

# ceForth\_33

**Dr. Chen-Hanson Ting,**  
**July, 2019**

## Chapter 1. How to Write Forth in C

In 1990, when Bill Muench and I developed eForth Model for microcontrollers, memory was scarce and the only way to implement eForth was assembly. At that time, there were several Forth systems written in C, the most notable ones were Wil Baden's thisForth, and Mitch Bradley's Forthmacs. However, these two implementations were targeted to large computers, base on Unix environment. I studied them, but could not understand them in their convoluted make processes. I did not have sufficient knowledge on C and Unix to build eForth from scratch.

In Silicon Valley Forth Interest Group, we intermittently had long discussions on how to write Forth in C. John Harbold, also an expert C programmer, assured me that it was possible to write Forth in C, and showed me code on how to do it. But, they were way above my head.

In 2009, I started to think seriously on my problems with C, and problems in writing Forth in C. I realized that a Virtual Forth Machine (VFM) could be written easily in C, just like in any other assembly language. VFM was simply a set of Forth primitive words, very simple to write in assembly of a particular microcontroller, or in C, which was designed for an idealized, general purpose CPU, to emulate algebra. My problems were in the construction of a Forth dictionary. Forth dictionary is a linked list of Forth words, in the form of records. Each record has 4 fields, a fixed length link field, a variable length name field, a fixed length code field, and a variable length parameter field. The elementary C compiler, as I understood, did not have data constructs for building and linking of these records. You needed the convoluted ways in thisForth and Forthmac to build and link these records.

Chuck Moore showed me how to write assembler and how to build the dictionary in MuP21, in a metacompiler. I had used his metacompiler to build eForth systems for P8, P24, eP16 and eP32 chips. A Forth metacompiler was much more powerful than any macro assembler, and C. All I had to do was to allocate a huge data array, and built the dictionary with all the records. This data array could then be copied into VFM code in an assembly file, or in a C header file. If I defined VFM with a set of byte code as its pseudo instructions, the dictionary would contain only data and no executable C code. The beauty in byte code was that it completely isolated the Forth system from the underlying microcontroller, and the Forth system could be ported to any microcontroller with a C compiler.

In a direct threaded Forth model, a record of primitive word contains only byte code. A colon word has one cell of byte code in its code field, and a token list in its parameter field. Tokens are code field address of other Words.

Embedding Forth dictionary into a data array fits nicely with the fundamental programming model of C, in that executable C code are compiled into code segments, and data and variables are compiled into data segments. C as a compiled language does not execute code in data segments, and consider writing code or data into code segments illegal. Forth as an interpretive language, does not distinguish code from data, and encourages user to add new code into its dictionary. I made the compromise to put all VFM code in a code segment, and all Forth words in a data segment. I accept the limitation that no new pseudo instruction will be added to the baseline VFM, while new colon words can be added to the Forth dictionary freely.

The design of a Forth system can now be separated into two independent tasks: building a VFM machine targeting to various microcontrollers, including C, and building a Forth dictionary. You can use independent tools which are best suited for the particular task. I chose F# to build the Forth dictionary, because I had used it for years. Currently, C, C++, and C#. in my understanding, do not have the necessary tools to build the dictionary together with the VFM.

In 2009, I wrote two versions of eForth in C: ceForth 1.0 with 64 primitives, and ceForth 1.1 with 32 primitives. They were compiled by `gcc` under `cygwin`. I did them for my own ego, just to show that I could. I did not expect they could be used for any practical purpose.

In 2011, I was attracted to Arduino Uno Kit and ported eForth to it as 328eForth. One of the problems with this implementation was that it was not compatible with the prevailing Arduino IDE tool chain. I needed to add new Forth words to the dictionary in flash memory. Under Arduino, you were not allowed to write to flash memory at run time. To get the privilege of writing to flash memory, I had to take over the bootloader section which was monopolized by Arduino IDE to write to flash memory.

To accommodate Arduino, I ported ceForth 1.1 to Arduino Uno in the form of a sketch, `ceForth_328.cpp`, which was essentially a C program. Observing the restriction that I could not write anything into flash memory, I extended Forth dictionary in the RAM memory. It worked. However, you had only 1.5KB of RAM memory left over for new Forth words, and you could not save these new words before you lost power. As I stated then, it was only a teaser to entice new people to try Forth on Arduino Uno. For real applications, you had to use 328eForth.

In 2016, a friend, Derek Lai, in the Taiwan FIG group gave me a couple of WiFiBoy Kits he and his son Ricky built. It used an ESP8266 chip with an integrated WiFi radio. I found that a simpler kit NodeMCU with the same chip cost only \$3.18 on eBay. It was the cheapest and most powerful microcontroller kit ever, with a 32 bit CPU at 160 MHz, 150 KB of RAM, 4 MB of flash, and many IO devices. On top of all these, it is 802.11 WiFi ready.

The manufacturer of ESP8266, Espressif Systems in Shanghai, China, released a number of Software Development Kits, and left it to the user community to provide software support for this chip. Many engineers took up the challenge and supplied a wide range of programming tools for the community. Espressif later hired a Russian engineer Ivan Grokhotkov to extend Arduino IDE to compile ESP8266 code. This new Arduino IDE extension made it possible for hobbyists like me to experiment with IoT. Large memories in ESP8266 solved the problems I had with ATmega328 on Arduino Uno and made ESP8266 a good host for Forth.

I was pleasantly surprised that ceForth was successfully ported to NodeMCU Kit in a couple of hours. There were only very few changes to fit it into ESP8266, and the Forth dictionary required no change at all. It was all because of the portability in C code. It generally took me two weeks to port Forth to a new microcontroller. Most of this time was wasted in dealing with quirks in a particular assembler, and to impose a VFM on an unyielding CPU architecture. Here C behaved like a sweet universal assembler.

With a Forth written in C on Arduino IDE, I was able to get several NodeMCU Kits to talk to one another over a WiFi network. I still did not understand the Tensilica L106 chip inside ESP8266 at all, and I did not understand WiFi and all its protocols. What I did was to look up library functions I needed to do the few things I had to do. IoT for Dummies!.

It looks that a simple Forth written in C does have values. Therefore, I updated ceForth 1.0 to ceForth 2.3, and hope that people will find some use of it. Several important improvements were implemented, like circular buffers for stacks, and a stream-lined Finite State Machine to run VFM.

It was moved to C++ under Microsoft Visual Studio Community 2017, so that one can compile and test it on a modern Windows PC. espForth for ESP8266 was showcased in SVFIG booth in 2017 Bay Area Maker Faire.

Recently, ESP8266 was upgraded to ESP32, with 3 CPU cores and much bigger RAM memory. Ron Golding in SVFIG decided to use it in his AIR (AI Robot), and I ported espForth to it, and designated it is esp32forth. It was demonstrated in 2019 Bay Area Maker Faire.

Just when I was preparing for the Faire, my wife got a stroke, and I sent her to emergency room. After the Faire she was sent to California Pacific Regional Rehabilitation Center. I went along and set up camp along her hospital bed, forgetting to bring my computer. I proved to her that I could survive without a computer. She had my undivided attention she rightfully deserved. I had lots of time to think about my Forth in C.

I pondered on my beautifully crafted Forth finite state machine:

```
{primitives[cData[P++]]() ; }
```

It reads consecutive byte code and executes them in sequence. If I could read consecutive bytes from a data array, I certainly could write consecutive bytes back into a data array. C does not have built-in variable length arrays, but it does not prevent me from writing variable length records into a big data array.

Two weeks later, my daughter brought my computer to the hospital, and I started to try writing my own data records. I first tried it on Python, which allowed me to write things into a big array and read them back. First I had `writeByte(c)` to append a byte to an array, and `writeInteger(n)` to append an integer. From them I define macro functions `CODE()` to assemble primitive Forth words, `COLON()` to assemble colon words, and `LABEL()` to append token lists to colon words. They work like a macro assembler. Since I had nothing else

to do, I built the entire dictionary of esp32forth\_54, and compared it byte-for-byte against the header file rom\_54.h produced by the Forth metacompiler in esp32forth\_54.

After my wife was discharged from the rehab center, I came back to my NodeMCU ESP32S Kit, added the macro assembler to esp32forth, and I got rid of the rom\_54.h header file. In this new esp32forth\_61 system, everything is in one esp32forth\_61.ino file. A single C file contained all the information necessary to bring up a complete Forth system on an ESP32 Kit. However, using labels in token lists to mark target addresses for branching and looping was not satisfactory, because forward references had to be resolved manually. The macro assembler was extended so that control structures could be built in a single pass, with all forward references resolved automatically. This was esp32forth\_62.

Subsequently, ceForth\_23 was upgraded to ceForth\_33 with the new macro assembler. It is very nice that I only have to distribute one file for people to try out a Forth system. It also saves me the trouble of documenting the F# metacompiler, and explaining it to people who are not familiar with Forth. It is silly to explain Forth in Forth. To explain Forth, you have to use some other languages like C, or assembly. This was the intent of the original eForth Model.

## Chapter 2. Running ceForth

A couple of years ago, I was asked the availability of my earlier books and Forth implementations. Paper copies were mostly gone. Electronic copies I saved on my computer seemed outdated. They all cried out loud asking for new lives, with new formats on newer computers.

My 86eForth 1.0 was the worst. It was compiled by MASM on a PC-DOS computer in 1990. MASM was long discontinued and I had to find better ways to resurrect it. Then I learnt that MASM was still available, but hidden behind C++ in Visual Studio.

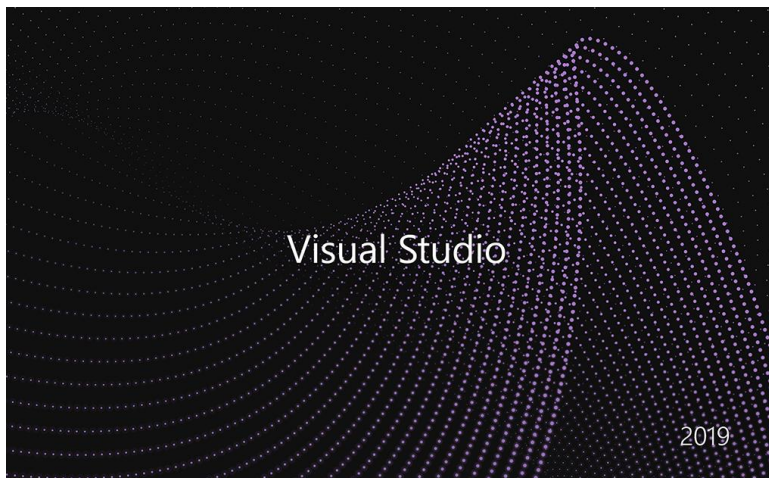
ceForth 1.0 and 1.1 were developed with gcc on cygwin. Cygwin was a crippled Linux running on PC, but it was a foreign system to Windows. I had totally forgotten how to compile and run it. Time to move on to Visual Studio.

### Install Visual Studio 2019 Community

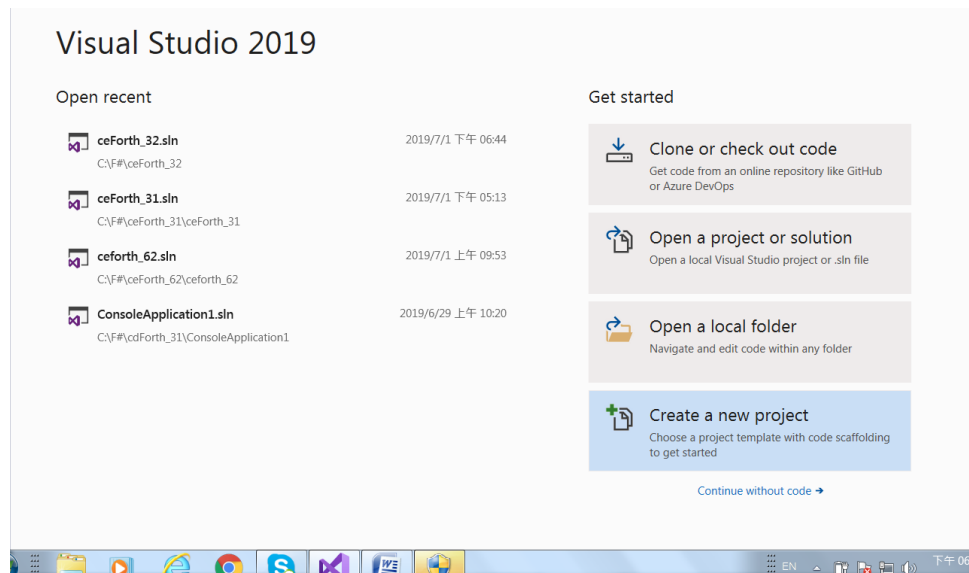
ceForth 1.0 was upgraded to ceForth\_23 on Visual Studio 2017 Community. Then it is upgraded to ceForth\_33 on Visual Studio 2019 Community.

ceForth\_33.cpp is a Visual Studio C++ Windows Console Application. It is a streamlined C program to be compiled by Visual Studio C++, and then run under Windows. To run ceForth, you have to first install Visual Studio IDE. Then you can copy ceForth\_33.cpp to it and get it running.

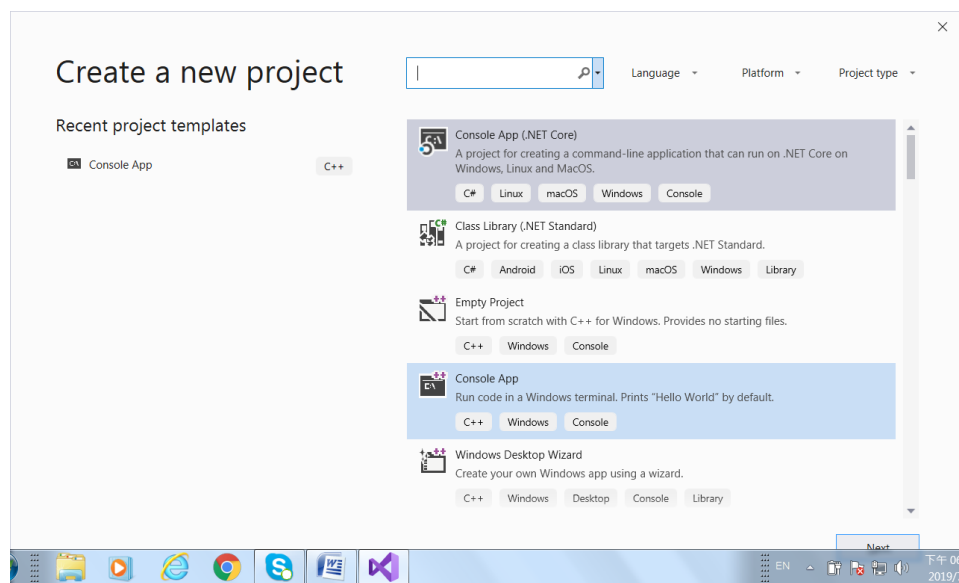
Download Visual Studio 2019 Community from [www.microsoft.com](http://www.microsoft.com) and install it on your PC. Open Visual Studio, and you will see its logo page:



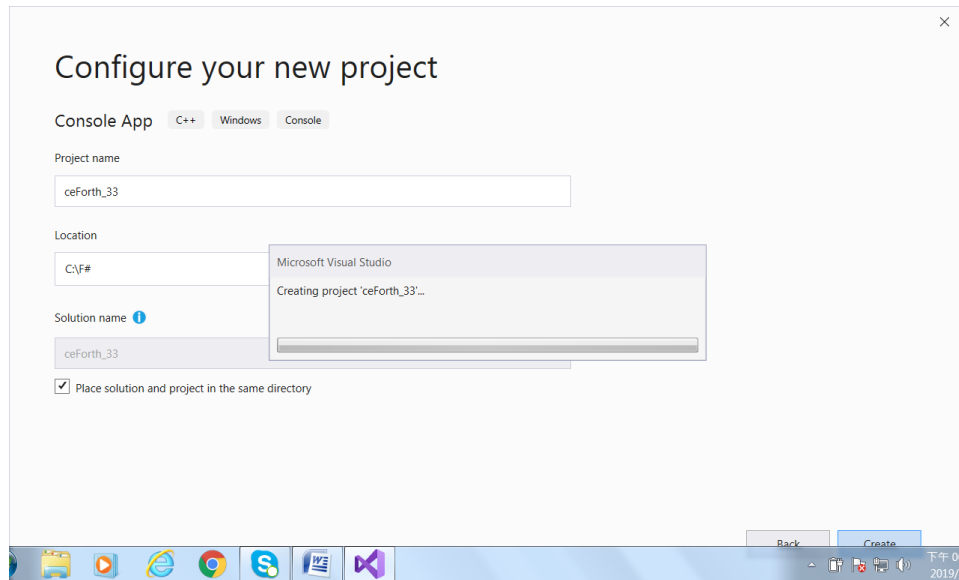
After a while, you will see its start page:



Click Create a new project



In the Create a new project Panel, select Console App. Then you will see the Configure your project page:



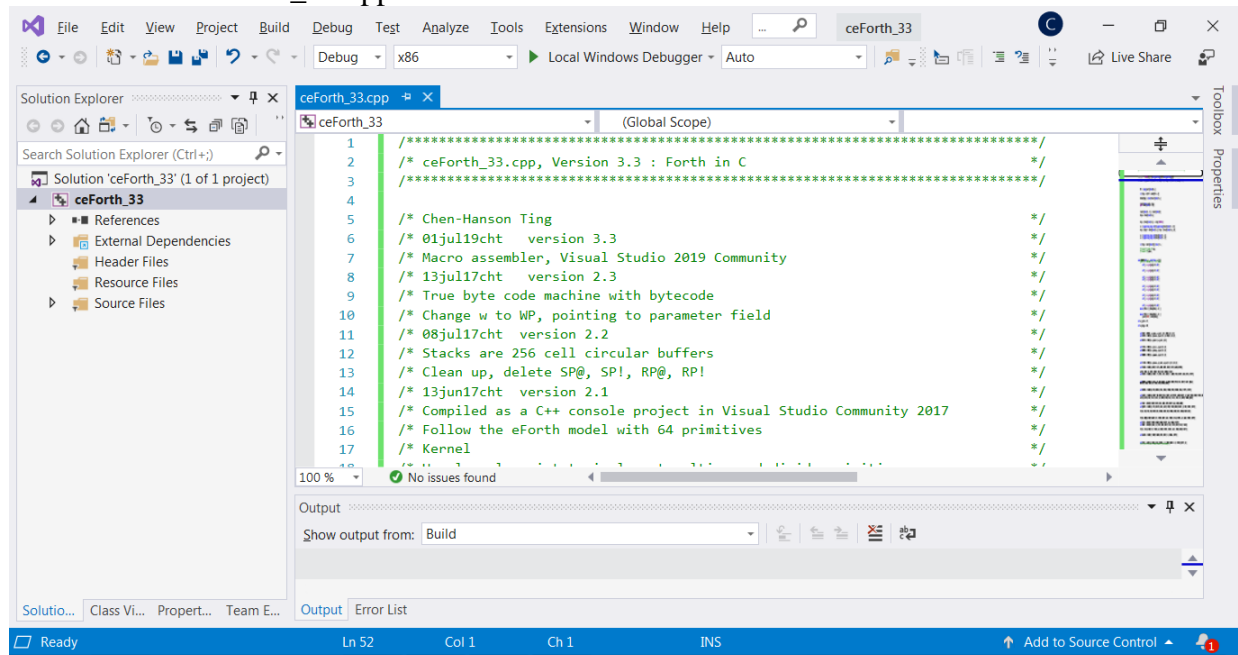
In the `Project name` box, enter `ceForth_33` for your project. In the `Location` box, select a file folder or browse to a folder you like to store your project.

Click `Create` button at the lower right corner to create the new `ceForth_33` project.

Visual Studio created a new project for you, and gives you a template file `ceForth_33.cpp`. Copy the contents of `ceForth_33.cpp` supplied in `ceForth_33.zip` file, and paste them into `ceForth_33.cpp` Edit Panel.

