ceForth_23

Preface--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, an expert C programmer, assured me that it was possible to write Forth in C, and showed me code fragments 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 commands, very simple to write in assembly of a particular microcontroller, or in C, which was designed for an idealized, general purpose CPU. My problems were in the construction of a Forth dictionary. Forth dictionary is a linked list of Forth commands, 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 source 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 C compiler.

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

Embedding eForth 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 commands in a data segment. I accept the limitation that no new pseudo instruction will be added to the baseline VFM, while new compound commands can be added to the Forth dictionary freely.

The design of an eForth system can now be separated into two independent tasks: building a VFM machine targeting to various microcontrollers, including C, and building an eForth dictionary. You can use independent tools which are best suited for the particular task.
I chose F# to build the eForth 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 kits 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 commands 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 bootload section which was used 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 eForth dictionary in the RAM memory. It worked. However, you had only 1.5KB of RAM memory left over for new Forth commands, and you could not save these new commands 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 single day. There were only very few changes to fit a VFM into ESP8266, and the eForth dictionary required no change at all. It was all because of the portability in C code. It generally took me two weeks to port eForth 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 do not understand the Tensilica L106 chip inside ESP8266 at all, and I do 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 you will find some use of it. Several important improvements were implemented:
It was moved to C++ under Microsoft Visual Studio Community 2017, so that you can compile and test it on a modern Windows PC.
A much simpler byte code sequencer replaced the finite state machine running VFM.
The stacks were changed to 256 level circular buffers, so they would never overflow or underflow.

Chapter 1. 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.

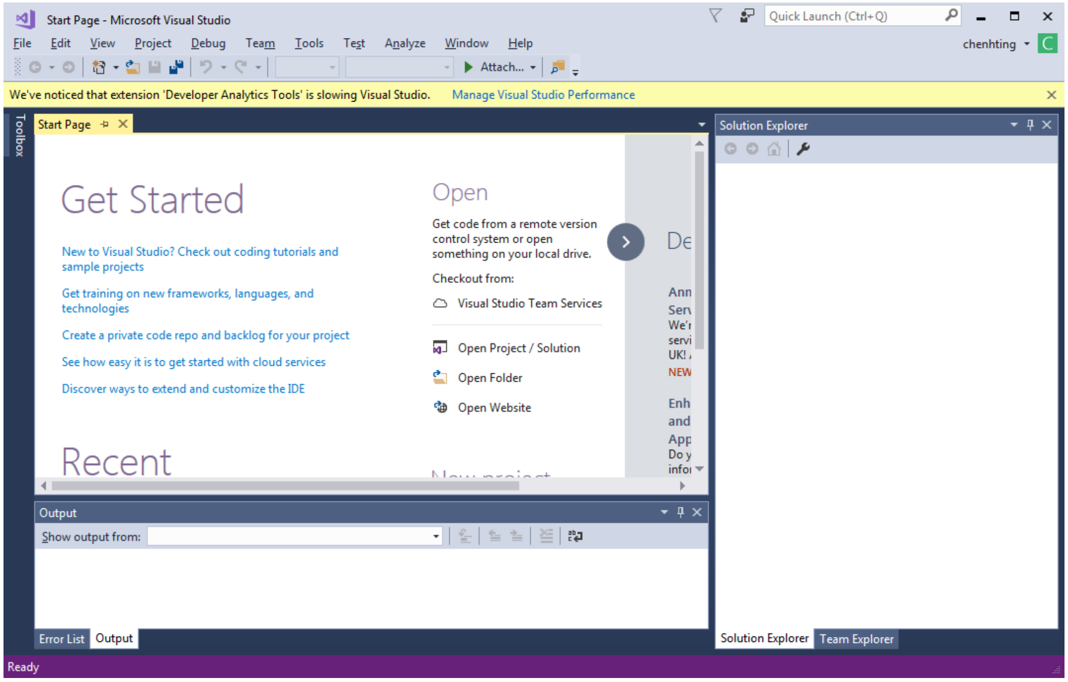
ceForth 1.0 and 1.1 were developed with gcc on cygwin. Cygwin was a reduced Linux running on PC, but it was a foreign system to Windows. I had totally forgotten how to compile and run it. Time to move on with Visual Studio.

Install Visual Studio Community 2017

`ceForth_23.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_23.cpp` to it and get it running.

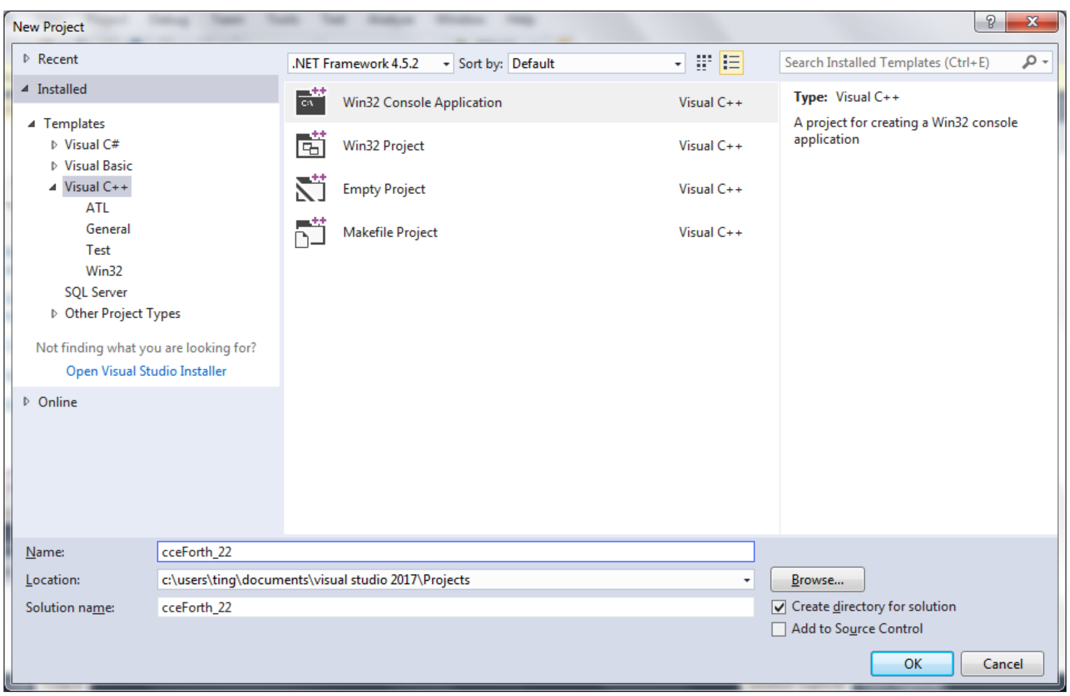
Download Visual Studio Community 2017 or the latest version from www.microsoft.com and install it on your PC. Open Visual Studio, and you will see its logo:

After a while, you will see its start page:

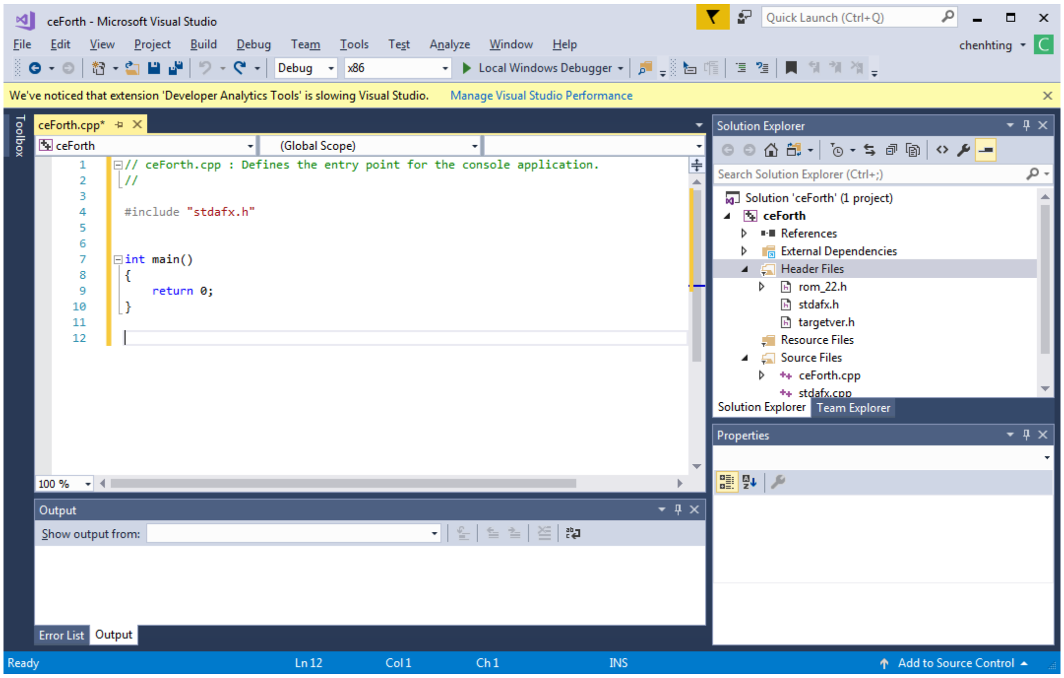


Click `File>New>Project`

In the Installed Panel, select `Templates>Visual C++`, and in the central panel, select `Win32 Console Application`. In the Name box at bottom, enter ceForth for your project, or a project name you prefer:



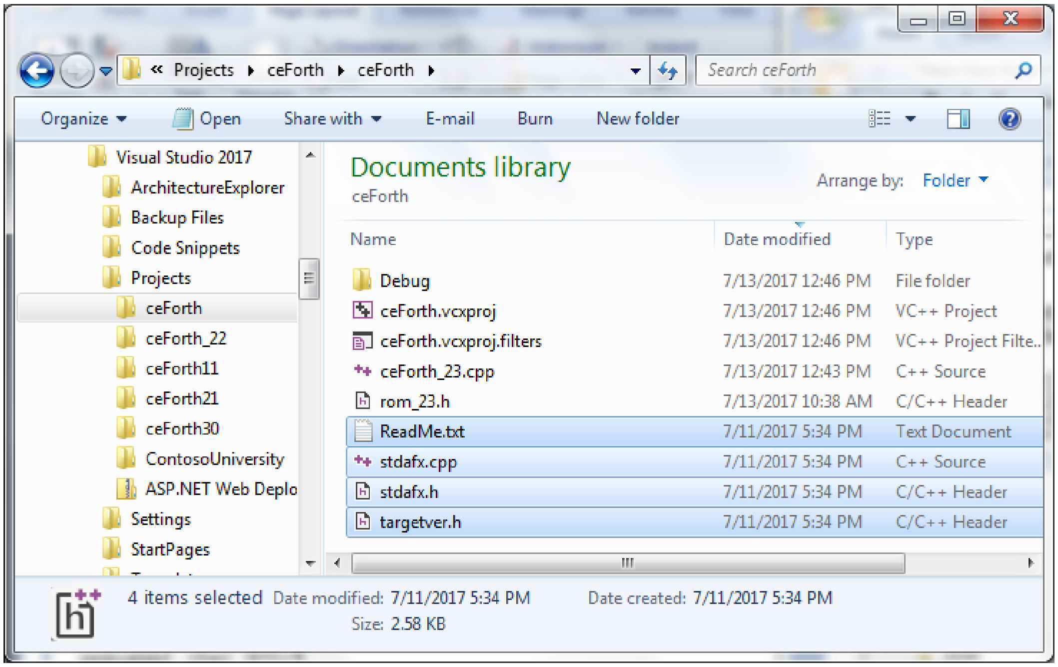
Click `OK` to create a new project. Watch the location of you new project in the Location box. Mine is in `c:\users\ting\documents\visual studio 2017\projects`. Yours surely will be different.



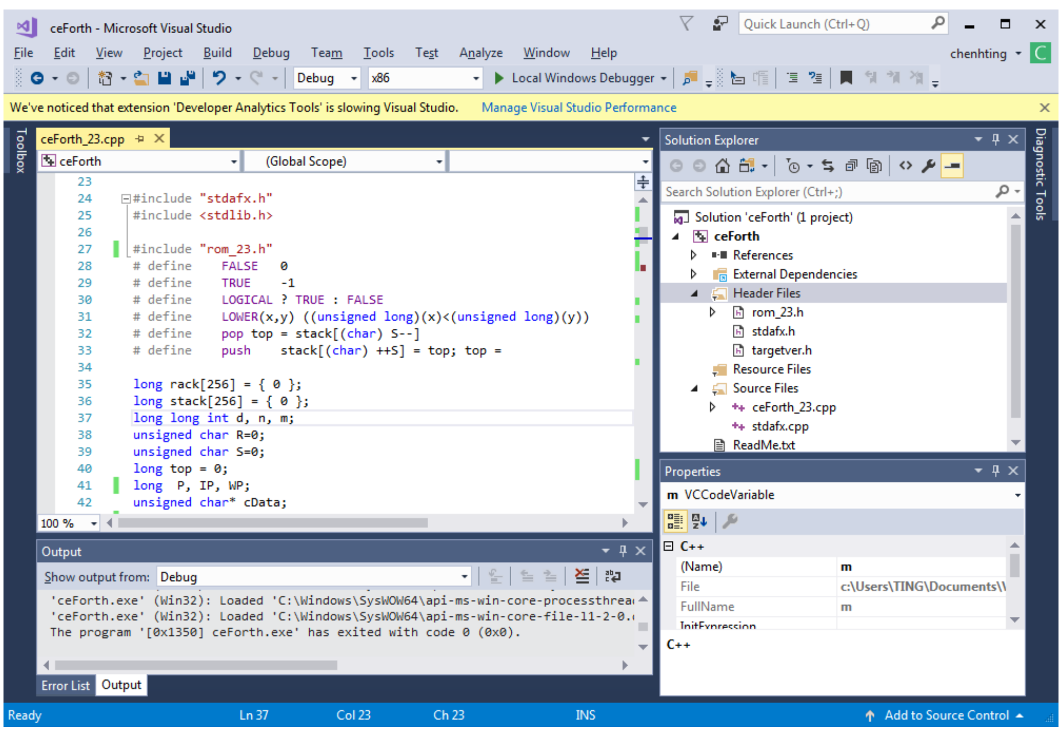
Visual Studio created a new project for you, and gives you a template file `ceForth.cpp`. Copy the contents of `ceForth_23.cpp` supplied in `ceForth_23.zip` file, and paste them into `ceForth.cpp`.

Compile ceForth

Next you have to copy the `rom_23.h` file into ceForth project, with ceForth.cpp and other files supplied by Visual Studio:



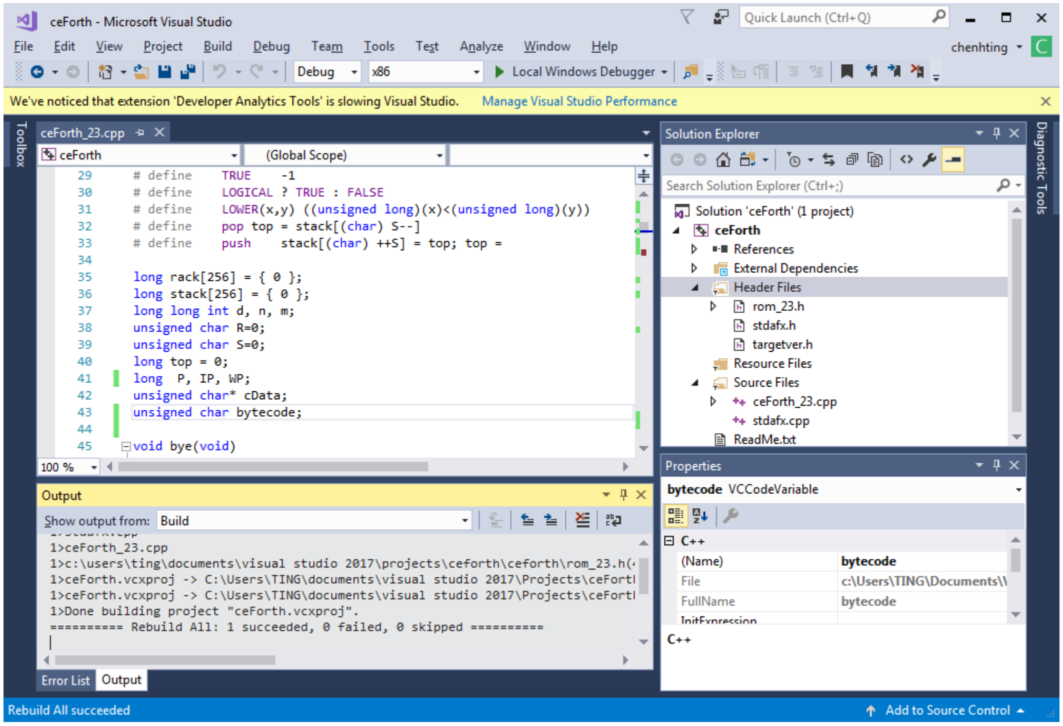
In the Solution Explorer Panel, right click `Header Files`, and then select `Add>Existing Items…`. Now select `rom_23.h` to add it to this project. Now your Visual Studio windows should look something like this:



With both `ceForth.cpp` and `rom_23.h` included in your project, you are ready to build `ceForth`.

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:

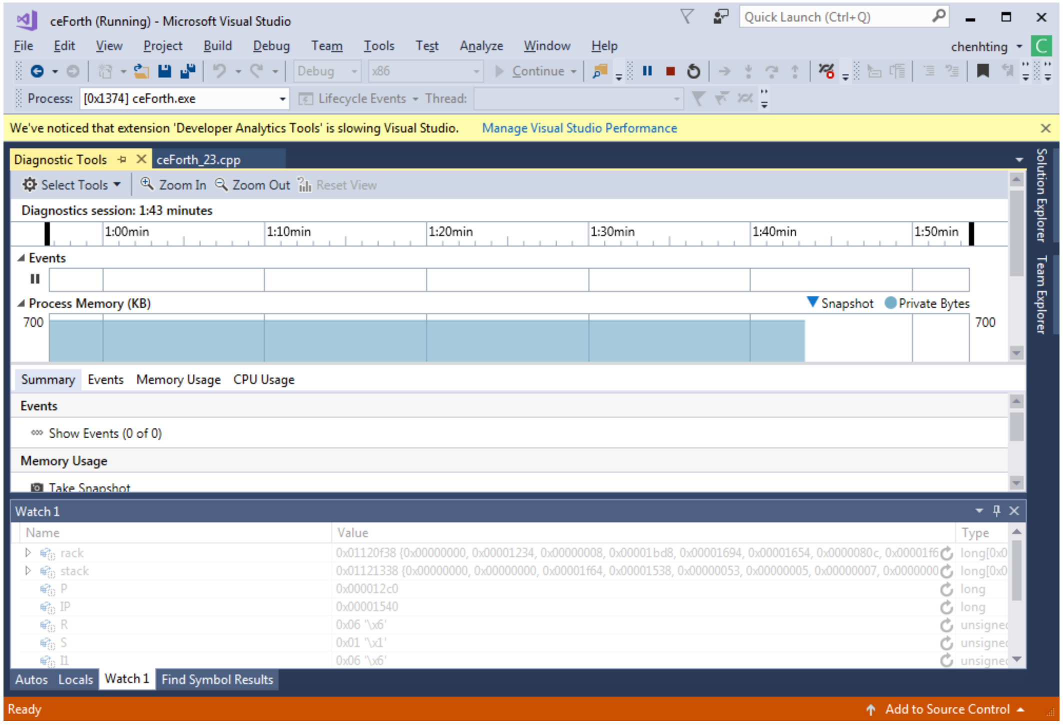
```
===== Rebuild All, 1 succeeded, 0 failed, 0 skipped =====
```



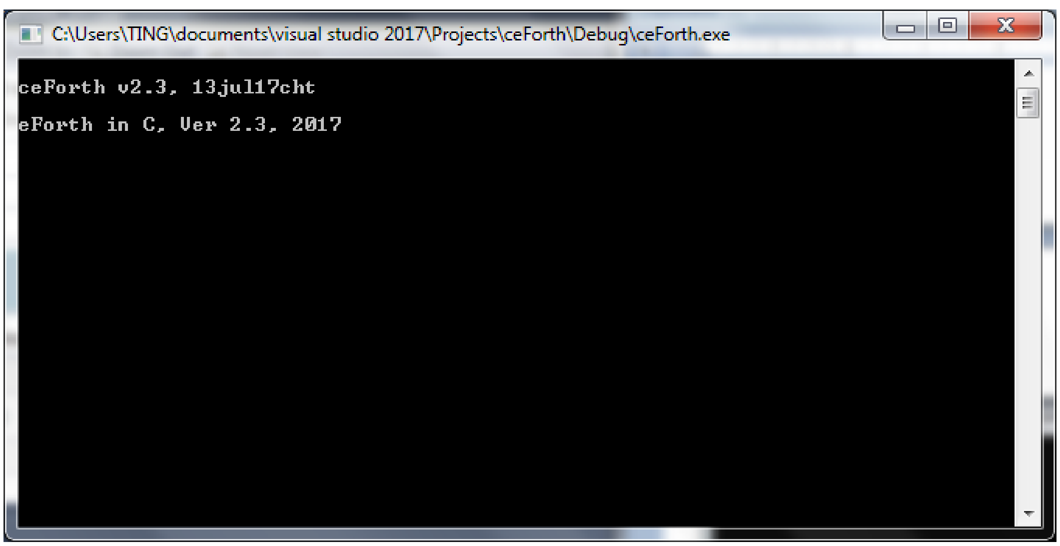
All is well. Ready to test.

