an immediate command, as the immediate bit in the header of a command is already cleared, and cannot be set again. Another problem is that it is very difficult to build turnkey system, because the table of initial values of user variables cannot be updated without erasing a whole sector wherein the table is located.

Happily, STM32F4 has 192 KB of RAM memory which can be used as program memory. In stm32eforth720, eForth system is first copied from flash to RAM, and executed in RAM. In RAM memory, eForth system can grow at will, without limitations posed by flash memory. When an application is completely debugged, the entire eForth dictionary can be saved into flash memory. Your application can start running after reset by saving its execution address in the user variable 'BOOT.

In STM32F4, there is a flash memory controller, which is just like another IO device. It is called 'Flash Memory Interface', and has a set of status, control, and data registers. Following the chapter on flash memory in the reference manual, it is not very difficult to program the flash memory to do what you want it to do.

In STM32F407 chip, 1 MB of flash memory are organized in 12 sectors. Sectors 0-3 have 16 KB each. Sector 4 has 64 KB. Sectors 5-11 have 128 KB each. Stm32eforth720 only uses sectors 0-3.

Regular eForth memory read commands @ and C@ can read flash memory. Memory write commands ! and C! have no effect on flash memory. I added a special command I! to write a 32-bit word into flash memory. While flash memory is unlocked, a 2 is also stored into the PSIZE field of FLASH_CR register. It specifies that we will only write 32-bit words into flash memory.

UNLOCK	Unlock flash memory to be writable. This is done only once on reset. Two special words 0x45670123 and 0xCDEF89AB are written to FLASH_KEYR register.

```
;**************************************************************************
; Flash memory interface

FLASH EQU   0x40023C00
FLASH_KEYR  EQU   0X04
FLASH_SR    EQU   0x0C
FLASH_CR    EQU   0X10
FLASH_KEY1  EQU   0x45670123
FLASH_KEY2  EQU   0xCDEF89AB

UNLOCK      ; unlock flash memory
      ldr   r0, =FLASH
      ldr   r4, =FLASH_KEY1
      str   r4, [r0, #0x4]
      ldr   r4, =FLASH_KEY2
      str   r4, [r0, #0x4]
      mov   r4, #0x200        ; PSIZE 32 bits
      str   r4, [r0, #0x10]

      _NEXT
```

WAIT_BSY	Wait until the busy flag BSY in the FLASH_SR register is cleared, so that we can start the next flash operation.

```
WAIT_BSY
      ldr   r0, =FLASH
WAIT1 ldr   r4, [r0, #0x0C]   ; FLASH_SR
      ands  r4, #0x1000 ; BSY
      bne   WAIT1
      _NEXT
```

```
	ALIGN
```

ERASE_SECTOR	Erase one sector (0-11) of flash memory. stm32eforth720 only uses the first 4 sectors (16 KB each) of flash memory.

```
;   ERASE_SECTOR     ( sector -- )
;       Erase one sector of flash memory.  Sector=0 to 11

	DCD	_QUIT-MAPOFFSET
_ESECT	DCB	12
	DCB	"ERASE_SECTOR"
	ALIGN

ESECT	; sector --
	_NEST
	bl	WAIT_BSY
	ldr	r4,[r0, #0x10]	; FLASH_CR
	bic	r4,r4,#0x78 ; clear SNB
	lsl	R5,R5,#3	; align sector #
	orr	r4,r4,r5	; put in sector #
	orr	R4,R4,#0x10000	; set STRT bit
	orr	R4,R4,#0x200	; PSIZE=32
	orr	R4,R4,#2	; set SER bit, enable erase
	str	r4,[r0, #0x10]	; start erasing
;	bl	WAIT_BSY
	_POP
	_UNNEST
```

I!	Write 32 bit `data` into flash memory location `address`. Enable flash writing before writing. Disable flash writing afterwards to protect flash memory.

```
;   I!          ( data address -- )
;        Write one word into flash memory

	DCD	_ESECT-MAPOFFSET
_ISTOR	DCB	2
	DCB	"I!"
	ALIGN

ISTOR ; data address --
	_NEST
	bl	WAIT_BSY
	ldr	r4, [r0, #0x10]	; FLASH_CR
	orr	r4,R4,#0x1	; PG
	str	r4, [r0, #0x10]	; enable programming
	bl	STORE
	bl	WAIT_BSY
	ldr	r4, [r0, #0x10]	; FLASH_CR
	bic	r4,R4,#0x1	; PG
	str	r4, [r0, #0x10]	; disable programming
	_UNNEST
	ALIGN
	LTORG
```

TURNKEY	Copy eForth dictionary from RAM to flash. The user variables are copied first from

	0xFF00-0xFF3F to 0xC0-0xFF so that the new eForth system will be boot up properly with current user variables. 'BOOT must be initialized correctly to point to an application command you wish to run after reset.

```
;   TURNKEY ( -- )
;     Copy dictionary from RAM to flash.

      DCD   _ISTOR-MAPOFFSET
_TURN DCB   7
      DCB   "TURNKEY"
      ALIGN
TURN  _NEST
      _DOLIT                  ; save user area
      DCD   0XFF00
      _DOLIT
      DCD   0xC0              ; to boot array
      _DOLIT
      DCD   0x40
      BL    MOVE
      _DOLIT
      DCD   0
      _DOLIT
      DCD   0x8000000
      BL    CPP
      BL    AT
      BL    CELLSL
      BL    TOR
TURN1 BL    OVER
      BL    AT
      BL    OVER
      BL    ISTOR
      BL    SWAP
      BL    CELLP
      BL    SWAP
      BL    CELLP
      BL    DONXT
      DCD   TURN1-MAPOFFSET
      BL    DDROP
      _UNNEST
      ALIGN
```

3.5 Forth Compiler

3.5.1 Compiler Loop

The Forth compile is the twin brother of the text interpreter. They share lot of code and they reside in the same interpreter loop QUIT. Let us use the same task sequence in the text interpreter section to show what the compiler does:

Step 1. Accept one line of text from the terminal.
Step 2. Parse out a space delimited name string.
Step 3. Search the dictionary for a command of this name.
Step 4. If it is an immediate command, execute it. Go to Step 9.

Step 5. If it is a command, compile it as a token. Go to Step 9.
Step 6. If it is not a command, convert it to a number.
Step 7. If it is a number, compile a integer literal structure. Go to Step 9.
Step 8. If it is not a number, abort. Go back to step 1.
Step 9. If the text line is not exhausted, go back to step 2.
Step 10. If the text line is exhausted, go back to Step 1.