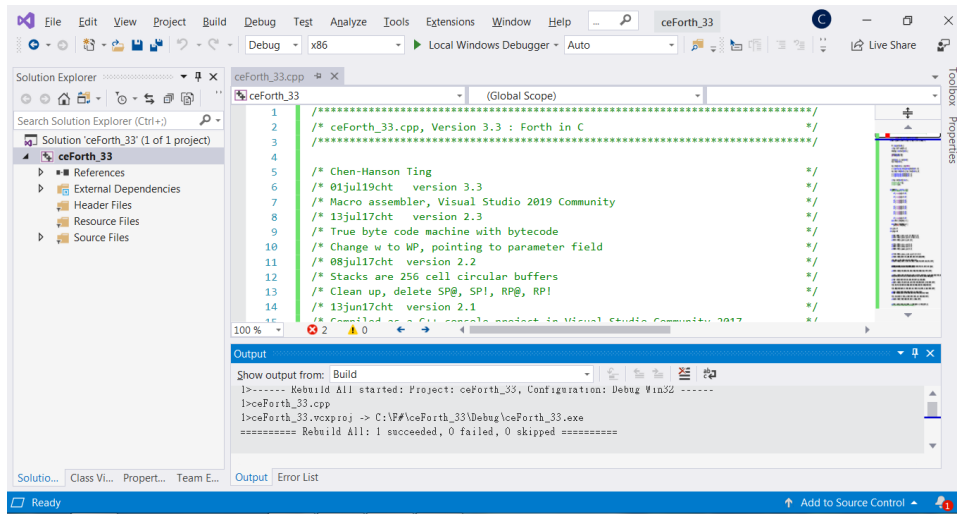
## Compile ceForth

Now we have `ceForth_33.cpp` in the Edit Panel:



Click Build>Rebuild Solution, and Visual Studio goes to work. After a while, in the Output Panel, it will report a few lines of progress, and end with this message:

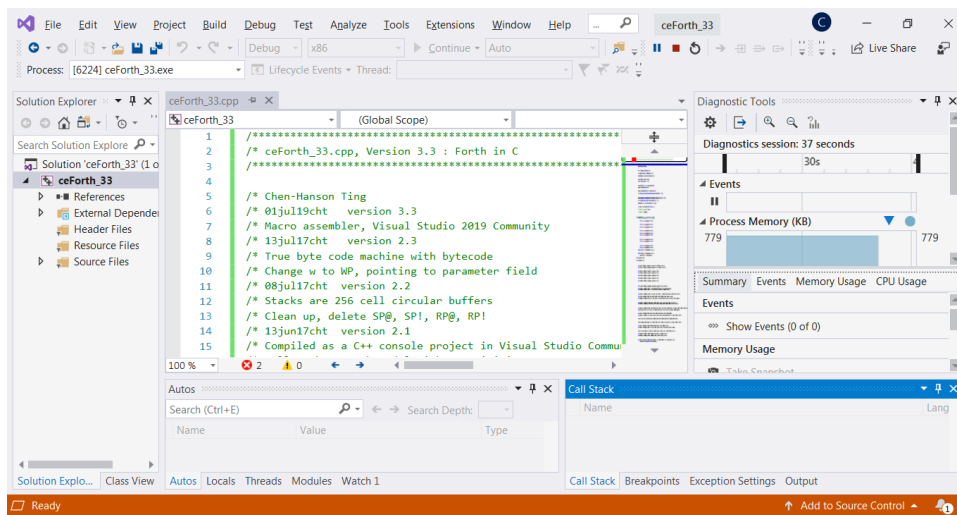
===== Rebuild All, 1 succeeded, 0 failed, 0 skipped =====



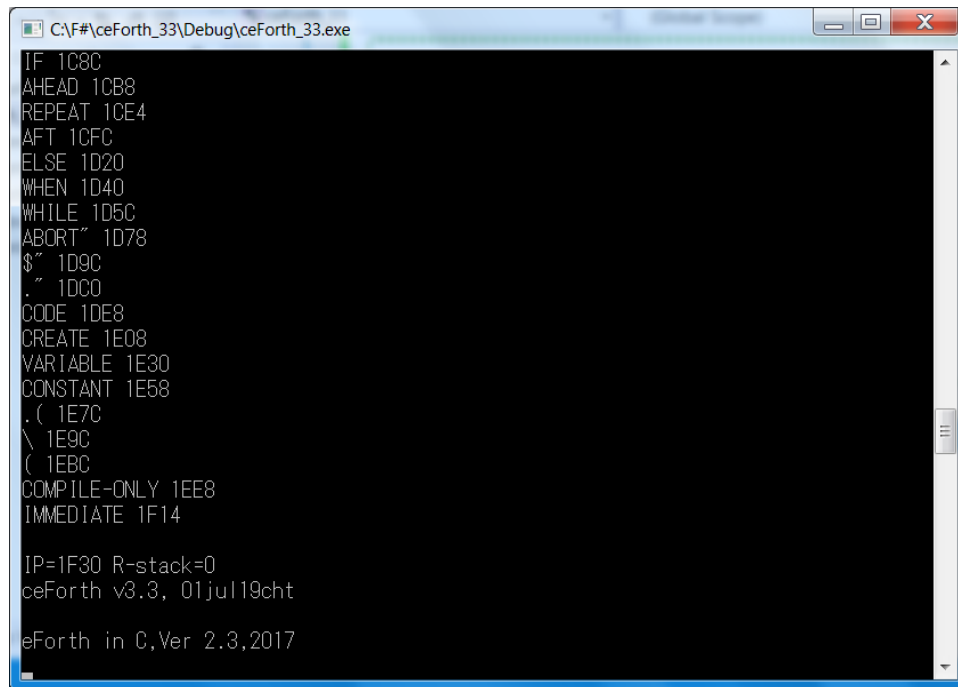
All is well. Ready to test.

## Test ceForth

Click Debug>Start without debugging. Wait some more. Finally, you will see the Debug window:



On top of it, you have the Console Window:

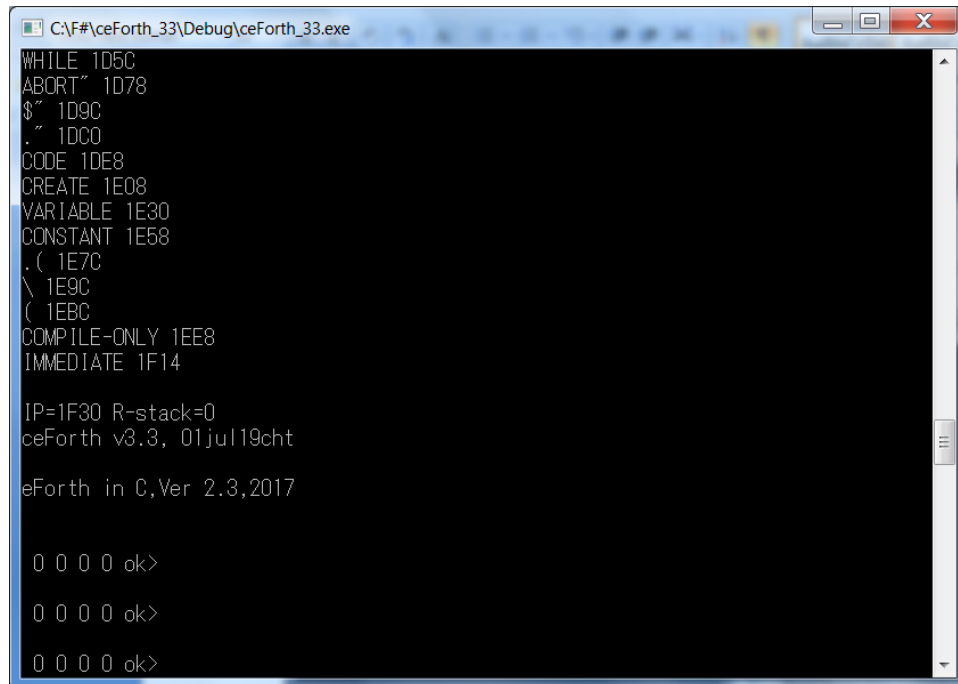


```
C:\F#\ceForth_33\Debug\ceForth_33.exe
IF 1C8C
AHEAD 1CB8
REPEAT 1CE4
AFT 1CFC
ELSE 1D20
WHEN 1D40
WHILE 1D5C
ABORT" 1D78
$" 1D9C
." 1DC0
CODE 1DE8
CREATE 1E08
VARIABLE 1E30
CONSTANT 1E58
.( 1E7C
\ 1E9C
( 1EBC
COMPILE-ONLY 1EE8
IMMEDIATE 1F14

IP=1F30 R-stack=0
ceForth v3.3, 01jul19cht
eForth in C,Ver 2.3,2017
```

Success! ceForth is running.

Press Enter key a number of times. ceForth displays an empty stack with 'ok>' prompts.



```
C:\F#\ceForth_33\Debug\ceForth_33.exe
WHILE 1D5C
ABORT" 1D78
$" 1D9C
." 1DC0
CODE 1DE8
CREATE 1E08
VARIABLE 1E30
CONSTANT 1E58
.( 1E7C
\ 1E9C
( 1EBC
COMPILE-ONLY 1EE8
IMMEDIATE 1F14

IP=1F30 R-stack=0
ceForth v3.3, 01jul19cht
eForth in C,Ver 2.3,2017

0 0 0 0 ok>
0 0 0 0 ok>
0 0 0 0 ok>
```

Type WORDS, and you get a screen of word names representing a complete ceForth system:

```
C:\F#\ceForth_33\Debug\ceForth_33.exe
0 0 0 0 ok>
0 0 0 0 ok>
0 0 0 0 ok>words
words
IMMEDIATE COMPILE-ONLY ( \ .( CONSTANT VARIABLE CREATE CODE ." $"
ABORT" WHILE WHEN ELSE AFT REPEAT AHEAD IF AGAIN UNTIL NEXT
BEGIN FOR THEN COLD FORGET WORDS .ID >NAME DUMP dm+ ;
: ] OVERT $COMPILE COMPILE [COMPILE] ' $.n ?UNIQUE $," ALLOT
LITERAL , QUIT EVAL .OK [ $INTERPRET ERROR abort" ABORT QUERY
EXPECT ACCEPT kTAP TAP ^H NAME? find SAME? NAME> WORD TOKEN
PARSE PACK$ (parse) ? . U. U.R .R ."| $"| do$
CR TYPE SPACES CHARS SPACE NUMBER? DIGIT? >upper wupper DECIMAL HEX
str #> SIGN #S # HOLD <# EXTRACT DIGIT FILL MOVE
CMOVE @EXECUTE TIB PAD HERE ALIGNED >CHAR WITHIN EMIT KEY ?KEY
DOVAR 1- 1+ CELL/ CELLS CELL- CELL+ CELL BL MIN MAX
COUNT 2@ 2! +! PICK */ */MOD M* * UM* /
MOD /MOD M/MOD UM/MOD < U< = ABS - DNEGATE NEGATE
NOT + 2DUP 2DROP ROT ?DUP NEXT UM+ XOR OR AND
O< OVER SWAP DUP DROP >R R@ R> C@ C! @
! BRANCH OBRANCH DONEXT EXECUTE EXIT DOLIST DOLIT DOCON TX! ?RX
BYE NOP tmp 'ABORT 'EVAL LAST CP CONTEXT BASE 'TIB #TIB
>IN SPAN HLD
0 0 0 0 ok>
```

Now, enter this universal greeting word:  
: TEST CR ." HELLO, WORLD!" ;  
and then type TEST:

```
C:\F#\ceForth_33\Debug\ceForth_33.exe
words
IMMEDIATE COMPILE-ONLY ( \ .( CONSTANT VARIABLE CREATE CODE ." $"
ABORT" WHILE WHEN ELSE AFT REPEAT AHEAD IF AGAIN UNTIL NEXT
BEGIN FOR THEN COLD FORGET WORDS .ID >NAME DUMP dm+ ;
: ] OVERT $COMPILE COMPILE [COMPILE] ' $.n ?UNIQUE $," ALLOT
LITERAL , QUIT EVAL .OK [ $INTERPRET ERROR abort" ABORT QUERY
EXPECT ACCEPT kTAP TAP ^H NAME? find SAME? NAME> WORD TOKEN
PARSE PACK$ (parse) ? . U. U.R .R ."| $"| do$
CR TYPE SPACES CHARS SPACE NUMBER? DIGIT? >upper wupper DECIMAL HEX
str #> SIGN #S # HOLD <# EXTRACT DIGIT FILL MOVE
CMOVE @EXECUTE TIB PAD HERE ALIGNED >CHAR WITHIN EMIT KEY ?KEY
DOVAR 1- 1+ CELL/ CELLS CELL- CELL+ CELL BL MIN MAX
COUNT 2@ 2! +! PICK */ */MOD M* * UM* /
MOD /MOD M/MOD UM/MOD < U< = ABS - DNEGATE NEGATE
NOT + 2DUP 2DROP ROT ?DUP NEXT UM+ XOR OR AND
O< OVER SWAP DUP DROP >R R@ R> C@ C! @
! BRANCH OBRANCH DONEXT EXECUTE EXIT DOLIST DOLIT DOCON TX! ?RX
BYE NOP tmp 'ABORT 'EVAL LAST CP CONTEXT BASE 'TIB #TIB
>IN SPAN HLD
0 0 0 0 ok>: TEST CR ." HELLO WORLD!" ;
: TEST CR ." HELLO WORLD!" ;
0 0 0 0 ok>TEST
TEST
HELLO WORLD!
0 0 0 0 ok>
```

ceForth is now fully functional.

## Chapter 3. ceForth Virtual Forth Engine

### ceForth\_33.cpp

The file ceForth\_33.cpp is a C++ program which can be compiled by Visual Studio IDE, and run as a Windows Console Application. This file serves perfectly as a specification of a Virtual Forth Engine, in terms of C functions.

Before diving directly into ceForth, I would like to give you an overview of a Forth system with a Virtual Forth Machine (VFM) under it, so you can better understand how the whole thing is implemented.

- A VFM executes a set of pseudo instructions, in the form byte code.
- All Forth words or commands are stored in a large data array, called a dictionary.
- Each word has a record. All records are linked in the dictionary.
- Each word record contains 4 fields, a link field, a name field, a code field, and a parameter field. The link field and name field allow the dictionary to be searched for a word from its ASCII name. The code field contains executable byte code. The parameter field contains optional code and data needed by the word.
- There are two types of words: primitive words containing executable byte code, and colon words containing token lists. A token is a code field address pointing to code field of a word.
- A sequencer executes byte code sequences stored in code fields of primitive words.
- An inner interpreter terminates a primitive word and executes the next token in a token list.
- An address interpreter executes nested token lists.
- A return stack is required to process nested token lists
- A data stack is required to pass parameters among words
- A text interpreter processes word lists entered from a terminal.
- A compiler turns word lists into new Forth words.

The text interpreter interprets or executes a list of Forth words, separated by spaces:

```
<list of words>
```

It also functions like a compiler creating new words to replace lists of words:

```
: <name> <list of words> ;
```

These new words are called colon words, because compilation starts by colon ':' and ends with semicolon ';'. Colon ':' and semicolon ';' are also Forth words.

All computable problems can be solved by repeatedly creating new words to replace lists of existing words. It is very similar to natural languages. New words are created to replace lists of existing words. Thoughts and ideas are thus abstracted to a higher level. Real intelligence is best represented by deeply nested lists. Forth is real intelligence, in contrast to artificial intelligence. It is also the simplest and most efficient way to explore solution spaces far and wide, and to arrive at optimized solutions for any computable problem.

Now let's read the source code in `ceForth_33.cpp`, to see how this VFM is actually implemented.

## Preamble

In the beginning of `ceForth_33.cpp`, I put in several comment lines to document the progressing of `ceForth` implementation. After the comments, there are several include instructions to pull in header files, and several macros to facilitate compilation the rest of C code.

```
//Preamble

#include <stdlib.h>
#include <stdio.h>
#include <tchar.h>
#include <stdarg.h>
#include <string.h>

# define      FALSE  0
# define      TRUE   -1
# define      LOGICAL ? TRUE : FALSE
# define      LOWER(x,y) ((unsigned long)(x)<(unsigned long)(y))
# define      pop    top = stack[(char) S--]
# define      push   stack[(char) ++S] = top; top =
# define      popR   rack[(unsigned char)R--]
# define      pushR  rack[(unsigned char)++R]
```

`stdlib.h`, `<tchar.h>` and `<stdio.h>` are standard library header files needed by `ceForth`. `<stdarg.h>` is needed for macro functions having variable parameter lists. `<string.h>` file is needed for `strlen()` to determine lengths of strings.

Default value of a `FALSE` flag is 0, and of `TRUE`, -1. `ceForth` considers any non-zero number as a `TRUE` flag. Nevertheless, all `ceForth` words which generate flags would return a -1 for `TRUE`. `LOGICAL` is a macro enforcing the above policy for logic words to return the correct `TRUE` and `FALSE` flags.

`LOWER(x,y)` returns a `TRUE` flag if `x<y`.

`pop` is a macros which stream-lines the often used operations to pop the data stack to a register or a memory location. As the top element of the data tack is cached in register `top`, popping is more complicated, and `pop` macro helps to clarify my intention.

Similarly, `push` is a macro to push a register or contents in a memory location on the data stack. Actually, the contents in `top` register must be pushed on the data stack, and the source data is copied into `top` register.

`pushR` is a macro to push a register or contents in a memory location on the return stack. `popR` does the reverse. The return stack is drafted to help compiling control structures in the token lists of colon words at compile time. At run time, the return stack hosts nested return addresses so tokens can call nested tokens.

## Registers and Arrays

`ceForth` uses a large data array to hold its dictionary of words. There are lots of variables and buffers declared in this array. Besides this data array, the VFM needs many registers and arrays to hold data to support all its operations. Here is the list of these registers and arrays:

```
long rack[256] = { 0 };
long stack[256] = { 0 };
long long int d, n, m;
unsigned char R = 0;
unsigned char S = 0;
long top = 0;
long P, IP, WP, thread, len;
unsigned char c;

long data[16000] = {};
unsigned char* cData = (unsigned char*)data;
```

The following table explains the functions of these registers and arrays:

Register/Array	Functions
P	Program counter, pointing to pseudo instructions in <code>data[]</code> .
IP	Instruction pointer for address interpreter
WP	Working register, usually pointing to a parameter field
<code>rack[]</code>	Return stack, a 256 cell circular buffer
<code>stack[]</code>	Data stack, a 256 cell circular buffer
R	One byte return stack pointer
S	One byte data stack pointer
top	Cached top element of the data stack
c	A 8 bit scratch register
<code>Data[]</code>	A huge data array with 16000 integers
<code>cData[]</code>	Alias of <code>data[]</code> array to access bytes

In `ceForth_33`, I allocated 1KB memory for each stack, `stack[256]` and `rack[256]`, and used a byte pointer for each stack, S and R. The stacks are 256 cell circular buffers, and will never underflow or overflow. However, the C compiler needs to be reminder constantly that the stack pointers have 8-bit values and must not be leaked to integer. R and S pointers must always be prefixed with `(unsigned char)` specification. Thus the stacks will never overshoot their boundaries. No stack overflows or stack underflows

ceForth\_33 interpreter displays top 4 elements on data stack. Always seeing these 4 elements, you do not need utility to dump or examine data stack. With circular buffers for stacks, I saved 4 byte code, about 10 stack managing words, and a ton of worrying.

All Forth words are stored in `data[16000]` array, as a linked list, and is generally called a dictionary. Each record in this list contains 4 fields: a 32 bit link field, a variable length name field, a 32 bit code field, and a variable length parameter field. In a primitive word, the parameter field may contain additional byte code. In high level colon words, the code field has a byte code 6, for nesting the token list in the subsequent parameter field. Tokens are code field addresses, pointing to other Forth words.

The dictionary in `data[]` array is filled in by a macro assembler. It has a byte array alias `cData[]`, because the macro assembler has to fill in both variable length byte fields and variable length integer fields. `P` is a byte code program counter, used to index byte array `cData[]`. `IP` is an instruction pointer or integer pointer, used to index integer array `data[]`.

Later, I will go through the macro assembler, and explain how this dictionary is constructed.