Test ceForth

Click `Debug>Start debugging`. Wait some more. Finally, you will see the Debug window:

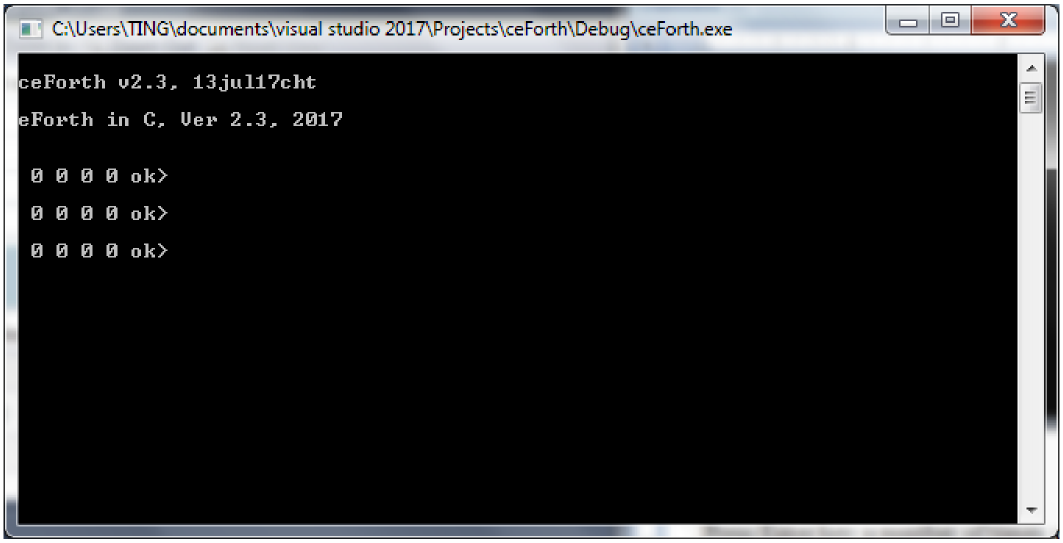


On top of it, you have the Console Window:

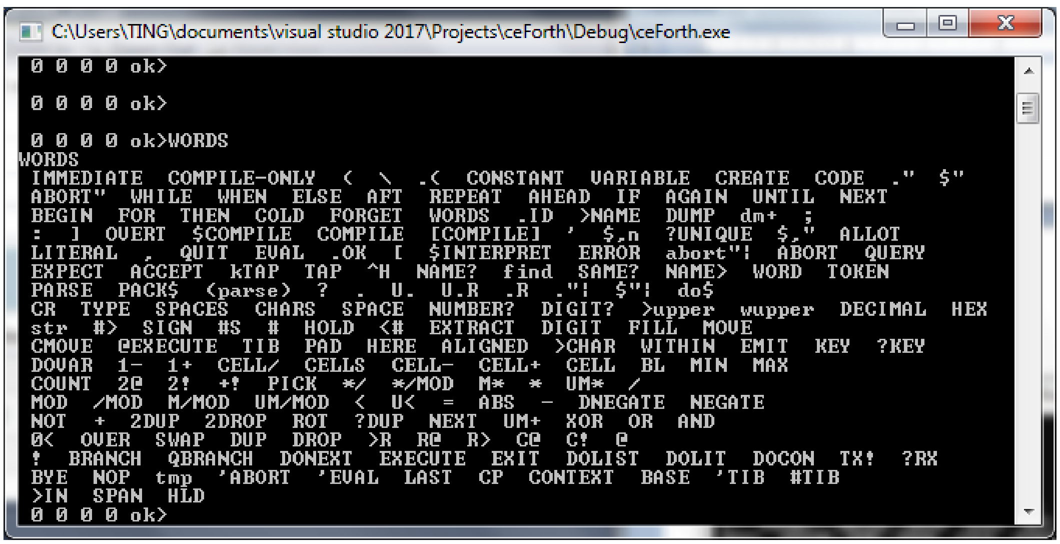


Success! ceForth is running.

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



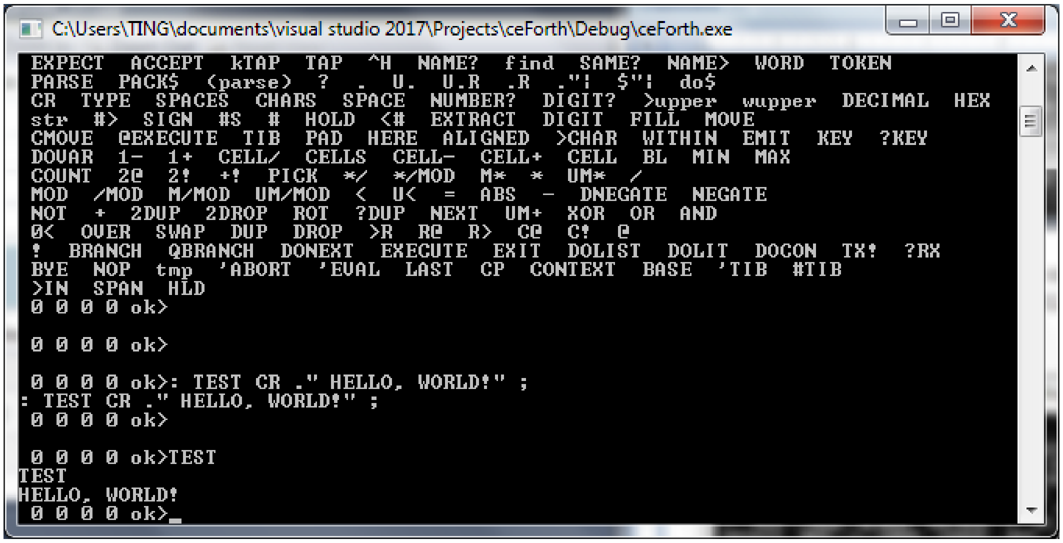
Type `WORDS`, and you get a screen of command names representing a complete ceForth system:



Now, enter this universal greeting command:

```
: TEST CR ." HELLO, WORLD!" ;
```

and then type `TEST`:



ceForth is now fully functioning.

Chapter 2. ceForth Virtual Forth Engine

ceForth_23.cpp

The file `ceForth_23.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 Firth 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 commands are stored in a large data array, called a dictionary.
- Each command is a record. All records are linked in the dictionary.
- Each command 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 command from its ASCII name. The code field contains executable byte code. The parameter field contains optional code and data needed by the command.
- There are two types of commands: primitive commands containing executable byte code, and compound commands containing token lists. A token is a code field address pointing to code field of a command.
- A sequencer executes byte code sequences stored in code fields of primitive commands.
- An inner interpreter terminates a primitive command 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 commands
- A text interpreter processes command lists entered from a terminal.

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

```
<list of commands>
```

It can also create new commands to replace lists of commands:

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

These new commands are called compound commands, in which lists of commands are compiled into token lists.

All computable problems can be solved by repeatedly creating new commands to replace lists of existing commands. 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_23.cpp`, to see how this VFM is actually implemented.

Preamble

In the beginning of `ceForth_23.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 macro define instructions to facilitate compilation the rest of C code.

`stdafx.h` is a precompiled header required by Visual Studio C++ compiler.

```
#include "stdafx.h"
```

`stdlib.h` is a standard library header file needed by ceForth.

```
#include <stdlib.h>
```

`rom_23.h` is the Forth dictionary produced by ceForth metacompiler.

```
#include "rom_23.h"
```

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 commands which generate flags would return a -1 for `TRUE`.

```
# define  FALSE  0
# define  TRUE  -1
```

`LOGICAL` is a macro enforcing the above policy for logic commands to return the correct `TRUE` and `FALSE` flags.

```
# define  LOGICAL ? TRUE : FALSE
```

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

```
# define  LOWER(x,y) ((unsigned long)(x)<(unsigned long)(y))
```

`pop` is a macros which stream-line 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.

```
# define   pop   top = stack[(char) S--]
```

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.

```
# define   push  stack[(char) ++S] = top; top =
```

Registers and Arrays

ceForth uses a large data array to hold it dictionary of commands. 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, S;
long top, I, P, IP;
unsigned char* cData;
unsigned char bytecode;
```

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

Register/Array	Functions
P	Program counter, pointing to byte code in data[].
IP	Instruction pointer for address interpreter for token lists
WP	Scratch register, generally pointing to parameter field
bytecode	Byte code register for byte code to be executed
Rack[256]	Return stack, a 256 cell circular buffer
R	One byte return stack pointer
Stack[256]	Data stack, a 256 cell circular buffer
S	One byte data stack pointer
top	Cached top element of the data stack
d,m,n	Scratch registers for 64 bit integers for multiply and divide
cData	A byte pointer to access data[] array in bytes
data[4096]	An array containing Forth dictionary

Data stack and return stack are implemented as 256 element circular buffers `stack[256]` and `rack[256].` Stack pointers `S` and `R` are byte values, ensuring that the stacks would never overflow or underflow. With this design, I eliminated many commands which would be necessary to manage the stacks.

Dictionary Array

All Forth commands are stored in a `data[4096]` 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 command, the parameter field has additional byte code. In a high level compound command, the parameter field contains a sequence of tokens which are code field addresses pointing to the code field of other Forth commands.

The dictionary in `data`[] array is generated by a separated Forth program called a metacompiler. It is discussed in the Chapter 3 of this manual. This dictionary is saved in a file rom_23.h, and incorporated into the Virtual Forth Machine by the command #include rom_23.h. The beginning and ending of rom_23.h are shown here:

```
long data[4096] = {
/* 00000000 */ 0x00000000,
/* 00000004 */ 0x00000000,
…
);
```

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

VFM Functions

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

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 command. WP helps retrieving the token list of a compound command, 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 command.

```
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 = P;
      next(); }
```

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

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

`execu` (a --) Take the execution address from data stack and executes that token. This powerful command 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)
```

```
{     w = top;
      top = stack[(char) S];
      stack[(char) S] = w; }
```

`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)
{     w = stack[(char) S - 1];
      stack[(char) S - 1] = stack[(char) S];
      stack[(char) S] = top;
      top = w; }
```

`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; }
```

`uless` (u1 u2 -- t) Unsigned compare of top two items.

```
void uless(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) {
            w = stack[(char) S] / top;
            stack[(char) S] %= top;
            top = w;
      } }
```

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

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

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

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

```
void mstar(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; }
```

`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 << 32;
      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 << 32;
      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 maxx(void)
{     if (top < stack[(char) S]) pop;
      else (char) S--; }
```

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

```
void minn(void)
{     if (top < stack[(char) S]) (char) S--;
```

```
      else pop; }
```

Byte Code Array

There are 67 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 commands which have these byte code in their code field. The byte code may spill over into the subsequent parameter field, if a primitive command 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 67 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 commands using the metacompiler I will discuss later, or to compile more compound commands 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 */ xorr,
```

```
	/* 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 */ dnega,
	/* 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,
};
```

These byte code are executed by a special function `execute(code)`, which passes the byte code `code` to the byte code array `primitives[code]` to call the corresponding function:

```
void execute(unsigned char code)
{	if (code < 64) { primitives[code](); }
	else { printf("\n Illegal code= %x P= %x", code, P); } }
```

Byte Code Sequencer

To start the VFM running, some of the registers have to be initialized in the `main()` function required by Visual Studio C++ IDE. At the very end of ceForth_23.cpp file, and we have a byte code sequencer to execute pseudo instructions stored in the `data[]` array.

```
/*
* Main Program
*/
int main(int ac, char* av[])
```

```
{     P = 0;
      WP = 4;
      IP = 0;
      S = 0;
      R = 0;
      top = 0;
      cData = (unsigned char *)data;
      printf("\nceForth v2.3, 13jul17cht\n");
      while (TRUE) {
            bytecode = (unsigned char)cData[P++];
            execute(bytecode);
      } }
```

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

In the earlier implementations of ceForth, I copied the Finite State Machine in eP32 chip. It assumed that 32 bit words were read in Phase 0, and 4 byte code were decoded. In the next 4 phases, these byte code were executed. Phase 5 is needed to return to Phase 0.

Since I can read data[] either in words or in bytes, it is much simpler to sequence through the byte code list one byte at a time. The sequencer discussed above is much simpler than the earlier finite state machines. However, tokens are 32 bit addresses, and are accessed more conveniently with work addressing.

`main`() initializes P to 0. VFM starts by executing byte code stored in memory location 0 in the Forth dictionary. The byte code at location 0 are:

```
dolist()  nop()  nop()  nop()
COLD
```

`dolist`() expects a token list at location 4. Therefore WP must be initialized to 4, pointing to `COLD`. This token list has only one element. `COLD` eventually drops into `QUIT`, which contains an infinite loop, and never returns back to this list.

Chapter 3. Metacompilation of the ceForth

Metacompilation

In Forth community, metacompilation means compiling a new Forth system using an existing Forth system. As Forth system is generally consider an operating system supporting Forth programming language, metacompilation is the highest level of art in Forth programming. For people not converse in the art of Forth, it is black magic.

Metacompilation always reminds me of metamorphosis, transformation of a caterpillar into a butterfly.

When Chuck Moore invented Forth in 1960's, he quickly found Forth was powerful enough to regenerated itself and easily migrated to any computer in his sight. It was so mysterious to others that he felt confident to release complete source code with his implementations, without the fear that others might reverse-engineer his Forth system. He stores his source code in 1024 byte blocks of memory on tapes and disks. Without his Forth system and his block editor, nobody could read his code. He was able to reprogram so many telescope computers, that the International Astronomy Society adopted Forth as the official programming language for observatory automation in 1974.

It was not until 1978 that the Forth Interest Group (FIG) in Silicon Valley engineered 6 figForth systems for the then most popular microcomputers. FIG released figForth in the form of assembly files so that non-Forth programmers could implement them on their own microcomputers with various operating systems, and understand what was going on in these Forth systems.

In 1990, Bill Muench and I developed eForth Model, with a very small set (33) of primitive commands so that it could be ported easily to microcontrollers. It was implemented on 30 some different microprocessors and microcontrollers by many volunteers. In the meantime, I also worked with Chuck Moore on his new Forth chip, MuP21. He reduced its instructions to 25, and fit a 20-bit microprocessor on a small die, with an NTSC video coprocessor and a DRAM memory coprocessor. It was a marvelous design, but we ran out of money before it was perfected.

I compared the designs of eForth and the MuP21, and found great similarity, in spite of the completely different origins of these two designs. eForth is a software design and the MuP21 is a hardware design. However, they both were based on primitive instruction sets with about 30 instructions. Many instructions were identical in these two instruction sets. Those instructions which were different, were different because of hardware constraints. I was able to implement eForth on the MuP21. A real Forth language on a real Forth chip. Chuck gave me a metacompiler to compile eForth on MuP21.

After the MuP21, Chuck and I went our separate ways. He founded iTV, and Intellesys, and Green Arrays to built multiprocessor chips based on the MuP21 core. I discovered FPGAs, and developed scalable P-series microcontrollers based on the same core, implementing 16-, 24- and 32-bit versions of the P-microcontrollers.

In 2000, a young fellow in Taiwan, Mr. Cheah-shen Yap, ported eForth to Windows to become the F# system. It could call all Windows APIs to build applications running on a PC. I used it to write metacompilers for P-microcontrollers, including P24, eP16 and eP32. For these processors, F# metacompiler had assemblers to assemble, and then built Forth dictionary, which was loaded into the memory to start Forth running on these machines. However, for the ceForth, I build a Virtual Forth Machine in C, and modified the F# metacompiler to compile a dictionary to be incorporated into the C program. The C program is compiled as a C++ Windows Console program on Visual Studio, and opens a DOS like window accepting your commands.

In this Chapter, I will discuss how the F# metacompiler construct the Forth dictionary used in ceForth_23.cpp. The discussion follows the source files in their loading sequence.

The immediate goal of ceForth metacompiler is to build three defining commands: `INST`, `CODE`, and `::` (colon-colon). They are cookie cutters which create classes of commands to build the target dictionary in a data array.

`INST` creates a set of byte code assembly commands. An assembly command assembles one byte code into the code fields of primitive commands in target dictionary. Examples are:

```
0 INST nop,
1 INST bye   ,   2 INST qrx,    3 INST txsto,  4 INST docon,
5 INST dolit,    6 INST dolist, 7 INST exit,   8 INST execu,
```

`CODE` creates new primitive commands. It compiles the link field and name field of a new primitive command in target dictionary. Its code field is now open to the byte code assemblers to assemble byte code. Examples are:

```
CODE NOP next,
CODE BYE bye, next,
CODE EMIT txsto, next,
CODE DOLIT dolit, next,
CODE DOLIST dolist, next,
CODE EXIT exit, next,
```

`::` (colon-colon) creates new compound commands. It compiles the link field and name field of a new compound command in target dictionary. It then adds a byte code `dolist,` to the code field. Its parameter field is now ready to build a new token list. Examples are:

```
:: WITHIN ( u ul uh -- t ) \ ul <= u < uh
  OVER - >R - R> U< ;;
:: >CHAR ( c -- c )
  $7F LIT AND DUP $7F LIT BL WITHIN
  IF DROP ( CHAR _ ) $5F LIT THEN ;;
:: ALIGNED ( b -- a ) 3 LIT + FFFFFFFC LIT AND ;;
:: HERE ( -- a ) CP @ ;;
```

```
:: PAD ( -- a ) HERE 50 LIT + ;;
:: TIB ( -- a ) 'TIB @ ;;
```

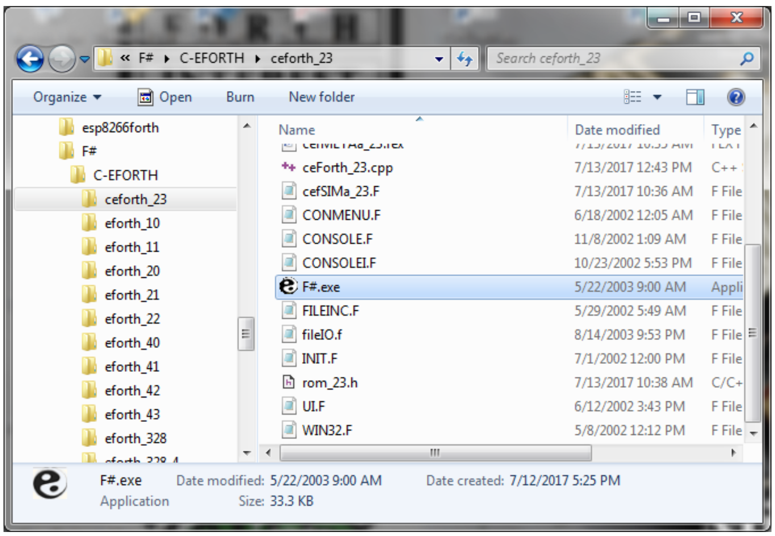
`CODE` and `::` (colon-colon) also mark the code field addresses of the new commands they created. When these commands are later executed, they compile their respective code field addresses into the token list under construction.

`INST,` `CODE` and `::` (colon-colon) are defined in cefASMa_23.f.

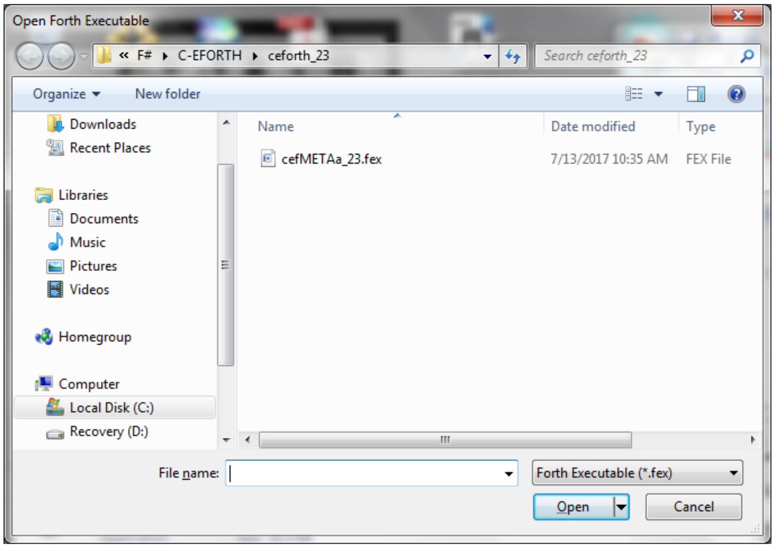
Building ceForth Dictionary

All source code of the ceForth eForth system is contained in the ceForth_23.zip file. F# system and its Windows utilities are also included here.

