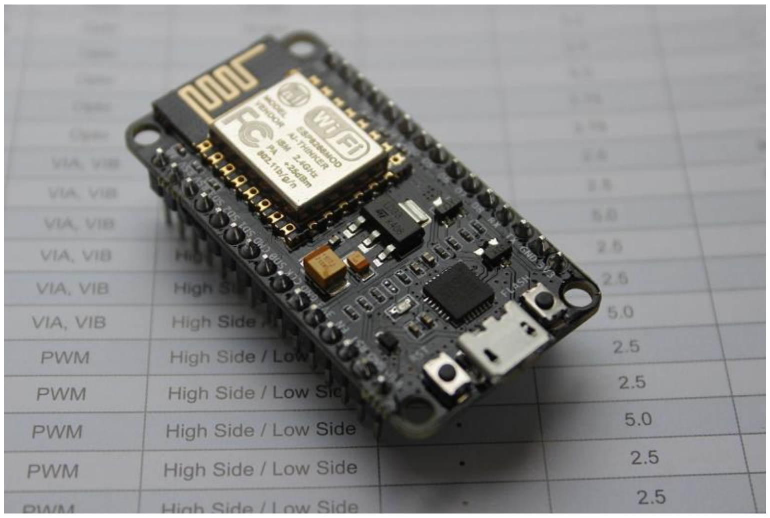
espForth for ESP8266

Preface—IoT For Fun!

NodeMCU is a thumb-size microcontroller kit selling for $3.18 on eBay. It has a 32 bit ESP8266 chip with 150 KB of RAM, 4 MB of flash, and a MicroUSB connector to communicate with a PC. The most amazing feature is that it has a WiFi radio, ready to be connected to the Internet, wirelessly. NodeMCU Kit is easily the most capable microcontroller kit under $5, and thus opens the door for all people to explore IoT applications.



WiFi is a very complicated subject involving many hardware and software issues. The original developers of ESP8266 in Espressif Systems, Shanghai, China, solved the hardware problem in silicon, and left a few software development kits (SDK) for software engineers to deal with software issues themselves. Software engineers all over the world took up the challenge and released programming tools known as IDE (Integrated Development Environment) for users to develop their own applications. Since there are 150 KB of RAM and 4 MB of flash memory on board, people ported quite sophisticated interactive programming languages like MicroPython and Lua to it, and allowed hobbyists to build IoT applications easily.

I am always of the opinion that Forth is the best programming language for microcontrollers, and have promoted a very simple eForth Model to implement Forth language on every microcontroller I laid my hands on. This NodeMCU Kit is an ideal platform to demonstrate the usefulness of Forth in programming microcontrollers. Microcontroller programming is very different from hardware engineering and software engineering, and I called it firmware engineering.

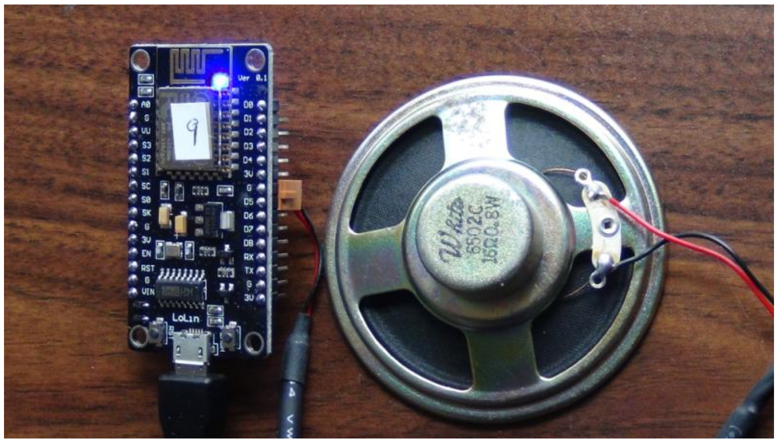
In good old days, firmware engineering meant programming a UV Erasable EPROM chip to run a standalone microcontroller system. Now, flash memory is integrated into microcontrollers, and a microcontroller system can be programmed very conveniently through a USB Serial cable. Would it be nice to program your target system remotely through WiFi, or even over the

Internet? NodeMCU give us the opportunity, and this possibility excited me. I got so excited that I went to DMV and got myself a new license plate for it.