Compiler and interpreter are both processing a linear list of names. However, interpreter is like talking, a simple linear list is generally sufficient. Compiler is like writing, and it can express deeply convoluted thoughts and ideas. These ideas cannot be expressed in a single line of names. You need a big sheet of paper, or a file, to put them down properly. In addition to compile linear lists of tokens, the Forth compile can build complicated branch structures, loop structures, and control structures embedded in token lists. These structures are built with the immediate commands, which are executed immediately by the compiler. These are things we will discuss in this section.

3.5.2 Compiler Tools

'	Search the dictionary for the following string. If the string is a valid command, return its code field address `ca`. If the string is not a valid command, print it with a ? mark.

```
;***************************************************************************
; The compiler

;   '    ( -- ca )
;     Search context vocabularies for the next word in input stream.

      DCD   _TURN-MAPOFFSET
_TICK DCB  1
      DCB   "'"
      ALIGN
TICK
      _NEST
      BL    TOKEN
      BL    NAMEQ ;?defined
      BL    QBRAN
      DCD   TICK1-MAPOFFSET
      _UNNEST     ;yes, push code address
TICK1 B.W   ABORT ;no, error
```

ALLOT	Allocate `n` bytes of memory on top of the dictionary. User variable `CP` points to the top of dictionary. Increment `CP` by `n`.

```
;   ALLOT   ( n -- )
;     Allocate n bytes to the ram area.

      DCD   _TICK-MAPOFFSET
_ALLOT      DCB  5
      DCB   "ALLOT"
      ALIGN
ALLOT
      _NEST
      BL    CPP
```

```
       BL    PSTOR
       _UNNEST                 ;adjust code pointer
```

, (comma)	It is the most primitive compiler command. It compiles an integer `w` to the top of dictionary. It usually adds a new item to the growing token list of the current command under construction. This is the primitive compiler upon which the Forth compiler rests.

```
;   ,    ( w -- )
;     Compile an integer into the code dictionary.

       DCD   _ALLOT-MAPOFFSET
_COMMA       DCB  1,","
       ALIGN
COMMA
       _NEST
       BL    HERE
       BL    DUPP
       BL    CELLP ;cell boundary
       BL    CPP
       BL    STORE
       BL    STORE
       _UNNEST     ;adjust code pointer, compile
```

[COMPILE]	Compile the code field address of the next command in the input stream. It is used to compile immediate commands, which would otherwise be executed while compiling.

```
;   [COMPILE]   ( -- ; string> )
;     Compile the next immediate word into code dictionary.

       DCD   _COMMA-MAPOFFSET
_BCOMP       DCB  IMEDD+9
       DCB   "[COMPILE]"
       ALIGN
BCOMP
       _NEST
       BL    TICK
       BL    COMMA
       _UNNEST
```

COMPILE	Compile the code field address of the next command in the input stream. It forces compilation of a command at run time.

```
;   COMPILE ( -- )
;     Compile the next address in colon list to code dictionary.

       DCD   _BCOMP-MAPOFFSET
_COMPI       DCB  COMPO+7
       DCB   "COMPILE"
       ALIGN
COMPI
       _NEST
       BL    RFROM
       BIC   R5,R5,#1
       BL    DUPP
       BL    AT
```

```
        BL     CALLC              ;compile BL instruction
        BL     CELLP
        ORR    R5,R5,#1
        BL     TOR
        _UNNEST                   ;adjust return address
```

LITERAL	Compile an integer literal structure. It first compiles a `BL doLIT` machine instruction, followed by an integer `w`. When `doLIT` is executed in run time, it extracts this integer in the next program word and pushes it on the parameter stack.

```
;   LITERAL ( w -- )
;     Compile tos to code dictionary as an integer literal.

        DCD    _COMPI-MAPOFFSET
_LITER         DCB  IMEDD+7
        DCB    "LITERAL"
        ALIGN
LITER
        _NEST
        BL     COMPI
        DCD    DOLIT-MAPOFFSET
        BL     COMMA
        _UNNEST
```

$,"	Compile a string literal structure. String text is taken from the input stream and terminated by a double quote. A string token (such as `."|` or `$"|`) must be compiled before the string to initiate this sting literal structure.

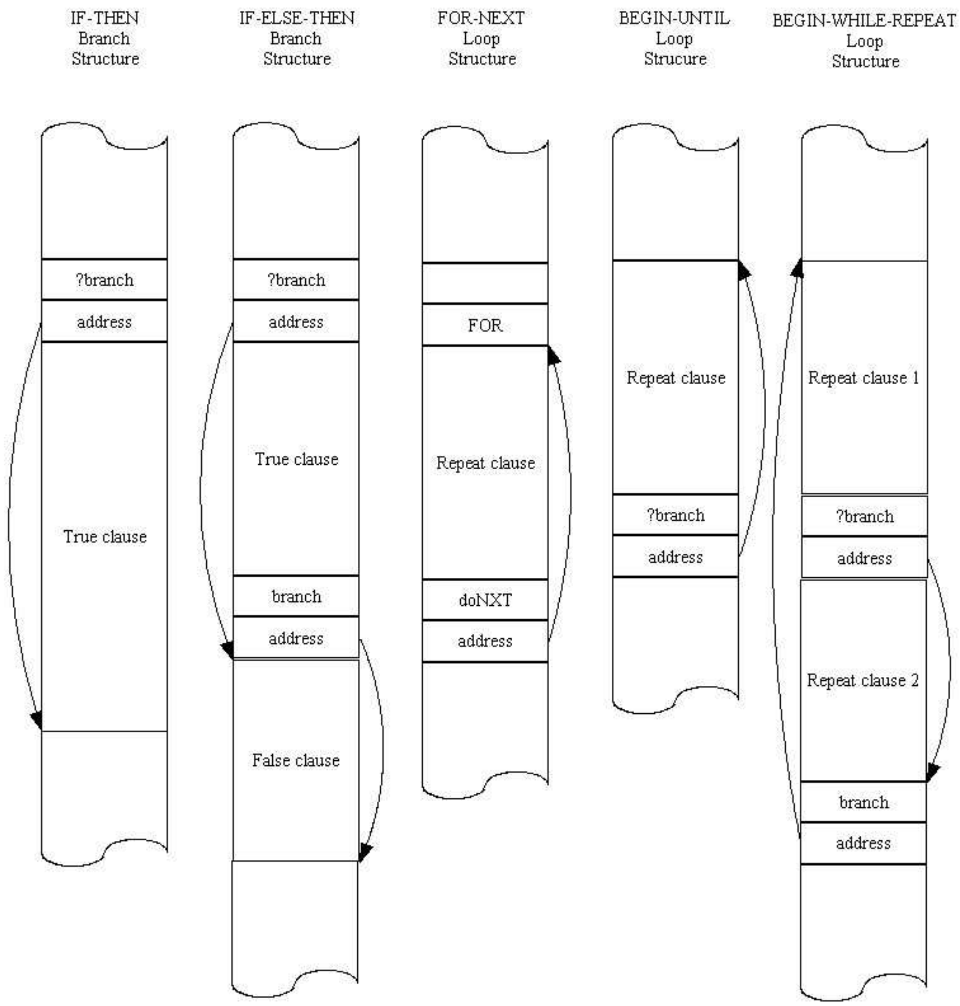
```
;   $,"      ( -- )
;     Compile a literal string up to next " .

;       DCD    _LITER-MAPOFFSET
;_STRCQ        DCB  3
;       DCB    "$$,"""
;       ALIGN
STRCQ
        _NEST
        _DOLIT
        DCD    -4
        BL     CPP
        BL     PSTOR
        _DOLIT
        DCD    '\"'
        BL     WORDD              ;moveDCB to code dictionary
        BL     COUNT
        BL     PLUS
        BL     ALGND              ;calculate aligned end ofDCB
        BL     CPP
        BL     STORE
        _UNNEST                   ;adjust the code pointer
```