Unzip file ceForth_23.zip and put all the files into a folder named “ceForth_23”.

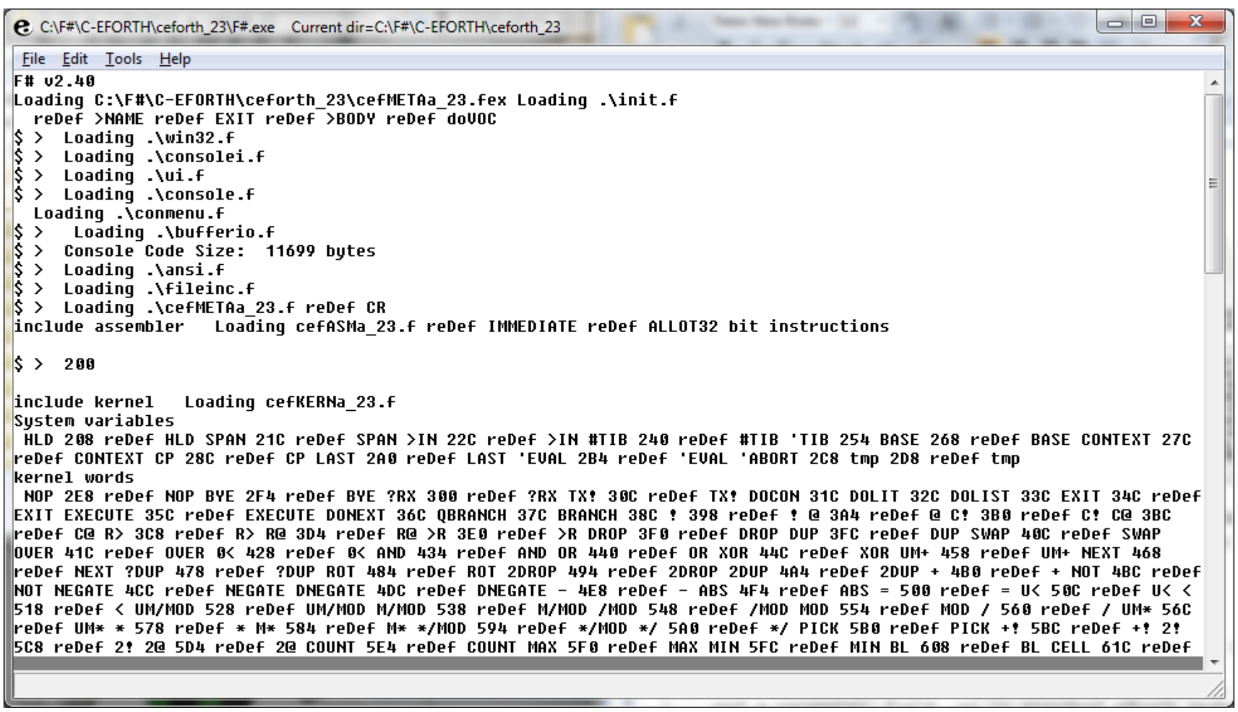


Start F# by double clicking F#.exe in the ceForth_23 folder, and a file select window is opened:



F# organizes projects using .fex files. Each .fex file loads all the files needed in a project. Metacompiler of ceForth_23 is contained in the ceMETAa_23.fex. Click Open button to run it.

F# opens a console window, loads the ceForth metacompiler and generates a new ceForth dictionary Lot of text scrolled up in this console window. Here is a picture of the lines scrolled up:



The first number of lines show a number of files loaded. The last file loaded is cefMETAa_23.f, which loads in the metacompiler, and start building ceForth dictionary. The contents of ceMETAa_23.fex are as following:

```
\ ceForth_23, 10jul17cht
\ cEFa  10sep09cht
\ Goal is to produce a dictionary file to be compiled by a C compiler
\ Assume 31 eForth primitives are coded in C
\ Each FORTH word contains a link field, a name field, a code field
\    and a parameter field, as in standard eForth model
\ The code field contains a token pointing to a primitive word
\ Low level primitive FORTH words has 1 cell of code field
\ High level FORTH word has doList in code field and a address list
\ Variable has doVAR in code field and a cell for value
\ Array is same as variable, but with many cells in parameter field
\ User variable has doUSE in code, and an offset as parameter
\
\

 FLOAD .\init.f         \ initial stuff
 FLOAD .\win32.f        \ win32 system interface
 FLOAD .\consolei.f     \ api and constant defination

 FLOAD .\ui.f           \ user interface helper routine ( reposition )

 FLOAD .\console.f      \ the main program
 FLOAD .\ansi.f
 FLOAD .\fileinc.f
 FLOAD .\cefMETAa_23.f

cr .( Version FIX 12SEP09CHT )
```

Files loaded by ceMETAa_23.fex are for these purposes:

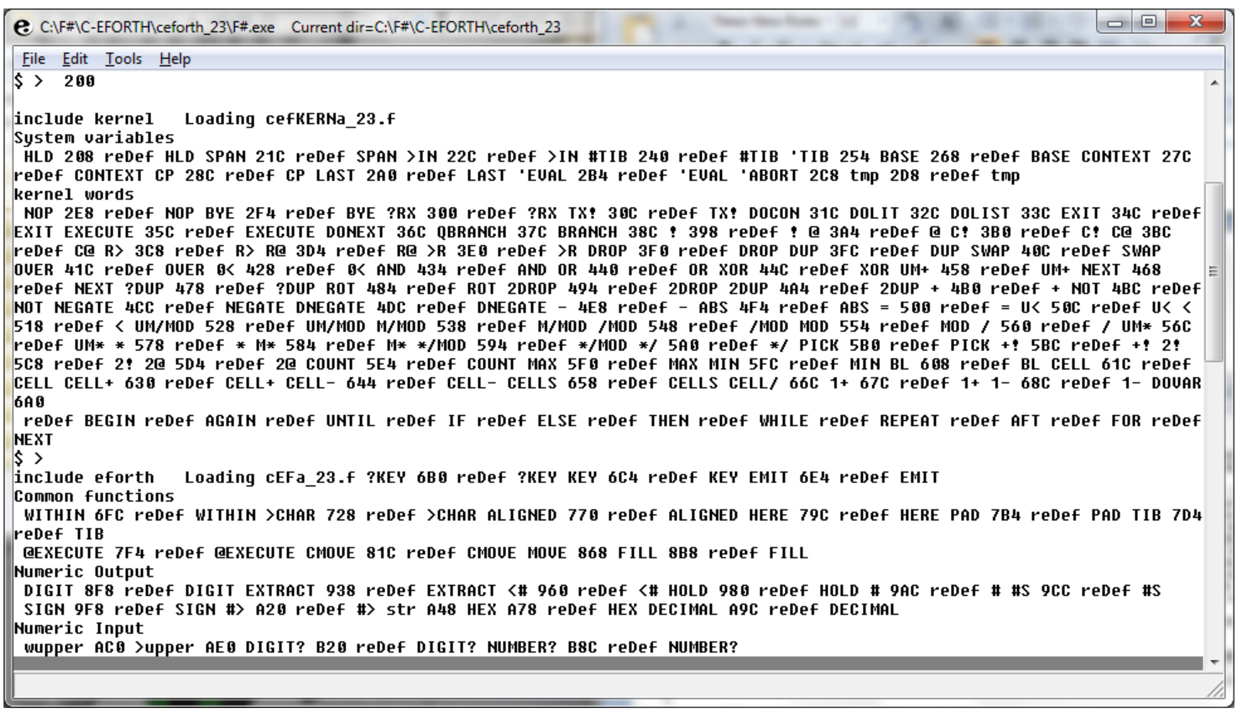
init.f	Extend F# core to manage vocabulary and constants
win32.f	WinAPI interface and CallBack
consolei.f	API constants and prototypes
ui.f	Windows user interface
console.f	Create main console window
conmenu.f	Windows console and menu
bufferio.f	Buffered console input and output
ansi.f	Add words for ANSI Forth compatibility
fileinc.f	Windows file interface API
cefMETAa_23.f	ceForth metacompiler
cefASMa_23.f	ceForth assembler of byte code pseudo instructions
cefKERNa_23.f	Compile ceForth primitive commands
cEFa_23.f	Compile ceForth compound commands

`cefMETAa_23.f` allocates a large data array `RAM` in F# memory, to host the dictionary target to ceForth system. The contents in the `RAM` array will be saved as a text file `rom_23.h`, which will eventually be copied into the `data`[] array in Visual Studio `ceForth_23.cpp` to run ceForth on Windows PC.

When `cefKERNa_23.f` is being loaded, new primitive commands are added to the `RAM` array dictionary. F# and ceForth are derived from the same eForth Model, and they have much the same command set. When a new command is added to the target dictionary, a new word is also added to the F# dictionary, representing a new token in the target dictionary. The name of this token is the same as the name of the new command in target, and the name of an existing word in F#. We are in fact redefine a F# word. For example the first primitive command HLD causes this line of message:

```
HLD 208 redef HLD
```

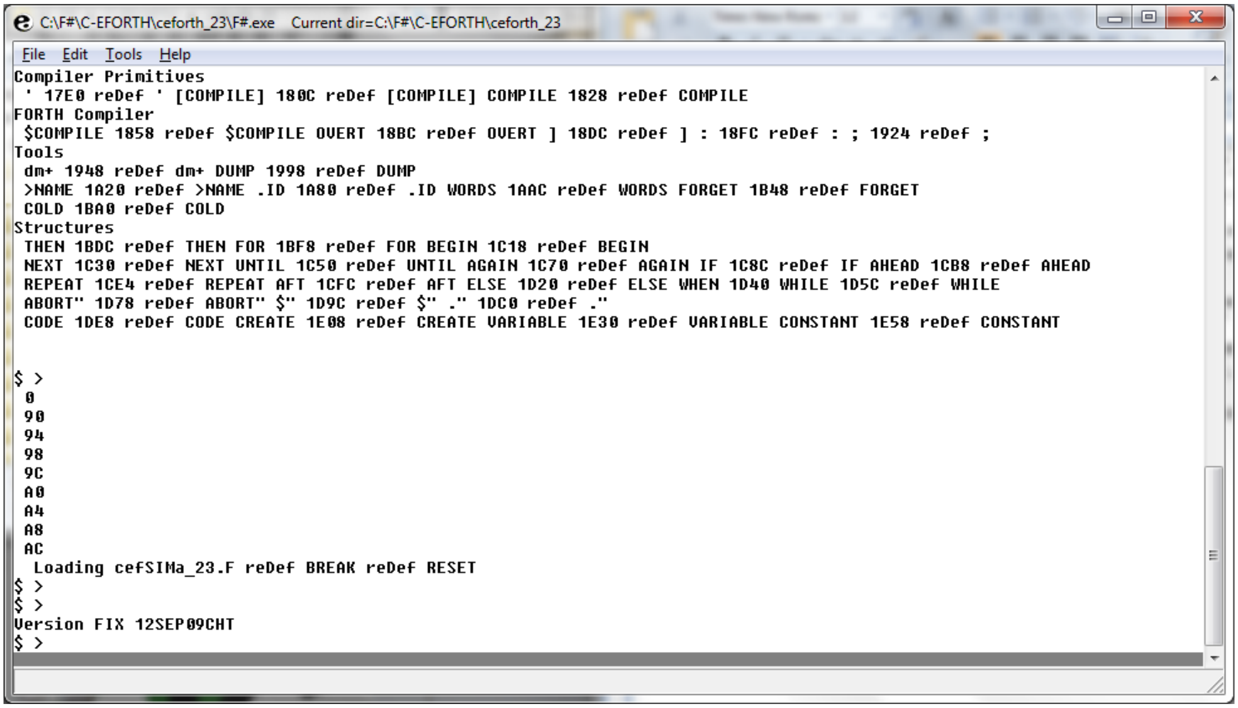
It means that a new command HLD is defined in target dictionary, and its code field address is 0x208, which will be used to compile a token of 0x208 when HLD is encountered in the token list of a compound command. Since HLD was an existing F# word, a warning message "`redef HLD`" is issued. The list continues until all primitive commands are compiled, as shown here:



```
C:\F#\C-EFORTH\ceforth_23\F#.exe    Current dir=C:\F#\C-EFORTH\ceforth_23
File  Edit  Tools  Help
$ >  200

include kernel   Loading cefKERNa_23.f
System variables
 HLD 208 reDef HLD SPAN 21C reDef SPAN >IN 22C reDef >IN #TIB 240 reDef #TIB 'TIB 254 BASE 268 reDef BASE CONTEXT 27C
reDef CONTEXT CP 28C reDef CP LAST 2A0 reDef LAST 'EVAL 2B4 reDef 'EVAL 'ABORT 2C8 tmp 2D8 reDef tmp
kernel words
 NOP 2E8 reDef NOP BYE 2F4 reDef BYE ?RX 300 reDef ?RX TX! 30C reDef TX! DOCON 31C DOLIT 32C DOLIST 33C EXIT 34C reDef
EXIT EXECUTE 35C reDef EXECUTE DONEXT 36C QBRANCH 37C BRANCH 38C ! 398 reDef ! @ 3A4 reDef @ C! 3B0 reDef C! C@ 3BC
reDef C@ R> 3C8 reDef R> R@ 3D4 reDef R@ >R 3E0 reDef >R DROP 3F0 reDef DROP DUP 3FC reDef DUP SWAP 40C reDef SWAP
OVER 41C reDef OVER 0< 428 reDef 0< AND 434 reDef AND OR 440 reDef OR XOR 44C reDef XOR UM+ 458 reDef UM+ NEXT 468
reDef NEXT ?DUP 478 reDef ?DUP ROT 484 reDef ROT 2DROP 494 reDef 2DROP 2DUP 4A4 reDef 2DUP + 4B0 reDef + NOT 4BC reDef
NOT NEGATE 4CC reDef NEGATE DNEGATE 4DC reDef DNEGATE - 4E8 reDef - ABS 4F4 reDef ABS = 500 reDef = U< 50C reDef U< <
518 reDef < UM/MOD 528 reDef UM/MOD M/MOD 538 reDef M/MOD /MOD 548 reDef /MOD MOD 554 reDef MOD / 560 reDef / UM* 56C
reDef UM* * 578 reDef * M* 584 reDef M* */MOD 594 reDef */MOD */ 5A0 reDef */ PICK 5B0 reDef PICK +! 5BC reDef +! 2!
5C8 reDef 2! 2@ 5D4 reDef 2@ COUNT 5E4 reDef COUNT MAX 5F0 reDef MAX MIN 5FC reDef MIN BL 608 reDef BL CELL 61C reDef
CELL CELL+ 630 reDef CELL+ CELL- 644 reDef CELL- CELLS 658 reDef CELLS CELL/ 66C 1+ 67C reDef 1+ 1- 68C reDef 1- DOVAR
6A0
 reDef BEGIN reDef AGAIN reDef UNTIL reDef IF reDef ELSE reDef THEN reDef WHILE reDef REPEAT reDef AFT reDef FOR reDef
NEXT
$ >
include eforth   Loading cEFa_23.f ?KEY 6B0 reDef ?KEY KEY 6C4 reDef KEY EMIT 6E4 reDef EMIT
Common functions
 WITHIN 6FC reDef WITHIN >CHAR 728 reDef >CHAR ALIGNED 770 reDef ALIGNED HERE 79C reDef HERE PAD 7B4 reDef PAD TIB 7D4
reDef TIB
 @EXECUTE 7F4 reDef @EXECUTE CMOVE 81C reDef CMOVE MOVE 868 FILL 8B8 reDef FILL
Numeric Output
 DIGIT 8F8 reDef DIGIT EXTRACT 938 reDef EXTRACT <# 960 reDef <# HOLD 980 reDef HOLD # 9AC reDef # #S 9CC reDef #S
 SIGN 9F8 reDef SIGN #> A20 reDef #> str A48 HEX A78 reDef HEX DECIMAL A9C reDef DECIMAL
Numeric Input
 wupper AC0 >upper AE0 DIGIT? B20 reDef DIGIT? NUMBER? B8C reDef NUMBER?
```

Then, cEFa_23.f file is loaded, and all the compound commands are compiled.



Finally, the system variables are compiled, the entire target dictionary is finished, and is written out to the `rom_23.h` file.

Copy `rom_23.h` into ceForth_23 project and you can build and test ceForth_23 on Visual Studio.

Metacompiler

Metacompiler is a term used by Forth programmers to describe a program to build a new Forth system on an existing Forth system. The new Forth system may run on the same platform as the old Forth system. It may be targeted to a new platform, or to a new CPU. The new Forth system may share a large portion of Forth code with the old system, hence the term metacompilation. In a sense, a metacompiler is very similar to a conventional cross assembler/compiler.

The ceForth metacompiler is contained in file `cefMETAa_23.f`. It allocates a data array `RAM`, and deposits records of primitive commands and compound commands to build a dictionary for ceForth. The Virtual Forth Machine (VFM) was programmed in `ceForth_23.cpp,` as a set of C functions represented by a set of byte code. These byte code are assembled into the code fields of primitive commands. The addresses of code fields become tokens compiled as token lists in the parameter fields of compound commands

After setting up the environment to build a target dictionary, `cefMETAa_23.` loads source code from three other files to do the building:

`cefASMa_23.f`	ceForth assembler

cefKERNa_23.f	Assemble primitive commands
cEFa_23.f	Compile compound commands

Source code in cefMETAa_23.f is lengthy, and it is best to comment each command to bring out its function and meaning.

debugging? (-- a) A variable containing a switch to turn break points on and off. When debugging? is set to -1, compilation will stop and the data stack is dumped when a “cr” command is executed. Sprinkling “cr” commands in the source code file allows you to watch the progress of metacompilation and even stops it when necessary.

```
variable debugging?
\ -1 debugging? !
```

.head (a – a) Display name of a command that is about to be compiled. It is used to display a symbol table. You can look up the code field address of any command in this table.

```
: .head ( addr -- addr )
   SPACE >IN @ 20 WORD COUNT TYPE >IN !
   DUP .
   ;
```

cr (--) Stop metacompilation if debugging? is -1, and dump data stack. If you press control-A, metacompilation is aborted. Otherwise, metacompilation continues. It is a NOOP if debugging? is 0.

```
: CR CR
   debugging? @
   IF .S KEY 0D = IF ." DONE" QUIT THEN
   THEN
   ;
```

break (--) Pause metacompilation and dump data stack. If you press Return, metacompilation is aborted. Otherwise, metacompilation continues. It sets a break point.

```
: BREAK CR
   .S KEY 0D = IF ." DONE" QUIT THEN
   ;
```

During metacompilation, Forth commands will be redefined so that they assemble byte code or compile tokens into the target dictionary. There are numerous occasions when the original behavior of a Forth command must be exercised. To preserve the original behavior of a Forth command, it is assigned a different name. Thereby after a command is redefined, you can still exercise its original behavior by invoking the alternate name.

For example, “DUP” is a Forth command that duplicates the top number on the data stack in the F# system. Then in the cefKERNa_23.f file, a new “DUP” command is defined to compile a DUP token in the target ceForth system. If you still need to duplicate a number, you must use the alternate command “forth_dup” as shown below. All the F# commands you need to use later must be redefined as “forth_xxx” commands. If you neglect to redefine them, you will find that the system behaves very strangely.

```
: forth_dup DUP ;
: forth_drop DROP ;
: forth_over OVER ;
: forth_swap SWAP ;
: forth_@ @ ;
: CRR cr ;
```

ceForth executes commands and accesses data in the dictionary, range 0-3FFF. In F# we allocated a 32k byte memory array, "RAM", to hold the ceForth target dictionary. This array contains code and data to be copied into ceForth `data[]` array, to be compiled by ceForth_23.cpp, and eventually to be executed as a Windows Console Application.