### **Virtual Forth Machine (VFM)**

Then come all the VFM pseudo instructions, coded as C functions. Each pseudo instruction will be assigned a byte code. A byte code sequencer or a Finite State Machine is designed to execute byte code placed in code fields of primitive words in the dictionary. Byte code are machine instructions of VFM, just like machine instructions of a real computer.

```
// Virtual Forth Machine
```

`bye ( -- )` Return control from Forth back to Windows. Close the Windows Console opened for Forth.

```
void bye(void)
{
    exit(0);
}
```

`qrx ( -- c T|F )` Return a character and a true flag if the character has been received. If no character was received, return a false flag

```
void qrx(void)
{
    push(long) getchar();
    if (top != 0) push TRUE;
}
```

`txsto ( c -- )` Send a character to the serial terminal.

```
void txsto(void)
{
    putchar((char)top);
}
```

```

        pop;
    }

```

`next()` is the inner interpreter of the Virtual Forth Machine. Execute the next token in a token list. It reads the next token, which is a code field address, and deposits it into Program Counter P. The sequencer then jumps to this address, and executes the byte code in it. It also deposits P+4 into the Work Register WP, pointing to the parameter field of this word. WP helps retrieving the token list of a colon word, or data stored in parameter field.

```

void next(void)
{
    P = data[IP >> 2];
    WP = P + 4;
    IP += 4;
}

```

`dovar( -- a )` Return the parameter field address saved in WP register.

```

void dovar(void)
{
    push WP;
}

```

`docon ( -- n )` Return integer stores in the parameter field of a constant word.

```

void docon(void)
{
    push data[WP >> 2];
}

```

`dolit ( -- w )` Push the next token onto the data stack as an integer literal. It allows numbers to be compiled as in-line literals, supplying data to data stack at run time.

```

void dolit(void)
{
    push data[IP >> 2];
    IP += 4;
    next();
}

```

`dolist ( -- )` Push the current Instruction Pointer (IP) the return stack and then pops the Program Counter P into IP from the data stack. When `next ( )` is executed, the tokens in the list are executed consecutively.

```

void dolist(void)
{
    rack[(char)++R] = IP;
    IP = WP;
    next();
}

```

**exit** ( -- ) Terminate all token lists in colon words. EXIT pops the execution address saved on the return stack back into the IP register and thus restores the condition before the colon word was entered. Execution of the calling token list will continue.

```
void exit(void)
{
    IP = (long)rack[(char)R--];
    next();
}
```

**execu** ( a -- ) Take the execution address from data stack and executes that token. This powerful word allows you to execute any token which is not a part of a token list.

```
void execu(void)
{
    P = top;
    WP = P + 4;
    pop;
}
```

**donext** ( -- ) Terminate a FOR-NEXT loop. The loop count was pushed on return stack, and is decremented by donext. If the count is not negative, jump to the address following donext; otherwise, pop the count off return stack and exit the loop.

```
void donext(void)
{
    if (rack[(char)R]) {
        rack[(char)R] -= 1;
        IP = data[IP >> 2];
    }
    else {
        IP += 4;
        R--;
    }
    next();
}
```

**qbran** ( f -- ) Test top as a flag on data stack. If it is zero, branch to the address following qbran; otherwise, continue execute the token list following the address.

```
void qbran(void)
{
    if (top == 0) IP = data[IP >> 2];
    else IP += 4;
    pop;
    next();
}
```

**bran** ( -- ) Branch to the address following bran.

```
void bran(void)
```

```

{
    IP = data[IP >> 2];
    next();
}

```

**store ( n a -- )** Store integer n into memory location a.

```

void store(void)
{
    data[top >> 2] = stack[(char)S--];
    pop;
}

```

**at ( a -- n )** Replace memory address a with its integer contents fetched from this location.

```

void at(void)
{
    top = data[top >> 2];
}

```

**cstor ( c b -- )** Store a byte value c into memory location b.

```

void cstor(void)
{
    cData[top] = (char)stack[(char)S--];
    pop;
}

```

**cat ( b -- n )** Replace byte memory address b with its byte contents fetched from this location.

```

void cat(void)
{
    top = (long)cData[top];
}

```

**rfrom ( n -- )** Pop a number off the data stack and pushes it on the return stack.

```

void rfrom(void)
{
    push rack[(char)R--];
}

```

**rat ( -- n )** Copy a number off the return stack and pushes it on the return stack.

```

void rat(void)
{
    push rack[(char)R];
}

```

**tor ( -- n )** Pop a number off the return stack and pushes it on the data stack.

```

void tor(void)
{
    rack[(char)++R] = top;
}

```

```

        pop;
    }

```

**drop ( w -- ) Discard top stack item.**

```

void drop(void)
{
    pop;
}

```

**dup ( w -- w w ) Duplicate the top stack item.**

```

void dup(void)
{
    stack[(char) ++S] = top;
}

```

**swap ( w1 w2 -- w2 w1 ) Exchange top two stack items.**

```

void swap(void)
{
    WP = top;
    top = stack[(char)S];
    stack[(char)S] = WP;
}

```

**over ( w1 w2 -- w1 w2 w1 ) Copy second stack item to top.**

```

void over(void)
{
    push stack[(char)S - 1];
}

```

**zless ( n - f ) Examine the top item on the data stack for its negativeness. If it is negative, return a -1 for true. If it is 0 or positive, return a 0 for false.**

```

void zless(void)
{
    top = (top < 0) LOGICAL;
}

```

**andd ( w w -- w ) Bitwise AND.**

```

void andd(void)
{
    top &= stack[(char)S--];
}

```

**orr ( w w -- w ) Bitwise inclusive OR.**

```

void orr(void)
{
    top |= stack[(char)S--];
}

```

**xorr ( w w -- w ) Bitwise exclusive OR.**

```
void xorr(void)
{
    top ^= stack[(char)S--];
}
```

**uplus ( w w -- w cy ) Add two numbers, return the sum and carry flag.**

```
void uplus(void)
{
    stack[(char)S] += top;
    top = LOWER(stack[(char)S], top);
}
```

**nop ( -- ) No operation.**

```
void nop(void)
{
    next();
}
```

**qdup ( w -- w w | 0 ) Dup top of stack if it is not zero.**

```
void qdup(void)
{
    if (top) stack[(char) ++S] = top;
}
```

**rot ( w1 w2 w3 -- w2 w3 w1 ) Rot 3rd item to top.**

```
void rot(void)
{
    WP = stack[(char)S - 1];
    stack[(char)S - 1] = stack[(char)S];
    stack[(char)S] = top;
    top = WP;
}
```

**ddrop ( w w -- ) Discard two items on stack.**

```
void ddrop(void)
{
    drop(); drop();
}
```

**ddup ( w1 w2 -- w1 w2 w1 w2 ) Duplicate top two items.**

```
void ddup(void)
{
    over(); over();
}
```

**plus ( w w -- sum )** Add top two items.

```
void plus(void)
{
    top += stack[(char)S--];
}
```

**inver ( w -- w )** One's complement of top.

```
void inver(void)
{
    top = -top - 1;
}
```

**negat ( n -- -n )** Two's complement of top.

```
void negat(void)
{
    top = 0 - top;
}
```

**dnega ( d -- -d )** Two's complement of top double.

```
void dnega(void)
{
    inver();
    tor();
    inver();
    push 1;
    uplus();
    rfrom();
    plus();
}
```

**subb ( n1 n2 -- n1-n2 )** Subtraction.

```
void subb(void)
{
    top = stack[(char)S--] - top;
}
```

**abss ( n -- n )** Return the absolute value of n.

```
void abss(void)
{
    if (top < 0)
        top = -top;
}
```

**great ( n1 n2 -- t )** Signed compare of top two items. Return true if n1>n2.

```
void great(void)
{
    top = (stack[(char)S--] > top) LOGICAL;
}
```

```
}
```

**less ( n1 n2 -- t ) Signed compare of top two items. Return true if  $n1 < n2$ .**

```
void less(void)
{
    top = (stack[(char)S--] < top) LOGICAL;
}
```

**equal ( w w -- t ) Return true if top two are equal.**

```
void equal(void)
{
    top = (stack[(char)S--] == top) LOGICAL;
}
```

**unless ( u1 u2 -- t ) Unsigned compare of top two items.**

```
void unless(void)
{
    top = LOWER(stack[(char)S], top) LOGICAL; (char)S--;
}
```

**ummod ( udl udh u -- ur uq ) Unsigned divide of a double by a single. Return mod and quotient.**

```
void ummod(void)
{
    d = (long long int)((unsigned long)top);
    m = (long long int)((unsigned long)stack[(char)S]);
    n = (long long int)((unsigned long)stack[(char)S - 1]);
    n += m << 32;
    pop;
    top = (unsigned long)(n / d);
    stack[(char)S] = (unsigned long)(n % d);
}
```

**msmod ( d n -- r q ) Signed floored divide of double by single. Return mod and quotient.**

```
void msmod(void)
{
    d = (signed long long int)((signed long)top);
    m = (signed long long int)((signed long)stack[(char)S]);
    n = (signed long long int)((signed long)stack[(char)S - 1]);
    n += m << 32;
    pop;
    top = (signed long)(n / d);
    stack[(char)S] = (signed long)(n % d);
}
```

**slmod ( n1 n2 -- r q ) Signed divide. Return mod and quotient.**

```
void slmod(void)
{
    if (top != 0) {
        WP = stack[(char)S] / top;
        stack[(char)S] %= top;
        top = WP;
    }
}
```

**mod ( n n -- r ) Signed divide. Return mod only.**

```
void mod(void)
{
    top = (top) ? stack[(char)S--] % top : stack[(char)S--];
}
```

**slash ( n n -- q ) Signed divide. Return quotient only.**

```
void slash(void)
{
    top = (top) ? stack[(char)S--] / top : (stack[(char)S--], 0);
}
```

**umsta ( u1 u2 -- ud ) Unsigned multiply. Return double product.**

```
void umsta(void)
{
    d = (unsigned long long int)top;
    m = (unsigned long long int)stack[(char)S];
    m *= d;
    top = (unsigned long)(m >> 32);
    stack[(char)S] = (unsigned long)m;
}
```

**star ( n n -- n ) Signed multiply. Return single product.**

```
void star(void)
{
    top *= stack[(char)S--];
}
```

**mstar ( n1 n2 -- d ) Signed multiply. Return double product.**

```
void mstar(void)
{

```

```

    d = (signed long long int)top;
    m = (signed long long int)stack[(char)S];
    m *= d;
    top = (signed long)(m >> 32);
    stack[(char)S] = (signed long)m;
}

```

**ssmod ( n1 n2 n3 -- r q )** Multiply n1 and n2, then divide by n3. Return mod and quotient.

```

void ssmod(void)
{
    d = (signed long long int)top;
    m = (signed long long int)stack[(char)S];
    n = (signed long long int)stack[(char)S - 1];
    n *= m;
    pop;
    top = (signed long)(n / d);
    stack[(char)S] = (signed long)(n % d);
}

```

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

```

void stasl(void)
{
    d = (signed long long int)top;
    m = (signed long long int)stack[(char)S];
    n = (signed long long int)stack[(char)S - 1];
    n *= m;
    pop; pop;
    top = (signed long)(n / d);
}

```

**pick ( ... +n -- ... w )** Copy the nth stack item to top.

```

void pick(void)
{
    top = stack[(char)S - (char)top];
}

```

**pstor ( n a -- )** Add n to the contents at address a.

```

void pstor(void)
{
    data[top >> 2] += stack[(char)S--], pop;
}

```

**dstor ( d a -- )** Store the double integer to address a.

```

void dstor(void)
{
    data[(top >> 2) + 1] = stack[(char)S--];
    data[top >> 2] = stack[(char)S--];
    pop;
}

```

**dat ( a -- d )** Fetch double integer from address a.

```

void dat(void)
{
    push data[top >> 2];
    top = data[(top >> 2) + 1];
}

```

**count ( b -- b+1 +n )** Return count byte of a string and add 1 to byte address.

```

void count(void)
{
    stack[(char) ++S] = top + 1;
    top = cData[top];
}

```

**maxx ( n1 n2 -- n )** Return the greater of two top stack items.

```

void max(void)
{
    if (top < stack[(char)S]) pop;
    else (char)S--;
}

```

**minn ( n1 n2 -- n )** Return the smaller of top two stack items.

```

void min(void)
{
    if (top < stack[(char)S]) (char) S--;
    else pop;
}

```

## Byte Code Array

There are 64 functions defined in VFM as shown before. Each of these functions is assigned a unique byte code, which become the pseudo instructions of this VFM. In the dictionary, there are primitive words which have these byte code in their code field. The byte code may spill over into the subsequent parameter field, if a primitive word is very complicated. VFM has a byte code sequencer, which will be discussed shortly, to sequence through byte code list. The numbering of these byte code in the following primitives[] array does not follow any perceived order.

Only 64 byte code are defined. You can extend them to 256 if you wanted. You have the options to write more functions in C to extend the VFM, or to assemble more primitive words using the metacompiler I will discuss later, or to compile more colon words in Forth, which is far easier. The same function defined in different ways should behave identically. Only the execution speed may differ, inversely proportional to the efforts in programming.

```
void(*primitives[64])(void) = {
    /* case 0 */ nop,
    /* case 1 */ bye,
    /* case 2 */ qrx,
    /* case 3 */ txsto,
    /* case 4 */ docon,
    /* case 5 */ dolit,
    /* case 6 */ dolist,
    /* case 7 */ exitt,
    /* case 8 */ execu,
    /* case 9 */ donext,
    /* case 10 */ qbran,
    /* case 11 */ bran,
    /* case 12 */ store,
    /* case 13 */ at,
    /* case 14 */ cstor,
    /* case 15 */ cat,
    /* case 16 */ rpat, /* nop,
    /* case 17 */ rpsto, /* nop,
    /* case 18 */ rfrom,
    /* case 19 */ rat,
    /* case 20 */ tor,
    /* case 21 */ spat, /* nop,
    /* case 22 */ spsto, /* nop,
    /* case 23 */ drop,
    /* case 24 */ dup,
    /* case 25 */ swap,
    /* case 26 */ over,
    /* case 27 */ zless,
    /* case 28 */ andd,
    /* case 29 */ orr,
    /* case 30 */ xor,
    /* case 31 */ uplus,
    /* case 32 */ next,
    /* case 33 */ qdup,
    /* case 34 */ rot,
    /* case 35 */ ddrop,
    /* case 36 */ ddup,
    /* case 37 */ plus,
    /* case 38 */ inver,
```

```

/* case 39 */ negat,
/* case 40 */ dnegat,
/* case 41 */ subb,
/* case 42 */ abss,
/* case 43 */ equal,
/* case 44 */ uless,
/* case 45 */ less,
/* case 46 */ ummod,
/* case 47 */ msmod,
/* case 48 */ slmod,
/* case 49 */ mod,
/* case 50 */ slash,
/* case 51 */ umsta,
/* case 52 */ star,
/* case 53 */ mstar,
/* case 54 */ ssmod,
/* case 55 */ stasl,
/* case 56 */ pick,
/* case 57 */ pstor,
/* case 58 */ dstor,
/* case 59 */ dat,
/* case 60 */ count,
/* case 61 */ dovar,
/* case 62 */ max,
/* case 63 */ min,
};

```

Let us skip over all the macro assembler and the dictionary it builds, to the end of `ceForth_33.cpp` to see how the Virtual Forth Machine starts running. The following code sets up the boot-up vector:

```

IP = 0;
int RESET = LABEL(2, 6, COLD);

```

It basically tells VFM to execute `COLD` word in Forth, which starts the Forth interpreter.

To start the VFM running, registers `P`, `WP`, `IP`, `S`, `R` and `top` have to be initialized in the `main()` function required by Visual Studio. As `P` is set to 0, execution starts by executing the byte code stored in virtual memory location 0.

```

/*
 * Main Program
 */
int main(int ac, char* av[])
{
    P = 0;
    WP = 4;
    IP = 0;
    S = 0;
    R = 0;
    top = 0;
}

```

```
printf("\nceForth v2.3, 13jul17cht\n");  
while (TRUE) {  
    primitives[(unsigned char)cData[P++]]();  
}  
}
```

The `while (TRUE)` loop loops forever. Going through each loop, the finite state machine reads the next byte code pointed to by `P`. The byte code is executed by `execute(bytecode)`. The next byte code is read and executed. And so forth. It behaves just like a real computer sequencing through its machine instructions stored in memory.

## Chapter 3. Forth Dictionary

Forth is a programming language to program computers, but it is very similar to natural languages like English. Forth has a set of words, similar to words in English. Forth words therefore are called words. Forth has a very simple grammar rule: words are separated by spaces. A Forth computer processes lists of words, executing words from left to right. It is like English: you read a sentence from left to right.

Forth is like a natural language, in which new words are defined in terms of existing words. Adding new words extends the language, pushes it towards higher and higher levels of abstraction, eventually solving all computable problems. It is the simplest and most powerful form of intelligence, essentially the way how human beings think, reason, communicate, and accumulate knowledge.

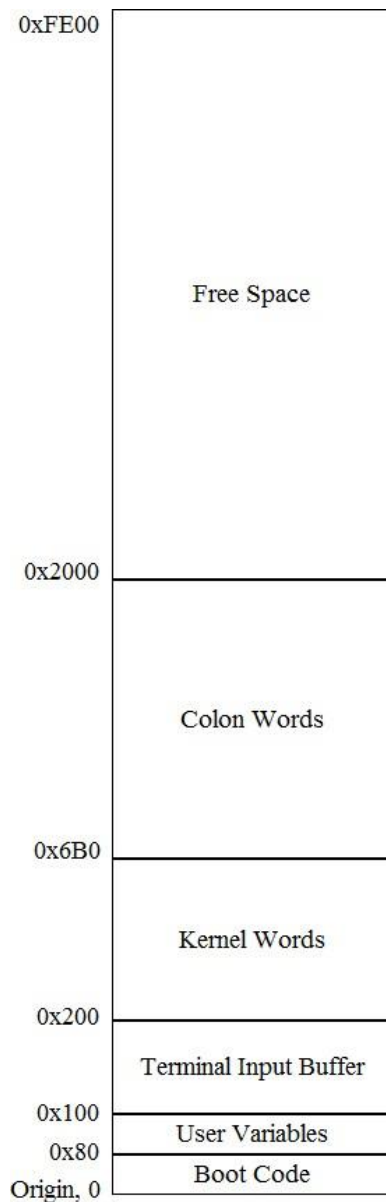
A more detailed and precise description of Forth is as follows:

- Forth has a set of words or words.
- Forth words are records stored in a computer memory area, called a dictionary.
- A Forth word has two representations: an external representation in the form of a name in ASCII characters; and an internal representation in the form of a token, which invokes executable code stored in memory.
- There are two principal types of Forth words: primitive words containing machine instructions, and colon words containing token lists.
- Forth has a text interpreter, which scans a list of words, finds tokens of words and executes the tokens in left to right order.
- Forth has a compiler, which compiles new words to replace lists of tokens.
- Tokens are often nested token lists. A return stack is thus required to nest and unnest token lists.
- Forth words pass numeric parameters implicitly on a first-in-last-out data stack or parameter stack, thus greatly simplify the language syntax.

In many Forth systems, a token is an address of executable code. However, a token can take other forms depending on implementation. In `ceForth_33`, tokens are 32-bit code field addresses for Forth words.

### Forth Dictionary

`ceForth_33` allocates a big array `data[16000]` for the Virtual Forth Engine to use. This array is used mostly to store the Forth dictionary, along with several buffers for other information. The most important areas in this data array are show in the following figure:



### Forth Dictionary

Reset vector is stored at location 0. 128 bytes are reserved for user to put his own system initialization code here. Then, 128 bytes are used to store user variables, which store pointers needed by Forth text interpreter to run. A big Terminal Input Buffer is allocated from 0x100 to 0c1FF. It used to be 80 bytes long to read a single punch card.

Area about 0x200 is the Forth dictionary. First come the primitive kernel words, and then all the colon word. This implementation has 80 primitive words and 110 colon words. The dictionary has 7984 bytes.

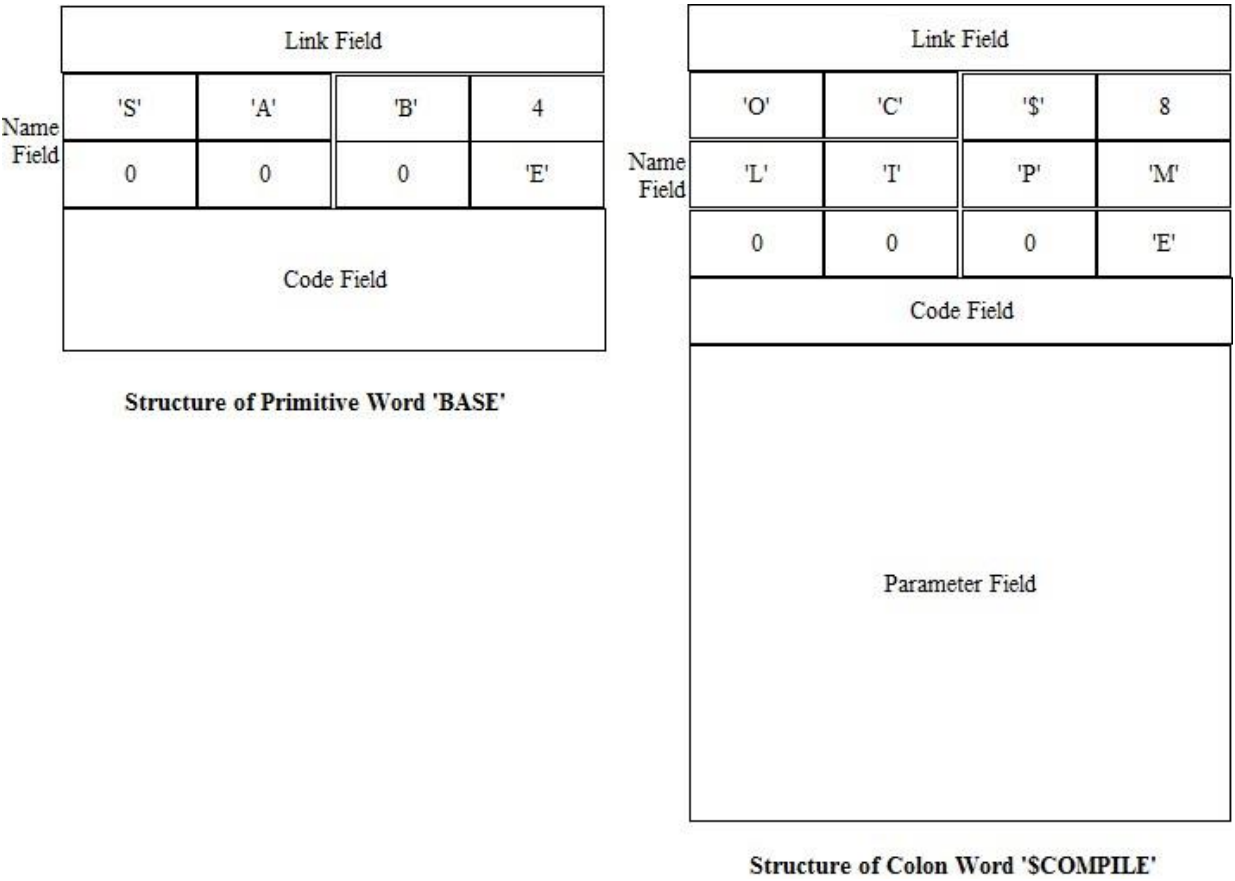
### Word Records

Predefined Forth words, both primitive words and colon words, are assembled into a linearly linked dictionary in memory. New colon words are added to the dictionary, and extends the capability of Forth system.