(Sorry to my Democrat friends about that Trump sticker on the car.)

In 2017, fellows in Silicon Valley Forth Interest Group got a big booth in the Bay Area Maker Faire, and we ran an “IoT for Fun!” WiFi Workshop, featuring this NodeMCU Kit. We posted a challenge to attendees to turn its on-board LED on and off, remotely over WiFi. Whoever succeeded got a free NodeMCU Kit. It was fun, and we gave away more than 50 kits. We provided tools to program NodeMCU in Forth, Arduino, MicroPython and Lua. This manual details the espForth system we promoted in the Faire.

Although the official challenge was only to control the LED, we also attached s small speaker to an output pin. In one of our examples, the speak beeps when LED is turned on. Here is a picture of the setup:



Programming ESP8266 is not easy, believe me. The CPU, its memories, and its peripherals are complicated, and not very well documented. Espressif Systems released several version of the Software Development Kits (SDK) under Linux. I am a simple person, and never had the patience to learn Unix or Linux. I could not get the make file to work, because I could not install Linux correctly. I could not find an assembler for its CPU, Tensilica L106. Even if I could, I had no way to deal with its peripherals, and especially, the WiFi hardware and software stack. I needed help.

I tried Lua and MicroPython. It is nice that they are interactive, similar to Forth. But they are very complicated underneath, i.e., object oriented, and carry great overhead for the convenience they offer.

Another option was Arduino IDE. I had used it to program Arduino Uno Kit, based on ATmega328P microcontroller from Atmel. Espressif Systems hired a Russian engineer Ivan Grokhotkov to extend Arduino IDE so that it could support ESP8266 chip. I was very impressed by the Arduino IDE, because it captured the essence of firmware programming in two routines `setup`() and `loop`(). You simply insert C code into these routines, and Arduino takes care of the rest. The only drawback was that it speaks only C language, so I had to write my Forth in C.

I did write a Forth in C on Arduino Uno Kit. It was ceForth_328, for. However, it was only a teaser, because ATmega328P chip had only 1.5 KB of RAM left to add new Forth commands, and you could not build substantial application on it. Now, ESP8266 has 150 KB of RAM, and it is more than enough to do serious applications. Arduino IDE supplies all library routines for whatever you have to do with NodeMCU Kit.

I was pleasant surprised to port ceForth_328 to ESP8266 after a single day's work. I took pride in porting eForth to a new microcontroller in about two weeks. Here because this Forth was written in C, it moved over to a new chip very easily. Only the hardware interface has to be changed. More than 95% of the code needs no modification.

I am very happy with this espForth. One reason is that it gave me an opportunity to re-examine my Forth in C implementation, and made a few significant improvements, like using circular buffers for stacks. The other reason is that it can be controlled by a serial monitor through USB cable, and by UDP packets sent and received over a WiFi network. These two communication channels work in parallel. I can now program NodeMCU interactively through a serial monitor, and can dispatch it and control it remotely. When you can access microcontrollers remotely, it will be possible to build large multiprocessing systems without limit. Forth is the best language not only for us human to control computers, but also for computers to communicate with one another.

I have to admit that I really do not understand ESP8266 chip at all. I do not understand the Ardhuino IDE which bridges the gap between me and my ESP8266. Nevertheless, for the privileges to talk to my ESP8266 remotely, I am willing to swallow my pride and accept C language and the tools it built to turn on the LED on my NodeMCU Kit, remotely.