`RAM` (-- a) Memory array in F# for the ceForth target dictionary. It has a logical base address of 0 for the ceForth. Commands and data words in the target are stored in this array.

```
CREATE RAM  8000 ALLOT
```

`RESET` (--) Clear "RAM" image array, preparing it to receive code and data for the ceForth.

```
: RESET   RAM 8000 0 FILL ;   RESET
```

`RAM@` (a – n) Replace a logical address on stack with data stored in "RAM" dictionary.

```
: RAM@    RAM +  @ ;
```

`RAMC@` (a – c) Replace a logical address on stack with byte data stored in "RAM" dictionary.

```
: RAMC@   RAM +  C@ ;
```

`RAM!` (a n --) Store second integer on stack into logical address of "RAM" dictionary.

```
: RAM!    RAM +  ! ;
```

`RAM!` (a c --) Store second byte on stack into logical address of "RAM" dictionary.

```
: RAMC!   RAM +  C! ;
```

`FOUR` (a --) Display four consecutive words in target dictionary.

```
: FOUR   (  a -- ) 4 FOR AFT  DUP RAM@ 9 U.R  4 + THEN NEXT  ;
```

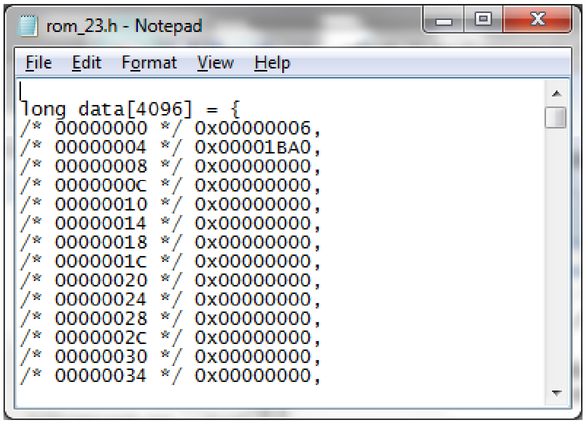
`SHOW` (a – a+128) Display 128 words in target from address "a". It also returns a+128 to "show" the next block of 128 words.

```
: SHOW ( a)   10 FOR AFT  CR  DUP 7 .R SPACE
      FOUR SPACE FOUR  THEN NEXT ;
```

`SHOWRAM` (--) Display the entire ceForth dictionary of 2K words.

```
: showRAM 0 0C FOR AFT SHOW THEN NEXT DROP ;
```

The ceForth metacompiler builds a target dictionary for the ceForth in `RAM`. This dictionary eventually will be imported to the ceForth_23 project so that this dictionary will be incorporated in ceForth. Visual Studio IDE requires that the dictionary be written in a file conforming to its `long data[]` array format, which consists of a header with a body containing memory information in hexadecimal numbers. The header and first few lines of the body are as follows:



This dictionary is written to a text file `rom_23.h`. Here are the commands to open this file, writing data to it, and closing it.

`hFile` (-- handle) A variable holding a file handle.

```
VARIABLE hFile
```

`CRLF-ARRAY` (-- a) A byte array containing CR and LF characters.

```
CREATE CRLF-ARRAY 0D C, 0A C,
```

`CRLF` (--) Insert a carriage return and a line feed into the currently opened file.

```
: CRLF
      hFile @
      CRLF-ARRAY 2
      PAD ( lpWrittenBytes )
      0 ( lpOverlapped )
      WriteFile
      IF ELSE ." write error" QUIT THEN
   ;
```

`open-mif-file` (--) Open a file named `rom_23.h` for writing.

```
: open-mif-file
   Z" rom_23.h"
   $40000000 ( GENERIC_WRITE )
   0 ( share mode )
   0 ( security attribute )
   2 ( CREATE_ALWAYS )
   $80 ( FILE_ATTRIBUTE_NORMAL )
   0 ( hTemplateFile )
   CreateFileA hFile !
   ;
```

`write-mif-header` (--) Write a header required by Visual Studio into current file.

```
: write-mif-header
   CRLF
      hFile @
      $" long data[4096] = {"
      PAD ( lpWrittenBytes )
```

```
      0 ( lpOverlapped )
      WriteFile
      IF ELSE ." write error" QUIT THEN
   ;
```

`write-mif-trailer` (--) Write last line of text into current file.

```
: write-mif-trailer
   CRLF
      hFile @
      $" 0 } ; "
      PAD ( lpWrittenBytes )
      0 ( lpOverlapped )
      WriteFile
      IF ELSE ." write error" QUIT THEN
   ;
```

`write-mif-data` (--) Write a 4K word image of the ceForth dictionary from memory array "`RAM`" to the `rom_23.h` file.

```
: write-mif-data
   0 ( initial RAM location )
   $800 FOR AFT
      CRLF
      hFile @
      OVER ( 4 / )  ( byte address )
      <# 2F HOLD 2A HOLD 20 HOLD
          7 FOR # NEXT
      20 HOLD 2A HOLD 2F HOLD #>
      PAD ( lpWrittenBytes )
      0 ( lpOverlapped )
      WriteFile
      IF ELSE ." write error" QUIT THEN
      hFile @
      OVER RAM@
      <# 2C HOLD 7 FOR # NEXT 78 HOLD 30 HOLD 20 HOLD #>
      PAD ( lpWrittenBytes )
      0 ( lpOverlapped )
      WriteFile
      IF ELSE ." write error" QUIT THEN
      CELL+
   THEN NEXT
   DROP ( discard RAM location )
   ;
```

`close-mif-file ( -- )` Close rom_23.h file.

```
: close-mif-file
   hFile @ CloseHandle DROP
   ;
```

`write-mif-file` (--) open `rom_23.h` file, write a header, write data, write trailer, and then closes the file. `rom_23.h` containing 4K words of the ceForth dictionary.

```
: write-mif-file
```

```
open-mif-file
write-mif-header
write-mif-data
write-mif-trailer
close-mif-file
;
```

The ceForth metacompiler continues to load the byte code assembler in `cefASM_23.f`. In the assembler, all byte code of VFM are defined, and the ways they are assembled into code fields of primitive commands. Means to compile link fields and name fields to form headers of commands are also defined. It is now almost ready to assemble primitive commands for ceForth.

```
CR .( include assembler )
FLOAD cefASMa_23.f
```

After the assembler is built, we are ready to build the kernel part of ceForth dictionary. All primitive commands are assembled in `cefKERNa_23.f` file. The kernel starts at location 0x200, leaving rooms for the Terminal Input Buffer TIB in the area 0x1000x1FF. System variables from 0x80-0xFF.

```
$200 ORG
CR .( include kernel )
FLOAD cefKERNa_23.f
```

With the kernel in place, high level compound commands are compiled immediately after the kernel, by loading `cEFa_23.f`. The top ceForth dictionary is at 0x1F70 so far. It is pushed on data stack by the commands '`H forth_@`', to be used later to initialize the system variable CP. With 4096 words allocated in `data[]` array, the space is about half full. You can compile substantial application in this dictionary. If you need more space, just allocate a bigger array.

```
CRR .( include eforth )
FLOAD cEFa_23.f
H forth_@
```

`ceASMa_23.f, cdKERNa_23.f,` and `cEFa_23.f` files will be discussed in separate chapters. Finally, several system variables must be initialized so that the Forth interpreter can work properly on boot.

When ceForth boots up, the P register is initialized to 0 and WP to 4, so we have to have some valid byte code at this location. The ceForth boot up routine is the command `COLD`. Therefore, in memory location 0, we assemble `dolist, COLD` commands.

```
CRR
0 ORG
dolist, aanew
COLD
```

System variables are in the area between 0x90-0xAF. They contain vital information for the ceForth system to work properly. Only the following system variables have to be initialized:

```
$90 ORG $100                              #,
```

```
$94 ORG $10                                  #,
$98 ORG lastH forth_@                        #,
$9C ORG                                      #,
$A0 ORG lastH forth_@                        #,
$A4 ORG $INTERPRET
$A8 ORG QUIT
$AC ORG 0                                    #,
```

System Variable	Address	Initial Value	Function
'TIB	0x90	0	Pointer to Terminal Input Buffer.
BASE	0x94	0x10	Number base for hexadecimal numeric conversions.
CONTEXT	0a98	0x1F48	Pointer to name field of last command in dictionary.
CP	0x9C	0x1F70	Pointer to top of dictionary, first free memory location to add new commands. It is saved by "`h forth_@`" on top of the source code page.
LAST	0xA0	0x1F48	Pointer to name field of last command.
'EVAL	0xA4	0x153C	Execution vector of text interpreter, initialized to point to `$INTERPRET`. It may be changed to point to `$COMPILE` in compiling mode.
'ABORT	0xA8	0x167C	Pointer to `QUIT` command to handle error conditions.
tmp	0xAC	0	Scratch pad.

At last, write the contents of dictionary to `rom_23.h`.

```
write-mif-file
```

Simulate ceForth

I wrote a simulator for ceForth. It is in cefSIMa_23.f file. It is loaded next:

```
FLOAD cefSIMa_23.F
```

Now, you can simulate ceForth, by typing:

```
-1 G
```

And the simulator signs on:

```
C:\F#\C-EFORTH\ceforth_23\F#.exe    Current dir=C:\F#\C-EFORTH\ceforth_23
File  Edit  Tools  Help
 dm+ 1948 reDef dm+ DUMP 1998 reDef DUMP
 >NAME 1A20 reDef >NAME .ID 1A80 reDef .ID WORDS 1AAC reDef WORDS FORGET 1B48 reDef FORGET
 COLD 1BA0 reDef COLD
Structures
 THEN 1BDC reDef THEN FOR 1BF8 reDef FOR BEGIN 1C18 reDef BEGIN
 NEXT 1C30 reDef NEXT UNTIL 1C50 reDef UNTIL AGAIN 1C70 reDef AGAIN IF 1C8C reDef IF AHEAD 1CB8 reDef AHEAD
 REPEAT 1CE4 reDef REPEAT AFT 1CFC reDef AFT ELSE 1D20 reDef ELSE WHEN 1D40 WHILE 1D5C reDef WHILE
 ABORT" 1D78 reDef ABORT" $" 1D9C reDef $" ." 1DC0 reDef ."
 CODE 1DE8 reDef CODE CREATE 1E08 reDef CREATE VARIABLE 1E30 reDef VARIABLE CONSTANT 1E58 reDef CONSTANT


$ >
 0
 90
 94
 98
 9C
 A0
 A4
 A8
 AC
  Loading cefSIMa_23.F reDef BREAK reDef RESET
$ >
$ >
Version FIX 12SEP09CHT
$ >
$ > -1 G
Press any key to stop.

eForth in C, Ver 2.3, 2017
```

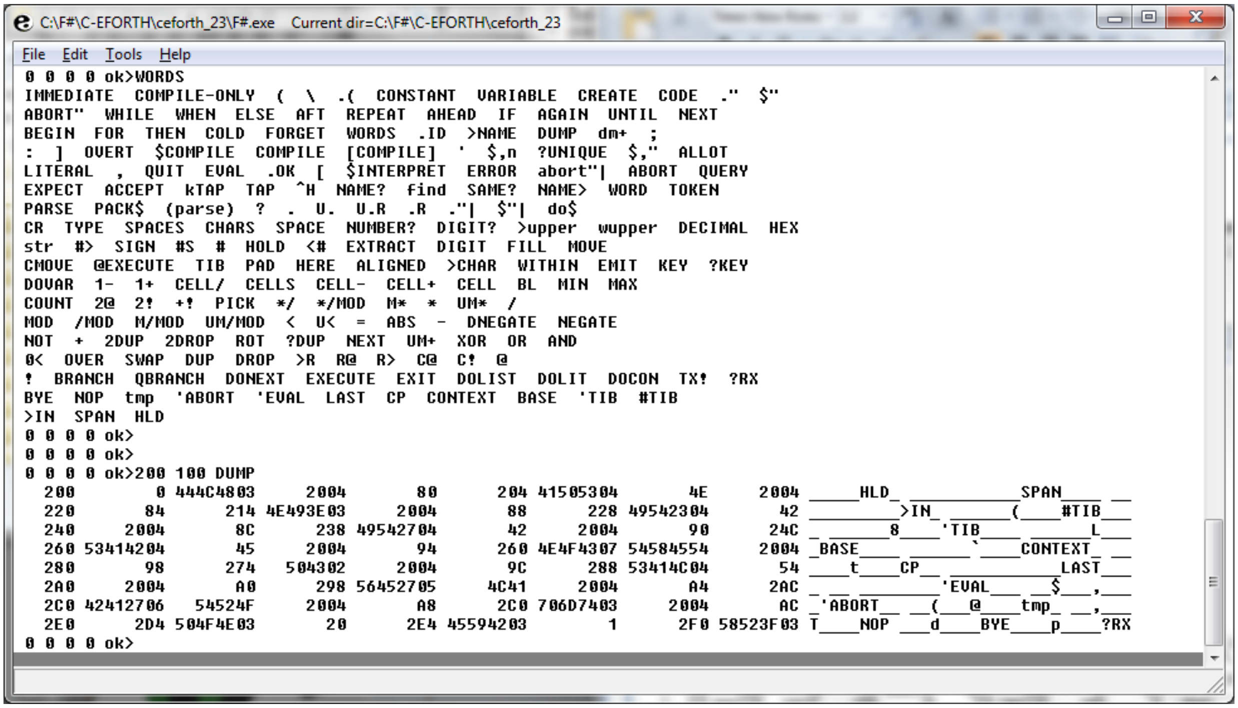
Type `WORDS` to display names of all Forth commands in ceForth:

```
C:\F#\C-EFORTH\ceforth_23\F#.exe    Current dir=C:\F#\C-EFORTH\ceforth_23
File  Edit  Tools  Help
$ >
Version FIX 12SEP09CHT
$ >
$ > -1 G
Press any key to stop.

eForth in C, Ver 2.3, 2017

 0 0 0 0 ok>
 0 0 0 0 ok>
 0 0 0 0 ok>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
 0<  OVER  SWAP  DUP  DROP  >R  R@  R>  C@  C!  @
 !  BRANCH  QBRANCH  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>
 0 0 0 0 ok>
```

Type 200 100 DUMP to dump the beginning of ceForth disctionary:



```
C:\F#\C-EFORTH\ceforth_23\F#.exe    Current dir=C:\F#\C-EFORTH\ceforth_23
File  Edit  Tools  Help
0 0 0 0 ok>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
0<  OVER  SWAP  DUP  DROP  >R  R@  R>  C@  C!  @
!  BRANCH  QBRANCH  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>
0 0 0 0 ok>
0 0 0 0 ok>200 100 DUMP
  200        0 444C4803     2004       80      204 41505304       4E     2004 _____HLD_ ___________SPAN____ __
  220       84      214 4E493E03     2004       88      228 49542304       42 _________>IN_ ______(____#TIB___
  240     2004       8C      238 49542704       42     2004       90      24C _ _______8____'TIB____ ______L___
  260 53414204       45     2004       94      260 4E4F4307 54584554     2004 _BASE____ _______`____CONTEXT_ __
  280       98      274   504302     2004       9C      288 53414C04       54 ____t____CP__ ____________LAST___
  2A0     2004       A0      298 56452705     4C41     2004       A4      2AC _ __ ________'EVAL___ __$___,___
  2C0 42412706   54524F     2004       A8      2C0 706D7403     2004       AC _'ABORT__ __(___@____tmp_ __,___
  2E0      2D4 504F4E03       20      2E4 45594203        1      2F0 58523F03 T____NOP ___d____BYE____p____?RX
0 0 0 0 ok>
```

In the ASCII dump on the right, you can see the names of the first few primitive commands. If you look carefully in the word dump on the left, you can probably identify the link fields, the name fields, and the codes field of these primitive commands.

This simulator reproduces very accurately what the ceForth system does on Windows. It is 99% assured that if ceForth works in the simulator, it would work in Windows. I relied on it heavily in getting ceForth to work.

The simulator will be discussed in details in a later Chapter 7.

Chapter 4. ceForth Assembler

Byte Code Assembler

The `cefASMa_23.f` file contains a byte code assembler for ceForth. It packs up to 4 byte code into one 32-bit program word. It first clears a program location pointed to by a variable "`hw`", initiating to 4 byte code of `NOP's`. Assembly commands are executed to insert byte code into consecutive bytes, from right to left. Unused bytes contain `NOP`. Assembly commands make necessary decisions as to whether to add more byte code to the current program word, or start a new program word.

ceForth uses 8-bit byte code in primitive commands. 4 Byte code are packed into one 32-bit word, and are executed from right to left, Byte code 1 to Byte code 4, as shown below:

31---------24	23---------16	15----------8	7------------0
Byte code 4	Byte code 3	Byte code 2	Byte code 1

Assembly commands for byte code are defined by a defining word `INST`. Defining words in Forth makes this byte code assembler very simple and very efficient.

The ceForth system is based on the Direct Threading Model, in which a primitive command has byte code in its code field and parameter field. To assemble a primitive command, the assembler first build a header, with a link field and a name field. After that, the assembler simply packs consecutive bytes with byte code until the primitive commands is completed.

`H ( -- a )` A variable pointing to the next free memory cell on top of the target dictionary.

```
VARIABLE H
```

`LASTH ( -- a )` A variable pointing to the name field of the current target command under construction.

```
variable lastH 0 lastH !                  \ init linkfield address lfa
```

`nameR! ( d -- )` Compile a 32-bit value, "d", in the name field of the current command in target dictionary.

```
: nameR! ( d -- )
   H @ RAM!                              \ store double to code buffer
   4 H +!                              \ bump nameH
   ;
```

`COMPILE-ONLY ( -- )` Patch Bit 6 in first byte of name field in current target command. Text interpreter checks it to avoid executing compiler commands.

```
: compile-only 40 lastH @ RAM@ XOR lastH @ RAM! ;
```

`IMMEDIATE ( -- )` Patch Bit 7 in first byte of name field in current target command. Compiler checks it to execute commands while compiling.

```
: IMMEDIATE    80 lastH @ RAM@ XOR lastH @ RAM! ;
```

`Hw ( -- a )` A variable pointing to a new program word being constructed.
`Hi ( -- a )` A variable pointing to a byte to pack the next byte code.
`Bi ( -- a )` A variable pointing to a byte to pack the next ASCII character.

```
VARIABLE Hi   VARIABLE Hw VARIABLE Bi ( for packing)
```

`ALIGN ( -- )` Initialize pointer "`Hi`" to start assembling a new program word.

```
: ALIGN   10 Hi ! ;
```

`ORG ( a -- )` Initialize pointer "`H`" to a new address to start assembling.

```
: ORG   DUP . CR H !  ALIGN ;
```

`mask ( -- a )` An array of 4 masks to isolate one 8-bit byte code from a 32-bit word. A byte code can be assembled in one of 4 bytes selected by "`Hi`".

```
CREATE mask  FF , FF00 , FF0000 , FF000000 ,
```

`#, ( d -- )` Compile "d" to top of target dictionary. It is the most basic assembler and compiler. The ceForth assembler is an extension of this basic assembly command.

```
: #,   ( d ) H @ RAM!  4 H +! ;
```

`,W ( d -- )` OR "d" to the program word pointed to by "`Hw`". It generally fills the address field in the current program word.

```
: ,w   ( d ) Hw @ RAM@  OR  Hw @ RAM! ;
```

`,I ( d -- )` Use "Hi" to select one machine instruction in "d" and assemble it into the program word selected by "`Hw`".

```
: ,I   ( d ) Hi @ 10 =  IF  0 Hi !  H @ Hw !  0 #,  THEN
             Hi @ mask + @ AND  ,w  4 Hi +! ;
```

`SPREAD ( b – d )` Repeat 8-bit byte code "b" in all 4 bytes to form a 32-bit word. "`mask`" uses it to select a byte for assembling.

```
: spread ( b - d ) DUP 100 * DUP 100 * DUP 100 * + + + ;
```

`,B ( b -- )` Pack byte "b" into current program word. Pointer "`Bi`" determines which byte field to pack.

```
: ,B   ( c ) Bi @ 0 = IF 1 Bi ! H @ Hw ! 0 #, ,w EXIT THEN
             Bi @ 1 = IF 2 Bi ! 100 * ,w EXIT THEN
             Bi @ 2 = IF 3 Bi ! 10000 * ,w EXIT THEN
             0 Bi ! 1000000 * ,w ;
```

`INST ( b -- )` Define a set of byte code assembly commands. It creates a byte code assembly command like a constant. When a byte code assembly command is later executed, this byte "b" is retrieved and a byte code is assembled into the current program word by command "`,I`".

```
: INST CONSTANT DOES> R> @ spread ,I ;
```

INST is a defining word, which is used to create a class of commands. These commands then assemble byte code in the code field of a primitive command in the target dictionary. INST takes a byte code number on data stack and creates a CONSTANT first. However, when this new command is executed, it retrieves the constant value, and actually assembles a byte code on top of target dictionary, which is allocated to a new code field for a new primitive command in target dictionary. This is accomplished by the commands R> @ spread ,I, between DOES> and ":". In fact, INST defines the ceForth assembler.

nop, (--) First byte code assembly command defined by "INST".

```
0 INST nop,
```

aanew (--) Fill current program word with nop, and initialize Hi and hw to assemble new byte code in the next program word.

```
: aanew BEGIN Hi @ 10 < WHILE nop, REPEAT 0 Bi ! ;
```

begin (– a) Mark the current top of dictionary. This is where new tokens will be compiled later.

```
: begin aanew H @ ;
```

ceForth Byte Code

Here are all the byte code assembly commands to be used to assemble byte code in primitive commands to build ceForth kernel. They are all created by INST.

```
decimal
\ 0 INST nop,
1 INST bye,      2 INST qrx,    3 INST txsto,  4 INST docon,
5 INST dolit,    6 INST dolist, 7 INST exit,   8 INST execu,
9 INST donext,  10 INST qbran, 11 INST bran,  12 INST store,
13 INST at,     14 INST cstor, 15 INST cat,   16 INST rpat,
17 INST rpsto , 18 INST rfrom, 19 INST rat,   20 INST tor,
21 INST spat  , 22 INST spsto, 23 INST drop,  24 INST dup,
25 INST swap,   26 INST over,  27 INST zless, 28 INST andd,
29 INST orr,    30 INST xorr,  31 INST uplus, 32 INST next,
33 INST qdup,   34 INST rot,   35 INST ddrop, 36 INST ddup,
37 INST plus,   38 INST inver, 39 INST negat, 40 INST dnega,
41 INST subb,   42 INST abss,  43 INST equal, 44 INST uless,
45 INST less,   46 INST ummod, 47 INST msmod, 48 INST slmod,
49 INST mod,    50 INST slash, 51 INST umsta, 52 INST star,
53 INST mstar,  54 INST ssmod, 55 INST stasl, 56 INST pick,
57 INST pstor,  58 INST dstor, 59 INST dat,   60 INST count,
61 INST dovar,  62 INST max,   63 INST min,
```

Command Headers

In ceForth, all commands are compiled in a target dictionary, and linked as a list. Each command has a link field of one 32-bit word, a variable length name field in which the first byte contains a length followed by the ASCII code of the name, null filled to a 32-bit word boundary, a one word code field, and a variable-length parameter field containing 32-bit tokens or byte code. A primitive command has byte code in its code field. A compound command has a `dolist`, byte code in its code field, and a token list in its parameter field. Here are commands to build headers, which include link and name fields.