3.5.3 Structure Commands

Immediate commands are not compiled as tokens by the compiler. Instead, they are executed by the compiler immediately. They are used to build control structures in the token lists of compound commands. Immediate commands has its `IMMEDIATE` lexicon bit set, in the length byte of the name field. The control structures used in eForth are the following:

Conditional branch	`IF ... THEN`
	`IF ... ELSE ... THEN`
Finite loop	`FOR ... NEXT`
	`FOR ... AFT ... THEN... NEXT`
Infinite loop	`BEGIN ... AGAIN`
Indefinite loop	`BEGIN ... UNTIL`
	`BEGIN ... WHILE ... REPEAT`

A control structure contains one or more address literals with `BL ?branch, BL branch` and `BL next` tokens, which cause execution to branch out of the normal sequence. The control structure commands are immediate commands which compile the address literal and resolve the branch address. These control structures are shown in the following figure:



One should note that `BEGIN` and `THEN` do not compile any token. They set up or resolve control structures in a token list. `IF`, `ELSE`, `WHILE`, `UNTIL`, and `AGAIN` do compile address literals with branching tokens.

I use two characters `a` and `A` to denote different addresses on the parameter stack. `a` points to a location to where a branch commands will jump to. `A` points to a location where a new address will be stored when the address is resolved.

FOR	Compile a `BL TOR` token and pushes the address of the next token `a` on the parameter stack. It starts a `FOR-NEXT` loop.

```
;***************************************************************************
; Structures

;   FOR      ( -- a )
;     Start a FOR-NEXT loop structure in a colon definition.

      DCD   _LITER-MAPOFFSET
_FOR  DCB  IMEDD+3
      DCB   "FOR"
      ALIGN
FOR
      _NEST
      BL    COMPI
      DCD   TOR-MAPOFFSET
      BL    HERE
      _UNNEST
```

BEGIN	Start a loop structure. It pushes an address `a` on the parameter stack. `a` points to the top of the dictionary where new tokens will be compiled. If begins an infinite loop or an indefinite loop.

```
;   BEGIN   ( -- a )
;     Start an infinite or indefinite loop structure.

      DCD   _FOR-MAPOFFSET
_BEGIN      DCB  IMEDD+5
      DCB   "BEGIN"
      ALIGN
BEGIN
      _NEST
      BL    HERE
      _UNNEST
```

NEXT	Compile a `BL next` token with a target address `a` on the top of the parameter stack. It resolves a `FOR NEXT` loop.

```
;   NEXT    ( a -- )
;     Terminate a FOR-NEXT loop structure.

      DCD   _BEGIN-MAPOFFSET
_NEXT DCB  IMEDD+4
```

```
       DCB   "NEXT"
       ALIGN
NEXT
       _NEST
       BL    COMPI
       DCD   DONXT-MAPOFFSET
       BL    COMMA
       _UNNEST
```

UNTIL	Compile a `BL ?branch` token with a target address `a` on the top of the parameter stack. It resolves a `BEGIN-UNTIL` indefinite loop.

```
;   UNTIL   ( a -- )
;     Terminate a BEGIN-UNTIL indefinite loop structure.

       DCD   _NEXT-MAPOFFSET
_UNTIL       DCB  IMEDD+5
       DCB   "UNTIL"
       ALIGN
UNTIL
       _NEST
       BL    COMPI
       DCD   QBRAN-MAPOFFSET
       BL    COMMA
       _UNNEST
```

AGAIN	Compile a `BL branch` token with a target address `a` on the top of the parameter stack. It resolves a `BEGIN-AGAIN` infinite loop.

```
;   AGAIN   ( a -- )
;     Terminate a BEGIN-AGAIN infinite loop structure.

       DCD   _UNTIL-MAPOFFSET
_AGAIN       DCB  IMEDD+5
       DCB   "AGAIN"
       ALIGN
AGAIN
       _NEST
       BL    COMPI
       DCD   BRAN-MAPOFFSET
       BL    COMMA
       _UNNEST
```

IF	Compile a `BL ?branch` address literal and pushes its address, `a`, is left on the parameter stack. It starts an `IF-ELSE-THEN` or an `IF-THEN` branch structure.

```
;   IF        ( -- A )
;     Begin a conditional branch structure.

       DCD   _AGAIN-MAPOFFSET
_IFF   DCB  IMEDD+2
       DCB   "IF"
       ALIGN
IFF
       _NEST
```

```
        BL      COMPI
        DCD     QBRAN-MAPOFFSET
        BL      HERE
        _DOLIT
        DCD     4
        BL      CPP
        BL      PSTOR
        _UNNEST
```

AHEAD	Compile a `BL branch` address literal and pushes its next address `A` on the parameter stack. It starts a `AHEAD-THEN` branch structure.

```
;   AHEAD   ( -- A )
;     Compile a forward branch instruction.

        DCD     _IFF-MAPOFFSET
_AHEAD          DCB     IMEDD+5
        DCB     "AHEAD"
        ALIGN
AHEAD
        _NEST
        BL      COMPI
        DCD     BRAN-MAPOFFSET
        BL      HERE
        _DOLIT
        DCD     4
        BL      CPP
        BL      PSTOR
        _UNNEST
```

REPEAT	Compile a `BL branch` token with a target address `a` on the top of the parameter stack. It resolves the address of `BL ?branch` token at `A` left by `WHILE`. It terminates a `BEGIN-WHILE-REPEAT` indefinite loop structure.

```
;   REPEAT  ( A a -- )
;     Terminate a BEGIN-WHILE-REPEAT indefinite loop.

        DCD     _AHEAD-MAPOFFSET
_REPEA          DCB     IMEDD+6
        DCB     "REPEAT"
        ALIGN
REPEA
        _NEST
        BL      AGAIN
        BL      HERE
        BL      SWAP
        BL      STORE
        _UNNEST
```