Chapter 1. Running espForth

Install Arduino IDE

espForth_44.ino is an Arduino sketch. It is a streamlined C program to be compiled by Arduino IDE, and then uploaded to NodeMCU Kit to execute. To run espForth, you have to first install Arduino IDE with ESP8266 extensions. Then you can copy espForth_44.ino to it and get it running.

Arduino was originally developed for a lowly 8-bit AVR microcontrollers ATmega328P from Atmel Corp, on an Arduino Uno kit. It greatly simplified the C programming language and made it very easy for you to write your own application on AVR chips. It gives you a very simple program template, which expects you to fill C code in two routines `setup`() and `loop`(). It captures the essences of firmware engineering and invites everybody to become a firmware engineer.

It is amazing that people in ESP8266 Community extended the Arduino IDE so that you can program this very sophisticated 32-bit ESP8266 chip with ease. It even supports our ESP8266 NodeMCU Kit!

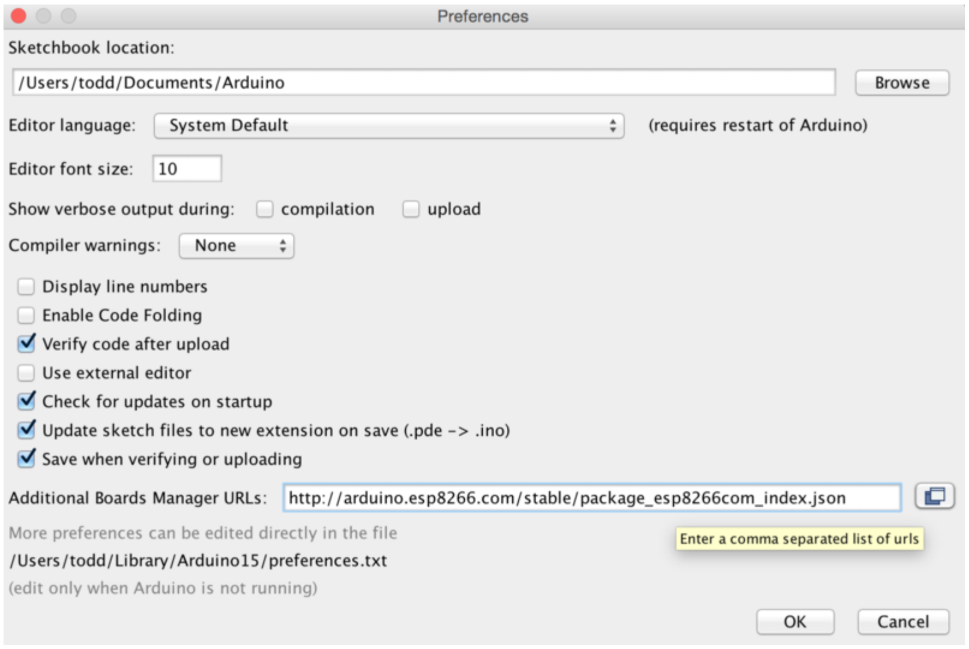
In the experiments with espForth and UDP Server, you will have to use Arduino IDE. If your computer does not have it, you have to install it first. Aduino IDE has to be extended so that it can compile programs for the ESP8266 chip, and to upload the compiled code to flash memory on NodeMCU Kit. After Arduino IDE is set up properly, it will be very easy to do experiments with NodeMCU Kit.

Download Arduino 1.8.2 IDE or the latest version from www.arduino.cc and install it on your PC. Open Auduino, and you will see its title page:

ARDUINO
Genuino
AN OPEN PROJECT WRITTEN, DEBUGGED, AND SUPPORTED BY ARDUINO.CC AND THE ARDUINO COMMUNITY WORLDWIDE
LEARN MORE ABOUT THE CONTRIBUTORS OF ARDUINO.CC on arduino.cc/credits

Click File>Preferences to open the Preferences window.