`(makehead) ( -- )` Build a header for a new target command. The header includes a link field and a name field. The address of the name field in the last target command is stored in "`lasth`", and is compiled into the link field. "`H`" points to the name field of the new command, and is copied into "`lasth`". Now, the following string is packed into the name field, starting with its length byte, and null filled to the word boundary. Now, "`H`" points to the code field of this new target command.

```
: (makeHead)
   aanew 20 word                        \ get name of new definition
   lastH @ nameR!                       \ fill link field of last word
   H @ lastH !                          \ save nfa in lastH
   DUP c@ ,B                            \ store count
   count FOR AFT
      count ,B                          \ fill name field
   THEN NEXT
   DROP aanew
   ;
```

`makehead ( -- )` Build a header with (`makehead`) and save the name string to define a compiler command in metacompiler. It displays the name and code field address. A string can be used repeatedly by saving and restoring its pointer in "`>IN`".

```
: makeHead
   >IN @ >R                             \ save interpreter pointer
   (makeHead)
   R> >IN !                             \ restore word pointer
   ;
```

`$LIT ( -- )` Compile a packed string for a string literal inside a token list. It works similarly as (`makehead`). However, the name string is delimited by space character (ASCII 0x20), while a string literal is delimited by a double-quote character (ASCII 0x22).

```
: $LIT ( -- )
   aanew 22 WORD
   DUP c@ ,B ( compile count )
   count FOR AFT
      count ,B ( compile characters )
   THEN NEXT    DROP aanew ;
```

Compilers for Primitive and Compound Commands

We are now at the peak of our metacompiler. We had built all the tools to compile new commands into the target dictionary, which will eventually run ceForth. All commands have a link field and a name field. Primitive commands have an additional code field. Compound commands have a code field and a parameter field. Two defining commands are now created to build the primitive and compound commands. `CODE` creates a headers for primitive commands, and its following code field can now be packed with byte code. `:: (colon-colon)` creates a header for a compound command, and its following parameter field can be stuffed with a token list.

`CODE ( -- )` Create a new primitive command in ceForth target dictionary. It creates a new header with a link field and a name field, and is ready to assemble byte code in the following code field. It also creates an assembly command in the metacompiler, storing its code field address. When this assembly command is encountered by metacompiler, it compiles its code field address as a token to extend the token list currently under construction.

```
: CODE makeHead begin .head CONSTANT  DOES> R> @ #, ;
```

`:: (colon-colon, -- )` Create a new compound command in ceForth target dictionary. It creates a link field and a name field, and then add a byte code `dolist`, in the code field. Now, a token list can be built in its parameter field, to become a new compound command in target dictionary. It also creates an assembly command in the metacompiler, storing its code field address. When this assembly command is encountered by metacompiler, it compiles its code field address as a token to extend the token list currently under construction.

```
: ::   makeHead begin .head CONSTANT dolist, aanew DOES> R> @
#, ;
```

Defining word is an advanced topic in Forth. A defining word is used to create a class of new commands. The command DOES> separates the compiling behavior of the defining word, and the run time behavior of the new commands it later creates. Another way to look at it, a defining word can be specified:

```
:  <Defining word>  <Compiler>  DOES>   <Interpreter>  ;
```

In `CODE`, it compiles a new command by making its header by `makeHead`, obtaining its code field address by `begin`, printing its name and code field address with `.head`, and finally create a constant with the code field address by `CONSTANT`. However, when the new command thus created is later executed, top of return stack is pointing at its parameter field, where the code field address was stored. Retrieve this cfa by `R>,` fetch it contents by `@`, and then add this address to the current token list by “`#,`”.

In `:: (colon-colon),` a new compound command is created much like `CODE` first as a `CONSTANT` with its cfa. Now, a byte code `dolist,` is assembled into the new code field. The parameter field is open for a new token list. All commands following will be compiled as tokens. The new compound command thus created, just like the primitive commands created by `CODE`, will add its code field address to the token list on top of the target dictionary.

Clear as mud? There are 4 levels of Forth expertise:
Level 1: Type a list of commands and get them executed.
Level 2: Create a new compound command to replace a list of existing commands.
Level 3: Create a defining word to create a class of similarly behaving commands.
Level 4: Metacompilation.

You are now between Level 3 and Level 4. Do not be discouraged if you do not understand everything. I am trying to show you a working Level 4 system. There is nothing beyond it. There is no Level 5.

Chapter 5. ceForth Primitive Commands

The kernel of ceForth_23 is defined in file `cefKERNa_23.f`. It is a collection of primitive commands, executing byte code. The byte code it refers to are C function defined in `ceForth_23.cpp`.

In Forth dictionary, each command has a record containing 4 fields: a link field pointing to the name field of its prior command, a name field containing ASCII characters for the name of this command, a code field pointing to executable byte code of this command, and a parameter field containing data used by this command. There are two types of commands used in eForth: low level primitive commands whose parameter field contains more byte code; and high level compound commands whose parameter fields contain tokens lists. In this ceForth_23 implementation, a token is the code field address of a command in dictionary. Tokens are 4 bytes in length. The length of a parameter field varies depending upon the complexity of the command.

In the code field of a primitive command, there are 4 byte code in the code field. More byte code may spill over into the following parameter field. A byte code sequence must be terminated by a special byte code named `next`(). The function of `next`() is to fetch the token pointed to by the Interpreter Pointer IP, increment IP to point to the next token in a token list, and execute the token just fetched. Since a token points to a code field containing executable byte code, executing a token means jumping indirectly to the code field pointed to by the token. `next`() thus allows the Virtual Forth Engine to execute a token list with very little CPU overhead. In ceForth_23, `next`() is defined as:

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

In a compound command, the code field contains one byte code (6) which executes a function `dolist`(),which processes the token list in the parameter field following the code field. `dolist`() pushes the contents in IP onto the return stack, copies the address of the first token in its code field into IP and then calls `next`(). `next`() then executes the token list in the parameter field:

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

The last token in the token list of a compound command must be a primitive command `EXIT`. It executes `exitt`(), and thus undoes what `dolist`() accomplished. `exitt`() pops the top item on the return stack into the IP register. Consequently, IP points to the token following the compound command just executed. `exitt`() then invokes next() which continues processing of the calling token list, briefly interrupted by the last compound command in this token list.

```
CODE EXIT exitt, next,
void exitt(void) { IP = (cell) rack[(unsigned char)R--]; next(); }
```

`next`() is the inner interpreter of primitive words, and `dolist`() is the interpreter of compound command and are often referred to as an address interpreter. They are the foundation of a Virtual Forth Engine.

The collection of primitive commands is generally called the kernel of a Forth system. These primitive commands contain byte code, which cause C function in the Virtual Forth Machine to be executed to perform atomic operations by the host computer.

In the kernel of ceForth_23, most primitive commands have one byte code followed by the inner interpreter `next()`, which has a byte code of 0x20. However, primitive commands may have one byte code, or as many byte code as necessary. The assembler in ceForth_23 uses a special command `COLD` to construct the header of a primitive command, which includes a link field and a name field. After the header is constructed, byte code are added at will. The names of assembly instructions are simply the names of the corresponding C functions, terminated with a , (comma) character.

To show you clearly the relationship between a primitive command and the corresponding C function in VFM, I will put them together, as much as I can. Hopefully, you can bridge the gap between a VFM and a Forth machine that you can interact with.

System Variables

A set of system variables are implemented as constants pointing to specific addresses in the variable area, allocated in the beginning of the dictionary.

`HLD` (-- a) Return a pointer to a buffer below PAD to build a numeric output string.

```
CODE HLD      80 docon, next, #,       \ scratch
```

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

```
CODE SPAN     84 docon, next, #,       \ #chars input by EXPECT
```

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

```
CODE >IN      88 docon, next, #,       \ input buffer offset
```

`#TIB` (-- a) Hold the current character count in terminal input buffer. Terminal Input Buffer is in the next variable 'TIB.

```
CODE #TIB     8C docon, next, #,       \ #chars in the input buffer
```

`'TIB` (-- a) Hold the current address of terminal input buffer.

```
CODE 'TIB     90 docon, next, #,       \ ptr to TIB
```

`BASE` (-- a) Store the current radix base for numeric I/O. Default to 0x10.

```
CODE BASE     94 docon, next, #,       \ number base
```

`CONTEXT` (-- a) Return a pointer to the last name field in Forth dictionary.

```
CODE CONTEXT  98 docon, next, #,       \ first search vocabulary
```

`CP` (-- a) Point to the top of the Forth dictionary. New commands are compiled here.

```
CODE CP       9C docon, next, #,       \ dictionary code pointer
```

LAST (-- a) Return a pointer to the last name field in Forth dictionary. It is updated only when a valid new command is compiled successfully.

```
CODE LAST     A0 docon, next, #,       \ ptr to last name compiled
```

'EVAL (-- a) Hold the execution vector for EVAL. Default to $INTERPRET while interpreting commands. It is changed to $COMPILE while compiling a new compound command.

```
CODE 'EVAL    A4 docon, next, #,       \ interpret/compile vector
```

'ABORT (-- a) Hold the execution vector for error handler. Default to QUIT.

```
CODE 'ABORT   A8 docon, next, #,       \ ptr to error handler
```

tmp (-- a) A temporary storage location used in parsing and searching commands.

```
CODE tmp      AC docon, next, #,       \ ptr to converted # string
```

System Interface Commands

NOP (--) No operation.

```
CODE NOP next,
void nop(void)
{     next(); }
```

BYE (--) Return to Windows.

```
CODE BYE bye,
void bye(void)
{     exit(0); }
```

?RX (-- c T | F) Return input character and true, or a false if no input.received.

```
CODE ?RX qrx, next,
void qrx(void)
{     push(long) getchar();
      if (top != 0) push TRUE; }
```

TX! (c --) Send a character to the serial terminal.

```
CODE TX! txsto, next,
void txsto(void)
{     putchar((char)top);
      pop; }
```

Inner Interpreter

next() is the inner interpreter of the Virtual Forth Machine. Execute the next token in a token list.

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

NOP (--) No operation.

```
CODE NOP next,
void nop(void) { jump(); }
```

`DOCON` (-- n) Return integer stores in the next cell in current token list.

```
CODE DOCON docon, next,
void next(void)
{     P = data[IP>>2];
      WP = P + 4;
      IP += 4; }
```

`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.

```
CODE DOLIT dolit, next,
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. `Dolist,` is in the code field of all compound commands. The token list in a compound command must be terminated by `EXIT`.

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

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

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

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

```
CODE EXECUTE execu, next,
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. `DONEXT` is compiled by `NEXT`.

```
CODE DONEXT donext, next,
void donext(void)
{     if (rack[(char)R]) {
            rack[(char)R] -= 1;
            IP = data[IP>>2]; }
```

```
      else {IP += 4;
            R--; }
      next(); }
```

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.

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

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

```
CODE BRANCH bran, next,
void bran(void)
{     IP = data[IP>>2];
      next(); }
```

Memory Access Commands

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

```
CODE ! store, next,
void store(void)
{     data[top>>2] = stack[(char) S--];
      pop; }
```

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

```
CODE @ at, next,
void at(void)
{     top = data[top>>2]; }
```

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

```
CODE C! cstor, next,
void cstor(void)
{     cData[top] = (char)stack[(char) S--];
      pop; }
```

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

```
CODE C@ cat, next,
void cat(void)
{     top = (long)cData[top]; }
```

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

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

`2!` (d a --) Store the double integer to address a.
```
CODE 2! dstor, next,
void dstor(void)
{     data[(top>>2) + 1] = stack[(char) S--];
      data[top>>2] = stack[(char) S--];
      pop; }
```

`2@` (a -- d) Fetch double integer from address a.
```
CODE 2@ dat, next,
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.
```
CODE COUNT count, next,
void count(void)
{     stack[(char) ++S] = top + 1;
      top = cData[top]; }
```

Stack Commands

`>R` (n --) Pop a number off the data stack and pushes it on the return stack.
```
CODE R> rfrom, next,
void tor(void)
{     rack[(char)++R] = top;
      pop; }
```

`R@` (-- n) Copy a number off the return stack and pushes it on the return stack.
```
CODE R@ rat, next,
void rat(void)
{     push rack[(char)R]; }
```

`>R`(-- n) Pop a number off the return stack and pushes it on the data stack.
```
CODE >R tor, next,
void tor(void)
{     rack[(char)++R] = top;
      pop; }
```

`DROP` (w --) Discard top stack item.
```
CODE DROP drop, next,
void drop(void)
{     pop; }
```

`DUP` (w -- w w) Duplicate the top stack item.
```
CODE DUP dup, next,
void dup(void)
{     stack[(char) ++S] = top; }
```

`SWAP` (w1 w2 -- w2 w1) Exchange top two stack items.
```
CODE SWAP swap, next,
```

```
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.

```
CODE OVER over, next,
void over(void)
{     push stack[(char) S - 1]; }
```

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

```
CODE ?DUP qdup, next,
void qdup(void)
{     if (top) stack[(char) ++S] = top; }
```

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

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

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

```
CODE 2DROP ddrop, next,
void ddrop(void)
{     drop(); drop(); }
```

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

```
CODE 2DUP ddup, next,
void ddup(void)
{     over(); over(); }
```

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

```
CODE PICK pick, next,
void pick(void)
{     top = stack[(char) S - (char)top]; }
```

Logic Commands

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.

```
CODE 0< zless, next,
void zless(void)
{     top = (top < 0) LOGICAL; }
```

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

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

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

```
CODE U< uless, next,
void uless(void)
{     top = LOWER(stack[(char) S], top) LOGICAL; (char) S--; }
```

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

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

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

```
CODE NOT inver, next,
void inver(void)
{     top = -top - 1; }
```

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

```
CODE AND andd, next,
void andd(void)
{     top &= stack[(char) S--]; }
```

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

```
CODE OR orr, next,
void orr(void)
{     top |= stack[(char) S--]; }
```

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

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

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

```
CODE MAX max, next,
void max(void)
{     if (top < stack[(char) S]) pop;
      else (char) S--; }
```

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

```
CODE MIN min, next,
void min(void)
{     if (top < stack[(char) S]) (char) S--;
      else pop; }
```

Math Commands

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

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

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

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

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

```
CODE NEGATE negat, next,
void negat(void)
{     top = 0 - top; }
```

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

```
CODE DNEGATE dnega, next,
void dnega(void)
{     inver();
      tor();
      inver();
      push 1;
      uplus();
      rfrom();
      plus(); }
```

`-` (n1 n2 -- n1-n2) Subtraction.

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

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

```
CODE ABS abss, next,
void abss(void)
{     if (top < 0)
            top = -top; }
```

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

```
CODE UM/MOD ummod, next,
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); }
```

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

```
CODE M/MOD msmod, next,
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); }
```

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

```
CODE /MOD slmod, next,
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.

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

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

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

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

```
CODE UM* umsta, next,
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; }
```

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

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

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

```
CODE M* mstar, next,
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; }
```

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

```
CODE */MOD ssmod, next,
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 << 32;
      pop;
      top = (signed long)(n / d);
      stack[(char) S] = (signed long)(n%d); }
```

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

```
CODE */ stasl, next,
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 << 32;
      pop; pop;
      top = (signed long)(n / d); }
```

Miscellaneous Commands

`BL` (– 0x20) Push blank character on stack.

```
CODE BL 20 docon, next, #,
```

`CELL` (– 4) Push cell size on stack.

```
CODE CELL 4 docon, next, #,
```

`CELL+` (a -- a) Add cell size in byte to address.

```
CODE CELL+ 4 docon, plus, next, #,
void cellp(void) { top += 4; }
```

`CELL-` (a -- a) Subtract cell size in byte from address.

```
CODE CELL- 4 docon, subb, next, #,
void cellm(void) { top -= 4; }
```

`CELLS` (n -- n) Multiply top by cell size in bytes.

```
CODE CELLS 4 docon, star, next, #,
void cells(void) { top *= 4; }
```

`CELL/` (n -- n) Divide top by cell size in bytes.

```
CODE CELL/ 4 docon, slash, next, #,
void cellsl(void) { top /= 4; }
```

`1+` (a – a+1) Increment top.

```
CODE 1+ 1 docon, plus, next, #,
```

`1-` (a – a-1) Decrement top.

```
CODE 1- 1 docon, subb, next, #,
```

`DOVAR`(-- a) Return address of the next cell in current token list

```
CODE DOVAR dovar, next,
void dovar(void)
{     push WP; }
```

Control Structure Commands

Above we have all the primitive commands in the kernel of ceForth. Many of the primitive commands are not used in programming, but are used to construct branching and looping control structures in token lists of compound commands. They compile integer and address literals in token lists, and resolve the address fields in address literals. They allow Forth to incorporate nested structures in simple linear token lists to solved complex problems.

`;; ( -- )` Terminate a token list in compound commands by compiling the token `EXIT`.

```
: ;; EXIT ;
```

`BEGIN` Mark current location in target for later address resolution.

```
: BEGIN ( -- a ) begin ;
```

`AGAIN` Terminate a begin-again loop, and assemble a bra instruction to "`BEGIN`".

```
: AGAIN ( a -- ) BRANCH #, ;
```

`UNTIL` Terminate a begin-until loop if top is not cleared.

```
: UNTIL ( a -- ) QBRANCH #, ;
```

`IF` Start a conditional branch structure. Assemble a `QBRANCH` instruction.

```
: IF ( -- a ) QBRANCH BEGIN 0 #, ;
```

`ELSE` Resolve branch instruction at "`IF`", and start a branch structure. Assemble a `BRANCH` instruction.

```
: ELSE ( a1 -- a2 ) BRANCH BEGIN 0 #, forth_swap
      BEGIN forth_swap RAM! ;
```

`THEN` Terminate a conditional branch structure by resolving the branch instruction at "`IF`" or "`ELSE`".

```
: THEN ( a -- ) BEGIN forth_swap RAM! ;
```

`WHILE` Start a conditional branch structure in a begin-while-repeat loop. Assemble a `QBRANCH` instruction.

```
: WHILE ( a1 -- a2 a1 ) IF forth_swap ;
```

`REPEAT` Terminate a begin-while-repeat loop, and assemble a bra instruction to "begin".

```
: REPEAT ( a -- ) BRANCH #, THEN ;
```

`AFT` Branch the THEN in a FOR-NEXT loop to skip this branch the first time through the loop.

```
: AFT ( a1 -- a3 a2 ) forth_drop BRANCH BEGIN 0 #, BEGIN forth_swap ;
```

`FOR` Start a finite FOR-NEXT loop.

```
: FOR ( -- a ) >R BEGIN ;
```

`NEXT` Terminate a FOR-NEXT loop.

```
: NEXT ( a -- ) DONEXT #, ;
```

`LIT` Compile an integer literal in a token list..

```
: LIT ( n -- ) DOLIT #, ;
```

Chapter 6. ceForth Compound Commands

The dictionary of ceForth system contains records of all Forth commands. The low level primitive commands are discussed in the Kernel section. The high level compound commands are discussed here. All compound commands are defined in the file `cEFa_23.f`. They are discussed in their loading order. The loading order is very important in the ceForth metacompiler, because forward referencing is not allowed. All assembling and compiling processes are accomplished in a single pass.

ceForth metacompiler behaves very similar to a regular Forth system. However, to compile primitive commands into a target dictionary, the command `CODE` was changed to accomplish this goal. To compile high level compound commands to the target dictionary, as we do in this `cEFa_23.f` file, a new set of commands `:: (colon-colon)` and `:: (semicolon-semicolon)` are used instead of the Forth commands `: (colon)` and `; (semicolon)`. Unlike `: (colon)`, `:: (colon-colon)` does not change to a compiling state, and the metacompiler remains in the interpretive state throughout. New commands defined by the metacompiler would just add new tokens to the target dictionary.

Control structure commands like `IF, ELSE, THEN, BEGIN, WHILE, REPEAT`, etc, are all retained in the metacompiler so they can construct control structures properly in target dictionary. The only exception is the handling of literals. An integer encountered by metacompiler would remain on data stack. If you intended to compiler it as a literal in target dictionary, you would have to use the special command `LIT`. If you are familiar with Forth language, you would notice that the compound commands in `cEFa_23`.f read identically like regular Forth code, except that integer literals have to be handled explicitly.

Common Commands

`?KEY` (-- c T| F) Inspect the terminal device and returns a character and a true flag if the character has been received and is waiting to be retrieved. If no character was received, `?KEY` simply returns a false flag.