THEN	Resolve the address in a `BL branch` token whose address is `A` on the top of the parameter stack. It resolves a `IF-ELSE-TEHN` or `IF-THEN` branch structure.

```
;   THEN    ( A -- )
;     Terminate a conditional branch structure.
```

```
        DCD     _REPEA-MAPOFFSET
_THENN          DCB   IMEDD+4
        DCB     "THEN"
        ALIGN
THENN
        _NEST
        BL      HERE
        BL      SWAP
        BL      STORE
        _UNNEST
```

AFT	Compile a `BL branch` literal and leaves its address as `A` on stack, It also replaces the address `a` left by `FOR` with the address `a1` of the next token. `A` will be used by THEN to resolve the `AFT-THEN` branch structure, and `a1` will be used by `NEXT` to resolve the loop structure.

```
;   AFT      ( a – a1 A )
;     Jump to THEN in a FOR-AFT-THEN-NEXT loop the first time through.

        DCD     _THENN-MAPOFFSET
_AFT    DCB   IMEDD+3
        DCB     "AFT"
        ALIGN
AFT
        _NEST
        BL      DROP
        BL      AHEAD
        BL      BEGIN
        BL      SWAP
        _UNNEST
```

ELSE	Compile a `BL branch` token, and use the address of the next token to resolve the address field of `BL ?branch` token in `a`, as left by `IF`. It also replaces `a` with `A`, the address of its address field for `THEN` to resolve. `ELSE` starts the false clause in the `IF-ELSE-THEN` branch structure.

```
;   ELSE    ( A -- A )
;     Start the false clause in an IF-ELSE-THEN structure.

        DCD     _AFT-MAPOFFSET
_ELSEE          DCB   IMEDD+4
        DCB     "ELSE"
        ALIGN
ELSEE
        _NEST
        BL      AHEAD
        BL      SWAP
        BL      THENN
        _UNNEST
```

WHILE	Compile a `BL ?branch` token and leave its address, `A`, on the stack. Address `a` left by `BEGIN` is swapped to the top of the parameter stack. `WHILE` is used to start the true clause

	in the BEGIN-WHILE-REPEAT loop.

```
;   WHILE   ( a -- A a )
;     Conditional branch out of a BEGIN-WHILE-REPEAT loop.

      DCD   _ELSEE-MAPOFFSET
_WHILE      DCB  IMEDD+5
      DCB   "WHILE"
      ALIGN
WHILE
      _NEST
      BL    IFF
      BL    SWAP
      _UNNEST
```

ABORT"	Compile an error message as a string literal structure. This error message is display at run time if the top item on the parameter stack is true, and the rest of the tokens in this compound command are skipped and eForth enters the interpreter loop in QUIT. This is the programmed response to an error condition.

```
;   ABORT"  ( -- ; string> )
;     Conditional abort with an error message.

      DCD   _WHILE-MAPOFFSET
_ABRTQ      DCB  IMEDD+6
      DCB   "ABORT\""
      ALIGN
ABRTQ
      _NEST
      BL    COMPI
      DCD   ABORQ-MAPOFFSET
      BL    STRCQ
      _UNNEST
```

$"	Compile a string literal structure. When it is executed in run time, only the address of the string is pushed on the parameter stack. Later commands can use this address to access the string and individual characters in the string as a string array.

```
;   $"      ( -- ; string> )
;     Compile an inlineDCB literal.

      DCD   _ABRTQ-MAPOFFSET
_STRQ DCB  IMEDD+2
      DCB   "$$"""
      ALIGN
STRQ
      _NEST
      BL    COMPI
      DCD   STRQP-MAPOFFSET
      BL    STRCQ
      _UNNEST
```

."	Compile a string literal structure which will print a text string when it is executed in run time. This is the best way to present messages to user in an application.

```
;   ."      ( -- ; string> )
;     Compile an inlineDCB literal to be typed out at run time.

      DCD   _STRQ-MAPOFFSET
_DOTQ DCB  IMEDD+2
      DCB   ".""
      ALIGN
DOTQ
      _NEST
      BL    COMPI
      DCD   DOTQP-MAPOFFSET
      BL    STRCQ
      _UNNEST
```

3.5.4 String Compiler

We had seen how tokens and structures are compiled into the code field of a compound command in the dictionary. To build a new command, we have to build its header first. A header consists of a link field and a name field. Here are the commands to build the header.

?UNIQUE	Display a warning message to show that the name of a new command already exists in the dictionary. Forth does not prevent your reusing the same name for different commands. However, giving the same name to many different commands often causes problems in software projects. It is to be avoided if possible and `?UNIQUE` reminds you of it.

```
;**************************************************************************
; Name compiler

;   ?UNIQUE ( a -- a )
;     Display a warning message if the word already exists.

      DCD   _DOTQ-MAPOFFSET
_UNIQU      DCB  7
      DCB   "?UNIQUE"
      ALIGN
UNIQU
      _NEST
      BL    DUPP
      BL    NAMEQ             ;?name exists
      BL    QBRAN
      DCD   UNIQ1-MAPOFFSET   ;redefinitions are OK
      BL    DOTQP
      DCB   7
      DCB   " reDef "         ;but warn the user
      ALIGN
      BL    OVER
      BL    COUNT
      BL    TYPEE             ;just in case its not planned
UNIQ1 BL    DROP
      _UNNEST
```

$,n	Build a new header with a name string at memory address `na`. It first builds a link field with an address pointing to the name field of the prior command. At this point, the parser

	had already packed the name into the name field. Move the dictionary pointer CP to the end of this name field, and the header is complete. The top of dictionary now is the code field of the new command, and tokens can be compiled.

```
;   $,n      ( na -- )
;     Build a new dictionary name using the data at na.

;     DCD   _UNIQU-MAPOFFSET
;_SNAME     DCB  3
;     DCB   "$$,n"
;     ALIGN
SNAME
      _NEST
      BL    DUPP              ; na na
      BL    CAT               ; ?null input
      BL    QBRAN
      DCD   SNAM1-MAPOFFSET
      BL    UNIQU             ; na
      BL    LAST              ; na last
      BL    AT                ; na la
      BL    COMMA             ; na
      BL    DUPP              ; na na
      BL    LAST              ; na na last
      BL    STORE             ; na , save na for vocabulary link
      BL    COUNT             ; na+1 count
      BL    PLUS              ; na+1+count
      BL    ALGND             ; word boundary
      BL    CPP
      BL    STORE             ; top of dictionary now
      _UNNEST
SNAM1
      BL    STRQP
      DCB   7," name? "
      B.W   ABORT
```