Each word is a record with 4 fields:

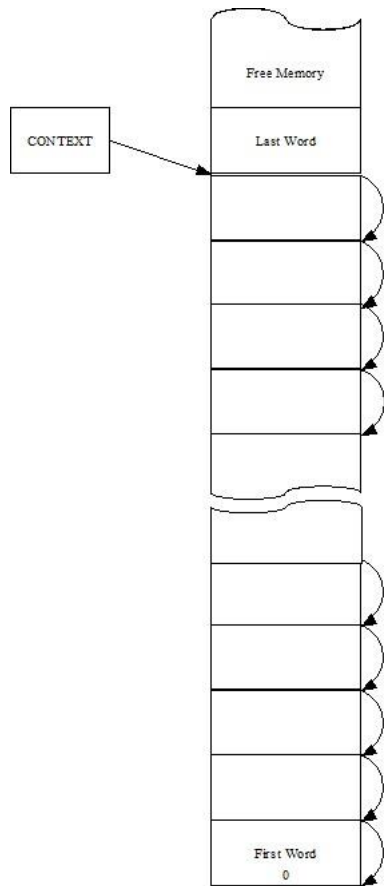
Field	Length	Function
Link	4 bytes	name field of previous word
Name	Variable	name of word and lexicon byte
Code	4 byte	executable byte code
Parameter	Variable	instructions, tokens, data

In colon words, the parameter field contains a list of tokens, In primitive words, the parameter field is an extension of code field. Following figure shows the structure of word records:



word boundary. Tokens are 32-bit integers, and therefore the parameter field is always terminated on word boundary.

All word records are linked in a uni-direction linked list, called a dictionary. The link field contains a pointer pointing to the name field of the prior word. The link list starts at the last word IMMEDIATE assembled in the dictionary. It is pointed to by a user variable `CONTEXT`. Word searching also starts here. The first word `HLD`, which terminates the list and stops dictionary searching, has a zero in its link field, indicating the end of list. The threading of records in a dictionary is shown in the following figure:



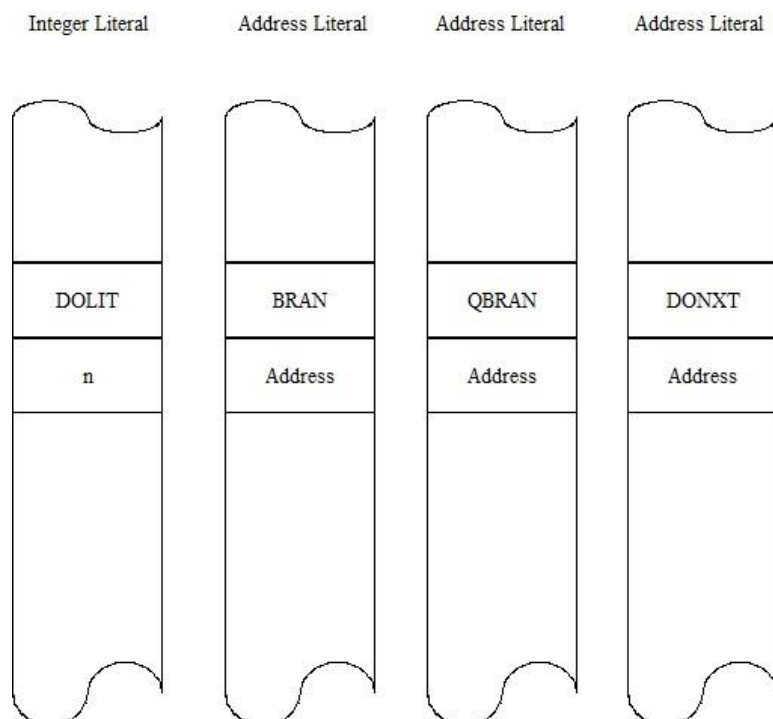
`ceForth_33` uses the 'Direct Thread Model'. Each word has a code field in its record. The code field address (`cfa`) is considered the token of this word. The code field contains executable byte code. In a primitive word, the code field contains a list of byte code, which may extend into the parameter field, terminated by a special byte code `next`, which fetches the next token from a token list and executes that token.

In a colon word, the code field contains a `DOLST` byte code, which processes the contents of the parameter field as a token list. The token list is generally terminated by a primitive word `EXIT`, which un-nests a token list started by `DOLST`. The code fields and parameter fields of these words are shown as the following figure:

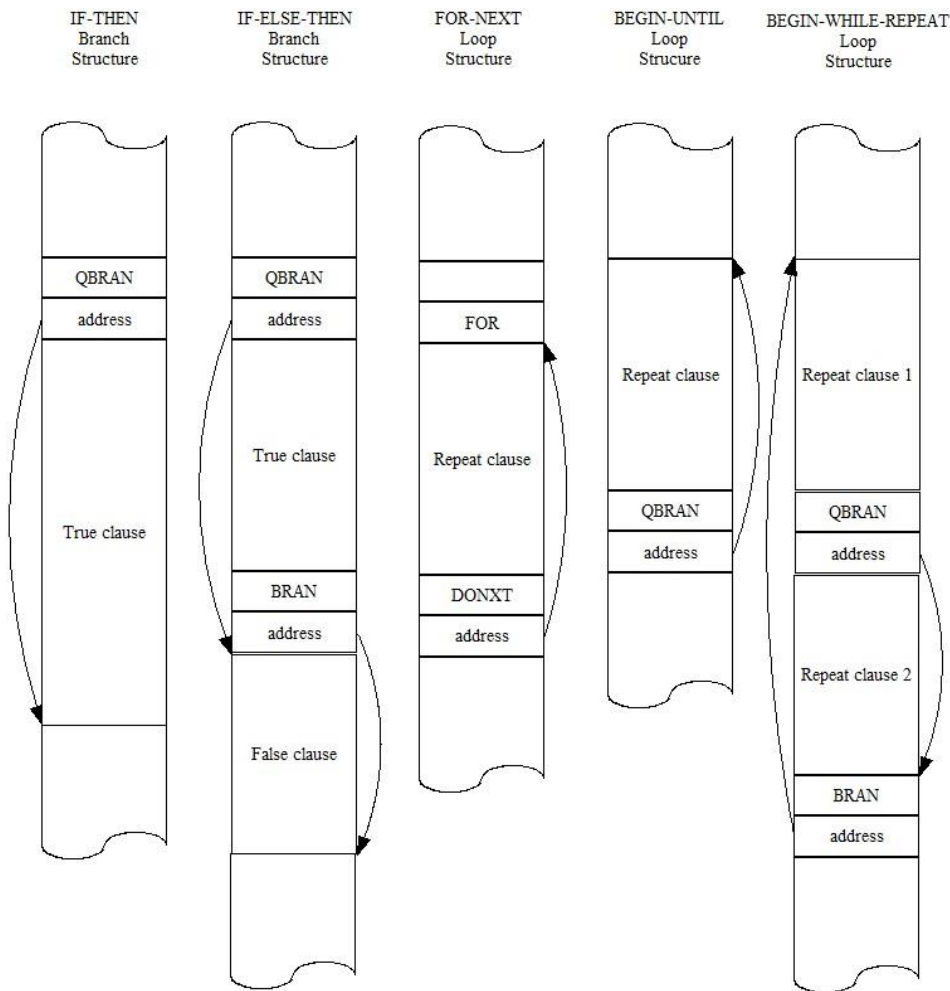


In the parameter field of a colon word, the token list generally contains tokens which are 32-bit code field addresses of other words. However, there are many other kinds of information embedded in a token list. In ceForth\_33, there are integer literals, address literals, and string literals. An integer literal is a token DOLIT followed by a 32-bit integer value. This integer will be pushed on the data stack at run time. An address literal starts with a token BRAN, QBRAN or DONXT, followed by a 32-bit address. This address will be used by BRAN, QBRAN or DONXT to branch to a new location in the token list containing these branching tokens.

The integer literal and address literals are shown in the following figure:



The address literals are used to build control structures in token lists. The following figure shows how they are used in structures like IF-ELSE-THEN and BEGIN-WHILE-REPEAT.



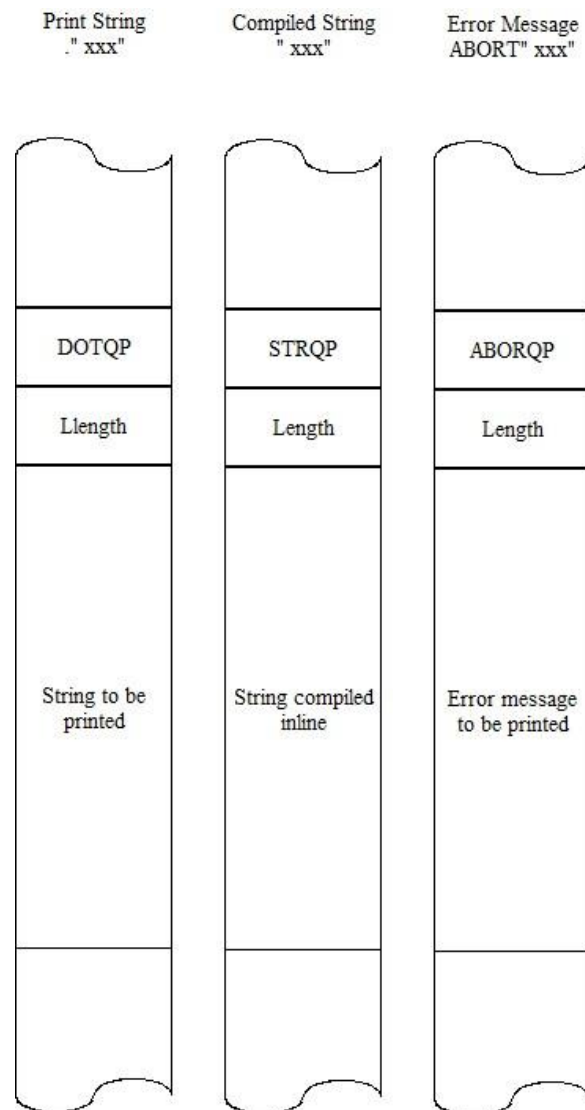
A string literal starts with a token, DOTQP, STRQP, or ABORQP, followed by a counted string of ASCII characters, null-filled to the word boundary. They are used to print a string in runtime, or make the string available for other words to use. They are used in the following ways:\

." print this message"

\$" push address of this string on stack"

ABORT" compile only"

These string literals are shown in the following figure:



In ceForth\_33, we have to build the entire Forth dictionary to work with the Virtual Forth Machine. Native C compilers do not provide good tools to build variable length fields required by an interpretive system like Forth. However, a set of macros can be defined in C as functions to assemble all fields in all Forth words, and link them all into a searchable dictionary. In the following Chapter, I will go through these macros and show you how to construct a dictionary for a Virtual Forth Machine to run in Visual Studio 2019 Community.

## Chapter 4. Macro Assembler for Forth in C

For a long time, my Forth in C had to import the Forth dictionary as a header file. The header file was produced by a Forth metacompiler, because I did not know how to generate the dictionary in a C program. Although C does not provide data structures for variable length and variable length parameter fields, one can write C code to place arbitrary byte and integer values in a data array. Placing bytes and integers into consecutive locations in an array can be orchestrated to build various fields and recorder in a dictionary.

Way back when, before Windows, Microsoft gave us a macro assembler called MASM, which I used to build the first eForth Model. I had a few macros to construct all 4 fields in a word record, and linked all word records into a linked dictionary. This mechanism of macro assembler can be realized in C to build my Forth dictionary. I first tried out this idea in Python, because Python is interactive, and I could see the dictionary as I was building it. I wrote three macros to build primitive words, colon worlds, and labels for partial token lists to allow branching and looping. Once verified in Python, these macros were ported to C straightforwardly.

Label macros allow branching and looping. However, it was difficult to make forward referencing working properly. MASM, as most other compilers, used 2 passes to resolve forward referencing. On the other hand, Chuck Moore designed Forth with a beautiful one pass compiler. It was not difficult to extend my macro assembler so that everything is compiled in a single pass. This is the macro assembler you will see as I go through the C code in ceForth\_33.

### Macro Assembler

In the Virtual Forth Machine, the dictionary is stored in an integer array `data[IP]`. It is aliased to a byte array `cData[P]`, where `IP` is an integer pointer and `P` is a byte pointer. To point to the same location in the dictionary,  $IP = P/4$ . To write consecutive bytes into the dictionary, we can do the following:

```
cData[P++] = char c;
```

To write consecutive integers, do the following:

```
Data[IP++] = int n;
```

Synchronizing  $IP = P/4$ , we can write anything and everything to build a Forth dictionary.

I first coded a simple macro assembler to build word records in the Forth dictionary. It consisted of four macros: `HEADER()` to build link fields and name fields, `CODE()` to build code fields for primitive words, `COLON()` to build code and parameter fields for colon words, and `LABEL()` to extend token lists in colon words for branching and looping. These four macros are as follows:

```
// Macro Assembler

int IMEDD = 0x80;
int COMPO = 0x40;
```

IMEDD and COMPO designate lexicon bits in the length byte of name field. IMEDD as bit 7 is called immediate bit, and it forces Forth compiler to execute this word instead of compiling its token into the dictionary. All Forth words building control structures are immediate words. COMPO as bit 6 is called compile-only bit. Many Forth words are dangerous. They may crash the system if executed by Forth interpreter. These words are marked by COMPO as compile-only, and they are only used by Forth compiler.

```
int BRAN=0, QBRAN=0, DONXT=0, DOTQP=0, STRQP=0, TOR=0, ABORQP=0;
```

BRAN, QBRAN, DONXT, DOTQP, STRQP, ABRQP, and TOR are forward references to primitive words the macro assembler needs to build control structures and string structures in colon words. They are initialized to 0 here, but will be resolved when these primitives are assembled.

HEADER() builds a link field and a name field for either a primitive or a colon word. A global variable `thread` contains the name field address (nfa) of the prior word in the dictionary. This address is first assembled as a 32-bit integer with `data[IP++]=thread;`. Now, `P` is pointing at the name field of the current word. This `P` is saved back to `thread`.

In the name field, the first byte is a lexicon byte, in which the lower 5 bits stores the length of the name, bit 6 indicates a compile-only word, and bit 7 indicates an immediate word. This lexicon byte is assembled first in the name field, and then the name string. The name field is then null-filled to the next 32-bit word boundary. Now, `P` is pointing to the code field, ready to assemble byte code.

```
void HEADER(int lex, const char seq[]) {
    IP = P >> 2;
    int i;
    int len = lex & 31;
    data[IP++] = thread;
    P = IP << 2;
    //printf("\n%X",thread);
    //for (i = thread >> 2; i < IP; i++)
    //{    printf(" %X",data[i]); }
    thread = P;
    cData[P++] = lex;
    for (i = 0; i < len; i++)
    {
        cData[P++] = seq[i];
    }
    while (P & 3) { cData[P++] = 0; }
    printf("\n");
    printf(seq);
    printf(" %X", P);
}
```

CODE () builds a code field for a primitive word. After HEADER () finishes building a name field, P has the code field address (cfa). This cfa is return and saved as an integer, which will be used as a parameter by macro COLON () as a token. A sequence of byte code can now be assembled by cData [P++] = c;. CODE () has a variable number of byte code as parameters. This number must be the first parameter int len. In this implementation, the length len is either 4 or 8. The code field will always be either 4 byte long or 8 byte long, and is always aligned to 32-bit word boundary.

CODE () , together with HEADER () , builds a record for a primitive word in Forth dictionary. It returns a cfa to be assigned as an integer token to construct token lists in colon words.

```
int CODE(int len, ...) {
    int addr = P;
    va_list argList;
    va_start(argList, len);
    for (; len; len--) {
        int j = va_arg(argList, int);
        cData[P++] = j;
        //printf(" %X",j);
    }
    va_end(argList);
    return addr;
}
```

COLON () builds a code field and a parameter field for a colon word. P has the code field address (cfa). This cfa is and saved as an integer, which will be used as a parameter by other COLON () macros as a token. A DOLST byte code with value 6 is assembled as an integer by data [IP++] = 6;. It then assembles a variable number of integer tokens as parameters. This number must be the first parameter int len.

COLON () , together with HEADER () , builds a record for a colon word in Forth dictionary. It returns a cfa to be assigned as an integer token to construct token lists in later colon words.

```
int COLON(int len, ...) {
    int addr = P;
    IP = P >> 2;
    data[IP++] = 6; // dolist
    va_list argList;
    va_start(argList, len);
    //printf(" %X ",6);
    for (; len; len--) {
        int j = va_arg(argList, int);
        data[IP++] = j;
        //printf(" %X",j);
    }
    P = IP << 2;
}
```

```

    va_end(argList);
    return addr;
}

```

`LABEL()` builds a partial token list in a colon word for branching and looping. `P` has the current token address. This address is return and saved as an integer, which will be used as a address following a branching token. It then assembles a variable number of integer tokens as parameters. This number must be the first parameter `int len`.

`LABEL()` builds a partial token list in a colon word. It returns an address for branching tokens as their target addresses.

`LABEL()` cannot handle forward reference. All references in its parameter list must be valid. Not-yet-defined label must be initialized to a value like 0. After a first pass of macro assembler, all references are resolved, and the valid addresses must be copied to replace the 0 initially assigned. This second pass must be done manually.

```

int LABEL(int len, ...) {
    int addr = P;
    IP = P >> 2;
    va_list argList;
    va_start(argList, len);
    //printf("\n%X ",addr);
    for (; len; len--) {
        int j = va_arg(argList, int);
        data[IP++] = j;
        //printf(" %X",j);
    }
    P = IP << 2;
    va_end(argList);
    return addr;
}

```

Using `LABEL()` to mark all locations for branching and looping, I built and tested `esp32Forth_61`. I checked the dictionary it produced byte-for-byte against the dictionary generated by `espForth_54`. It proved that a C program could correctly build a Forth dictionary. However, manually resolving forward references was not satisfactory, and a better set of macro is needed to assemble Forth dictionary automatically. The new scheme was tried out in `esp32forth_62`, and it was adapted in `ceForth_33`.

## Macros for Structured Programming

I could handle labels myself to demonstrate that a Forth dictionary could be built in C, but I would not expect users to manually resolve forward references. Chuck Moore demonstrated that all nested control structures could be compiled correctly in a single pass. It is certainly possible to do it in this macro assembler. The way to do it is to write all the Forth immediate

words in macros to set up branch and loop structures and resolve all references, forward or backward.