Enter http://arduino.esp8266.com/stable/package_esp8266com_index.json into Additional Board Manager URLs field:


Preferences

Sketchbook location:

/Users/todd/Documents/Arduino

Editor language: System Default (requires restart of Arduino)

Editor font size: 10

Show verbose output during: ☐ compilation ☐ upload

Compiler warnings: None

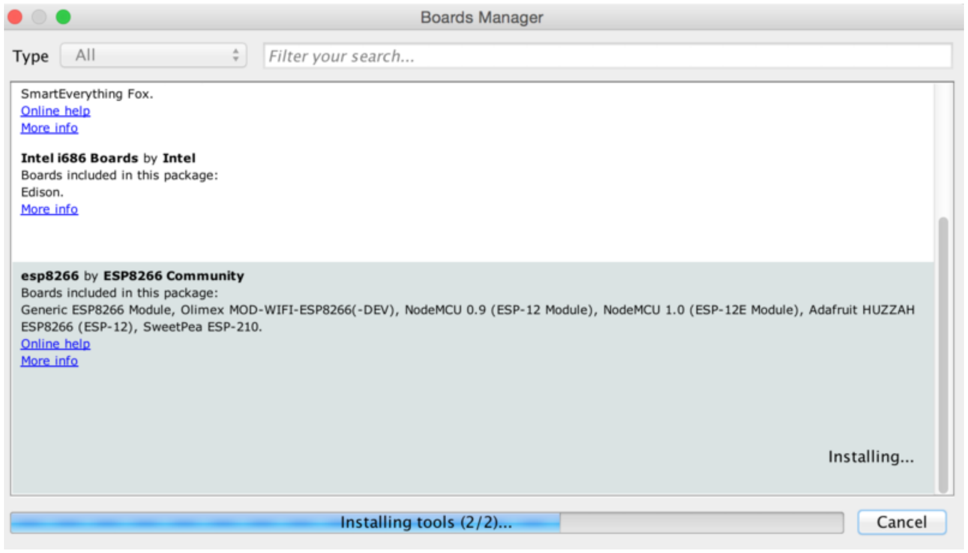
☐ Display line numbers
☐ Enable Code Folding
☑ Verify code after upload
☐ Use external editor
☑ Check for updates on startup
☑ Update sketch files to new extension on save (.pde -> .ino)
☑ Save when verifying or uploading

Additional Boards Manager URLs: http://arduino.esp8266.com/stable/package_esp8266com_index.json

Enter a comma separated list of urls

More preferences can be edited directly in the file
/Users/todd/Library/Arduino15/preferences.txt
(edit only when Arduino is not running)

Next, Click Tools>Board:xxxxxxx>Boards Manager. Scroll to the bottom of the display, and click on the panel named esp8266 by ESP8266 Community to select it:


Boards Manager

Type All   Filter your search...

SmartEverything Fox.
Online help
More info

Intel i686 Boards by **Intel**
Boards included in this package:
Edison.
More info

esp8266 by **ESP8266 Community**
Boards included in this package:
Generic ESP8266 Module, Olimex MOD-WIFI-ESP8266(-DEV), NodeMCU 0.9 (ESP-12 Module), NodeMCU 1.0 (ESP-12E Module), Adafruit HUZZAH ESP8266 (ESP-12), SweetPea ESP-210.
Online help
More info