```
:: ?KEY  ?RX ;;
```

`KEY` (-- c) Wait until a character is received from the terminal device and returns its ASCII code.

```
:: KEY   BEGIN ?RX UNTIL ;;
```

`EMIT` (c --) Send a character on data stack to the terminal device.

```
:: EMIT  TX! ;;
```

`WITHIN` checks whether the third item on the data stack is within the range as specified by 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 data stack. Otherwise, a false flag is returned. All numbers are assumed to be unsigned integers.

```
:: WITHIN ( u ul uh -- t ) \ ul <= u < uh
  OVER - >R - R> U< ;;
```

`>CHAR` is very important in converting a non-printable character to a harmless 'underscore' character (ASCII 95). As eForth is designed to communicate with you through a serial I/O device, it is important that eForth 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`.

```
:: >CHAR ( c -- c )
  $7F LIT AND DUP $7F LIT BL WITHIN
  IF DROP ( CHAR _ ) $5F LIT THEN ;;
```

`ALIGNED` changes the address to the next cell boundary so that it can be used to address 32 bit word in memory.

```
:: ALIGNED ( b -- a ) 3 LIT + FFFFFFFC LIT AND ;;
```

`HERE` returns the address of the first free location above the code dictionary, where new commands are compiled.

```
:: HERE ( -- a ) CP @ ;;
```

`PAD` returns the address of the text buffer where numbers are constructed and text strings are stored temporarily.

```
:: PAD ( -- a ) HERE 50 LIT + ;;
```

`TIB` returns the terminal input buffer where input text string is held.

```
:: TIB ( -- a ) 'TIB @ ;;
```

`@EXECUTE` is a special command supporting the vectored execution commands in eForth. It fetches the code field address of a token and executes the token.

```
:: @EXECUTE ( a -- ) @ ?DUP IF EXECUTE THEN ;;
```

`CMOVE` copies a memory array from one location to another. It copies one byte at a time.

```
:: CMOVE ( b b u -- )
  FOR AFT OVER c@ OVER c! >R 1+ R> 1+ THEN NEXT 2DROP ;;
```

`MOVE` copies a memory array from one location to another. It copies one word at a time.

```
:: MOVE ( b b u -- )
  CELL/ FOR AFT OVER @ OVER ! >R CELL+ R> CELL+ THEN NEXT 2DROP ;;
```

FILL fills a memory array with the same byte.

```
:: FILL ( b u c -- )
  SWAP FOR SWAP AFT 2DUP c! 1+ THEN NEXT 2DROP ;;
```

Numeric Output

DIGIT converts an integer to an ASCII digit.

```
:: DIGIT ( u -- c )
  9 LIT OVER < 7 LIT AND +
  ( CHAR 0 ) 30 LIT + ;;
```

`EXTRACT` extracts the least significant digit from a number n. n is divided by the radix in `BASE` and returned on the stack.

```
:: EXTRACT ( n base -- n c )
  0 LIT SWAP UM/MOD SWAP DIGIT ;;
```

`<#` 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.

```
:: <# ( -- ) PAD HLD ! ;;
```

`HOLD` 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.

```
:: HOLD ( c -- ) HLD @ 1- DUP HLD ! C!  ;;
```

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

```
:: # ( u -- u ) BASE @ EXTRACT HOLD ;;
```

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

```
:: #S ( u -- 0 ) BEGIN # DUP WHILE REPEAT ;;
```

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

```
:: SIGN ( n -- ) 0< IF ( CHAR - ) 2D LIT HOLD THEN ;;
```

`#>` terminates the numeric conversion and pushes the address and length of output numeric string on the parameter stack.

```
:: #> ( w -- b u ) DROP HLD @ PAD OVER - ;;
```

`str` converts a signed integer on the top of data stack to a numeric output string.

```
:: str ( n -- b u ) DUP >R ABS <# #S R> SIGN #> ;;
```

`HEX` sets numeric conversion radix in `BASE` to 16 for hexadecimal conversions.

```
:: HEX ( -- ) 10 LIT BASE ! ;;
```

`DECIMAL` sets numeric conversion radix in `BASE` to 10 for decimal conversions.

```
:: DECIMAL ( -- ) 0A LIT BASE ! ;;
```

Numeric Input

`wupper` converts 4 bytes in a word to upper case characters.

```
:: wupper ( w -- w' ) 5F5F5F5F LIT AND ;;
```

`>upper` converts a character to upper case.

```
:: >upper ( c -- UC )
  dup 61 LIT 7B LIT WITHIN IF 5F LIT AND THEN ;;
```

`DIGIT?` converts a digit to its numeric value according to the current base, and `NUMBER?` converts a number string to a single integer.

```
:: DIGIT? ( c base -- u t )
  >R ( CHAR 0 ) >upper 30 LIT - 9 LIT OVER <
  IF 7 LIT - DUP 0A LIT  < OR THEN DUP R> U< ;;
```

`NUMBER?` 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.

```
:: NUMBER? ( a -- n T | a F )
  BASE @ >R  0 LIT OVER COUNT ( a 0 b n)
  OVER c@ ( CHAR $ ) 24 LIT =
  IF HEX SWAP 1+ SWAP 1- THEN ( a 0 b' n')
  OVER c@ ( CHAR - ) 2D LIT = >R ( a 0 b n)
  SWAP R@ - SWAP R@ + ( a 0 b" n") ?DUP
  IF 1- ( a 0 b n)
    FOR DUP >R c@ BASE @ DIGIT?
      WHILE SWAP BASE @ * +  R> 1+
    NEXT DROP R@ ( b ?sign) IF NEGATE THEN SWAP
      ELSE R> R> ( b index) 2DROP ( digit number) 2DROP 0 LIT
      THEN DUP
  THEN R> ( n ?sign) 2DROP R> BASE ! ;;
```

Text Output

`SPACE` outputs a blank space character.

```
:: SPACE ( -- ) BL EMIT ;;
```

`CHARS` output n characters c.

```
:: CHARS ( +n c -- )
  SWAP 0 LIT MAX
  FOR AFT DUP EMIT THEN NEXT DROP ;;
```

`SPACES` output n blank space characters.

```
:: SPACES ( +n -- ) BL CHARS ;;
```

`TYPE` outputs n characters from a string in memory. Non ASCII characters are replaced by a underscore character.

```
:: TYPE ( B U -- )
  FOR AFT COUNT >CHAR EMIT THEN NEXT DROP ;;
```

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

```
:: CR ( -- ) ( =CR )
  0A LIT 0D LIT EMIT EMIT ;;
```

do$ retrieve 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 $"| and ."| use the command do$,

```
:: do$ ( -- $adr )
  R> R@ R> COUNT + ALIGNED >R SWAP >R ;;
```

$″| push the address of the following string on stack. Other commands can use this address to access data stored in this string. The string is a counted string. Its first byte is a byte count.

```
:: $"| ( -- a ) do$ ;;
```

″| displays the following string on stack. This is a very convenient way to send helping messages to you at run time.

```
:: ."| ( -- ) do$ COUNT TYPE ;;
```

.R 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.

```
::  .R ( n +n -- )
  >R str R> OVER - SPACES TYPE ;;
```

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

```
:: U.R ( u +n -- )
  >R <# #S #> R> OVER - SPACES TYPE ;;
```

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

```
:: U. ( u -- ) <# #S #> SPACE TYPE ;;
```

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

```
::  . ( n -- )
  BASE @ 0A LIT  XOR
  IF U. EXIT THEN str SPACE TYPE ;;
```

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

```
:: ? ( a -- ) @ . ;;
```

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.

```
:: (parse) ( b u c -- b u delta ; <string> )
  tmp c! OVER >R DUP \ b u u
  IF 1- tmp c@ BL =
    IF              \ b u' \ 'skip'
      FOR BL OVER c@ - 0< NOT
        WHILE 1+
      NEXT ( b) R> DROP 0 LIT DUP EXIT \ all delim
        THEN  R>
    THEN OVER SWAP  \ b' b' u' \ 'scan'
    FOR tmp c@ OVER c@ -  tmp c@ BL =
      IF 0< THEN WHILE 1+
    NEXT DUP >R
      ELSE R> DROP DUP 1+ >R
      THEN OVER -  R>  R> - EXIT
  THEN ( b u) OVER R> - ;;
```

`PACK$` 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.

```
:: PACK$ ( b u a -- a ) \ always word-aligned
  DUP >R
  2DUP + $FFFFFFFC LIT AND 0 LIT SWAP ! \ LAST WORD FILL 0 1ST
  2DUP C! 1+ SWAP CMOVE  R> ;;
```

`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.

```
:: PARSE ( c -- b u ; <string> )
  >R  TIB >IN @ +
  #TIB @ >IN @ -
  R> (parse) >IN +! ;;
```

`TOKEN` 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.

```
:: TOKEN ( -- a ;; <string> )
  BL PARSE $1F LIT MIN
  HERE CELL+          \ S D N
  PACK$  ;;
```

`WORD` 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.

```
:: WORD ( c -- a ; <string> )
  PARSE HERE CELL+ PACK$ ;; \ BM+
```

Dictionary Search

`NAME>` (nfa – cfa) Return a code field address from the name field address of a command.

```
:: NAME> ( a -- xt ) COUNT 1F LIT AND + ALIGNED ;;
```

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 1st word input string and a name. If these two words are the same, SAME? is called to compare the rest of two strings

```
:: SAME? ( a a u -- a a f \ -0+ )
 $1F LIT AND CELL/
 FOR AFT OVER R@ 4 LIT * + @ wupper
   OVER R@ 4 LIT * + @ wupper
   - ?DUP IF R> DROP EXIT THEN
 THEN NEXT
 0 LIT ;;
```

find (a va --cfa nfa, a F) searches the dictionary for a command. 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.

```
:: find ( a va -- xt na | a 0 )
  SWAP         \ va a
  DUP @ tmp !  \ va a  \ get cell count
  DUP @ >R     \ va a  \ #XOR --- count and 1st 3 char
  cell+ SWAP      \ a' va  a'=a(#XOR)+4
  BEGIN @ DUP  \ a' na na
    IF DUP @ $FFFFFF3F LIT AND wupper
      R@ wupper XOR \ ignore lexicon bits
      IF cell+ -1 LIT
      ELSE cell+ tmp @ SAME?
      THEN
    ELSE R> DROP SWAP cell- SWAP EXIT \ a 0
    THEN
  WHILE cell- cell-  \ a' la
  REPEAT R> DROP SWAP DROP
  cell- DUP NAME> SWAP ;;
:: NAME? ( a -- cfa na | a 0 )
  CONTEXT find ;;
```

Text Interpreter

The text interpreter interprets source text received from an input device and stored in the Terminal Input Buffer. To process characters in the Terminal Input Buffer, we need special commands to deal with the special conditions of backspace character and carriage return:

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

```
:: ^H ( b b b -- b b b ) \ backspace
  >R OVER R> SWAP OVER XOR
  IF ( =BkSp ) 8 LIT EMIT
     1-         BL EMIT \ distructive
     ( =BkSp ) 8 LIT EMIT \ backspace
```

```
  THEN ;;
```

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

```
:: TAP ( bot eot cur c -- bot eot cur )
  DUP EMIT OVER C! 1+ ;;
```

`kTAP ( bot eot cur c -- bot eot cur )` Processes character “c”. “bot” is the beginning of the input buffer, and “eot” is the end. “cur” points to the current character in the input buffer. “c” is normally stored at “cur”, which is incremented by 1. 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”.

```
:: kTAP ( bot eot cur c -- bot eot cur )
  DUP ( =Cr ) 0D LIT XOR
  OVER ( =Lf ) 0A LIT XOR AND
  IF ( =BkSp ) 8 LIT XOR
    IF BL TAP ELSE ^H THEN
    EXIT
  THEN DROP SWAP DROP DUP  ;;
```

`ACCEPT` (b u – b u) Accepts “u” characters into buffer at “b”, or until a carriage return. The value of “u” returned is the actual count of characters received.

```
:: ACCEPT ( b u -- b u )
  OVER + OVER
  BEGIN 2DUP XOR
  WHILE  KEY  DUP BL -  5F LIT U<
    IF TAP ELSE kTAP THEN
  REPEAT DROP  OVER -
 ;;
```

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

```
:: EXPECT ( b u -- ) ACCEPT SPAN ! DROP ;;
```

`QUERY( -- )` is the command 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`.

```
:: QUERY ( -- )
  TIB 50 LIT ACCEPT #TIB !
  DROP 0 LIT >IN ! ;;
```

`ABORT ( -- )` resets system and re-enters into the text interpreter loop `QUIT`. It actually executes `QUIT` stored in ‘`ABORT`. This avoids forward-referencing to `QUIT`, as `QUIT` is yet to be defined.

```
:: ABORT ( -- ) 'ABORT @EXECUTE ;;
```

`abort"|` (f --) A runtime string command 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.

```
:: abort" ( f -- )
```

```
  IF do$ COUNT TYPE ABORT THEN do$ DROP ;;
```

ERROR displays an error message at a with a ? mark, and ABORT.

```
:: ERROR ( a -- )
  space count type $3F LIT EMIT
  $1B LIT ( ESC) EMIT
  CR ABORT
```

$INTERPRET executes a command whose string address is on the stack. If the string is not a command, convert it to a number. If it is not a number, ABORT.

```
:: $INTERPRET ( a -- )
  NAME? ?DUP
  IF C@ $40 LIT AND
    abort" $LIT  compile only" ( ?even)  EXECUTE  EXIT
  THEN
  NUMBER? IF EXIT ELSE ERROR THEN
```

[(left-bracket) 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.

```
:: [ ( -- )  DOLIT $INTERPRET 'EVAL ! ;; IMMEDIATE
```

.OK used to be a command which displays the familiar 'ok' prompt after executing to the end of a line. In ceForth_23, 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.

```
:: .OK ( -- ) CR
  DOLIT $INTERPRET 'EVAL @ =
  IF >R >R >R DUP . R> DUP . R> DUP . R> DUP . ."| $LIT  fg>"
  THEN ;;
```

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.

```
:: EVAL ( -- )
  BEGIN TOKEN DUP @
  WHILE 'EVAL  @EXECUTE \ ?STACK
  REPEAT DROP .OK ;;
```

QUIT (--) is the operating system, or a shell, of the eForth system. It is an infinite loop eForth will never 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.

```
:: QUIT ( -- ) [ BEGIN QUERY EVAL AGAIN
```

Command Compiler

`, (comma)` 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 command currently under construction.

```
:: , ( w -- )  HERE DUP CELL+ CP ! ! ;;
```

`LITERAL` compiles an integer literal to the current compound command under construction. The integer literal is taken from the data stack, and is preceded by the token `DOLIT`. When this compound command 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`.

```
:: LITERAL ( n -- ) DOLIT DOLIT , , ;; IMMEDIATE
```

`ALLOT` 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 command`’,’(comma).`

```
:: ALLOT ( n -- ) ALIGNED CP +! ;;
```

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

```
:: $," ( -- ) ( CHAR " ) 22 LIT WORD COUNT + ALIGNED CP ! ;;
```

?`UNIQUE` is used to display a warning message to show that the name of a new command already existing in dictionary. eForth does not mind your reusing the same name for different commands. However, giving many commands 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.

```
:: ?UNIQUE ( a -- a )
  DUP NAME?
  ?DUP IF COUNT 1F LIT AND SPACE TYPE ."| $LIT  reDef "
  THEN DROP ;;
```

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

```
:: $,n ( a -- )
  DUP @ IF ?UNIQUE
    ( na ) DUP NAME> CP !
    ( na ) DUP LAST ! \ for OVERT
    ( na ) CELL-
    ( la ) CONTEXT @ SWAP ! EXIT
  THEN ERROR
```

`' (tick)` 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 displays an error message.

```
:: ' ( -- xt )
  TOKEN NAME? IF EXIT THEN
  ERROR
```

`[COMPILE]` acts similarly, except that it compiles the next command immediately. It causes the following command to be compiled, even if the following command is usually an immediate command which would otherwise be executed.

```
:: [COMPILE] ( -- ; <string> )
  '  , ;; IMMEDIATE
```

`COMPILE` is used in a compound command. 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.

```
:: COMPILE ( -- ) R> DUP @ , CELL+ >R ;;
```

$`COMPILE` builds the body of a new compound command. A complete compound command 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 commands are the most prevailing type of commands in eForth. In addition, eForth has a few other defining commands which create other types of new commands in the dictionary.

```
:: $COMPILE ( a -- )
  NAME? ?DUP
  IF @ $80 LIT AND
    IF EXECUTE
    ELSE ,
    THEN EXIT
  THEN
  NUMBER?
  IF LITERAL EXIT
  THEN ERROR
```

`OVERT` links a new command to the dictionary and thus makes it available for dictionary searches.

```
:: OVERT ( -- ) LAST @ CONTEXT ! ;;
```

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

```
:: ] ( -- ) DOLIT $COMPILE 'EVAL ! ;;
```

`: (colon)` creates a new header and start a new compound command. It takes the following string in the input stream to be the name of the new compound command, 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-bracket)` 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 command is terminated by `; (semi-colon),` which compiles an `EXIT` to terminate the token list, and executes `[ (left-bracket)` to turn the compiler back to text interpreter.

```
:: : ( -- ; <string> ) TOKEN $,n ] 6 LIT , ;;
```

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

```
:: ; ( -- ) DOLIT EXIT , [ OVERT  ;; IMMEDIATE
```

Debugging Tools

`dm+` 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.

```
:: dm+ ( b u -- b )
  OVER 6 LIT U.R
  FOR AFT DUP @ 9 LIT U.R CELL+
  THEN NEXT ;;
```

`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.

```
:: DUMP ( b u -- )
  BASE @ >R HEX  1F LIT + 20 LIT /
  FOR AFT CR 8 LIT 2DUP dm+
  >R SPACE CELLS TYPE R>
  THEN NEXT DROP R> BASE ! ;;
```

`>NAME( xt -- na | F )` finds the name field address na of a token from its code field address xt. If the token does not exist in the dictionary, it returns a false flag. `>NAME` is the mirror image of the command `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 get the name field address.

```
:: >NAME ( xt -- na | F )
  CONTEXT
  BEGIN @ DUP
  WHILE 2DUP NAME> XOR
    IF 1-
    ELSE SWAP DROP EXIT
    THEN
  REPEAT SWAP DROP ;;
```

`.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.

```
:: .ID ( a -- )
  COUNT $01F LIT AND TYPE SPACE ;;
```

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

```
:: WORDS ( -- )
  CR CONTEXT
  0 LIT TMP !
  BEGIN @ ?DUP
  WHILE DUP SPACE  .ID CELL-
    TMP @ 10 LIT <
    IF 1 LIT TMP +!
    ELSE CR 0 LIT TMP ! THEN
  REPEAT ;;
```

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

```
:: FORGET ( -- )
  TOKEN NAME? ?DUP
  IF CELL- DUP CP !
     @ DUP CONTEXT ! LAST !
     DROP EXIT
  THEN ERROR
```

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

```
:: COLD ( -- )
   CR ."| $LIT ceForth V4.3, 2017 " CR
   QUIT
```

Control Structures

This set of commands is now compiled into target dictionary, to be used by the ceForth system under Windows. They are very similar to the set of metacompiler commands defined in cefKERNa_23.f, which were used to build control structures in the target dictionary. If you understand the differences between these two sets of control structure commands with identical names, you have pretty much mastered the metacompiler.

Control structures in target dictionary are built with address literals BRANCH, QBRANCH, and DONEXT. The address fields following these tokens are resolved and filled in a single pass.

In following discussion, I will use two address symbols 'A' and 'a', in the stack pictures to indicate two types of addresses. 'A' is the address of an empty cell, which will receive an address when later resolved. 'a' is generally the address of the current token list, and this address will be used to resolve the address at 'A'.

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.

```
:: THEN ( A -- ) HERE SWAP ! ;; IMMEDIATE
```

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.

```
:: FOR ( -- a ) COMPILE >R HERE ;; IMMEDIATE
```

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.

```
:: BEGIN ( -- a ) HERE ;; IMMEDIATE
```

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

```
:: NEXT ( a -- )  COMPILE DONEXT , ;; IMMEDIATE
```

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

```
:: UNTIL ( a -- ) COMPILE QBRANCH , ;; IMMEDIATE
```

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

```
:: AGAIN ( a -- ) COMPILE BRANCH , ;; IMMEDIATE
```

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.

```
:: IF ( -- A )   COMPILE QBRANCH HERE 0 LIT , ;; IMMEDIATE
```

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.

```
:: AHEAD ( -- A ) COMPILE BRANCH HERE 0 LIT , ;; IMMEDIATE
```

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

```
:: REPEAT ( A a -- ) AGAIN THEN ;; IMMEDIATE
```

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.