The macros we need are used to build the following structures:

```
BEGIN...AGAIN
BEGIN...UNTIL
BEGIN...WHILE...REPEAT
IF...ELSE...THEN
FOR...AFT...THEN...NEXT
```

Some macros generally assemble a branch token with a branch address, followed by a variable number of tokens. Other macros resolves a forward reference. The structures can be nested; therefore, a stack is necessary to pass address field locations, and resolved addresses when they are available. I draft the return stack mechanism to accomplish address parameter passing. Earlier I defined two replacement macros to clarify the return stack operations:

```
# define popR rack[(unsigned char)R--]
# define pushR rack[(unsigned char)++R]
```

BEGIN() starts an indefinite loop. It first pushes the current word address IP on the return stack, so that the loop terminating branch token can assemble the correct return address. It then assemble a token list with the parameters passing to BEGIN() macro. Number of parameters is indicated by the first parameter int len.

```
void BEGIN(int len, ...) {
    IP = P >> 2;
    //printf("\n%X BEGIN ",P);
    pushR = IP;
    va_list argList;
    va_start(argList, len);
    for (; len; len--) {
        int j = va_arg(argList, int);
        data[IP++] = j;
        //printf(" %X",j);
    }
    P = IP << 2;
    va_end(argList);
}
```

AGAIN() closes an indefinite loop. It first assembles a BRAN token, and then assembles the address BEGIN() left on the return stack to complete the loop structure. It then assemble a token list with the parameters passing to AGAIN() macro. Number of parameters is indicated by the first parameter int len.

```
void AGAIN(int len, ...) {
    IP = P >> 2;
    //printf("\n%X AGAIN ",P);
```

```

    data[IP++] = BRAN;
    data[IP++] = popR << 2;
    va_list argList;
    va_start(argList, len);
    for (; len; len--) {
        int j = va_arg(argList, int);
        data[IP++] = j;
        //printf(" %X",j);
    }
    P = IP << 2;
    va_end(argList);
}

```

UNTIL ( ) closes an indefinite loop. It first assembles a QBRAN token, and then assembles the address BEGIN ( ) left on the return stack to complete the loop structure. It then assemble a token list with the parameters passing to AGAIN ( ) macro. Number of parameters is indicated by the first parameter `int len`.

```

void UNTIL(int len, ...) {
    IP = P >> 2;
    //printf("\n%X UNTIL ",P);
    data[IP++] = QBRAN;
    data[IP++] = popR << 2;
    va_list argList;
    va_start(argList, len);
    for (; len; len--) {
        int j = va_arg(argList, int);
        data[IP++] = j;
        //printf(" %X",j);
    }
    P = IP << 2;
    va_end(argList);
}

```

WHILE ( ) closes a always clause and starts a true clause in an indefinite loop. It first assembles a QBRAN token with a null address. The words address after the null address is now pushed on the return stack under the address left by BEGIN ( ) . Two address on the return stack will be used by REPEAT ( ) macro to close the loop, and to resolve the address after QBRAN(). It then assemble a token list with the parameters passing to WHILE ( ) macro. Number of parameters is indicated by the first parameter `int len`.

```

void WHILE(int len, ...) {
    IP = P >> 2;
    int k;
    //printf("\n%X WHILE ",P);
    data[IP++] = QBRAN;

```

```

    data[IP++] = 0;
    k = popR;
    pushR = (IP - 1);
    pushR = k;
    va_list argList;
    va_start(argList, len);
    for (; len; len--) {
        int j = va_arg(argList, int);
        data[IP++] = j;
        //printf(" %X",j);
    }
    P = IP << 2;
    va_end(argList);
}

```

**REPEAT ()** closes a **BEGIN-WHILE-REPEAT** indefinite loop. It first assembles a **BRAN** token with the address left by **BEGIN ()** . The words address after the **BEGIN** address is now stored into the location whose address was left by **WHILE ()** on the return stack. It then assemble a token list with the parameters passing to **REPEAT ()** macro. Number of parameters is indicated by the first parameter `int len`.

```

void REPEAT(int len, ...) {
    IP = P >> 2;
    //printf("\n%X REPEAT ",P);
    data[IP++] = BRAN;
    data[IP++] = popR << 2;
    data[popR] = IP << 2;
    va_list argList;
    va_start(argList, len);
    for (; len; len--) {
        int j = va_arg(argList, int);
        data[IP++] = j;
        //printf(" %X",j);
    }
    P = IP << 2;
    va_end(argList);
}

```

**IF ()** starts a true clause in a branch structure. It first assembles a **QBRAN** token with a null address. The words address of the null address field is now pushed on the return stack. The null address field will be filled by **ELSE ()** or **THEN ()** , when they resolve the branch address. It then assemble a token list with the parameters passing to **IF ()** macro. Number of parameters is indicated by the first parameter `int len`.

```

void IF(int len, ...) {
    IP = P >> 2;

```

```

        //printf("\n%X IF ",P);
        data[IP++] = QBRAN;
        pushR = IP;
        data[IP++] = 0;
        va_list argList;
        va_start(argList, len);
        for (; len; len--) {
            int j = va_arg(argList, int);
            data[IP++] = j;
            //printf(" %X",j);
        }
        P = IP << 2;
        va_end(argList);
    }

```

ELSE () closes a true clause and starts a false clause in an IF-ELSE-THEN branch structure. It first assembles a BRAN token with a null address. The words address after the null address is now used to resolve the branch address assembled by IF (). The address of the null address field is pushed on the return stack to be used by THEN (). It then assemble a token list with the parameters passing to ELSE () macro. Number of parameters is indicated by the first parameter `int len`.

```

void ELSE(int len, ...) {
    IP = P >> 2;
    //printf("\n%X ELSE ",P);
    data[IP++] = BRAN;
    data[IP++] = 0;
    data[popR] = IP << 2;
    pushR = IP - 1;
    va_list argList;
    va_start(argList, len);
    for (; len; len--) {
        int j = va_arg(argList, int);
        data[IP++] = j;
        //printf(" %X",j);
    }
    P = IP << 2;
    va_end(argList);
}

```

THEN () closes an IF-THEN or IF-ELSE-THEN branch structure. It resolved the null address assembled by IF () or ELSE () with the current word address after THEN (). It then assemble a token list with the parameters passing to THEN () macro. Number of parameters is indicated by the first parameter `int len`.

```

void THEN(int len, ...) {

```

```

    IP = P >> 2;
    //printf("\n%X THEN ",P);
    data[popR] = IP << 2;
    va_list argList;
    va_start(argList, len);
    for (; len; len--) {
        int j = va_arg(argList, int);
        data[IP++] = j;
        //printf(" %X",j);
    }
    P = IP << 2;
    va_end(argList);
}

```

**FOR ()** starts a definite loop structure. It first assembles a TOR token and pushes the address of the current address field on the return stack. This address will be used by **NEXT ()** in its loop back address field. It then assemble a token list with the parameters passing to **FOR ()** macro. Number of parameters is indicated by the first parameter `int len`.

```

void FOR(int len, ...) {
    IP = P >> 2;
    //printf("\n%X FOR ",P);
    data[IP++] = TOR;
    pushR = IP;
    va_list argList;
    va_start(argList, len);
    for (; len; len--) {
        int j = va_arg(argList, int);
        data[IP++] = j;
        //printf(" %X",j);
    }
    P = IP << 2;
    va_end(argList);
}

```

**NEXT ()** closes an definite loop. It first assembles a DONXT token, and then assembles the address **FOR ()** left on the return stack to complete the loop structure. It then assemble a token list with the parameters passing to **NEXT ()** macro. Number of parameters is indicated by the first parameter `int len`.

```

void NEXT(int len, ...) {
    IP = P >> 2;
    //printf("\n%X NEXT ",P);
    data[IP++] = DONXT;
    data[IP++] = popR << 2;
    va_list argList;

```

```

    va_start(argList, len);
    for (; len; len--) {
        int j = va_arg(argList, int);
        data[IP++] = j;
        //printf(" %X",j);
    }
    P = IP << 2;
    va_end(argList);
}

```

AFT ( ) closes a always clause and starts a skip-once-only clause in an definite loop. It first assembles a BRAN token with a null address. The words address after the null address is now pushed on the return stack replacing the address left by FOR ( ) . The address of the null address field is pushed on the return stack. Top address on the return stack will be used by THEN ( ) macro to close skip-once-only clause, and to resolve the address after AFT ( ) . It then assemble a token list with the parameters passing to AFT ( ) macro. Number of parameters is indicated by the first parameter `int len`.

```

void AFT(int len, ...) {
    IP = P >> 2;
    int k;
    //printf("\n%X AFT ",P);
    data[IP++] = BRAN;
    data[IP++] = 0;
    k = popR;
    pushR = IP;
    pushR = IP - 1;
    va_list argList;
    va_start(argList, len);
    for (; len; len--) {
        int j = va_arg(argList, int);
        data[IP++] = j;
        //printf(" %X",j);
    }
    P = IP << 2;
    va_end(argList);
}

```

A token list in a colon word contains mostly tokens, cfa of other words. However, other information can be embedded in the token list as literals. We have seen lots of address literals following BRAN, QBRAN and DONXT to build control structures. There is an integer literal following DOLIT. The integer literal will be pushed on the data stack at run time when the token list is executed. Another class of literals is string literals to embed strings in token list. We need three more macros to build string structures:

```

." <string"
$" <string>"

```

ABORT" <string>"

DOTQ ( ) starts a string literal to be printed out at run time. It first assembles a DOTQP token. Then the string, terminated by another double quote, is assembled as a counted string. The string is null-filled to the 32-bit word boundary, similar to what HEADER ( ) does.

```
void DOTQ(const char seq[]) {
    IP = P >> 2;
    int i;
    int len = strlen(seq);
    data[IP++] = DOTQP;
    P = IP << 2;
    cData[P++] = len;
    for (i = 0; i < len; i++)
    {
        cData[P++] = seq[i];
    }
    while (P & 3) { cData[P++] = 0; }
    //printf("\n%X ",P);
    //printf(seq);
}
```

STRQ ( ) starts a string literal to be accessed at run time. When it executes, it leaves the address of the count byte of this string on the data stack. Other Forth words will make use of this string, knowing its count byte address. It first assembles a STRQP token. Then the string, terminated by another double quote, is assembled as a counted string. The string is null-filled to the 32-bit word boundary, similar to what HEADER ( ) does.

```
void STRQ(const char seq[]) {
    IP = P >> 2;
    int i;
    int len = strlen(seq);
    data[IP++] = STRQP;
    P = IP << 2;
    cData[P++] = len;
    for (i = 0; i < len; i++)
    {
        cData[P++] = seq[i];
    }
    while (P & 3) { cData[P++] = 0; }
    //printf("\n%X ",P);
    //printf(seq);
}
```

ABORQ ( ) starts a string literal as a warning message to be printed out when Forth is aborted. It first assembles a ABORQP token. Then the string, terminated by another double quote, is

assembled as a counted string. The string is null-filled to the 32-bit word boundary, similar to what `HEADER()` does.

```
void ABORQ(const char seq[]) {
    IP = P >> 2;
    int i;
    int len = strlen(seq);
    data[IP++] = ABORQP;
    P = IP << 2;
    cData[P++] = len;
    for (i = 0; i < len; i++)
    {
        cData[P++] = seq[i];
    }
    while (P & 3) { cData[P++] = 0; }
    //printf("\n%X ",P);
    //printf(seq);
}
```

`Checksum()` dumps 32 bytes of memory to Serial Terminal, roughly in the Intel dump format. First display 4 bytes of address and a space. Then 32 bytes of data, 2 hex characters to a byte. Finally the sum of these 32 bytes is displayed in 2 hex characters. The checksums are very useful for me to compare the dictionary assembled by this macro assembler against the dictionary produced by my earlier F# metacompiler.

```
void CheckSum() {
    int i;
    char sum = 0;
    printf("\n%4X ", P);
    for (i = 0; i < 16; i++) {
        sum += cData[P];
        printf("%2X", cData[P++]);
    }
    printf(" %2X", sum & 0xFF);
}
```

The macro assembler is complete. The macros thus defined will be used to construct primitive words and colon words to form a complete Forth dictionary for VFM to run.

## Chapter 5. Primitive Words

### Byte Code Mnemonic

To facilitate the macro assembler to assemble consecutive byte code, individual byte code are given mnemonic names so we don't have to use hard numbers. Mnemonic names are simply names of the corresponding primitive routines, prefixed with "as\_", to show that they are assembly mnemonics.

```
// Byte Code Assembler
```

```
int as_nop = 0;
int as_bye = 1;
int as_qrx = 2;
int as_txsto = 3;
int as_docon = 4;
int as_dolit = 5;
int as_dolist = 6;
int as_exit = 7;
int as_execu = 8;
int as_donext = 9;
int as_qbran = 10;
int as_bran = 11;
int as_store = 12;
int as_at = 13;
int as_cstor = 14;
int as_cat = 15;
int as_rpat = 16;
int as_rpsto = 17;
int as_rfrom = 18;
int as_rat = 19;
int as_tor = 20;
int as_spat = 21;
int as_spsto = 22;
int as_drop = 23;
int as_dup = 24;
int as_swap = 25;
int as_over = 26;
int as_zless = 27;
int as_andd = 28;
int as_orr = 29;
int as_xorr = 30;
int as_uplus = 31;
int as_next = 32;
int as_qdup = 33;
```

```

int as_rot = 34;
int as_ddrop = 35;
int as_ddup = 36;
int as_plus = 37;
int as_inver = 38;
int as_negat = 39;
int as_dnega = 40;
int as_subb = 41;
int as_abss = 42;
int as_equal = 43;
int as_uless = 44;
int as_less = 45;
int as_ummod = 46;
int as_msmod = 47;
int as_slmod = 48;
int as_mod = 49;
int as_slash = 50;
int as_umsta = 51;
int as_star = 52;
int as_mstar = 53;
int as_ssmod = 54;
int as_stasl = 55;
int as_pick = 56;
int as_pstor = 57;
int as_dstor = 58;
int as_dat = 59;
int as_count = 60;
int as_dovar = 61;
int as_max = 62;
int as_min = 63;

```

## Assembling Forth Dictionary

The macro assembler run first in the `main()` loop required by Visual Studio:

```

/*
 * Main Program
 */
int main(int ac, char* av[])
{
    cData = (unsigned char*)data;
    P = 512;
    R = 0;

```

`cData[]` byte array must be aligned with `data[]` array, to allow bytes to be accessed in the integer `data[]` array.

The macros `HEADER()` and `CODE()` defined previously in the macro assembler are now used to assemble the primitive words to the dictionary. The dictionary is divided in two sections, first the primitive words as a kernel of FVM, and next are all the colon words, which constitutes the bulk of Forth dictionary.

The first 512 byte of the dictionary are allocated to a terminal input buffer, and an area storing initial values of user variables. `P` is therefore initialized to 512. `thread` was already initialized to 0, ready to build the linked records of the first Forth word.

## User Variables

User variables are in the area between 0x80-0xAC. They contain vital information for the Forth interpreter to work properly.

User Variable	Address	Initial Value	Function
HLD	0x80	0	Pointer to text buffer for number output.
SPAN	0x84	0	Number of input characters.
>IN	0x88	0	Pointer to next character to be interpreted.
#TIB	0x8C	0	Number of characters received in terminal input buffer.
'TIB	0x90	0	Pointer to Terminal Input Buffer.
BASE	0x94	0x10	Number base for hexadecimal numeric conversions.
CONTEXT	0a98	0x1DDC	Pointer to name field of last word in dictionary.
CP	0x9C	0x1DE8	Pointer to top of dictionary, first free memory location to add new words. It is saved by "h forth @" on top of the source code page.
LAST	0xA0	0x1DDC	Pointer to name field of last word.
'EVAL	0xA4	0x13D4	Execution vector of text interpreter, initialized to point to <code>\$INTERPRET</code> . It may be changed to point to <code>\$COMPILE</code> in compiling mode.
'ABORT	0xA8	0x1514	Pointer to <code>QUIT</code> word to handle error conditions.
tmp	0xAC	0	Scratch pad.

// Kernel

```

HEADER(3, "HLD");
int HLD = CODE(8, as_docon, as_next, 0, 0, 0x80, 0, 0, 0);
HEADER(4, "SPAN");
int SPAN = CODE(8, as_docon, as_next, 0, 0, 0x84, 0, 0, 0);
HEADER(3, ">IN");
int INN = CODE(8, as_docon, as_next, 0, 0, 0x88, 0, 0, 0);
HEADER(4, "#TIB");
int NTIB = CODE(8, as_docon, as_next, 0, 0, 0x8C, 0, 0, 0);
HEADER(4, "'TIB");

```

```

int TTIB = CODE(8, as_docon, as_next, 0, 0, 0X90, 0, 0, 0);
HEADER(4, "BASE");
int BASE = CODE(8, as_docon, as_next, 0, 0, 0X94, 0, 0, 0);
HEADER(7, "CONTEXT");
int CNTXT = CODE(8, as_docon, as_next, 0, 0, 0X98, 0, 0, 0);
HEADER(2, "CP");
int CP = CODE(8, as_docon, as_next, 0, 0, 0X9C, 0, 0, 0);
HEADER(4, "LAST");
int LAST = CODE(8, as_docon, as_next, 0, 0, 0XA0, 0, 0, 0);
HEADER(5, "'EVAL");
int TEVAL = CODE(8, as_docon, as_next, 0, 0, 0XA4, 0, 0, 0);
HEADER(6, "'ABORT");
int TABRT = CODE(8, as_docon, as_next, 0, 0, 0XA8, 0, 0, 0);
HEADER(3, "tmp");
int TEMP = CODE(8, as_docon, as_next, 0, 0, 0XAC, 0, 0, 0);

```

## Kernel Words

Forth words have a link field, a name field, a code field, and an optional parameter field. Link field and name field are constructed by `HEADER()` macro. Primitive words have variable length code field and are assembled by `CODE()` macro. `CODE()` returns the code field address (cfa) of a word. This cfa is assigned to an integer, which will be invoked by `COLON()` macro to assemble a token into the token list in the parameter field of a colon word.

Kernel words reveal the 32-bit nature of the eP32 microcontroller this VFM emulates. Most kernel words execute one byte code followed by the return byte code `next()`. Two empty bytes are null-filled to make up the 32-bit word. In all the user variable words listed above, and many other words which return only a constant, the byte code are `docon()` and `next()`, followed by 2 null bytes. The constant value is stored in the next 32-bit word. These words have 8 bytes in their code fields. `docon()` has to read the constant value stored in the next 32-bit word.

```
// primitive words
```

`NOP(--)` No operation. Break FVM and returns to Arduino `loop()`.

```

HEADER(3, "NOP");
int NOP = CODE(4, as_next, 0, 0, 0);

```

`BYE(--)` Return to Visual Studio OS.

```

HEADER(3, "BYE");
int BYE = CODE(4, as_bye, as_next, 0, 0);

```

`?RX(-- c t | f)` Read a character from terminal input device. Return character c and a true flag if available. Else return a false flag.

```

HEADER(3, "?RX");
int QRX = CODE(4, as_qrx, as_next, 0, 0);

```

**TX! ( c -- )** Send a character to the Windows console.

```
HEADER(3, "TX!");  
int TXSTO = CODE(4, as_txsto, as_next, 0, 0);
```

**DOCON ( -- w)** Push the next token onto the data stack as a constant.

```
HEADER(5, "DOCON");  
int DOCON = CODE(4, as_docon, as_next, 0, 0);
```

**DOLIT ( -- w)** Push the next token onto the data stack as an integer literal. It allows numbers to be compiled as in-line literals, supplying data to the data stack at run time.

```
HEADER(5, "DOLIT");  
int DOLIT = CODE(4, as_dolit, as_next, 0, 0);
```

**DOLIST ( -- )** Push the current Instruction Pointer (IP) on the return stack and then pops the Program Counter P into IP from the data stack. When `next()` is executed, the tokens in the list are executed consecutively. `Dolist`, is in the code field of all colon words. The token list in a colon word must be terminated by `EXIT`.

```
HEADER(6, "DOLIST");  
int DOLST = CODE(4, as_dolist, as_next, 0, 0);
```

**EXIT ( -- )** Terminate all token lists in colon words. `EXIT` pops the execution address saved on the return stack back into the IP register and thus restores the condition before the colon word was entered. Execution of the calling token list will continue.

```
HEADER(4, "EXIT");  
int EXITT = CODE(4, as_exit, as_next, 0, 0);
```

**EXECUTE ( a -- )** Take the execution address from the data stack and executes that token. This powerful word allows you to execute any token which is not a part of a token list.

```
HEADER(7, "EXECUTE");  
int EXECU = CODE(4, as_execu, as_next, 0, 0);
```

**DONEXT ( -- )** Terminate a FOR-NEXT loop. The loop count was pushed on return stack, and is decremented by `DONEXT`. If the count is not negative, jump to the address following `DONEXT`; otherwise, pop the count off return stack and exit the loop. `DONEXT` is compiled by `NEXT`.

```
HEADER(6, "DONEXT");  
DONXT = CODE(4, as_donext, as_next, 0, 0);
```

**QBRANCH ( f -- )** Test top element as a flag on data stack. If it is zero, branch to the address following `QBRANCH`; otherwise, continue execute the token list following the address.