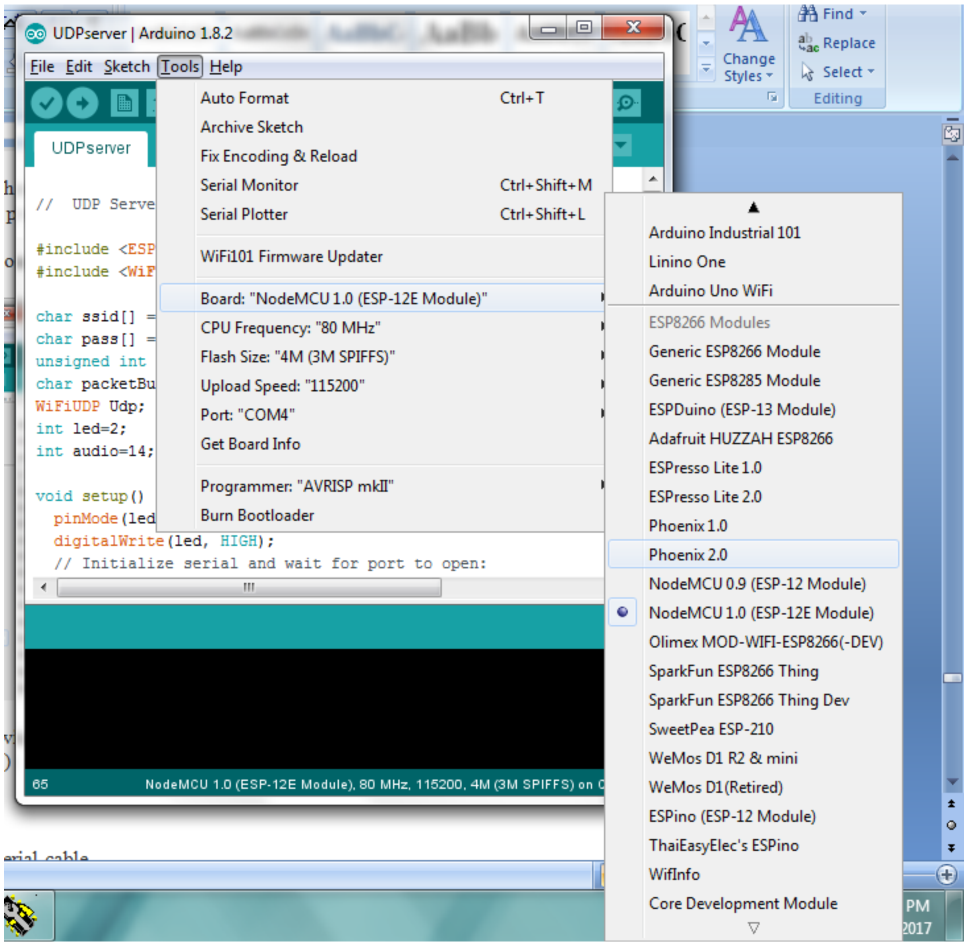
Installing...

Installing tools (2/2)...

Click the Install button at bottom right to install the ESP8266 package.

After the install process, you should see that ESP8266 package is marked INSTALLED. Close the Boards Manager window once the install process has completed.

Select NodeMCU 1.0 from the Tools->Board dropdown menu:



Board: "NodeMCU 1.0 (ESP-12E Module)"
CPU Frequency: "80 MHz"
Flash Size: "4M (3M SPIFFS)"
Upload Speed: "115200"
Port: "COM4"
Programmer: "AVRISP mkII"

```
//  UDP Serve

#include <ESP
#include <WiF

char ssid[] =
char pass[] =
unsigned int
char packetBu
WiFiUDP Udp;
int led=2;
int audio=14;

void setup()
  pinMode(led
  digitalWrite(led, HIGH);
  // Initialize serial and wait for port to open:
```

```
65    NodeMCU 1.0 (ESP-12E Module), 80 MHz, 115200, 4M (3M SPIFFS) on C
```

In the Tools menu, you will see the following selections:

```
Board: NodeMCU 1.0 (ESP-12E Module)
CPU frequency: 80 MHz
Flash Size: 4M (3M SPIFFS)
Upload Speed: 115200 baud
Port: COM port for your FTDI or USB-Serial cable
```

Arduino IDE is now set up properly. You can now proceed to do espForth or UDP Server experiments.