$COMPILE	Build the token list of a new compound command in its code field, which is on the top of the dictionary. It takes a string address a on the top of the parameter stack, search dictionary for a matching token, and appends the token to the token list. If the string is not a valid command, it is converted to a number, and a integer literal is appended to the token list. If the string is not a number, abort the compilation process and return to the text interpreter loop in QUIT. If the string is the name of an immediate command, this command is not compiled, but executed immediately. Immediate commands are tools used by the compiler to build structures in a token list.

```
;   $COMPILE      ( a -- )
;     Compile next word to code dictionary as a token or literal.

      DCD   _UNIQU-MAPOFFSET
_SCOMP      DCB  8
      DCB   "$$COMPILE"
      ALIGN
SCOMP
      _NEST
      BL    NAMEQ
```

```
        BL    QDUP  ;defined?
        BL    QBRAN
        DCD   SCOM2-MAPOFFSET
        BL    AT
        _DOLIT
        DCD   IMEDD
        BL    ANDD  ;immediate?
        BL    QBRAN
        DCD   SCOM1-MAPOFFSET
        BL    EXECU
        _UNNEST                 ;it's immediate, execute
SCOM1 BL    CALLC               ;it's not immediate, compile
        _UNNEST
SCOM2 BL    NUMBQ
        BL    QBRAN
        DCD   SCOM3-MAPOFFSET
        BL    LITER
        _UNNEST                 ;compile number as integer
SCOM3 B.W   ABORT               ;error
```

OVERT	Link a new command to the dictionary and thus makes it available for dictionary searches. When a new header is build, its name field address is stored in system variable LAST, and it is not yet linked to the dictionary which starts at CONTEXT. OVERT copies the name field address in LAST to CONTEXT and links the new command to the dictionary. It is used to protect the dictionary so that new commands not compiled successfully will not be linked incorrectly into the dictionary.

```
;   OVERT   ( -- )
;     Link a new word into the current vocabulary.

        DCD   _SCOMP-MAPOFFSET
_OVERT      DCB  5
        DCB   "OVERT"
        ALIGN
OVERT
        _NEST
        BL    LAST
        BL    AT
        BL    CNTXT
        BL    STORE
        _UNNEST
```

;	Terminate a new compound command. It compiles an _UNNEST machine instruction to terminate the new token list, links this new command to the dictionary, and then returns to interpreting mode by storing the code field address of $INTERPRET into user variable 'EVAL.

```
;   ;    ( -- )
;     Terminate a colon definition.

        DCD   _OVERT-MAPOFFSET
_SEMIS      DCB  IMEDD+COMPO+1
        DCB   ";"
        ALIGN
```

```
SEMIS
      _NEST
      _DOLIT
      _UNNEST
      BL    COMMA
      BL    LBRAC
      BL    OVERT
      _UNNEST
```

]	Turn the text interpreter to a compiler by storing the code field address of `$COMPILE` into user variable `'EVAL`..

```
;   ]    ( -- )
;     Start compiling the words in the input stream.

      DCD   _SEMIS-MAPOFFSET
_RBRAC      DCB  1
      DCB   "]"
      ALIGN
RBRAC
      _NEST
      _DOLIT
      DCD   SCOMP-MAPOFFSET
      BL    TEVAL
      BL    STORE
      _UNNEST
```

3.5.5 Branch and Link Token

In STM32F4, subroutine call uses the Branch and Link `BL<addr>` instruction. All high level compound commands are assembled as tokens of BL instructions. BL instruction, as invented in the ARM RISC architecture, assumed a return stack of 1 level. If the called subroutine had to call other subroutines, the return address in LR had to be saved on a real return stack of adequate depth. (In eForth, the return stack and the parameter stack run to about 20 levels deep. 64 levels are reserved for the return stack. About 16K levels are available for the parameter stack.)

In the uVision5 debugger, I watched the disassembled BL instructions while single stepping through the code, but could not figure out how the instructions were encoded. Only when I was testing the decompiler command SEE, I had to figure it out without a shiver of doubt. It is composed of two 16-bit THUMB2 instructions in the form of:

11111	Address Bits 22-11	11110	Address Bits 11-1
Byte 1	Byte 0	Byte 3	Byte 2

Very strange, indeed! But, I was able to shift the bits around and eventually get the correct address out.

BL.W	Compile or assemble a `BL` instruction as a token. The destination address `ca` is on the parameter stack. Compound commands are compiled as lists of BL tokens.

```
;   BL.W    ( ca -- )
;     Assemble a branch-link long instruction to ca.
;     BL.W is split into 2 16 bit instructions with 11 bit address fields.

;     DCD   _RBRAC-MAPOFFSET
;_CALLC     DCB  5
;     DCB   "call,"
;     ALIGN
CALLC
      _NEST
      BIC   R5,R5,#1          ; clear b0 of address from R>
      BL    HERE
      BL    SUBB
      SUB   R5,R5,#4          ; pc offset
      MOVW  R0,#0x7FF         ; 11 bit mask
      MOV   R4,R5
      LSR   R5,R5,#12         ; get bits 22-12
      AND   R5,R5,R0
      LSL   R4,R4,#15         ; get bits 11-1
      ORR   R5,R5,R4
      ORR   R5,R5,#0xF8000000
      ORR   R5,R5,#0xF000
      BL    COMMA             ; assemble BL.W instruction
      _UNNEST
      ALIGN
```

: (colon)	Create a new header and start a new compound command. It takes the following string in the input stream to be the name of the new command. The dictionary is ready to accept a token list.] turns the text interpreter into compiler, which will compile the following text strings to build a new compound command. The new compound command will then be terminated by ;.