`QBRANCH` is compiled by `IF`, `WHILE` and `UNTIL`.

```
HEADER(7, "QBRANCH");  
QBRAN = CODE(4, as_qbran, as_next, 0, 0);
```

**BRANCH ( -- )** Branch to the address following `BRANCH`. `BRANCH` is compiled by `AFT`, `ELSE`, `REPEAT` and `AGAIN`.

```
HEADER(6, "BRANCH");
BRAN = CODE(4, as_bran, as_next, 0, 0);
```

**! ( n a -- ) Store integer n into memory location a.**

```
HEADER(1, "!");
int STORE = CODE(4, as_store, as_next, 0, 0);
```

**@ ( a -- n ) Replace memory address a with its integer contents fetched from this location.**

```
HEADER(1, "@");
int AT = CODE(4, as_at, as_next, 0, 0);
```

**C! ( c b -- ) Store a byte value c into memory location b.**

```
HEADER(2, "C!");
int Cstor = CODE(4, as_cstor, as_next, 0, 0);
```

**C@ ( b -- n ) Replace byte memory address b with its byte contents fetched from this location.**

```
HEADER(2, "C@");
int CAT = CODE(4, as_cat, as_next, 0, 0);
```

**R> ( n -- ) Pop a number off the data stack and pushes it on the return stack.**

```
HEADER(2, "R>");
int RFROM = CODE(4, as_rfrom, as_next, 0, 0);
```

**R@ ( -- n ) Copy a number off the return stack and pushes it on the return stack.**

```
HEADER(2, "R@");
int RAT = CODE(4, as_rat, as_next, 0, 0);
```

**>R( -- n ) Pop a number off the return stack and pushes it on the data stack.**

```
HEADER(2, ">R");
TOR = CODE(4, as_tor, as_next, 0, 0);
```

**DROP ( w -- ) Discard top stack item.**

```
HEADER(4, "DROP");
int DROP = CODE(4, as_drop, as_next, 0, 0);
```

**DUP ( w -- w w ) Duplicate the top stack item.**

```
HEADER(3, "DUP");
int DUPP = CODE(4, as_dup, as_next, 0, 0);
```

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

```
HEADER(4, "SWAP");
int SWAP = CODE(4, as_swap, as_next, 0, 0);
```

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

```
HEADER(4, "OVER");
int OVER = CODE(4, as_over, as_next, 0, 0);
```

0< ( n – f ) Examine the top item on the data stack for its negativeness. If it is negative, return a -1 for true. If it is 0 or positive, return a 0 for false.

```
HEADER(2, "0<");
int ZLESS = CODE(4, as_zless, as_next, 0, 0);
```

AND ( w w -- w ) Bitwise AND.

```
HEADER(3, "AND");
int ANDD = CODE(4, as_andd, as_next, 0, 0);
```

OR ( w w -- w ) Bitwise inclusive OR.

```
HEADER(2, "OR");
int ORR = CODE(4, as_orr, as_next, 0, 0);
```

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

```
HEADER(3, "XOR");
int XORR = CODE(4, as_xorr, as_next, 0, 0);
```

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

```
HEADER(3, "UM+");
int UPLUS = CODE(4, as_uplus, as_next, 0, 0);
```

NEXT ( -- ) Jump to the next token in the token list under processing.

```
HEADER(4, "NEXT");
int NEXTT = CODE(4, as_next, as_next, 0, 0);
```

?DUP ( w -- w w | 0 ) Dup top of stack if its is not zero.

```
HEADER(4, "?DUP");
int QDUP = CODE(4, as_qdup, as_next, 0, 0);
```

ROT ( w1 w2 w3 -- w2 w3 w1 ) Rot 3rd item to top.

```
HEADER(3, "ROT");
int ROT = CODE(4, as_rot, as_next, 0, 0);
```

2DROP ( w w -- ) Discard two items on stack.

```
HEADER(5, "2DROP");
int DDROP = CODE(4, as_ddrop, as_next, 0, 0);
```

**2DUP ( w1 w2 -- w1 w2 w1 w2 ) Duplicate top two items.**

```
HEADER(4, "2DUP");
int DDUP = CODE(4, as_ddup, as_next, 0, 0);
```

**+ ( w w -- sum ) Add top two items.**

```
HEADER(1, "+");
int PLUS = CODE(4, as_plus, as_next, 0, 0);
```

**NOT ( w -- w ) One's complement of top.**

```
HEADER(3, "NOT");
int INVER = CODE(4, as_inver, as_next, 0, 0);
```

**NEGATE ( n -- -n ) Two's complement of top.**

```
HEADER(6, "NEGATE");
int NEGAT = CODE(4, as_negat, as_next, 0, 0);
```

**DNEGATE ( d -- -d ) Two's complement of top double.**

```
HEADER(7, "DNEGATE");
int DNEGA = CODE(4, as_dnega, as_next, 0, 0);
```

**- ( n1 n2 -- n1-n2 ) Subtraction.**

```
HEADER(1, "-");
int SUBBB = CODE(4, as_subb, as_next, 0, 0);
```

**ABS ( n -- n ) Return the absolute value of n.**

```
HEADER(3, "ABS");
int ABSS = CODE(4, as_abss, as_next, 0, 0);
```

**= ( w w -- t ) Return true if top two are equal.**

```
HEADER(1, "=");
int EQUAL = CODE(4, as_equal, as_next, 0, 0);
```

**U< ( u1 u2 -- t ) Unsigned compare of top two items.**

```
HEADER(2, "U<");
int ULESS = CODE(4, as_uless, as_next, 0, 0);
```

**< ( n1 n2 -- t ) Signed compare of top two items.**

```
HEADER(1, "<");
```

```
int LESS = CODE(4, as_less, as_next, 0, 0);
```

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

```
HEADER(6, "UM/MOD");  
int UMMOD = CODE(4, as_ummod, as_next, 0, 0);
```

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

```
HEADER(5, "M/MOD");  
int MSMOD = CODE(4, as_msmmod, as_next, 0, 0);
```

/MOD ( n1 n2 -- r q ) Signed divide. Return mod and quotient.

```
HEADER(4, "/MOD");  
int SLMOD = CODE(4, as_slmod, as_next, 0, 0);
```

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

```
HEADER(3, "MOD");  
int MODD = CODE(4, as_mod, as_next, 0, 0);
```

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

```
HEADER(1, "/");  
int SLASH = CODE(4, as_slash, as_next, 0, 0);
```

UM\* ( u1 u2 -- ud ) Unsigned multiply. Return double product.

```
HEADER(3, "UM*");  
int UMSTA = CODE(4, as_umsta, as_next, 0, 0);
```

\* ( n n -- n ) Signed multiply. Return single product.

```
HEADER(1, "*");  
int STAR = CODE(4, as_star, as_next, 0, 0);
```

M\* ( n1 n2 -- d ) Signed multiply. Return double product.

```
HEADER(2, "M*");  
int MSTAR = CODE(4, as_mstar, as_next, 0, 0);
```

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

```
HEADER(5, "*/MOD");  
int SSMOD = CODE(4, as_ssmod, as_next, 0, 0);
```

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

```
HEADER(2, "*/");  
int STASL = CODE(4, as_stasl, as_next, 0, 0);
```

**PICK ( ... +n -- ... w )** Copy the nth stack item to top.

```
HEADER(4, "PICK");  
int PICK = CODE(4, as_pick, as_next, 0, 0);
```

**+! ( n a -- )** Add n to the contents at address a.

```
HEADER(2, "+!");  
int PSTOR = CODE(4, as_pstor, as_next, 0, 0);
```

**2! ( d a -- )** Store the double integer to address a.

```
HEADER(2, "2!");  
int DSTOR = CODE(4, as_dstor, as_next, 0, 0);
```

**2@ ( a -- d )** Fetch double integer from address a.

```
HEADER(2, "2@");  
int DAT = CODE(4, as_dat, as_next, 0, 0);
```

**COUNT ( b -- b+1 +n )** Return count byte of a string and add 1 to byte address.

```
HEADER(5, "COUNT");  
int COUNT = CODE(4, as_count, as_next, 0, 0);
```

**MAX ( n1 n2 -- n )** Return the greater of two top stack items.

```
HEADER(3, "MAX");  
int MAX = CODE(4, as_max, as_next, 0, 0);
```

**MIN ( n1 n2 -- n )** Return the smaller of top two stack items.

```
HEADER(3, "MIN");  
int MIN = CODE(4, as_min, as_next, 0, 0);
```

**BL ( -- 32 )** Return the blank ASCII code 32.

```
HEADER(2, "BL");  
int BLANK = CODE(8, as_docon, as_next, 0, 0, 32, 0, 0, 0);
```

**CELL ( -- 4 )** Return number of bytes in a 32-bit integer.

```
HEADER(4, "CELL");
```

```
int CELL = CODE(8, as_docon, as_next, 0, 0, 4, 0, 0, 0);
```

**CELL+ ( n – n+4 ) Add 4 to n.**

```
HEADER(5, "CELL+");  
int CELLP = CODE(8, as_docon, as_plus, as_next, 0, 4, 0, 0, 0);
```

**CELL- ( n – n-4 ) Subtract 4 from n.**

```
HEADER(5, "CELL-");  
int CELLM = CODE(8, as_docon, as_subb, as_next, 0, 4, 0, 0, 0);
```

**CELLS ( n – n\*4 ) Multiply n by 4.**

```
HEADER(5, "CELLS");  
int CELLS = CODE(8, as_docon, as_star, as_next, 0, 4, 0, 0, 0);
```

**CELL/ ( n – n+4 ) Divide n by 4.**

```
HEADER(5, "CELL/");  
int CELLD = CODE(8, as_docon, as_slash, as_next, 0, 4, 0, 0, 0);
```

**1+ ( n – n+1 ) Increment n.**

```
HEADER(2, "1+");  
int ONEP = CODE(8, as_docon, as_plus, as_next, 0, 1, 0, 0, 0);
```

**1- ( n – n-1 ) Decrement n.**

```
HEADER(2, "1-");  
int ONEM = CODE(8, as_docon, as_subb, as_next, 0, 1, 0, 0, 0);
```

**DOVAR ( – a ) Return address of a variable.**

```
HEADER(5, "DOVAR");  
int DOVAR = CODE(4, as_dovar, as_next, 0, 0);
```

## Chapter 7. Colon Words

HEADER ( ) assembles link field and name field for colon words. COLON ( ) adds a dolst byte code to form the code field. COLON ( ) then assembles a variable length token list. Tokens are cfas of other words.

Complicated colon words contain control structures, integer literals, and string literals. The macro assembler has all the macros to build these structures automatically in token lists. These macros allowed me to transcribe almost literally original Forth code into listing in C.

### Common Words

// Common Colon Words

?KEY ( -- c T|F ) Return a character and a true flag if the character has been received. If no character was received, return a false flag

```
HEADER(4, "?KEY");
int QKEY = COLON(2, QRX, EXITT);
```

KEY ( -- c ) wait for the console to receive a character c.

```
HEADER(3, "KEY");
int KEY = COLON(0);
BEGIN(1, QKEY);
UNTIL(1, EXITT);
```

TX! ( c -- ) Send a character to the Windows console.

```
HEADER(4, "EMIT");
int EMIT = COLON(2, TXSTO, EXITT);
```

WITHIN ( u ul uh -- t ) checks whether the third item on the data stack is within the range as specified by the top two numbers on the data stack. The range is inclusive as to the lower limit and exclusive to the upper limit. If the third item is within range, a true flag is returned on the data stack. Otherwise, a false flag is returned. All numbers are assumed to be unsigned integers.

```
HEADER(6, "WITHIN");
int WITHI = COLON(7, OVER, SUBBB, TOR, SUBBB, RFROM, ULESS, EXITT);
```

>CHAR ( c -- c ) is very important in converting a non-printable character to a harmless 'underscore' character (ASCII 95). As Forth is designed to communicate with you through a serial I/O device, it is important that Forth will not emit control characters to the host and causes unexpected behavior on the host computer. >CHAR thus filters the characters before they are sent out by TYPE.

```
HEADER(5, ">CHAR");
int TCHAR = COLON(8, DOLIT, 0x7F, ANDD, DUPP, DOLIT, 0x7F, BLANK, WITHI);
IF(3, DROP, DOLIT, 0x5F);
THEN(1, EXITT);
```

**ALIGNED ( b -- a )** changes the address to the next cell boundary so that it can be used to address 32 bit word in memory.

```
HEADER(7, "ALIGNED");
int ALIGN = COLON(7, DOLIT, 3, PLUS, DOLIT, 0FFFFFFFC, ANDD, EXITT);
```

**HERE ( -- a )** returns the address of the first free location above the code dictionary, where new words are compiled.

```
HEADER(4, "HERE");
int HERE = COLON(3, CP, AT, EXITT);
```

**PAD ( -- a )** returns the address of the text buffer where numbers are constructed and text strings are stored temporarily.

```
HEADER(3, "PAD");
int PAD = COLON(5, HERE, DOLIT, 0X50, PLUS, EXITT);
```

**TIB ( -- a )** returns the terminal input buffer where input text string is held.

```
HEADER(3, "TIB");
int TIB = COLON(3, TTIB, AT, EXITT);
```

**@EXECUTE ( a -- )** is a special word supporting the vectored execution words in Forth. It fetches the code field address of a token and executes the token.

```
HEADER(8, "@EXECUTE");
int ATEXE = COLON(2, AT, QDUP);
IF(1, EXECU);
THEN(1, EXITT);
```

**CMOVE ( b b u -- )** copies a memory array from one location to another. It copies one byte at a time.

```
HEADER(5, "CMOVE");
int CMOVEE = COLON(0);
FOR(0);
AFT(8, OVER, CAT, OVER, CSTOR, TOR, ONEP, RFROM, ONEP);
THEN(0);
NEXT(2, DDROP, EXITT);
```

**MOVE ( b b u -- )** copies a memory array from one location to another. It copies one word at a time.

```
HEADER(4, "MOVE");
int MOVE = COLON(1, CELLD);
FOR(0);
AFT(8, OVER, AT, OVER, STORE, TOR, CELLP, RFROM, CELLP);
THEN(0);
NEXT(2, DDROP, EXITT);
```

**FILL( b u c -- )** fills a memory array with the same byte c.

```
HEADER(4, "FILL");
```

```

int FILL = COLON(1, SWAP);
FOR(1, SWAP);
AFT(3, DDUP, CSTOR, ONEP);
THEN(0);
NEXT(2, DDROP, EXITT);

```

## Numeric Output

Forth is interesting in its special capabilities in handling numbers across the man-machine interface. It recognizes that the machine and the human prefer very different representations of numbers. The machine prefers a binary representation, but the human prefers decimal Arabic digital representations. 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 the binary form in the CPU and in memory. Only when numbers are imported or exported for human consumption are they converted to the external ASCII representation. The radix of external representation is controlled by the radix value stored in the variable BASE.

Since BASE is a system variable, you can select any reasonable radix for entering numbers into the computer and formatting numbers to be shown to you. Most programming languages can handle a small set of radices, like decimal, octal, hexadecimal and binary, with explicit prefix or postfix characters. In Forth, radix for number conversion is implicit, stored in BASE. It had caused endless grieves, even to very experience Forth programmers, because one did not know the true value of a number without knowing what was in BASE at the moment.

BASE is one of the very important inventions by Chuck Moore who gave us the Forth language. It gives us freedom of expression, in representing numbers any way we choose for whatever the reason at the moment.

```
// Number Conversions
```

DIGIT ( u -- c ) converts an integer to an ASCII digit.

```

HEADER(5, "DIGIT");
int DIGIT = COLON(12, DOLIT, 9, OVER, LESS, DOLIT, 7, ANDD, PLUS, DOLIT, 0X30,
PLUS, EXITT);

```

EXTRACT ( n base -- n c ) extracts the least significant digit from a number n. n is divided by the radix in BASE and returned on the stack.

```

HEADER(7, "EXTRACT");
int EXTRC = COLON(7, DOLIT, 0, SWAP, UMMOD, SWAP, DIGIT, EXITT);

```

<# ( -- ) initiates the output number conversion process by storing PAD buffer address into variable HLD, which points to the location next numeric digit will be stored.

```

HEADER(2, "<#");
int BDIGS = COLON(4, PAD, HLD, STORE, EXITT);

```

**HOLD ( c -- )** appends an ASCII character whose code is on the top of the parameter stack, to the numeric output string at HLD. HLD is decremented to receive the next digit.

```

HEADER(4, "HOLD");
int HOLD = COLON(8, HLD, AT, ONEM, DUPP, HLD, STORE, CSTOR, EXITT);

```

**# (dig) ( u -- u )** extracts one digit from integer on the top of the parameter stack, according to radix in BASE, and add it to output numeric string.

```

HEADER(1, "#");
int DIG = COLON(5, BASE, AT, EXTRC, HOLD, EXITT);

```

**#S (digs) ( u -- 0 )** extracts all digits to output string until the integer on the top of the parameter stack is divided down to 0.

```

HEADER(2, "#S");
int DIGS = COLON(0);
BEGIN(2, DIG, DUPP);
WHILE(0);
REPEAT(1, EXITT);

```

**SIGN ( n -- )** inserts a - sign into the numeric output string if the integer on the top of the parameter stack is negative.

```

HEADER(4, "SIGN");
int SIGN = COLON(1, ZLESS);
IF(3, DOLIT, 0X2D, HOLD);
THEN(1, EXITT);

```

**#> ( w -- b u )** terminates the numeric conversion and pushes the address and length of output numeric string on the parameter stack.

```

HEADER(2, "#>");
int EDIGS = COLON(7, DROP, HLD, AT, PAD, OVER, SUBBB, EXITT);

```

**str ( n -- b u )** converts a signed integer on the top of data stack to a numeric output string.

```

HEADER(3, "str");
int STRR = COLON(9, DUPP, TOR, ABSS, BDIGS, DIGS, RFROM, SIGN, EDIGS, EXITT);

```

**HEX ( -- )** sets numeric conversion radix in BASE to 16 for hexadecimal conversions.

```

HEADER(3, "HEX");
int HEXX = COLON(5, DOLIT, 16, BASE, STORE, EXITT);

```

**DECIMAL ( -- )** sets numeric conversion radix in BASE to 10 for decimal conversions.

```

HEADER(7, "DECIMAL");
int DECIM = COLON(5, DOLIT, 10, BASE, STORE, EXITT);

```

**wupper ( w -- w' )** converts 4 bytes in a word to upper case characters.

```

HEADER(6, "wupper");
int UPPER = COLON(4, DOLIT, 0x5F5F5F5F, ANDD, EXITT);

```

>upper ( c -- UC ) converts a character to upper case.

```

HEADER(6, ">upper");
int TOUPP = COLON(6, DUPP, DOLIT, 0x61, DOLIT, 0x7B, WITHI);
IF(3, DOLIT, 0x5F, ANDD);
THEN(1, EXITT);

```

DIGIT? ( c base -- u t ) converts a digit to its numeric value according to the current base, and NUMBER? converts a number string to a single integer.

```

HEADER(6, "DIGIT?");
int DIGTQ = COLON(9, TOR, TOUPP, DOLIT, 0x30, SUBBB, DOLIT, 9, OVER, LESS);
IF(8, DOLIT, 7, SUBBB, DUPP, DOLIT, 10, LESS, ORR);
THEN(4, DUPP, RFROM, ULESS, EXITT);

```

NUMBER? ( a -- n T | a F ) converts a string of digits to a single integer. If the first character is a \$ sign, the number is assumed to be in hexadecimal. Otherwise, the number will be converted using the radix value stored in BASE. For negative numbers, the first character should be a - sign. No other characters are allowed in the string. If a non-digit character is encountered, the address of the string and a false flag are returned. Successful conversion returns the integer value and a true flag. If the number is larger than  $2^{**n}$ , where n is the bit width of a single integer, only the modulus to  $2^{**n}$  will be kept.

```

HEADER(7, "NUMBER?");
int NUMBQ = COLON(12, BASE, AT, TOR, DOLIT, 0, OVER, COUNT, OVER, CAT, DOLIT,
0X24, EQUAL);
IF(5, HEXX, SWAP, ONEP, SWAP, ONEM);
THEN(13, OVER, CAT, DOLIT, 0X2D, EQUAL, TOR, SWAP, RAT, SUBBB, SWAP, RAT, PLUS,
QDUP);
IF(1, ONEM);
FOR(6, DUPP, TOR, CAT, BASE, AT, DIGTQ);
WHILE(7, SWAP, BASE, AT, STAR, PLUS, RFROM, ONEP);
NEXT(2, DROP, RAT);
IF(1, NEGAT);
THEN(1, SWAP);
ELSE(6, RFROM, RFROM, DDROP, DDROP, DOLIT, 0);
THEN(1, DUPP);
THEN(6, RFROM, DDROP, RFROM, BASE, STORE, EXITT);

```

## Number Output

The output number string is built below the PAD buffer. The least significant digit is extracted from an integer on the top of data stack by dividing it by the current radix in BASE. One digit thus extracted is added to the output string backwards from PAD to lower 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 outputting.

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 tokens is very versatile and can handle many different output formats.

// Terminal Output

SPACE ( -- ) outputs a blank space character.

```
HEADER(5, "SPACE");
int SPACE = COLON(3, BLANK, EMIT, EXITT);
```

CHARS ( +n c -- ) outputs n characters c.

```
HEADER(5, "CHARS");
int CHARS = COLON(4, SWAP, DOLIT, 0, MAX);
FOR(0);
AFT(2, DUPP, EMIT);
THEN(0);
NEXT(2, DROP, EXITT);
```

SPACES ( +n -- ) outputs n blank space characters.

```
HEADER(6, "SPACES");
int SPACS = COLON(3, BLANK, CHARS, EXITT);
```

TYPE ( b u -- ) outputs n characters from a string in memory. Non ASCII characters are replaced by a underscore character.

```
HEADER(4, "TYPE");
int TYPES = COLON(0);
FOR(0);
AFT(3, COUNT, TCHAR, EMIT);
THEN(0);
NEXT(2, DROP, EXITT);
```

CR ( -- ) outputs a carriage-return and a line-feed.

```
HEADER(2, "CR");
int CR = COLON(7, DOLIT, 10, DOLIT, 13, EMIT, EMIT, EXITT);
```

do\$ ( -- \$adr ) retrieves the address of a string stored as the second item on the return stack. do\$ is a bit difficult to understand, because the starting address of the following string is the second item on the return stack. This address is pushed on the data stack so that the string can be accessed. This address must be changed so that the address interpreter will return to the token right after the compiled string. This address will allow the address interpreter to skip over the string literal and continue to execute the token list as intended. Both \$"| and ."| use the word do\$,

```
HEADER(3, "do$");
int DOSTR = COLON(10, RFROM, RAT, RFROM, COUNT, PLUS, ALIGN, TOR, SWAP, TOR,
EXITT);
```

`$" | ( -- a )` pushes the address of the following string on stack. Other words can use this address to access data stored in this string. The string is a counted string. Its first byte is a byte count.

```
HEADER(3, "$\" |");
int STRQP = COLON(2, DOSTR, EXITT);
```

`" | ( -- )` displays the following string on stack. This is a very convenient way to send helping messages to you at run time.

```
HEADER(3, ".\" |");
DOTQP = COLON(4, DOSTR, COUNT, TYPES, EXITT);
```

`.R ( u +n -- )` displays 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.

```
HEADER(2, ".R");
int DOTR = COLON(8, TOR, STRR, RFROM, OVER, SUBBB, SPACS, TYPES, EXITT);
```

`U.R ( u +n -- )` displays an unsigned integer `n` right-justified in a field of `+n` characters.

```
HEADER(3, "U.R");
int UDOTR = COLON(10, TOR, BDIGS, DIGS, EDIGS, RFROM, OVER, SUBBB, SPACS, TYPES, EXITT);
```

`U. ( u -- )` displays an unsigned integer `u` in free format, followed by a space.

```
HEADER(2, "U.");
int UDOT = COLON(6, BDIGS, DIGS, EDIGS, SPACE, TYPES, EXITT);
```

`. (dot) ( n -- )` displays a signed integer `n` in free format, followed by a space.

```
HEADER(1, ".");
int DOT = COLON(5, BASE, AT, DOLIT, OXA, XORR);
IF(2, UDOT, EXITT);
THEN(4, STRR, SPACE, TYPES, EXITT);
```

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

```
HEADER(1, "?");
int QUEST = COLON(3, AT, DOT, EXITT);
```

## Parsing

Parsing is always thought of as a very advanced topic in computer sciences. However, because Forth uses very simple syntax rules, parsing is easy. Forth source code consists of words, which are ASCII strings separated by spaces and other white space characters like tabs, carriage returns, and line feeds. The text interpreter scans the source code, isolates words and interprets them in sequence. After a word is parsed out of the input text stream, the text interpreter will 'interpret' it--execute it if it is a token, compile it if the text interpreter is in the compiling mode, and convert it to a number if the word is not a Forth token.