Loading espForth

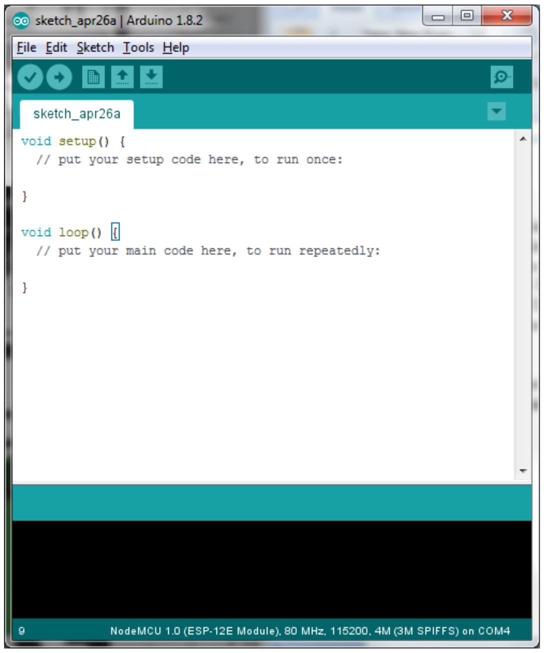
Forth is the simplest programming language, and has been widely used for industrial, scientific, military, space, and embedded applications. eForth is the simplest Forth implementation for microcontrollers. I ported it to ESP8266 under Arduino IDE as espForth. Once espForth is loaded on NodeMCU, it allows you to explore this chip, and test its IO devices interactively. Since espForth accepts input from both the USB-serial COM port, and UDP packets simultaneously, you can turn its on-board LED on and off interactively through the Serial

Monitor in Arduino, or through UDP packets through a UDP terminal like Hercules SETUP Utility.

Connect your NodeMCU Kit to your PC, with the MicroUSP-Serial cable.

Unzip espForth_44.zip, and place it in a folder like C:/espForth_44.

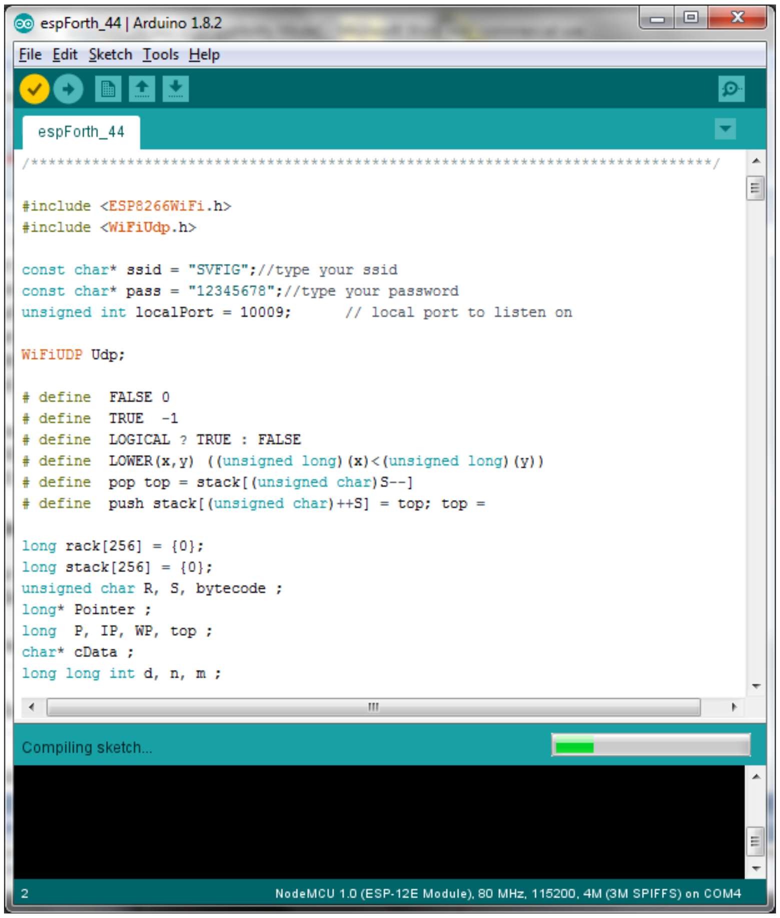
Open Arduino IDE. If it is the first time you do anything with Arduino, you will probably have a default program template like this:



```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

9 NodeMCU 1.0 (ESP-12E Module), 80 MHz, 115200, 4M (3M SPIFFS) on COM4

Click File>Open…, and select C: /espForth_44/espForth_44.ino, supplied in the espForth_44 project folder.