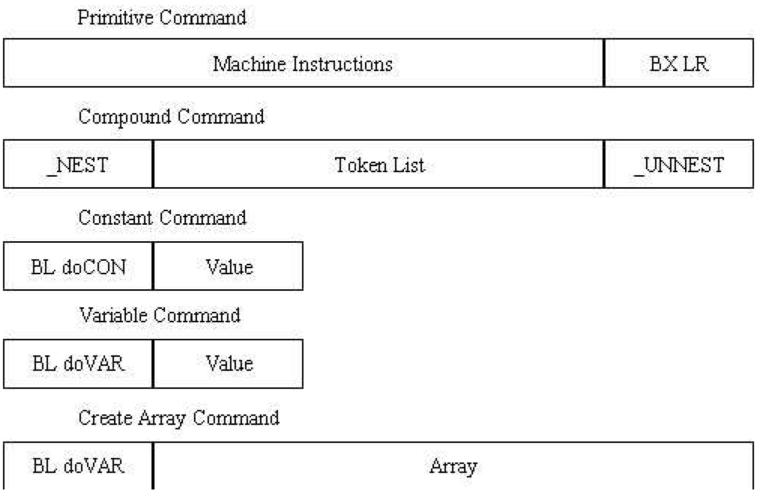
```
;     :     ( -- ; string> )
;     Start a new colon definition using next word as its name.

      DCD   _RBRAC-MAPOFFSET
_COLON      DCB  1
      DCB   ":"
      ALIGN
COLON
      _NEST
      BL    TOKEN
      BL    SNAME
      _DOLIT
      _NEST
      BL    COMMA
      BL    RBRAC
      _UNNEST
```

IMMEDIATE	Set the immediate lexicon bit in the name field of the new command. When the compiler encounters a command with this bit set, it will not compile this command into the token list under construction, but execute it immediately. This bit allows immediate commands to build special structures in compound commands, and to deal with special conditions while compiling.

```
;   IMMEDIATE   ( -- )
;     Make the last compiled word an immediate word.

      DCD   _COLON-MAPOFFSET
_IMMED      DCB  9
      DCB   "IMMEDIATE"
      ALIGN
IMMED
      _NEST
      _DOLIT
      DCD   IMEDD
      BL    LAST
      BL    AT
      BL    AT
      BL    ORR
      BL    LAST
      BL    AT
      BL    STORE
      _UNNEST
```

3.5.6 Defining Commands

Defining commands are molds which can be used to create classes of commands which share the same run time behavior. In stm32eForth720, we have the following defining commands: `:` , `CREATE`, `CONSTANT` and `VARIABLE`. The contents of the code fields in different classes of commands are shown in the following figure:



CONSTANT	Create a new constant command with a `BL doCON` token followed by the constant value `u`. When a constant command is executed, it pushes the constant value on the parameter stack.

```
;*************************************************************************
; Defining words

;   CONSTANT      ( u -- ; string> )
```

```
;     Compile a new constant.

      DCD   _IMMED-MAPOFFSET
_CONST      DCB  8
      DCB   "CONSTANT"
      ALIGN
CONST
      _NEST
      BL    TOKEN
      BL    SNAME
      BL    OVERT
      _DOLIT
      _NEST
      BL    COMMA
      _DOLIT
      DCD   DOCON-MAPOFFSET
      BL    CALLC
      BL    COMMA
      _UNNEST
```

CREATE	Create a new command with a `BL doVAR` token. It creates a data array in dictionary without allocating memory. When a command created by `CREATE` is executed, it will push the address after `BL doVAR` token on the parameter stack. Memory space of an actual array is allocated using `ALLOT` command.

```
;   CREATE  ( -- ; string> )
;     Compile a new array entry without allocating code space.

      DCD   _CONST-MAPOFFSET
_CREAT      DCB  6
      DCB   "CREATE"
      ALIGN
CREAT
      _NEST
      BL    TOKEN
      BL    SNAME
      BL    OVERT
      _DOLIT
      _NEST
      BL    COMMA
      _DOLIT
      DCD   DOVAR-MAPOFFSET
      BL    CALLC
      _UNNEST
```

VARIABLE	Create a new variable command with a `BL doVAR` token followed by one 32-bit memory cell. This memory cell is initialized to 0, its address is returned when the variable command is executed. Its contents can be read by @ command and written by ! command.

```
;   VARIABLE      ( -- ; string> )
;     Compile a new variable initialized to 0.

      DCD   _CREAT-MAPOFFSET
```

```
_VARIA      DCB  8
      DCB   "VARIABLE"
      ALIGN
VARIA
      _NEST
      BL    CREAT
      _DOLIT
      DCD   0
      BL    COMMA
      _UNNEST
```

3.6 Debugging Tools

Stm32eForth720 is a very small system and only a very small set of tool commands is provided for debugging. Nevertheless, this set of tool commands is powerful enough to help you debug new commands you add to the system. They are also very interesting programming examples on how to use the commands in eForth to build substantial applications.

Generally, the tool commands present information stored in different parts of the CPU in appropriate formats to let you inspect the results as you execute commands in the eForth system and commands you defined yourself. The tool commands include memory dump, stack dump, dictionary dump, and a compound command decompiler..

3.6.1 Memory Dump

This tool allows you inspect memory at any address, RAM, flash, and IO registers. You can dump data and inspect code. You can use it to monitor and control IO devices. It makes you feel that you are the master of your computer.

dm+	Print `u` bytes of data starting at address `a` to the terminal. It returns a new address `a+u` on the stack to facilitate dumping of the next line of memory.

```
;**************************************************************************
; Tools

;   dm+      ( a u -- a )
;     Dump u bytes from a, leaving a+u on the stack.

;     DCD   _VARIA-MAPOFFSET
;_DMP DCB  3
;     DCB   "dm+"
;     ALIGN
DMP
      _NEST
      BL    OVER
      _DOLIT
      DCD   4
      BL    UDOTR             ;display address
      BL    SPACE
      BL    TOR               ;start count down loop
      B.W   PDUM2             ;skip first pass
PDUM1   BL  DUPP
      BL    CAT
      _DOLIT
      DCD   3
      BL    UDOTR             ;display numeric data
      BL    ONEP              ;increment address
PDUM2   BL  DONXT
      DCD   PDUM1-MAPOFFSET   ;loop till done
      _UNNEST
```

DUMP	Print an array, `u` bytes of data starting at address `b,` to the terminal. It dumps 16 bytes to a line. A line begins with the address of the first byte, followed by 16 bytes shown in hex, 3

	columns per bytes. At the end of a line are the 16 bytes shown in ASCII characters. Non-printable characters are replaced by underscores (ASCII 95).

```
;   DUMP    ( a u -- )
;     Dump u bytes from a, in a formatted manner.

      DCD   _VARIA-MAPOFFSET
_DUMP DCB  4
      DCB   "DUMP"
      ALIGN
DUMP
      _NEST
      BL    BASE
      BL    AT
      BL    TOR
      BL    HEX                ;save radix,set hex
      _DOLIT
      DCD   16
      BL    SLASH              ;change count to lines
      BL    TOR
      B.W   DUMP4              ;start count down loop
DUMP1   BL  CR
      _DOLIT
      DCD   16
      BL    DDUP
      BL    DMP                ;display numeric
      BL    ROT
      BL    ROT
      BL    SPACE
      BL    SPACE
      BL    TYPEE              ;display printable characters
DUMP4   BL  DONXT
      DCD   DUMP1-MAPOFFSET    ;loop till done
DUMP3   BL  DROP
      BL    RFROM
      BL    BASE
      BL    STORE              ;restore radix
      _UNNEST
```

3.6.2 Parameter Stack Dump

One important discipline in learning Forth is to learn how to use the parameter stack correctly and effectively. All commands must consume their input parameters on the stack and leave only their intended results on the stack. Sloppy usage of the parameter stack is often the cause of bugs which are very difficult to detect later, as unexpected items left on the stack could result in unpredictable behavior. `.S` should be used liberally during programming and debugging to ensure that the correct parameters are consumed and left on the parameter stack.