**PARSE** scans the source string in the terminal input buffer from where **>IN** points to till the end of the buffer, for a word delimited by character **c**. It returns the address and length of the word parsed out. **PARSE** calls **(parse)** to do the dirty work.

// Parser

**(parse) ( b1 u1 c --b2 u2 n )** From the source string starting at **b1** and of **u1** characters long, parse out the first word delimited by character **c**. Return the address **b2** and length **u2** of the word just parsed out and the difference **n** between **b1** and **b2**. Leading delimiters are skipped over. **(parse)** is used by **PARSE**.

```

HEADER(7, "(parse)");
int PARS = COLON(5, TEMP, CSTOR, OVER, TOR, DUPP);
IF(5, ONEM, TEMP, CAT, BLANK, EQUAL);
IF(0);
FOR(6, BLANK, OVER, CAT, SUBBB, ZLESS, INVER);
WHILE(1, ONEP);
NEXT(6, RFROM, DROP, DOLIT, 0, DUPP, EXITT);
THEN(1, RFROM);
THEN(2, OVER, SWAP);
FOR(9, TEMP, CAT, OVER, CAT, SUBBB, TEMP, CAT, BLANK, EQUAL);
IF(1, ZLESS);
THEN(0);
WHILE(1, ONEP);
NEXT(2, DUPP, TOR);
ELSE(5, RFROM, DROP, DUPP, ONEP, TOR);
THEN(6, OVER, SUBBB, RFROM, RFROM, SUBBB, EXITT);
THEN(4, OVER, RFROM, SUBBB, EXITT);

```

**PACK\$ ( b u a -- a )** copies a source string (**b u**) to target address at **a**. The target string is null filled to the cell boundary. The target address **a** is returned.

```

HEADER(5, "PACK$");
int PACKS = COLON(18, DUPP, TOR, DDUP, PLUS, DOLIT, 0xFFFFFFFFC, ANDD, DOLIT, 0,
SWAP, STORE, DDUP, CSTOR, ONEP, SWAP, CMOVEE, RFROM, EXITT);

```

**PARSE ( c -- b u ; <string> )** scans the source string in the terminal input buffer from where **>IN** points to till the end of the buffer, for a word delimited by character **c**. It returns the address and length of the word parsed out. **PARSE** calls **(parse)** to do the dirty work.

```

HEADER(5, "PARSE");
int PARSE = COLON(15, TOR, TIB, INN, AT, PLUS, NTIB, AT, INN, AT, SUBBB, RFROM,
PARS, INN, PSTOR, EXITT);

```

**TOKEN ( -- a ;; <string> )** parses the next word from the input buffer and copy the counted string to the top of the name dictionary. Return the address of this counted string.

```

HEADER(5, "TOKEN");
int TOKEN = COLON(9, BLANK, PARSE, DOLIT, 0x1F, MIN, HERE, CELLP, PACKS, EXITT);

```

WORD ( c -- a ; <string> ) parses out the next word delimited by the ASCII character c. Copy the word to the top of the code dictionary and return the address of this counted string.

```
HEADER(4, "WORD");
int WORDD = COLON(5, PARSE, HERE, CELLP, PACKS, EXITT);
```

## Dictionary Search

In Forth, word records are linked into a dictionary which can be searched to find valid words. A header contains four fields: a link field holding the name field address of the previous header, a name field holding the name as a counted string, a code field holding execution address of the word, and a parameter field holding data to be processed. The dictionary is a list linked through the link fields and the name fields. The basic searching function is performed by the word `find`. `find` scans the linked list to find a name which matches an input text string, and returns the code field address and the name field address of an executable token, if a match is found.

NAME> ( nfa - cfa) Return a code field address from the name field address of a word.

```
HEADER(5, "NAME>");
int NAMET = COLON(7, COUNT, DOLIT, 0x1F, ANDD, PLUS, ALIGN, EXITT);
```

SAME? ( a1 a2 n - a1 a2 f) Compare n/4 words in strings at a1 and a2. If the strings are the same, return a 0. If string at a1 is higher than that at a2, return a positive number; otherwise, return a negative number. `FIND` compares the 1<sup>st</sup> word input string and a name. If these two words are the same, `SAME?` is called to compare the rest of two strings

```
HEADER(5, "SAME?");
int SAMEQ = COLON(4, DOLIT, 0x1F, ANDD, CELLD);
FOR(0);
AFT(14, OVER, RAT, CELLS, PLUS, AT, UPPER, OVER, RAT, CELLS, PLUS, AT, UPPER,
SUBBB, QDUP);
IF(3, RFROM, DROP, EXITT);
THEN(0);
THEN(0);
NEXT(3, DOLIT, 0, EXITT);
```

`find` ( a va --cfa nfa, a F) searches the dictionary for a word. A counted string at a is the name of a token to be looked up in the dictionary. The last name field address of the dictionary is stored in location va. If the string is found, both the code field address and the name field address are returned. If the string is not the name a token, the string address and a false flag are returned.

```
HEADER(4, "find");
int FIND = COLON(10, SWAP, DUPP, AT, TEMP, STORE, DUPP, AT, TOR, CELLP, SWAP);
BEGIN(2, AT, DUPP);
IF(9, DUPP, AT, DOLIT, 0xFFFFF3F, ANDD, UPPER, RAT, UPPER, XORR);
IF(3, CELLP, DOLIT, 0xFFFFFFFF);
ELSE(4, CELLP, TEMP, AT, SAMEQ);
THEN(0);
ELSE(6, RFROM, DROP, SWAP, CELLM, SWAP, EXITT);
THEN(0);
```

```

WHILE(2, CELLM, CELLM);
REPEAT(9, RFROM, DROP, SWAP, DROP, CELLM, DUPP, NAMET, SWAP, EXITT);
HEADER(5, "NAME?");
int NAMEQ = COLON(3, CNTXT, FIND, EXITT);

```

## Terminal Input

The text interpreter interprets input text stream stored in the terminal input buffer. None of us can type perfectly. We have to allow mistyped characters and give us opportunities to back up and correct mistakes. To allow some minimal editing, we need three special words to deal with backspaces and carriage return thus received: ^H, TAP and KTAP. These words are hard to understand because they manipulate three addresses on data stack: bot is bottom of terminal buffer, eot is end of terminal buffer, and cur is current character pointer.

```
// Terminal Input
```

^H (bot eot cur -- bot eot cur) Process the back-space character. Erase the last character and decrement cur. If cur=bot, do nothing because you cannot backup beyond the beginning of the input buffer.

```

HEADER(2, "^H");
int HATH = COLON(6, TOR, OVER, RFROM, SWAP, OVER, XORR);
IF(9, DOLIT, 8, EMIT, ONEM, BLANK, EMIT, DOLIT, 8, EMIT);
THEN(1, EXITT);

```

TAP (bot eot cur c -- bot eot cur) Echo c to output device, store c in cur, and bump cur.

```

HEADER(3, "TAP");
int TAP = COLON(6, DUPP, EMIT, OVER, CSTOR, ONEP, EXITT);

```

kTAP (bot eot cur c -- bot eot cur) Process a character c in input buffer. bot is the starting address of the input buffer. eot is the end of the input buffer. cur is the current character pointer. Character c is normally stored into cur, which is increment by 1. In this case, cur is the same as eot. If c is a carriage-return, echo a space and make eot=cur. If c is a back-space, erase the last character and decrement cur.

```

HEADER(4, "kTAP");
int KTAP = COLON(9, DUPP, DOLIT, OXD, XORR, OVER, DOLIT, OXA, XORR, ANDD);
IF(3, DOLIT, 8, XORR);
IF(2, BLANK, TAP);
ELSE(1, HATH);
THEN(1, EXITT);
THEN(5, DROP, SWAP, DROP, DUPP, EXITT);

```

ACCEPT ( b u1 --b u2 ) Accept u1 characters to b. u2 returned is the actual number of characters received.

```

HEADER(6, "ACCEPT");
int ACCEP = COLON(3, OVER, PLUS, OVER);
BEGIN(2, DDUP, XORR);

```

```

WHILE(7, KEY, DUPP, BLANK, SUBBB, DOLIT, 0X5F, ULESS);
IF(1, TAP);
ELSE(1, KTAP);
THEN(0);
REPEAT(4, DROP, OVER, SUBBB, EXITT);

```

EXPECT ( b u1 -- ) accepts u1 characters to b. Number of characters accepted is stored in SPAN.

```

HEADER(6, "EXPECT");
int EXPEC = COLON(5, ACCEP, SPAN, STORE, DROP, EXITT);

```

QUERY is the word which accepts text input, up to 80 characters, from an input device and copies the text characters to the terminal input buffer. It also prepares the terminal input buffer for parsing by setting #TIB to the received character count and clearing >IN.

```

HEADER(5, "QUERY");
int QUERY = COLON(12, TIB, DOLIT, 0X50, ACCEP, NTIB, STORE, DROP, DOLIT, 0, INN,
STORE, EXITT);

```

## Text Interpreter

Text interpreter is the heart of Forth. It is like the operating system of a computer. It is the primary interface between you and a computer. Since Forth uses very simple syntax rules-- words are separated by spaces, the text interpreter is also very simple. It accepts a line of text you type on a terminal keyboard, parses out a word delimited by spaces, searches the token of this word in the dictionary and then executes it. The process is repeated until the line of 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 Forth, the text interpreter is encoded in the word QUIT. QUIT contains an infinite loop which repeats the QUERY and EVAL commands. QUERY accepts a line of text from the terminal and copies the text into the Terminal Input Buffer (TIB). EVAL interprets the text one word at a time till end of the line.

One of the unique features in Forth is its error handling mechanism. While EVAL is interpreting a line of text, it could encounter many error conditions: a word is not found in the dictionary and it is not a number, a compile-only word is accidentally executed interpretively, and the interpretive process may be interrupted by the words ABORT or abort". Wherever the error occurs, the text interpreter resets and starts over at ABORT.

```
// Text Interpreter
```

ABORT ( -- ) resets system and re-enters into the text interpreter loop EVAL. It actually executes EVAL stored in 'ABORT.

```

HEADER(5, "ABORT");
int ABORT = COLON(2, TABRT, ATEXE);

```

`abort` | ( `f --` ) A runtime string word compiled in front of a string of error message. If flag `f` is true, display the following string and jump to `ABORT`. If flag `f` is false, ignore the following string and continue executing tokens after the error message.

```
HEADER(6, "abort\");
ABORQP = COLON(0);
IF(4, DOSTR, COUNT, TYPES, ABORT);
THEN(3, DOSTR, DROP, EXITT);
```

`ERROR` ( `a --` ) displays an error message at `a` with a `?` mark, and `ABORT`.

```
HEADER(5, "ERROR");
int ERRORR = COLON(11, SPACE, COUNT, TYPES, DOLIT, 0x3F, EMIT, DOLIT, 0x1B, EMIT,
CR, ABORT);
```

`$INTERPRET` ( `a --` ) executes a word whose string address is on the stack. If the string is not a word, convert it to a number. If it is not a number, `ABORT`.

```
HEADER(10, "$INTERPRET");
int INTER = COLON(2, NAMEQ, QDUP);
IF(4, CAT, DOLIT, COMPO, ANDD);
ABORQ(" compile only");
int INTER0 = LABEL(2, EXECU, EXITT);
THEN(1, NUMBQ);
IF(1, EXITT);
ELSE(1, ERRORR);
THEN(0);
```

[ `(left-paren)` ( `--` ) activates the text interpreter by storing the execution address of `$INTERPRET` into the variable `'EVAL`, which is executed in `EVAL` while the text interpreter is in the interpretive mode.

```
HEADER(IMEDD + 1, "[");
int LBRAC = COLON(5, DOLIT, INTER, TEVAL, STORE, EXITT);
```

`.OK` ( `--` ) used to be a word which displays the familiar `'ok'` prompt after executing to the end of a line. In `ceForth_33`, it displays the top 4 elements on data stack so you can see what is happening on the stack. It is more informative than the plain `'ok'`, which only give you a warm and fuzzy feeling about the system. When text interpreter is in compiling mode, the display is suppressed.

```
HEADER(3, ".OK");
int DOTOK = COLON(6, CR, DOLIT, INTER, TEVAL, AT, EQUAL);
IF(14, TOR, TOR, TOR, DUPP, DOT, RFROM, DUPP, DOT, RFROM, DUPP, DOT, RFROM, DUPP,
DOT);
DOTQ(" ok>");
THEN(1, EXITT);
```

`EVAL` ( `--` ) has a loop which parses tokens from the input stream and invokes whatever is in `'EVAL` to process that token, either execute it with `$INTERPRET` or compile it with `$COMPILE`. It exits the loop when the input stream is exhausted.

```
HEADER(4, "EVAL");
int EVAL = COLON(0);
BEGIN(3, TOKEN, DUPP, AT);
WHILE(2, TEVAL, ATEXE);
REPEAT(3, DROP, DOTOK, EXITT);
```

QUIT ( -- ) is the operating system, or a shell, of the Forth system. It is an infinite loop Forth will not leave. It uses QUERY to accept a line of text from the terminal and then let EVAL parse out the tokens and execute them. After a line is processed, it displays the top of data stack and wait for the next line of text. When an error occurred during execution, it displays the command which caused the error with an error message. After the error is reported, it re-initializes the system by jumping to ABORT. 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.

```
HEADER(4, "QUIT");
int QUITT = COLON(5, DOLIT, 0X100, TTIB, STORE, LBRAC);
BEGIN(2, QUERY, EVAL);
AGAIN(0);
```

## Chapter 9. Colon Compiler

After wading through the text interpreter, the Forth compiler will be an easy piece of cake, because the compiler uses almost all the modules used by the text interpreter. What the compiler does, over and above the text interpreter, is to build various structures required by the new words you add to the .data segment. Here is a list of these structures:

- Colon words
- Constants
- Variables
- Integer literals
- String literals
- Address literals and control structures

A special concept of immediate words is difficult to grasp at first. It is required in the compiler because of the needs in building different data and control structures in a colon word. To understand the Forth compiler fully, you have to be able to differentiate and relate the actions taken during compile time and actions taken during run time. Once these concepts are clear, the whole Forth system will become transparent.

The Forth compiler is the twin brother of the Forth text interpreter. They share many common properties and use lots of common code. In Forth, the implementation of the compiler clearly reflects this special duality. Two interesting words `[` and `]` cause the text interpreter to switch back and forth between the compiler mode and interpreter mode.

Since `'EVAL @EXECUTE` is used in `EVAL` to process a token parsed out of a line of text, the contents in `'EVAL` determines the behavior of the text interpreter. If `$INTERPRET` is stored in `'EVAL`, as `[` does, tokens are executed or interpreted. If we invoke `]` to store `$COMPILE` into `'EVAL`, the token will not be executed, but compiled to the top of dictionary. This is exactly the behavior desired by the colon word compiler in building a list of tokens in the parameter field of a new colon word in dictionary.

`$COMPILE` normally adds a token to the dictionary. However, there are two exceptions it must handle. If a string parsed out of the input stream is not a word in the dictionary, the string will be converted to a number. If the string can be converted to an integer, the integer is then compiled into the dictionary as an integer literal, which consists of a special token `DOLIT` followed by the integer. The other exception is that a token found in the dictionary could be an immediate word, which must be executed immediately, not compiled to the dictionary. Immediate words are used to compile structures in colon words.

```
// Colon Word Compiler
```

```
, (comma) ( w -- )
```

adds the execution address of a token on the top of the data stack to the code dictionary, and thus compiles a token to the growing token list of the word currently under construction.

```

HEADER(1, ",");
int COMMA = COLON(7, HERE, DUPP, CELLP, CP, STORE, STORE, EXITT);

```

**LITERAL ( n -- )** compiles an integer literal to the current compound word under construction. The integer literal is taken from the data stack, and is preceded by the token **DOLIT**. When this compound word is executed, **DOLIT** will extract the integer from the token list and push it back on the data stack. **LITERAL** compiles an address literal if the compiled integer happens to be an execution address of a token. The address will be pushed on the data stack at the run time by **DOLIT**.

```

HEADER(1MEDD + 7, "LITERAL");
int LITER = COLON(5, DOLIT, DOLIT, COMMA, COMMA, EXITT);

```

**ALLOT ( n -- )** allocates n bytes of memory on the top of the dictionary. Once allocated, the compiler will not touch the memory locations. It is possible to allocate and initialize this array using the word', (comma) '.

```

HEADER(5, "ALLOT");
int ALLOT = COLON(4, ALIGN, CP, PSTOR, EXITT);

```

**\$, " ( -- )** extracts next word delimited by double quote. Compile it as a string literal.

```

HEADER(3, "$,\"");
int STRCQ = COLON(9, DOLIT, 0x22, WORDD, COUNT, PLUS, ALIGN, CP, STORE, EXITT);

```

**?UNIQUE ( a -- a )** is used to display a warning message to show that the name of a new word already existing in dictionary. Forth does not mind your reusing the same name for different words. However, giving many words the same name is a potential cause of problems in maintaining software projects. It is to be avoided if possible and **?UNIQUE** reminds you of it.

```

HEADER(7, "?UNIQUE");
int UNIQ = COLON(3, DUPP, NAMEQ, QDUP);
IF(6, COUNT, DOLIT, 0x1F, ANDD, SPACE, TYPES);
DOTQ(" reDef");
THEN(2, DROP, EXITT);

```

**\$, n ( a -- )** builds a new name field in dictionary using the name already moved to the top of dictionary by **PACK\$**. It pads the link field with the address stored in **LAST**. A new token can now be built in the code dictionary.

```

HEADER(3, "$,n");
int SNAME = COLON(2, DUPP, AT);
IF(14, UNIQ, DUPP, NAMEQ, CP, STORE, DUPP, LAST, STORE, CELLM, CNTXT, AT, SWAP,
STORE, EXITT);
THEN(1, ERRORR);

```

**' (tick) ( -- cfa )** searches the next word in the input stream for a token in the dictionary. It returns the code field address of the token if successful. Otherwise, it aborts and displays an error message.

```

HEADER(1, "'");
int TICK = COLON(2, TOKEN, NAMEQ);
IF(1, EXITT);

```

```
THEN(1, ERRORR);
```

[COMPILE] ( -- ; <string> ) acts similarly, except that it compiles the next word immediately. It causes the following word to be compiled, even if the following word is usually an immediate word which would otherwise be executed.

```
HEADER(IMEDD + 9, "[COMPILE]");
int BCOMP = COLON(3, TICK, COMMA, EXITT);
```

COMPILE ( -- ) is used in a compound word. It causes the next token after COMPILE to be added to the top of the code dictionary. It therefore forces the compilation of a token at the run time.

```
HEADER(7, "COMPILE");
int COMPI = COLON(7, RFROM, DUPP, AT, COMMA, CELLP, TOR, EXITT);
```

\$COMPILE ( a -- ) builds the body of a new compound word. A complete compound word also requires a header in the name dictionary, and its code field must start with a dolist, byte code. These extra works are performed by : (colon) . Compound words are the most prevailing type of words in eForth. In addition, eForth has a few other defining words which create other types of new words in the dictionary.

```
HEADER(8, "$COMPILE");
int SCOMP = COLON(2, NAMEQ, QDUP);
IF(4, AT, DOLIT, IMEDD, ANDD);
IF(1, EXECU);
ELSE(1, COMMA);
THEN(1, EXITT);
THEN(1, NUMBQ);
IF(2, LITER, EXITT);
THEN(1, ERRORR);
```

OVERT ( -- ) links a new word to the dictionary and thus makes it available for dictionary searches.

```
HEADER(5, "OVERT");
int OVERT = COLON(5, LAST, AT, CNTXT, STORE, EXITT);
```

] (right paren) ( -- ) turns the interpreter to a compiler.

```
HEADER(1, "]");
int RBRAC = COLON(5, DOLIT, SCOMP, TEVAL, STORE, EXITT);
```

: (colon) ( -- ; <string> ) creates a new header and start a new compound word. It takes the following string in the input stream to be the name of the new compound word, by building a new header with this name in the name dictionary. It then compiles a dolist, byte code at the beginning of the code field in the code dictionary. Now, the code dictionary is ready to accept a token list. ] (right paren) is now invoked to turn the text interpreter into a compiler, which will compile the following words in the input stream to a token list in the code dictionary. The new compound word is terminated by ;, which compiles an EXIT to terminate the token list, and executes [ (left paren) to turn the compiler back to text interpreter.

```
HEADER(1, ":");
```

```
int COLN = COLON(7, TOKEN, SNAME, RBRAC, DOLIT, 0x6, COMMA, EXITT);
```

; (semi-colon) ( -- ) terminates a compound word. It compiles an EXIT to the end of the token list, links this new word to the dictionary, and then reactivates the interpreter.

```
HEADER(IMEDD + 1, ";");
```

```
int SEMIS = COLON(6, DOLIT, EXITT, COMMA, LBRAC, OVERT, EXITT);
```

## Debugging Tools

eForth is a very small system and only a very small set of tools are provided in the system. Nevertheless, this set of tools is powerful enough to help you debug new words you add to the system. They are also very interesting programming examples on how to use the words in eForth to build applications.