```
:: AFT ( a -- a A ) DROP AHEAD HERE SWAP ;; IMMEDIATE
```

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.

```
:: ELSE ( A -- A )  AHEAD SWAP THEN ;; IMMEDIATE
```

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.

```
:: WHILE ( a -- A a )    IF SWAP ;; IMMEDIATE
```

String Literals

ABORT" (--) compiles an error message. This error message is display if top item on the stack is non-zero. The rest of the commands in the command is skipped and eForth resets to ABORT. If top of stack is 0, ABORT" skips over the error message and continue executing the following token list.

```
:: ABORT" ( -- ; <string> ) DOLIT abort" HERE ! $," ;; IMMEDIATE
```

$" (--) 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.

```
:: $"     ( -- ; <string> ) DOLIT $"|    HERE ! $," ;; IMMEDIATE
```

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

```
:: ."     ( -- ; <string> ) DOLIT ."|    HERE ! $," ;; IMMEDIATE
```

Defining Commands

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

```
:: CODE ( -- ; <string> ) TOKEN $,n OVERT align ;;
```

CREATE creates a new array without allocating memory. Memory is allocated using ALLOT.

```
:: CREATE ( -- ; <string> ) CODE $203D LIT , ;;
```

VARIABLE (--) creates a new variable, initialized to 0.

```
:: VARIABLE ( -- ; <string> ) CREATE 0 LIT , ;;
```

CONSTANT (n --) creates a new constant, initialized to the value on top of stack.

```
:: CONSTANT CODE $2004  LIT , , ;;
```

Immediate Commands

In Forth, immediate commands are executed even during compilation of compound commands. As I discussed before, the text interpreter has two states: interpreting state and compiling state. In interpreting state, all commands are executed. In compiling state, all commands are compiled to build new token lists, except immediate command, which are executed even in the compiling state. Immediate commands are used to build structures in token lists. All control structure commands are declared by IMMEDIATE command so they are executed in compiling state.

A few commands are used to insert comments into command list, and the text interpreter ignores these commands and their associated comment strings. However, once a comment command is defined, you cannot use it to insert comments anymore, because after the metacompiler redefines a comment command, executing the redefine commands would add another token to the token list, and would not function as a comment command. This is why these immediate commands must be defined at the very end of the metacompilation process.

`.( (dot-paren)` types the following string till the next). It is used to output text to the terminal.

```
(makeHead) .( ( -- ) dolist, aanew 29 LIT PARSE TYPE ;; IMMEDIATE
```

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

```
(makeHead) \ ( -- )  dolist, aanew $A LIT WORD DROP  ;; IMMEDIATE
```

`( ( paren)` ignores the following string till the next). It is used to place comments in source text.

```
(makeHead) ( dolist, aanew 29 LIT PARSE 2DROP ;;        IMMEDIATE
```

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

```
 (makeHead) COMPILE-ONLY dolist, aanew $40 LIT LAST @ +! ;;
```

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

```
 (makeHead) IMMEDIATE dolist, aanew $80 LIT LAST @ +! ;;
```

The entire ceForth target dictionary is built. As shown in `cefMETAa_23.f` file, the dictionary will be written to `rom_23.h` file, after the system variables are initialized correctly.

Chapter 7. ceForth Simulator

When I designed and built P series Forth chips, I wrote a simulator to simulate the logic functions of these CPU's, and to test the eForth system based on these chips. I modified eP32 for ceForth, and the simulator was carried over. The simulator was very helpful in developing and debugging of a CPU and its associated operating system. Once the simulator said `'ok'`, the whole system, hardware and software, worked.

After I changed the finite state machine in ceForth VFM, to a much simpler byte code sequencer, the simulator still behaved correctly, and did not need modification. Here I will show it in the finite state machine model.

The source code of this simulator is in `cefSIMa_23.F`. It is loaded at the end of `cefMETAa_23.F`, which builds an eForth dictionary in memory array "`RAM`". The simulator reads program words from this array and executes pseudo instructions contained in these program words.

An accurate and fast logic simulator is extremely valuable in designing and testing a new CPU. It is also very useful in separating hardware and software development, so that hardware and software can be developed simultaneously. I wrote a simulator for 32 bit eP32 chip and it served me well in the process of designing and testing eP32 CPU and its associated eForth system simultaneously.

Registers and Arrays

A large array, REGISTER, is opened to host all registers and stacks. It is divided in two banks: a FROM bank and a TO bank. The FROM bank contains current values of all registers and all stack elements. A pseudo instruction takes data in the FROM bank, modifies them, and writes updated data into the TO bank. The rising edge of the master clock copies the TO to the FROM bank, and thus simulates a pseudo instruction. Logic functions are performed by Forth words and update values from the FROM bank to the TO bank.

The clock in Finite State Machine, which fetches a program word from memory, and executes 4 pseudo instructions in this word, is simulated by a 32-bit counter. The least significant 3 bits in this counter steps through phases 0 to 5 in 6 clock cycles. Then this 3-bit field is cleared to zero and the upper 29-bit field is incremented. Therefore, the upper 29-bit field in this counter gives an accurate program word count.

First, we have to remove the metacompiler, and restore eForth so that it can now compile the simulator. "`forth_forget h`" truncates the eForth dictionary back to where "`H`" was defined. It thus deletes words defined in the metacompiler, assembler, kernel, and target commands. F# is cleaned to a pristine state to host a new application, which is the simulator.

```
forth_forget H
```

Command	Function

LIMIT	Limit stacks depths are 256 levels.
RANGE	Limit program size to 32kB, the size of the 'RAM' array
CLOCK	A variable that has a 29-bit program word count field and a 3-bit SLOT field. The SLOT field sequences program word fetch and execution of up to 5 instructions in the program word.
BREAK	A variable holding breakpoint address.
C+!	(b c --) Add c to the byte pointed to by b.
REGISTER	Base address of registers and stack arrays.

```
HEX
$1F CONSTANT LIMIT ( stack depth )
$7FFF CONSTANT RANGE ( program memory size in words )
VARIABLE CLOCK ( phase is in the last 3 bits )
VARIABLE BREAK

CREATE REGISTER $300 ALLOT
: C+! DUP >R C@ + R> C! ;

DECIMAL
REGISTER CONSTANT P
REGISTER 4 + CONSTANT I
REGISTER 8 + CONSTANT I1
REGISTER 9 + CONSTANT I2
REGISTER 10 + CONSTANT I3
REGISTER 11 + CONSTANT I4
REGISTER 13 + CONSTANT RP
REGISTER 14 + CONSTANT SP
REGISTER 16 + CONSTANT T
REGISTER 24 + CONSTANT IP
REGISTER 32 + CONSTANT WP
REGISTER $100 + CONSTANT RSTACK0
REGISTER $200 + CONSTANT SSTACK0
HEX
: RSTACK  RP C@ LIMIT AND 2 LSHIFT RSTACK0 + ;
: SSTACK  SP C@ LIMIT AND 2 LSHIFT SSTACK0 + ;
```

Register	Function
P	Program counter
I	Instruction latch
I1	Machine instruction in slot1
I2	Machine instruction in slot2
I3	Machine instruction in slot3
I4	Machine instruction in slot4
RP	Return stack pointer
SP	Data stack pointer
T	Accumulator, top item on data stack
IP	Interpreter pointer
WP	Scratch register

RSTACK0	Origin of return stack
SSTACK0	Origin of data stack
RSTACK	Address of top of return stack
SSTACK	Address of top of data stack

Virtual Forth Machine

The Finite State Machine paces the simulator through byte code stored in 'RAM' memory. Instead of using a single phase clock as master clock, we use a CLOCK variable as source of a multiple phase clock. The lowest three bits in CLOCK define 6 phases of the Finite State Machine. In phase 0, the next program word is fetched into Instruction latch I, and byte code in it are decoded and stored in registers I1-I4. In phase 1, byte code in I1 is executed. In phase 2, byte code in I2 is executed. Etc. In phase 5, the least significant 3 bits in CLOCK are set to 7. Then CLOCK is incremented and the least significant 3 bits are cleared to 0. Now we enter phase 0 again and fetch the next program word.

JUMP also set the least significant 3 bits in CLOCK to 7. JUMP is used by all transfer instructions to force the finite state machine to enter phase 0 on the rising edge of the next clock.

```
: CYCLE 1 CLOCK +! ;
: JUMP  CLOCK @ 7 OR CLOCK ! ;
: RPUSH ( n -- , push n on return stack )
        1 RP C+! RSTACK ! ;
: RPOPP ( -- n , pop n from return stack )
        RSTACK @ -1 RP C+! ;
: SPUSH ( n -- , push n on data stack )
        1 SP C+!
        T @ SSTACK !
        T ! ;
: SPOPP ( -- n , pop n from data stack )
        T @
        SSTACK @ T !
      -1 SP C+! ;
```

Command	Function
CYCLE	Simulate rising edge of master clock by incrementing CLOCK.
JUMP	Fetch next program word by forcing a 7 into Slot Counter in CLOCK. On the rising edge of the master clock, CLOCK is incremented and clears Slot Counter to 0. The upper 29-bit field in CLOCK is incremented, indicating that a new word is fetched from memory. Thus the upper 29 bits in CLOCK keeps an accurate count of eP32 words that have been executed.
RPUSH	Push double integer d on return stack.
RPOPP	Pop return stack and leave double integer on system stack.
SPUSH	Push double integer d on data stack.
SPOPP	Pop data stack and leave double integer on system stack.

`continue` simulates functions performed in phase 0 in the Finite State Machine, which fetches the next program word from memory and stores it in instruction register I. Byte code are extracted to I1 to I5, and used to perform functions assigned to them.

`continue` also increments the P register by 4, pointing to next program word to be executed.

To execute a pseudo instruction or a byte code, the simulator takes current values in registers and stacks in the FROM bank, computes desired new values, and deposits them back in registers and stacks in the TO bank. On the rising edge of the master clock, which is simulated by command CYCLE, the contents of the TO bank are copied to the FROM bank. Pseudo instructions are defined as Forth commands in this simulator, and they read values in the FROM bank, make necessary changes, and store new values in the TO bank.

```
: continue
        P @ RAM@ DUP I !
        100 /MOD SWAP I1 C!
        100 /MOD SWAP I2 C!
        100 /MOD SWAP I3 C!
        FF AND I4 C!
      4 P +! ;
```

`next,` is the inner interpreter of eForth. IP register is pointing to a token in a token list. The token contains a code field address of a Forth command. This address is fetched and stored in P register. IP is incremented by 4, pointing to the next token in the same token list. JUMP now jumps to this code field address and starts executing the first byte code in this address.

```
: next,     IP @ RAM@ P !
      4 IP +! JUMP ;
```

Pseudo Instructions

Following are the pseudo instructions of ceForth Virtual Forth Machine. They were defined as C routines in `ceForth_23.cpp`. Here they are defined using Forth commands. Please refer to Chapter 2 for their original definitions.

```
: nop,    JUMP ;
: bye,    ABORT" Simulation done." ;
\ : qrx,    ?RX ?DUP IF SPUSH -1 ELSE 0 THEN SPUSH ;
: qrx,    KEY DUP SPUSH SPUSH ;
: txsto,  SPOPP TX! ;
: inline, P @ RAM@ SPUSH 4 P +! ;
: dolit,  IP @ RAM@ SPUSH 4 IP +! next, ;
: dolist, IP @ RPUSH P @ IP ! next, ;
: exit,   RPOPP IP ! next, ;
: execu,  IP @ RPUSH SPOPP P ! JUMP ;
: donext, RPOPP ?DUP IF 1- RPUSH IP @ RAM@ IP !
       ELSE 4 IP +! THEN next, ;
: qbran,  SPOPP IF 4 IP +! ELSE IP @ RAM@ IP ! THEN next, ;
: bran,   IP @ RAM@ IP ! next, ;
: store,  SPOPP SPOPP SWAP RAM! ;
```

```
: at,     SPOPP RAM@ SPUSH ;
: cstor,  SPOPP SPOPP SWAP RAMC! ;
: cat,    SPOPP RAMC@ SPUSH ;
: rpat,   B0 RAM@ SPUSH ;
: rpsto,  SPOPP B0 RAM! ;
: rfrom,  RPOPP SPUSH ;
: rat,    RPOPP DUP RPUSH SPUSH ;
: tor,    SPOPP RPUSH ;
: spat,   B4 RAM@ SPUSH ;
: spsto,  SPOPP B4 RAM! ;
: drop,   SPOPP DROP ;
: dup,    SPOPP DUP SPUSH SPUSH ;
: swap,   SPOPP SPOPP SWAP SPUSH SPUSH ;
: over,   SPOPP SPOPP DUP SPUSH SWAP SPUSH SPUSH ;
: zless,  SPOPP 0< SPUSH ;
: andd,   SPOPP SPOPP AND SPUSH ;
: orr,    SPOPP SPOPP OR SPUSH ;
: xorr,   SPOPP SPOPP XOR SPUSH ;
: uplus,  SPOPP SPOPP UM+ SWAP SPUSH SPUSH ;
: qdup,   SPOPP DUP SPUSH ?DUP IF SPUSH THEN ;
: rot,    SPOPP SPOPP SPOPP ROT ROT SPUSH SPUSH SPUSH ;
: ddrop,  SPOPP SPOPP 2DROP ;
: ddup,   SPOPP SPOPP 2DUP SPUSH SPUSH SPUSH SPUSH ;
: plus,   SPOPP SPOPP + SPUSH ;
: inver,  SPOPP NOT SPUSH ;
: negat,  SPOPP NEGATE SPUSH ;
: dnega,  SPOPP SPOPP SWAP DNEGATE SWAP SPUSH SPUSH ;
: subb,   SPOPP SPOPP SWAP - SPUSH ;
: abss,   SPOPP ABS SPUSH ;
: equal,  SPOPP SPOPP = SPUSH ;
: uless,  SPOPP SPOPP SWAP U< SPUSH ;
: less,   SPOPP SPOPP SWAP < SPUSH ;
: ummod,  SPOPP SPOPP SPOPP SWAP ROT UM/MOD SWAP SPUSH SPUSH ;
: msmod,  SPOPP SPOPP SPOPP SWAP ROT M/MOD SWAP SPUSH SPUSH ;
: slmod,  SPOPP SPOPP SWAP /MOD SWAP SPUSH SPUSH ;
: mod,        SPOPP SPOPP SWAP MOD SPUSH ;
: slash,  SPOPP SPOPP SWAP / SPUSH ;
: umsta,  SPOPP SPOPP UM* SWAP SPUSH SPUSH ;
: star,   SPOPP SPOPP * SPUSH ;
: mstar,  SPOPP SPOPP M* SWAP SPUSH SPUSH ;
: ssmod,  SPOPP SPOPP SPOPP SWAP ROT */MOD SWAP SPUSH SPUSH ;
: stasl,  SPOPP SPOPP SPOPP SWAP ROT */ SPUSH ;
: pick,   SPOPP SP C+! SSTACK SPUSH ;
: pstor,  SPOPP SPOPP OVER RAM@ + SWAP RAM! ;
: dstor,  SPOPP SPOPP OVER CELL+ RAM! SPOPP SWAP RAM! ;
: dat,    SPOPP DUP RAM@ SPUSH CELL+ RAM@ SPUSH ;
: count,  SPOPP DUP 1+ SPUSH RAMC@ SPUSH ;
: dovar,  P @ SPUSH ;
: max,    SPOPP SPOPP MAX SPUSH ;
: min,    SPOPP SPOPP MIN SPUSH ;
```

The pseudo instructions are called from a byte code table, by the command `executecode`:

```
CREATE CODE-TABLE
' nop,    , ' bye,    , ' qrx,    , ' txsto, ,
' inline, , ' dolit,  , ' dolist, , ' exit,  ,
' execu,  , ' donext, , ' qbran,  , ' bran,  ,
' store,  , ' at,     , ' cstor,  , ' cat,   ,
' rpat,   , ' rpsto,  , ' rfrom,  , ' rat,   ,
' tor,    , ' spat,   , ' spsto,  , ' drop,  ,
' dup,    , ' swap,   , ' over,   , ' zless, ,
' andd,   , ' orr,    , ' xorr,   , ' uplus, ,
' next,   , ' qdup,   , ' rot,    , ' ddrop, ,
' ddup,   , ' plus,   , ' inver,  , ' negat, ,
' dnega,  , ' subb,   , ' abss,   , ' equal, ,
' uless,  , ' less,   , ' ummod,  , ' msmod, ,
' slmod,  , ' mod,    , ' slash,  , ' umsta, ,
' star,   , ' mstar,  , ' ssmod,  , ' stasl, ,
' pick,   , ' pstor,  , ' dstor,  , ' dat,   ,
' count,  , ' dovar,  , ' max,    , ' min,   ,

: executecode ( code -- )
   DUP 3F > ABORT" Illegal code "
   CELLS CODE-TABLE + @ EXECUTE ;
```

Finite State Machine

Here are the commands that run the Finite State Machine, and show the contents of pertinent registers and stacks. Originally, I thought of implementing a set of break points to allow user the freedom to break execution at a number of different memory locations. Eventually, I realized that only one break point is necessary and a simple ‘GO’ command is sufficient. This is the `G` command show below.

```
: .stack ( add # ) FOR AFT DUP @ U. 4 - THEN NEXT DROP CR ;
: .sstack ." S:" T @ U.
        SSTACK SP C@ .stack ;
: .rstack ." R:" RSTACK RP C@ .stack ;
: .registers ."  P=" P @ . ."  I=" I @ U.
        ."  I1=" I1 C@ . ."  I2=" I2 C@ .
        ."  I3=" I3 C@ . ."  I4=" I4 C@ .
        ."  IP=" IP @ . CR ;
: S  CR ."  CLOCK=" CLOCK @ . .registers
        .sstack .rstack ;
```

Command	Function
`.stack`	Display the contents of a stack.
`.sstack`	Display the contents of data stack.
`.rstack`	Display the contents of return stack.
`.registers`	Display the contents of all the relevant registers.
`S`	Show all the registers and stacks at this cycle.

```
: SYNC0  continue ;
```

```
: SYNC1  I1 C@ executecode ;
: SYNC2  I2 C@ executecode ;
: SYNC3  I3 C@ executecode ;
: SYNC4  I4 C@ executecode ;

CREATE SYNC-TABLE
' continue , ' SYNC1 , ' SYNC2 , ' SYNC3 ,
' SYNC4 , ' JUMP , ' JUMP , ' JUMP ,

: sync  CLOCK @ 7 AND cells
        SYNC-TABLE + @ EXECUTE ;
: C     sync CYCLE S ;
: RESET REGISTER $300 ERASE 0 CLOCK !
      ;
RESET
```

Command	Function
SYNC0	Fetch and decode next program word.
SYNC1	Execute byte code in I1.
SYNC2	Execute byte code in I2.
SYNC3	Execute byte code in I3.
SYNC4	Execute byte code in I4.
SYNC-TABLE	Table of execution routines for 6 phases.
sync	Execute the current pseudo instruction using CLOCK to determine which phase is being executed. CLOCK points to one of the routines in SYNC-TABLE, which contains the following entries: CONTINUE, fetch next program word SYNC1, execute instruction in I1 SYNC2, execute instruction in I2 SYNC2, execute instruction in I2 SYNC3, execute instruction in I3 SYNC4, execute instruction in I4 JUMP , terminate current program word.
C	Run one clock cycle and display all registers and stacks.
reset	Clear the REGISTER array, simulating hardware reset.

"C" is the single stepper in simulator. It runs the Finite State Machine for one cycle, and displays all registers and stacks. This is the most useful command to debug the ceForth in the early development stage. You can see all data in all registers and stacks. In the many eForth system, the first command executed is COLD, which executes a diagnostic word, DIAGNOSE. DIAGNOSE runs simple tests on most pseudo instructions. By single stepping through DIAGNOSE, you can validate most pseudo instructions. If all tests in DIAGNOSE run successfully, it is very likely the eForth will run correctly in the target.

"reset" clears the REGISTER array, and initializes the simulator to run at memory location 0.

This simulator has a very simple text-based user interface. The most used commands are:

Command	Stack Effects	Function
G	a --	Run and stop at address given on Forth stack. This is a very efficient way to set breakpoints and then run till a breakpoint is triggered. It allows the user to execute a large portion of the program and stop only at a specified location.
PUSH	n --	Push a new integer into the T register and data stack.
POP	--	Discard contents in T and pop data stack back into T.
D	--	Display memory starting at address in P.
M	a --	Dump 128 words in memory using “show” command.
RUN	--	Continue stepping with any key, terminated by ESC.