The parameter stack is the center for arithmetic and logic operations. It is where commands receive their parameters and also where they left their results. In debugging a new command which may use stack items and leave items on the stack, the best was to debug it is to inspect the parameter stack, before and after its execution. To inspect the parameter stack non-destructively, use the command `.S`.

.S	Print the contents of the parameter stack in the free format. The bottom of the stack is aligned to the left margin. The top item is shown towards the right and followed by the characters `ok`. `.S` does not change the parameter stack so it can be used to inspect the parameter stack non-destructively at any time.

```
;   .S        ( ... -- ... )
;     Display the contents of the data stack.

      DCD   _DUMP-MAPOFFSET
_DOTS DCB  2
      DCB   ".S"
      ALIGN
DOTS
      _NEST
      BL    SPACE
      BL    DEPTH            ;stack depth
      BL    TOR              ;start count down loop
      B.W   DOTS2            ;skip first pass
DOTS1   BL  RAT
      BL    PICK
      BL    DOT              ;index stack, display contents
DOTS2   BL  DONXT
      DCD   DOTS1-MAPOFFSET  ;loop till done
      BL    SPACE
      _UNNEST
```

`>NAME` finds the name field address of a word from the corresponding code field address in a command record. If the command does not exist in the dictionary, it returns a false flag. It is the mirror image of the command `NAME>`, which returns the code field address of a command from its name field address. However, it is very difficult to scan backward from code field to locate the beginning of the name field, because we do not know how long the name field is. `>NAME` is therefore more complicated because the entire dictionary must be searched to locate its name field.

>NAME	Return a name field address, `na`, of a command from its code field address, `ca`. If `ca` is not a valid code field address, or if the code field does not have an header, return 0. It follows the linked list of the dictionary, and from every name field address we can get a corresponding code field address. If this address is not the same as `ca`, we go to the name field of the next command. If `ca` is a valid code field address with an header, we surely will find it. If the entire dictionary is searched and `ca` is not found, it is not a valid code field address or it does not have an header, and a false flag is returned.

```
;   >NAME   ( ca -- na | F )
;     Convert code address to a name address.

      DCD   _DOTS-MAPOFFSET
_TNAME      DCB  5
      DCB   ">NAME"
      ALIGN
TNAME
      _NEST
      BL    TOR              ;
```

```
      BL    CNTXT             ; va
      BL    AT                ; na
TNAM1
      BL    DUPP              ; na na
      BL    QBRAN
      DCD   TNAM2-MAPOFFSET   ; vocabulary end, no match
      BL    DUPP              ; na na
      BL    NAMET             ; na ca
      BL    RAT               ; na ca code
      BL    XORR              ; na f --
      BL    QBRAN
      DCD   TNAM2-MAPOFFSET
      BL    CELLM             ; la
      BL    AT                ; next_na
      B.W   TNAM1
TNAM2
      BL    RFROM
      BL    DROP              ; 0|na --
      _UNNEST                 ;0
```

.ID	Display the name of a command, given the name field address na of this command. It replaces non-printable characters in a name by under-scores.

```
;   .ID      ( na -- )
;     Display the name at address.

      DCD   _TNAME-MAPOFFSET
_DOTID      DCB  3
      DCB   ".ID"
      ALIGN
DOTID
      _NEST
      BL    QDUP              ;if zero no name
      BL    QBRAN
      DCD   DOTI1-MAPOFFSET
      BL    COUNT
      _DOLIT
      DCD   0x1F
      BL    ANDD              ;mask lexicon bits
      BL    TYPEE
      _UNNEST                 ;display name string
DOTI1   BL  DOTQP
      DCB   9
      DCB   " {noName}"
      ALIGN
      _UNNEST
```

3.6.3 Compound Command Decompiler

In the cold field of a compound command, there is a token list of BL instructions. It is very easy to extract the code field addresses from the BL tokens. If the token has a name field, we can display its name. This is the decompiler. If the token does not have a name field, or it is a piece of data, the decompiler simply displays its value, and let you figure out what it really means.

The decompiler is very useful in recovering the source code of a command in the dictionary when the source code listing is not immediately available, or non-existent. It is also useful to check on a new command you just compiled, to see if the computer is thinking what you are thinking. Computer is a “Do what you say, not what you mean” device. It always helps to check that what you say is actually what you mean with the decompiler.

SEE	Search dictionary for a command with the name in the following string. If it is a valid command, decompile the token list in its code field.

```
;   SEE      ( -- ; string> )
;     A simple decompiler.

      DCD   _DOTID-MAPOFFSET
_SEE  DCB  3
      DCB   "SEE"
      ALIGN
SEE
      _NEST
      BL    TICK  ; ca --, starting address
      BL    CR
      _DOLIT
      DCD   20
      BL    TOR
SEE1  BL    CELLP            ; a
      BL    DUPP             ; a a
      BL    DECOMP           ; a
      BL    DONXT
      DCD   SEE1-MAPOFFSET
      BL    DROP
      _UNNEST
```

DECOMILE	Search dictionary for a command whose code field address is in memory address a. If it is a valid command, display its name; otherwise, display its value.

```
;     DECOMPILE ( a -- )
;     Convert code in a.  Display name of command or as data.

      DCD   _SEE-MAPOFFSET
_DECOM      DCB  9
      DCB   "DECOMPILE"
      ALIGN

DECOMP
      _NEST
      BL    DUPP             ; a a
;     BL    TOR              ; a
      BL    AT               ; a code
      BL    DUPP             ; a code code
      _DOLIT
      DCD   0xF800F800
      BL    ANDD
      _DOLIT
      DCD   0xF800F000
      BL    EQUAL            ; a code ?
```

```
      BL    QBRAN
      DCD   DECOM2-MAPOFFSET  ; not a command
      ; a valid_code --, extract address and display name
      MOVW  R0,#0xFFE
      MOV   R4,R5
      LSL   R5,R5,#21         ; get bits 22-12
      ASR   R5,R5,#9          ; with sign extension
      LSR   R4,R4,#15         ; get bits 11-1
      AND   R4,R4,R0          ; retain only bits 11-1
      ORR   R5,R5,R4          ; get bits 22-1
      NOP
      BL    OVER              ; a offset a
      BL    PLUS              ; a target-4
      BL    CELLP             ; a target
      BL    TNAME             ; a na/0 --, is it a name?
      BL    QDUP              ; name address or zero
      BL    QBRAN
      DCD   DECOM1-MAPOFFSET
      BL    SPACE             ; a na
      BL    DOTID             ; a --, display name
;     BL    RFROM             ; a
      BL    DROP
      _UNNEST
DECOM1      ;BL   RFROM       ; a
      BL    AT                ; data
      BL    UDOT              ; display data
      _UNNEST
DECOM2      BL    UDOT
;     BL    RFROM
      BL    DROP
      _UNNEST
```