Generally, the tools presents the information stored in different parts of the memory in the appropriate format to let the use inspect the results as he executes words in the eForth system and words he defined himself. The tools are memory dump and dictionary dump.

```
// Debugging Tools
```

dm+ ( b u - b+u ) dumps u bytes starting at address b to the terminal. It dumps 8 words. A line begins with the address of the first byte, followed by 8 words shown in hex, and the same data shown in ASCII. Non-printable characters by replaced by underscores. A new address b+u is returned to dump the next line.

```
HEADER(3, "dm+");
```

```
int DMP = COLON(4, OVER, DOLIT, 6, UDOTR);
```

```
FOR(0);
```

```
AFT(6, DUPP, AT, DOLIT, 9, UDOTR, CELLP);
```

```
THEN(0);
```

```
NEXT(1, EXITT);
```

DUMP ( b u -- ) dumps u bytes starting at address b to the terminal. It dumps 8 words to a line. A line begins with the address of the first byte, followed by 8 words shown in hex. At the end of a line are the 32 bytes shown in ASCII code.

```
HEADER(4, "DUMP");
```

```
int DUMP = COLON(10, BASE, AT, TOR, HEXX, DOLIT, 0x1F, PLUS, DOLIT, 0x20, SLASH);
```

```
FOR(0);
```

```
AFT(10, CR, DOLIT, 8, DDUP, DMP, TOR, SPACE, CELLS, TYPES, RFROM);
```

```
THEN(0);
```

```
NEXT(5, DROP, RFROM, BASE, STORE, EXITT);
```

>NAME ( cfa -- nfa | F ) finds the name field address of a token from its code field address. If the token does not exist in the dictionary, it returns a false flag. >NAME is the mirror image of the word NAME>, which returns the code field address of a token from its name field address. Since the code field is right after the name field, whose length is stored in the lexicon

byte, NAME> is trivial. >NAME is more complicated because we have to search the dictionary to ascertain the name field address.

```
HEADER(5, ">NAME");
int TNAME = COLON(1, CNTXT);
BEGIN(2, AT, DUPP);
WHILE(3, DDUP, NAMET, XORR);
IF(1, ONEM);
ELSE(3, SWAP, DROP, EXITT);
THEN(0);
REPEAT(3, SWAP, DROP, EXITT);
```

.ID ( a -- ) displays the name of a token, given its name field address. It also replaces non-printable characters in a name by under-scores.

```
HEADER(3, ".ID");
int DOTID = COLON(7, COUNT, DOLIT, 0x1F, ANDD, TYPES, SPACE, EXITT);
```

WORDS ( -- ) displays all the names in the dictionary. The order of words is reversed from the compiled order. The last defined word is shown first.

```
HEADER(5, "WORDS");
int WORDS = COLON(6, CR, CNTXT, DOLIT, 0, TEMP, STORE);
BEGIN(2, AT, QDUP);
WHILE(9, DUPP, SPACE, DOTID, CELLM, TEMP, AT, DOLIT, 0xA, LESS);
IF(4, DOLIT, 1, TEMP, PSTOR);
ELSE(5, CR, DOLIT, 0, TEMP, STORE);
THEN(0);
REPEAT(1, EXITT);
```

FORGET ( -- , <name>) searches the dictionary for a name following it. If it is a valid word, trim dictionary below this word. Display an error message if it is not a valid word.

```
HEADER(6, "FORGET");
int FORGT = COLON(3, TOKEN, NAMEQ, QDUP);
IF(12, CELLM, DUPP, CP, STORE, AT, DUPP, CNTXT, STORE, LAST, STORE, DROP, EXITT);
THEN(1, ERRORR);
```

COLD ( -- ) is a high level word executed upon power-up. It sends out sign-on message, and then falls into the text interpreter loop through QUIT.

```
HEADER(4, "COLD");
int COLD = COLON(1, CR);
DOTQ("eForth in C,Ver 2.3,2017 ");
int DOTQ1 = LABEL(2, CR, QUITT);
```

## Control Structures

A set of immediate words are defined in Forth to build control structures in colon words. The control structures used in Forth 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

This set of words is more powerful than the ones in figForth model because they do not do error checking and thus permit multiple entries into and exits from a control structure. However, it is not recommended that you overlap the control structures. In the learning stage of Forth language, it will do you good to remember that:

*Control structures can be nested, but not overlapped.*

A control structure contains one or more address literals, which causes execution to branch out of the normal linear sequence. Control structure words are immediate words which compile address literals and resolve branch addresses.

One should note that `BEGIN` and `THEN` do not compile any code. They execute during compilation to set up and to resolve branch addresses in address literals. `IF`, `ELSE`, `WHILE`, `UNTIL`, and `AGAIN` do compile address literals with `BRANCH` and `?BRANCH` tokens. To set up a counted loop, `FOR` compiles `>R` to begin the loop, and `NEXT` compiles a `DONXT` address literal to terminate the loop. There are many excellent examples using `COMPILE` and `[COMPILE]`, and they are worthy of your attention.

This macro assembler uses the return stack to resolve forward and backward reference. The stack picture thus refers to return stack, not the data stack used at run time. Upper case `A` means a pointer to an address literal to be filled with the correct branch address. Lower case `a` means the a branch address to be assemble.

In the stack comments of the following control structure words, I will use a lower case `'a'` to indicate a pointer to the address field in an address literal. The address field is initialized to 0, and will be filled later when the target address is known. I will use an upper case `'A'` to indicate a target address which will be used to fill the address field in an address literal.

// Structure Compiler

`THEN ( A -- )` terminates a conditional branch structure. It uses the address of next token to resolve the address literal at `A` left by `IF` or `ELSE`.

```
HEADER( IMEDD + 4, "THEN" );
int THENN = COLON( 4, HERE, SWAP, STORE, EXITT );
```

`FOR ( -- a )` starts a `FOR-NEXT` loop structure in a colon definition. It compiles `>R`, which pushes a loop count on return stack. It also leaves the address of next token on data stack, so that `NEXT` will compile a `DONEXT` address literal with the correct branch address.

```
HEADER( IMEDD + 3, "FOR" );
```

```
int FORR = COLON(4, COMPI, TOR, HERE, EXITT);
```

**BEGIN ( -- a )** starts an infinite or indefinite loop structure. It does not compile anything, but leave the current token address on data stack to resolve address literals compiled later.

```
HEADER(IMEDD + 5, "BEGIN");  
int BEGIN = COLON(2, HERE, EXITT);
```

**NEXT ( a -- )** Terminate a FOR-NEXT loop structure, by compiling a DONEXT address literal, branch back to the address A on data stack.

```
HEADER(IMEDD + 4, "NEXT");  
int NEXT = COLON(4, COMPI, DONXT, COMMA, EXITT);
```

**UNTIL ( a -- )** terminate a BEGIN-UNTIL indefinite loop structure. It compiles a QBRANCH address literal using the address on data stack.

```
HEADER(IMEDD + 5, "UNTIL");  
int UNTIL = COLON(4, COMPI, QBRAN, COMMA, EXITT);
```

**AGAIN ( a -- )** terminate a BEGIN-AGAIN infinite loop structure. . It compiles a BRANCH address literal using the address on data stack.

```
HEADER(IMEDD + 5, "AGAIN");  
int AGAIN = COLON(4, COMPI, BRAN, COMMA, EXITT);
```

**IF ( -- A )** starts a conditional branch structure. It compiles a QBRANCH address literal, with a 0 in the address field. It leaves the address of this address field on data stack. This address will later be resolved by ELSE or THEN in closing the true clause in the branch structure.

```
HEADER(IMEDD + 2, "IF");  
int IFF = COLON(7, COMPI, QBRAN, HERE, DOLIT, 0, COMMA, EXITT);
```

**AHEAD ( -- A )** starts a forward branch structure. It compiles a BRANCH address literal, with a 0 in the address field. It leaves the address of this address field on data stack. This address will later be resolved when the branch structure is closed.

```
HEADER(IMEDD + 5, "AHEAD");  
int AHEAD = COLON(7, COMPI, BRAN, HERE, DOLIT, 0, COMMA, EXITT);
```

**REPEAT ( A a -- )** terminates a BEGIN-WHILE-REPEAT indefinite loop structure. It compiles a BRANCH address literal with address a left by BEGIN, and uses the address of next token to resolve the address literal at A.

```
HEADER(IMEDD + 6, "REPEAT");  
int REPEA = COLON(3, AGAIN, THENN, EXITT);
```

**AFT ( a -- a A )** jumps to THEN in a FOR-AFT-THEN-NEXT loop the first time through. It compiles a BRANCH address literal and leaves its address field on stack. This address will be resolved by THEN. It also replaces address A left by FOR by the address of next token so that NEXT will compile a DONEXT address literal to jump back here at run time.

```
HEADER(IMEDD + 3, "AFT");  
int AFT = COLON(5, DROP, AHEAD, HERE, SWAP, EXITT);
```

ELSE ( A -- A ) starts the false clause in an IF-ELSE-THEN structure. It compiles a BRANCH address literal. It uses the current token address to resolve the branch address in A, and replace A with the address of its address literal.

```
HEADER( IMEDD + 4, "ELSE" );
int ELSEE = COLON( 4, AHEAD, SWAP, THENN, EXITT );
```

WHEN ( a - a A a ) compiles a QBRANCH address literal. The address a of this address literal is copied over A.

```
HEADER( IMEDD + 4, "WHEN" );
int WHEN = COLON( 3, IFF, OVER, EXITT );
```

WHILE ( a -- A a ) compiles a QBRANCH address literal in a BEGIN-WHILE-REPEAT loop. The address A of this address literal is swapped with address a left by BEGIN, so that REPEAT will resolve all loose ends and build the loop structure correctly.

```
HEADER( IMEDD + 5, "WHILE" );
int WHILEE = COLON( 3, IFF, SWAP, EXITT );
```

## String Literals

Character strings are very important data structures for the program to communicate with you. Error messages, appropriate warnings and suggestions must be displayed to help you using the system in a friendly way. Character strings are compiled in the colon words as string literals. Each string literal consists of a string token which will use the compiled string to do things, and a counted string. The first byte in a counted string is the length of the string. Thus a string may have 0 to 255 characters in it.

ABORT" compiles an error message. This error message is displayed if top item on the stack is non-zero. The rest of the words in the word is skipped and Forth resets to ABORT. If top of stack is 0, ABORT" skips over the error message and continues executing the following token list.

```
HEADER( IMEDD + 6, "ABORT\" );
int ABRTQ = COLON( 6, DOLIT, ABORQP, HERE, STORE, STRCQ, EXITT );
```

\$" ( -- ; <string> ) compiles a character string. When it is executed, only the address of the string is left on the data stack. You will use this address to access the string and individual characters in the string as a string array.

```
HEADER( IMEDD + 2, "$\" );
int STRQ = COLON( 6, DOLIT, STRQP, HERE, STORE, STRCQ, EXITT );
```

." (dot-quot) ( -- ; <string> ) compiles a character string which will be displayed when the word containing it is executed in the runtime. This is the best way to present messages to the user.

```
HEADER( IMEDD + 2, ".\" );
int DOTQQ = COLON( 6, DOLIT, DOTQP, HERE, STORE, STRCQ, EXITT );
```

## Defining Words

The concept of defining word is a very unique feature of Forth, in that it allows you to define new classes of words which can make specific use of data stored in their parameter fields. Each class of words share the same interpreter encoded in its code field.

In `ceForth_33`, I provide the following defining words: `,` `CODE`, `CREATE`, `CONSTANT` and `VARIABLE`. `CREATE` and `VARIABLE` use the same inner interpreter `DOVAR`, and `CONSTANT` uses `DOCON`. `CONSTANT` and `VARIABLE` allocate only 4 bytes for their parameter fields. `CRATE`, however, let you specific the size of parameter field.

`CODE ( -- ; <string> )` creates a word header, ready to accept byte code for a new primitive word. Without a byte code assembler, you can use the word `,` (comma) to add words with byte code in them.

```
HEADER(4, "CODE");
int CODE = COLON(4, TOKEN, SNAME, OVERT, EXITT);
```

`CREATE ( -- ; <string> )` creates a new array without allocating memory. Memory is allocated using `ALLLOT`.

```
HEADER(6, "CREATE");
int CREAT = COLON(5, CODE, DOLIT, 0x203D, COMMA, EXITT);
```

`VARIABLE ( -- ; <string> )` creates a new variable, initialized to 0.

```
HEADER(8, "VARIABLE");
int VARIA = COLON(5, CREAT, DOLIT, 0, COMMA, EXITT);
```

`CONSTANT ( n -- ; <string> )` creates a new constant, initialized to the value on top of stack.

```
HEADER(8, "CONSTANT");
int CONST = COLON(6, CODE, DOLIT, 0x2004, COMMA, COMMA, EXITT);
```

## Comments

Comments are strings ignored by the interpreter and the compiler. They serve as reminders to ourself about the code we wrote.

`.( (dot-paren) ( -- ; <string> )` types the following string till the next `)`. It is used `CONSTANT ( n -- ; <string> )` creates a new constant, initialized to the value on top of stack.

```
HEADER(IMEDD + 2, ".(");
int DOTPR = COLON(5, DOLIT, 0x29, PARSE, TYPES, EXITT);
```

`\ (back-slash) ( -- ; <string> )` ignores all characters till end of input buffer. It is used to insert comment lines in text.

```
HEADER(IMEDD + 1, "\\");
int BKSLA = COLON(5, DOLIT, 0xA, WORDD, DROP, EXITT);
```

( (paren) ( -- ; <string> ) ignores the following string till the next ). It is used to place comments in source text.

```
HEADER(IMEDD + 1, "(");
int PAREN = COLON(5, DOLIT, 0x29, PARSE, DDROP, EXITT);
```

## Lexicon Bits

Remember bits 6 and 7 of the length byte in a name field? They are called lexicon bits, which request special treatment by Forth interpreter and Forth compiler. Bit 7 is called immediate bit, and it forces Forth compiler to execute this word instead of compiling its token into the dictionary. All Forth words building control structures are immediate words. Bit 6 is called compile-only bit. Many Forth words are dangerous. They may crash the system if executed by Forth interpreter. These words are marked compile-only, and they can only be used by Forth compiler.

COMPILE-ONLY ( -- ) sets the compile-only lexicon bit in the name field of the new word just compiled. When the interpreter encounters a word with this bit set, it will not execute this word, but spit out an error message. This bit prevents structure words to be executed accidentally outside of a compound word.

```
HEADER(12, "COMPILE-ONLY");
int ONLY = COLON(6, DOLIT, 0x40, LAST, AT, PSTOR, EXITT);
```

IMMEDIATE ( -- ) sets the immediate lexicon bit in the name field of the new word just compiled. When the compiler encounters a word with this bit set, it will not compile this word into the token list under construction, but execute the token immediately. This bit allows structure words to build special structures in a compound word, and to process special conditions when the compiler is running.

```
HEADER(9, "IMMEDIATE");
int IMMED = COLON(6, DOLIT, 0x80, LAST, AT, PSTOR, EXITT);
int ENDD = P;
```

## Checking Macro Assembler

You might have noticed that all the macro calls are placed inside the `main()` loop, and they are executed when `main()` starts. The Forth dictionary is build at runtime, not compiled into the `data[]` array. The macros have `printf()` statements which are commented out. If you like to see what the macro assembler assembles into the `data[]` array, un-comment some of these `printf()` statements. I used these `printf()` statements to verify that the macros behave correctly.

In the release version of `ceForth_33`, I only asked `HEADER()` to print out names and cfa's of words it assembles. After Forth dictionary is completed, its size is printed for your information. It also prints out the return stack pointer, which has to be 0 if all the control structures are constructed properly.

There is an option to dump the contents in the dictionary in Intel-Dump-like format, with checksums calculated for every 16 bytes of data. I used the checksums to verify that this dictionary is identical to the dictionary I used in ceForth\_23 system. It is a good exercise for you to read this dictionary dump to see the records of Forth words, and the fields in these word records.

```
// Boot Up
```

Print sized of Forth dictionary, and the return stack pointer to verify that the macro assembler works properly.

```
printf("\n\nIP=%X R-stack=%X", P, (popR << 2));
```

Set up the Reset Vector at memory location 0 to DOLST COLD.

```
P = 0;
int RESET = LABEL(2, 6, COLD);
```

Set up the user variables at memory location 0x90 to 0xAC to make Forth interpreter to work correctly.

```
P = 0x90;
int USER = LABEL(8, 0x100, 0x10, IMMED - 12, ENDD, IMMED - 12, INTER, QUITT, 0);
```

Optionally dump the dictionary in Intel Dump Format to verify its contents. Unfortunately, Windows console is not big enough to show the entire dump. Do this to see the second half of the dictionary. To see the first half, limit `len` to 0x100.

```
// dump dictionary
//P = 0;
//for (len = 0; len < 0x200; len++) { CheckSum(); }
```

Now, initialized all the most important registers in VFM, and the Finite State Machine brings up ceForth.

```
P = 0;
WP = 4;
IP = 0;
S = 0;
R = 0;
top = 0;
printf("\nceForth v3.3, 01jul19cht\n");
while (TRUE) {
    primitives[(unsigned char)cData[P++]>();
}
}
```

## Finite State Machine

Originally, ceForth was designed to emulate a 32-bit Forth microcontroller eP32, which read and writes only 32-bit words. A Finite State Machine (FSM) executes 8-bit machine instructions stored in 32-bit words. In State0, FSM fetches a 32-bit word from memory. In

state1-4, it executes 4 byte codes in this word. CALL and JMP instructions only execute in State1, and force FSM to State 0, fetching next program word from memory. RET may execute in any of State1-4, and then forces FSM to State 0.

Implement ceForth on a byte oriented machine, FSM was greatly simplified since VFM can fetch and execute consecutive bytes. In C, FSM can be written simply as:

```
primitives[cData[P++]]();
```

This FSM implies two states. In State0 a byte code is fetched from memory, and in State1, the byte code is executed. P is post-incremented, which can be done in either State0 or State1. However, it is useful to keep the concept of FSM, as the Virtual Forth Machine must first fetch a byte code and then execute it.

An interesting question I used to ask myself was that is it possible for the FSM to fetch a non-existing byte code in the virtual memory, outside of the code fields of existing primitive words? The answer is Never, Ever! I set up a trap to catch FSM executing an invalid byte code, and had never caught anything. When the Forth dictionary is constructed correctly, FSM will always fetch and execute byte code in some code fields. Only very experienced Forth programmers or a complete idiot can crash a Forth system by forcing its FSM to go outside a code field.

We have gone through the ceforth\_33 system in its entirety. It is in an cpp file written completely in C. We started with a Virtual Forth Machine with 64 byte code. Using these byte code, we assemble 80 primitive words. With these primitive words, we assembler or compile a complete Forth language interpreter and compiler, in 110 colon words. It is a complete and unambiguous specification of Forth as a programming language, and as an operating system. Since it is written in C, it can be easily implemented on computers and microcontrollers with a standard C compiler.

## Postlude

It had been 30 years since I wrote the first eForth for 8086 processor in 1990. It was 10 years since I write ceForth v1.0. In all these years, I have thought that the eForth Model is a good model useful for all different processors and microcontrollers, and for all different applications. It is a very simple model for anybody who like to learn Forth and to use it for their own applications.

Here, let me summarize lessons I have learnt in implementing Forth in C, over the last 10 years:

- Forth written in C has value, in programming modern advanced microcontrollers.
- It emulates a 32-bit microcontroller eP32 faithfully.
- Forth dictionary is stored in a virtual memory array to bypass the cardinal C restriction that you cannot write into code segment and that you cannot execute code in data segments.
- Circular buffers are ideal for return stack and data stack. They do not overflow or underflow. They require no maintenance.
- A Virtual Forth Machine could be implemented with a set of byte code.
- The Finite State Machine executing VFM byte code could be reduced to a single line of C code: `while TRUE {primitives[cData[P++]]() ;}`
- A one-pass macro assembler could be used to assemble the Forth dictionary.
- Forth can be easily configured as a module callable by large applications or OS.

I loved to recite a very short poem by a Tang poet name Ja Dao (779~843 AD).

### 剑客 贾岛

### Swordsman by Ja Dao

十年磨一剑，  
霜刃未曾试。  
今日把示君，  
谁有不平事？

I polish this sword for ten years.  
The shining blade has never been tested.  
Today I show it to you.  
Is there any injustice to avenge?



I have polished this ceForth system for 10 years. I am very happy with it, in that the cumbersome F# metacompiler was eliminated. I hope you will like it and make use of it. I wish you good luck.