```
: G      ( addr -- )
         CR ." Press any key to stop." CR
         BREAK !
         BEGIN sync P @ BREAK @ =
               IF CYCLE C EXIT
               ELSE CYCLE
               THEN
               ?KEY
         UNTIL ;
: PUSH   ( n ) T @ SPUSH T ! ;
: POP    SPOPP ;
: D      P @ CELL- FOUR FOUR ;
: M      SHOW ;
: RUN    CR ." Press ESC to stop." CR
         BEGIN C KEY 1B = UNTIL ;

: HELP   CR ." cEF Simulator, copyright Offete Enterprises, 2009"
         CR ." C: execute next cycle"
         CR ." S: show all registers"
         CR ." D: display next 8 words"
         CR ." addr M: display 128 words from addr"
         CR ." addr G: run and stop at addr"
         CR ." RUN: execute, one key per cycle"
         CR ;
```

This simulator is most effective in debugging short sequences of program words to verify that the sequences are executed correctly. After pseudo instructions are verified, use the G command to execute a long stretch of program and break only at a specified location. This allows large segments of programs to be tested. If the simulator runs forever and cannot reach the break point you specified, you can stop the G command by hitting a key on the keyboard to terminate it.

When F# runs the metacompiler to compile ceForth, it displays names and code field addresses of all commands compiled into the target image. The display is a symbol table. You can look up a command and find its code field address. The code field addresses are the best place to set your break point. To debug a command, find its code field address and enter it with the G command. The simulator will break at the beginning of this command, and you can use the C command to single step through it.

Typing lots of "C" commands is tedious. The RUN command lessens your typing chore. After executing RUN, the simulator displays registers and stacks and pauses. Pressing any key will single step The Finite State Machine for one cycle. You can run many steps easily this way. When you want to stop RUN, press the ESC key.

To examine memory, type an address followed by the "M" command. It will display 128 words of memory starting from that address. The "D" command displays 8 program words starting at this address.

Chapter 8. Implementation Notes

Byte Code Sequencer vs Finite State Machine

A Finite State Machine (FSM) was adopted from eP32 chip design to run VFM in ceForth. This FSM assumed that we had a 32 bit machine, running on 32 bit memory. It used 6 phases to execute code stored in memory. In phase 0, it read a 32 bit program word, and decoded 4 byte code in it. In phase 1 to 4 it executes these 4 byte code in sequence. In phase 5, it resets the phase counter to 0, so it will fetch the next program word from memory, and run through the phases again. This FSM is described completely in the `main()` routine:

```
int main(int ac, char* av[])
{     int phase;
      clock = 0;
      P = 0;
      IP = 0;
      S = 0;
      R = 0;
      top = 0;
      phase = 0;
      cData = (unsigned char *)data;
      printf("\nceForth v2.2, 08jul17cht\n");
      while (TRUE) {
            phase = clock & 7;
            switch (phase) {
            case 0: fetch_decode(); break;
            case 1: execute(I1); break;
            case 2: execute(I2); break;
            case 3: execute(I3); break;
            case 4: execute(I4); break;
            case 5: jump(); break;
            case 6: jump(); break;
            case 7: jump();
            }
            clock += 1;
      } }
```

In ceForth, the dictionary is an array of 32 bit words. However, this array can be read either in 32 bit words, or in 8 bit bytes. Therefore, byte code in the dictionary can be fetched directly and executed without a FSM. A much simpler byte code sequencer can be coded as follows:

```
int main(int ac, char* av[])
{     P = 0;
      WP = 4;
      IP = 0;
      S = 0;
      R = 0;
      top = 0;
      cData = (unsigned char *)data;
```

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

The sequence has only two steps: fetching next byte from memory, and execute the byte code. It is just like a hardware computer, sequencing through its memory to execute machine instructions.

In the design of ceForth VFM, byte code are packed into code fields of primitive commands, and can be accessed either by 32 bit words, or by byte sequence. The same dictionary accommodate both design equally well. No modification in ceForth dictionary is necessary.

The byte code sequencer is made even simpler, because the clock and phase counter are also eliminated.

Stacks as Circular Buffers

Stacks are big headaches in operating systems, and in application programs. In C programming, stacks are hidden from you to prevent you from messing them up. However, in Forth programming, data stack and return stack are open to you, and most of the times, data stack becomes the focus of your attention. Both stacks have to work perfectly. There is no margin of error.

With stacks of finite size implemented in memory, the most obvious problems are stack overflow and stack underflow. Generally, operating systems allocate large chucks of memory for stacks, and impose traps on overflow and underflow conditions. With these traps, you can write interrupt routines to handle these error conditions in your software. These traps are very difficult to handle, especially for those without advanced computer science degrees.

The most prevalent problem in Forth programming is underflow of data stack, when you try to access data below the memory allocated to data stack. After Forth interpreter finished interpreting a list of Forth words, it always check the stack pointer. If the stack pointer is below mark, Forth interpreter executes the `ABORT` command, and reinitialized the stacks.

In designing eP32 chip, I put both stacks in the CPU. I allowed 32 levels of stack space, and the system seemed to be happy. I checked often water marks on both stacks, and the water marks were mostly about 12 levels. 32 levels are adequate for most applications, and do not impose a big burden on CPU designs. The stacks used 5 bit stack pointers, and behaved like circular buffers. I also found that it was not really necessary to check the stack pointers. Using circular buffers, underflow and overflow are really not life-threatening error conditions. If useful data were actually overwritten, the system would not behave correctly, but in no danger of crashing. The stack pointers needed not be reset. The system would restart with the present pointers.

In ceForth_23, I allocated 1KB memory for each stack, and used one byte for each stack pointer. The stacks are 256 cell circular buffers, and will never underflow or overflow. However, the C compiler needed to be reminder constantly that the stack pointers were 8-bit values and should not be leaked to integer or longer number. R and S pointers must always be prefixed with `(char)` specification. I struggled with data stack underflow conditions for half a year, until I found that the stack pointers tended to overshoot the byte boundary in my back.

ceForth interpreter always displays top 4 elements on data stack. Always seeing these 4 elements, you do not need utility to dump or examine data stack. I believe this is the best way to use data stack. It relieves you from the anxiety of worrying your misusing it.

Metacompiler

Conceptually, metacompilation is not much different that the ordinary Forth compiler. Forth compiler compiles new commands on top of its dictionary. CP is the pointer to top of dictionary. If we changed CP to point to another memory location, like the target dictionary array we allocated for a target system, then we could compile a new dictionary for the target.

Of course, the devil is in the details. The target memory is a virtual memory. Addresses used by the target machine are virtual addresses relative to the beginning of the dictionary array, not the absolute addresses used in the host Forth system. The target machine may have a different machine instruction set. Byte addressable machine vs word addressable machine. Different linking schemes. On and on.

The art of metacompilation had been practices since Chuck Moore invented Forth. I documented it for polyForth, F83, and FPC, three of the most popular Forth implementations. They all used vocabularies to segregate names of same commands used at various stages of metacompilation. For example, + (plus) command had 3 different behaviors: a regular + (plus) version to add two integers in text interpreter, a version defined in target dictionary which will be used by a target system to add two integers, which is never executed during metacompilation, and one version used by metacompiler to compile a + (plus) token in the body (token list) of a compound command in target dictionary.

A dictionary is a linked list of command records. A vocabulary is a branch of a dictionary, which can be searched independent of the main dictionary. Vocabularies allow a command to be redefined multiple times, and different behavior is selected by specifying search order of vocabularies.

In the original eForth Model, Bill Muench reserved system variables to allow building up to 8 vocabularies. However, over the years I had not used this feature in all my applications, and decided to rid of it. Without vocabularies, I could still do metacompilation by carefully arranging the sequence in defining commands to build target dictionary correctly. As commands are redefined, the Forth system morphs and shows unexpected behavior. All eForth commands are redefined to compile tokens. At the end of metacompilation, you can type in any valid eForth command, and the system responds with ‘ok’, but does not seem to do anything.

The data stack does not change. All the commands do is to add a token to the top of target dictionary, which you cannot see without great efforts.

After the metacompiler finishes building the target dictionary, it is useless for any other purposes.

Byte Code

I use byte code to bridge the Virtual Forth Machine and the eForth system. The dictionary is stored as an integer array `data[]`. This data array can be addressed either by 32 bit words or by bytes. When addressing by bytes, the array is referred as `cData[]`.

Command records and the fields in them are all word aligned. The link field is a 32 bit word. The name field has a length byte followed by variable length name string, null-filled to the word boundary. In a primitive command, the code field contains byte code, and is null-filled to word boundary. In a compound command, the code field is a 32 bit word, containing the `dolist,` byte code. The parameter field contains a token list. All tokens are 32 bit words.

This dictionary design was copied from eP32, which was a 32 bit microcontroller. There I used 6 bit machine instructions, and a 32 bit word contained up to 5 machine instructions. In one of the earlier designs, I used 5 bit machine instruction, and I could pack 6 machine instructions to a word. The assembler was designed so that it could pack as many instructions as a program word would allow. In ceForth, I already had 67 machine instructions, and 6-bit fields were not enough for them. For convenience, I just allocate 8 bits for instructions, and give you the possibility of using 256 byte code for machine instructions.

I was not particularly concerned about the numbering of byte code. They were assign consecutive numbers as I coded them. However, there is no reason why the numbering could not follow some preconceived order, like Java Byte Code. In fact, there is no reason that you could not build a Virtual Java Machine with this ceForth design.

Appendix ceForth v2.3 Command Reference

Stack Comments:

Stack inputs and outputs are shown in the form: (input1 input2 ... -- output1 output2 ...)

Stack Abbreviations of Data Types

n	32 bit integer
d	64 bit integer
flag	Boolean flag, either 0 or -1
char	ASCII character or a byte
addr	32 bit address

Stack

?DUP	n -- n n | 0	Duplicate top of stack if it is not 0.
DUP	n1 -- n2	Duplicate top of stack.
DROP	n --	Discard top of stack.
SWAP	n1 n2 -- n2 n1	Exchange top two stack items.
OVER	n1 n2 -- n1 n2 n1	Make copy of second item on stack.
ROT	n1 n2 n3 -- n2 n3 n1	Rotate third item to top.
PICK	n -- n1	Zero based, duplicate nth item to top. (e.g. 0 PICK is DUP).
>R	n --	Move top item to return stack for temporary storage.
R>	-- n	Retrieve top item from return stack.
R@	-- n	Copy top of return stack onto stack.
2DUP	d -- d d	Duplicate double number on top of stack.
2DROP	d1 d2 --	Discard two double numbers on top of stack

Arithmetic

+	n1 n2 -- n3	Add n1 and n2.
-	n1 n2 -- n3	Subtract n2 from n1 (n1-n2=n3).
*	n1 n2 -- n3	Multiply. n3=n1*n2
/	n1 n2 -- n3	Division, signed (n3= n1/n2).
1+	n -- n+1	Increment n.
1-	n -- n-1	Decrement n.
CELL	n -- n/2	Logic right shift.
CELL+	n -- n+2	Increment n by 4.
CELL-	n -- n-2	Decrement n by 4.
CELLS	n -- n*2	Logic left shift 2 bits.
CELL/	n -- n/2	Logic right shift.
UM+	n1 n2 -- nd	Unsigned addition, double precision result.
UM*	n1 n2 -- nd	Unsigned multiply, double precision result.
M*	n n -- d	Signed multiply. Return double product.
UM/MOD	nd n1 -- mod quot	Unsigned division with double precision dividend.
M/MOD	d n -- mod quot	Signed floored divide of double by single. Return mod and quotient.
MOD	n1 n2 -- mod	Modulus, signed (remainder of n1/n2).
/MOD	n1 n2 -- mod quot	Division with both remainder and quotient.

*/MOD	n1 n2 n3 -- n4 n5	Multiply and then divide (n1*n2/n3)
*/	n1 n2 n3 -- n4	Like */MOD, but with quotient only.
ABS	n1 -- n2	If n1 is negative, n2 is its two's complement.
NEGATE	n1 -- n2	Two's complement.
DNEGATE	d1 -- d2	Negate double number. Two's complement.
D+	d1 d2 -- d3	Add double numbers.

Logic and Comparison

AND	n1 n2 -- n3	Logical bit-wise AND.
OR	n1 n2 -- n3	Logical bit-wise OR.
XOR	n1 n2 -- n3	Logical bit-wise exclusive OR.
NOT	n1 -- n2	Bit-wise one's complement.
0<	n -- flag	True if n is negative.
U<	n1 n2 -- flag	True if n1 less than n2. Unsigned compare.
<	n1 n2 -- flag	True if n1 less than n2.
=	n1 n2 -- flag	True if n1 equals n2.
MAX	n1 n2 -- n3	n3 is the larger of n1 and n2.
MIN	n1 n2 -- n3	n3 is the smaller of n1 and n2.
WITHIN	n1 n2 n3 -- flag	Return true if n1 is within range of n2 and n3. (n2 <= n1 < n3)

Memory

@	addr -- n	Replace addr by integer at addr.
C@	addr -- char	Fetch least-significant byte only.
!	n addr --	Store n at addr.
C!	char addr --	Store least-significant byte only.
2@	addr -- d	Fetch double integer d at addr.
2!	d addr --	Store double integer d at addr.
+!	n addr --	Add n to integer at addr.
COUNT	addr -- addr+1 char	Move string count from memory onto stack.
ALLOT	n --	Add n bytes to the RAM pointer DP.
HERE	-- addr	Address of next available RAM memory location.
PAD	-- addr	Address of a scratch area of at least 64 bytes.
TIB	-- addr	Address of terminal input buffer.
CMOVE	addr1 addr2 n --	Move n bytes starting at memory addr1 to addr2.
FILL	addr n char --	Fill n bytes of memory at addr with char.
ERASE	addr n --	Zero fill n bytes starting at addr
PACK$	addr1 u addr2 --	Build a string at addr2 from u characters at addr1

System Variables

BASE	-- addr	Radix for number conversion
tmp	-- addr	Temporary scratch pad
SPAN	-- addr	Actual number of characters received by EXPECT
>IN	-- addr	Character offset into the input stream buffer.
#TIB	-- addr	Current length of terminal input buffer (TIB.

'TIB	-- addr	Current address of terminal input buffer (TIB
'EVAL	-- addr	Interpreter or compiler to evaluate a command.
HLD	-- addr	Pointer to numeric string under construction.
CONTEXT	-- addr	Name field address of last command in dictionary
CP	-- addr	First free address in .data segment of memory
LAST	-- addr	Name field address of command under compilation

Terminal Input-Output

EMIT	char --	Display char.
KEY	-- char	Get an ASCII character from the keyboard.
?KEY	-- char -1 | 0	Return an ASCII character from the keyboard and a true flag. Return false flag if no character available.
.	n --	Display number n with a trailing blank.
U.	n --	Display an unsigned integer with a trailing blank.
.R	n1 n2 --	Display signed number n1 right justified in n2 character field.
U.R	n1 n2 --	Display unsigned number n1 right justified in n2 character field.
?	addr --	Display contents at memory addr.
<#	--	Start numeric output string conversion.
#	n1 -- n2	Convert next digit of number and add to output string
#S	n --	Convert all significant digits in n to output string.
HOLD	char --	Add char to output string.
SIGN	n --	If n is negative, add a minus sign to the output string.
#>	d -- addr n	Terminate numeric string, leaving addr and count for TYPE.
CR	--	Display a new line. Send carriage return and line feed.
SPACE	--	Display a space.
SPACES	n --	Display n spaces.
ACCEPT	addr n --	Accept n characters into buffer at addr.
TYPE	addr n --	Display a string of n characters starting at address addr.
BL	-- 32	Return ASCII Blank character.
DECIMAL	--	Set number base to decimal.
HEX	--	Set number base to hexadecimal.

Compiler and Interpreter

:<name>	--	Begin a colon definition of <name>.
;	--	Terminate execution of a colon definition.
CREATE <name>	--	Dictionary entry with no parameter field space reserved.
VARIABLE <name>	--	Defines a variable. At run-time, <name> leaves its address.
CONSTANT <name>	n --	Defines a constant. At run-time, n is left on the stack.
,	n --	Compile n to the dictionary in flash memory
IMMEDIATE	--	Cause last-defined command to execute even within a colon definition.
COMPILE <name>	--	<name> is compiled to dictionary.
[COMPILE] <name>	--	Immediate command. <name> is compiled to dictionary.
LITERAL	n --	Compile literal number n. At run-time, n is pushed on the stack.
[	--	Switch from compilation to interpretation.
]	--	Switch from interpretation to compilation.

WORD<text>	char -- addr	Get the char delimited string <text> from the input stream and leave as a counted string at addr.
(<comment>)	--	Ignore comment text.
\ <comment>	--	Ignore comment till end of line.
." <text>"	--	Compile <text> message. At run-time display text message.
.(<text>)	--	Display <text> from the input stream.
$" <text>"	-- addr	Compile <text> message. At run-time return its address.
ABORT	--	Jump to QUIT on error.
ABORT" <text>"	flag --	Compile <test> message. At run-time display message and abort if flag is true. Otherwise, ignore message and continue.
COLD	--	Start eForth system.
QUIT	--	Return to interpret mode, clear data and return stacks.
QUERY	--	Accept input stream to terminal input buffer.
EXECUTE	addr --	Execute command definition at addr.
@EXECUTE	addr --	Execute command definition whose execution address is in addr.

Structures

IF	flag --	If flag is zero, branches forward to ELSE or THEN.
ELSE	--	Branch forward to THEN.
THEN	--	Terminate a IF-ELSE-THEN structure.
FOR	n --	Setup loop with n as index. Repeat loop n+1 times.
NEXT	--	Decrement loop index by 1 and branch back to FOR. Terminate FOR-NEXT loop when index is negative.
AFT	--	Branch forward to THEN in a loop to skip the first round
BEGIN	--	Start an indefinite loop.
AGAIN	--	Branch backward to BEGIN.
UNTIL	flag --	Branch backward to BEGIN if flag is false. If flag is true, terminate BEGIN-UNTIL loop.
WHILE	flag --	If flag is false, branch forward to terminate BEGIN-WHILE-REPEAT loop. If flag is true, continue execution till REPEAT.
REPEAT	--	Resolve WHILE clause. Branch backward to BEGIN.
AHEAD	--	Resolve WHILE clause. Branch backward to BEGIN.

Utility

' <name>	-- addr	Look up <name> in the dictionary. Return execution address.
DUMP	addr --	Dump 128 bytes of RAM memory starting from addr.
WORDS	--	Display all eForth commands
BYE	--	Terminate eForth and return to Windows.

Inner Interpreters

next()	--	Jump to next token at end of all primitive commands
DOLIST	--	Address interpreter to start executing following token list
EXIT	--	Terminate execution of a token list
DOVAR	-- addr	Variable interpreter to return parameter field address
DOCON	-- n	Constant interpreter to return value in parameter field
DOLIT	-- n	Integer literal interpreter to return following literal value
BRANCH	--	Unconditionally branch to following literal address
?BRANCH	flag --	Branch to following literal address if flag is false
DONXT	--	Decrement count on return stack. Branch to following literal address if count is not negative; else pop return stack and exit loop.

do$	-- addr	Return address of a compiled string literal
$”|	-- addr	String literal interpreter returning address of following string
.”|	--	String literal interpreter displaying following string
ABORT”|	flag --	If flag is true, display following string and ABORT.

Supporting Words

DIGIT	u -- char	Convert digit u to a character.
EXTRACT	n1 base – n2 char	Extract the least significant digit from n1. Leave quotient n2 and digit char.
>CHAR	char -- char	Filter non-printing character to an underscore.
str	n – addr count	Convert a signed integer to a numeric string.
wupper	n – n	Convert 4 characters in a word to upper case.
>upper	char – char	Convert a character to upper case.
DIGIT?	char base– n flag	Convert a character to its numeric value. A flag indicates success.
(parse)	addr n char – addr n delta	Scan string delimited by c. Return found string and its offset delta.
PARSE	char – addr n	Scan input stream and return counted string delimited by char.
TOKEN	-- addr	Parse a word from input stream and copy it to name dictionary.
NUMBER?	addr -- n -1 | addr 0	Convert a number string to integer. Push a flag on data stack.
NAME>	nfa -- cfa	Return a code field address given a name field address.
>NAME	cfa -- nfa	Convert code field address to a name field address.
NAME?	addr -- cfa nfa | addr flag	Search dictionary for a string at addr. Return cfa and nfa if found. Else push a false flag above addr
SAME?	a1 a2 n – a1 a2 flag	Compare u-2 bytes in two strings. Return 0 if identical.
FIND	a va -- cfa nfa | a flag	Search a dictionary for a string. Return cfa and nfa if succeeded. Else, return a and false flag.
^H	bot eot cur -- bot eot cur	Backup the cursor by one character.
TAP	bot eot cur char -- bot eot cur	Accept and echo the key stroke and bump the cursor.
kTAP	bot eot cur char -- bot eot cur	Process a key stroke, CR or backspace.
?STACK	--	Abort if the data stack underflows.
.OK	--	Display the data stack only while interpreting.
EVAL	--	Interpret the input stream.
PRESET	--	Reset data stack pointer.
$INTERPRET	addr --	Interpret a word. If failed, try to convert it to an integer. Failing that, ABORT
$COMPILE	addr --	Compile a word to dictionary as a token or literal. Failing both, ABORT
?UNIQUE	addr -- addr	Display a warning message if the word at addr already exists.
$,”	--	Compile a literal string up to next " . nfa --
$,n	addr --	Build a new dictionary name using the string at addr
OVERT	--	Link a new word into the current dictionary.
.ID	nfa --	Display the name at name field address.