3.6.4 Dictionary Dump

The dictionary contains all command records defined in the system, ready for execution and compilation. `WORDS` command allows you to examine the dictionary and to look for the correct names of commands in case you are not sure of their spellings. `WORDS` follows the dictionary link in the system variable `CONTEXT` and displays the names of all commands in the dictionary. The dictionary links can be traced easily because the link field in the header of a command points to the name field of the previous command, and the link field is two bytes below the corresponding name field.

WORDS	Display all the names in the dictionary. The order of words is reversed from the compiled order. The last defined command is shown first.

```
;   WORDS   ( -- )
;     Display the names in the context vocabulary.

      DCD   _DECOM-MAPOFFSET
_WORDS      DCB  5
      DCB   "WORDS"
      ALIGN
WORDS
      _NEST
```

```
      BL    CR
      BL    CNTXT
      BL    AT                ;only in context
WORS1
      BL    QDUP              ;?at end of list
      BL    QBRAN
      DCD   WORS2-MAPOFFSET
      BL    DUPP
      BL    SPACE
      BL    DOTID             ;display a name
      BL    CELLM
      BL    AT
      B.W   WORS1
WORS2
      _UNNEST
      ALIGN
```

3.6.5 Cold Start

After the STM32F407 is turned on, it starts executing initial machine code at `Reset_Handler` to set up the CPU hardware. Then it jumps to `COLD` to initialize the Virtual Forth Machine. It finally jumps to `QUIT` and starts the text interpreter. `COLD` and `QUIT` are the topmost layers of stm32eForth720 system.

Before falling into `QUIT` to enter into the text interpreter loop, `COLD` command executes an application routine whose code address is stored in user variable `'BOOT`. This code address can be vectored to a command which defines the proper behavior of the system on power-up and on reset. Initially `'BOOT` contains the code field address of `HI,` which simply displays a sign-on message.

VER	Combine the major version number VER and minor version number EXT and return a 32-bit number to be displayed in the sign-on message. VER and EXT are assembler equate constants.

```
;***************************************************************************
; cold start

;   VER      ( -- n )
;     Return the version number of this implementation.

;     DCD   _WORDS-MAPOFFSET
;_VERSN     DCB  3
;     DCB   "VER"
;     ALIGN
VERSN
      _NEST
      _DOLIT
      DCD   VER*256+EXT
      _UNNEST
```

HI	The default start-up routine in stm32eForth720. It displays a sign-on message with the correct version number. This is the default start up routine whose code field address is stored in the user variable ‘BOOT. From ‘BOOT you can initialize the system to start your

	own application.

```
;   HI        ( -- )
;     Display the sign-on message of eForth.

      DCD   _WORDS-MAPOFFSET
_HI   DCB  2
      DCB   "HI"
      ALIGN
HI
      _NEST
      BL    CR
      BL    DOTQP
      DCB   13
      DCB   "stm32eForth v"   ;model
      ALIGN
      BL    BASE
      BL    AT
      BL    HEX   ;save radix
      BL    VERSN
      BL    BDIGS
      BL    DIG
      BL    DIG
      _DOLIT
      DCD   '.'
      BL    HOLD
      BL    DIGS
      BL    EDIGS
      BL    TYPEE ;format version number
      BL    BASE
      BL    STORE
      BL    CR
      _UNNEST                 ;restore radix
```

COLD	A high level compound command executed upon cold start, called froml `Reset_Hanlder` routine. Its initializes the CPU registers including the parameter stack, the return stack, and user variables, executes the boot-up routine vectored in `'BOOT`, and then falls into the text interpreter loop `QUIT`.

```
;   COLD    ( -- )
;     The high level cold start sequence.

      DCD   _HI-MAPOFFSET
LASTN DCB  4
      DCB   "COLD"
      ALIGN
COLD
; Initiate Forth registers
      MOVW R3,#0xFF00         ; user area
;     MOVT R3,#0x2000         ;
      MOV R2,R3         ; return stack
      SUB   R1,R2,#0x100      ; data stack
      MOV R5,#0               ; tos
      NOP
      _NEST
```

```
COLD1
      _DOLIT
      DCD   UZERO-MAPOFFSET
      _DOLIT
      DCD   UPP
      _DOLIT
      DCD   ULAST-UZERO
      BL    MOVE               ;initialize user area
      BL    PRESE              ;initialize stack and TIB
      BL    TBOOT
      BL    ATEXE              ;application boot
      BL    OVERT
      B.W   QUIT               ;start interpretation
      ALIGN
COLD2
CTOP
      DCD   0XFFFFFFFF         ; keep CTOP even
      END
```

3.7 Final Remarks

Never mind my badmouthing, STM32F4 is my dream Forth computer. All these years, I am looking for a microcontroller with lots of RAM, lots of programmable ROM, lots of GPIO pins, lots of communication channels, lots of counter-timers, lots of ADC, lots of DAC, fast clocks, low power consumption, small package, etc, etc. And it has to be cheap, too. Remember this saying?

Fast, big, and cheap. Pick two.

Actually, STM32F4 has them all.

Remember the old microcontroller development systems? The Intel blue box? You have a 19” rack with a big bus cage. A CPU board, a RAM board, a ROM board, many different IO boards, an EPROM programmer, an UV eraser, two floppy drives, and an expansive hard disk drive. All these things are now squeezed into a single chip, assembled on a small pc card, and selling for $20! What else do you want?

Coming with it, the software is complexity beyond belief. Black box approach? Third party libraries? C++ compiler? I don’t think these tools work at chip level for microcontrollers. You have to dive into the devices yourself and gain control over them. Only Forth gives you a fighting chance.

I tried to get Arduino Uno to play Bach’s organ pieces. It has only 3 counter-timers, and I could only play his 3-part music. Now, STM32F4 has 14 counter-timers. Old Bach will be very pleased with it.

There are 80 IO pins on STM32F4-Discovery Kit. A walking robot, perhaps?

A digital storage oscilloscope? Well, I need a good LCD display.

A remotely controlled telescope?

A high resolution digital spectrometer?

Well. Where is my retirement plan?