```
/*****************************************************************************/

#include <ESP8266WiFi.h>
#include <WiFiUdp.h>

const char* ssid = "SVFIG";//type your ssid
const char* pass = "12345678";//type your password
unsigned int localPort = 10009;      // local port to listen on

WiFiUDP Udp;

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

long rack[256] = {0};
long stack[256] = {0};
unsigned char R, S, bytecode ;
long* Pointer ;
long  P, IP, WP, top ;
char* cData ;
long long int d, n, m ;
```

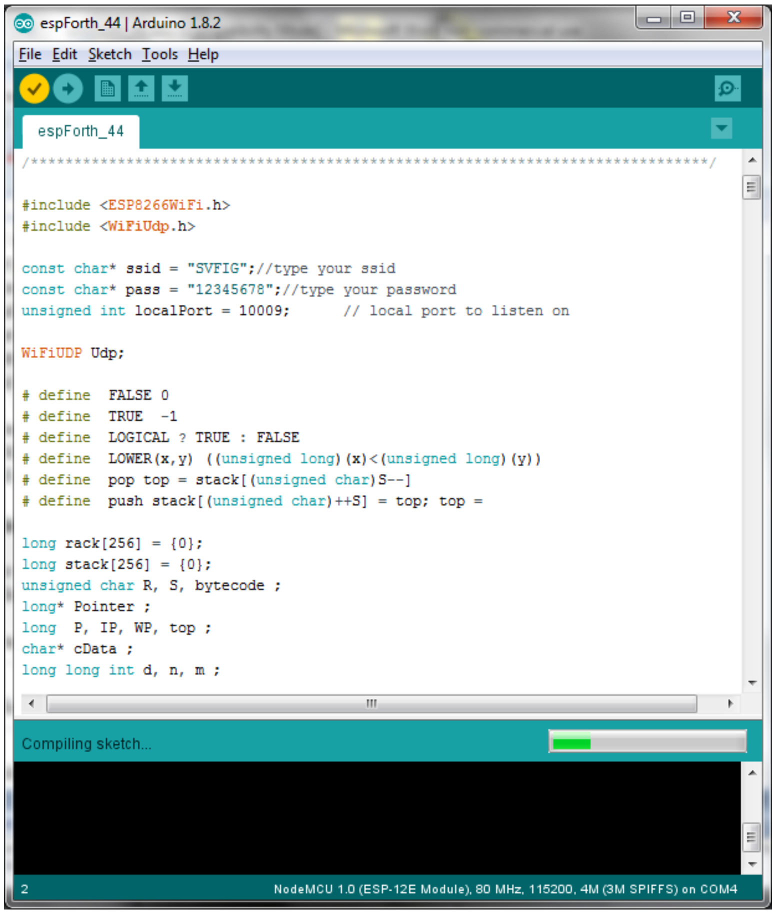
```
Compiling sketch...
```

```
2                NodeMCU 1.0 (ESP-12E Module), 80 MHz, 115200, 4M (3M SPIFFS) on COM4
```

espForth is a file of 66 KB size, very small comparing to software of this age, but it is a complete interactive operating system with a high level Forth programming language. Here in this experiment, I will not bother you with its features and its capabilities. I just want to lead you to meet our challenge in turning a LED on and off remotely.

Please note that espForth is connected to our local WiFi network, with a name of ‘SVFIG’ and a password of ‘12345678’. All NodeMCU Kits and all PC’s used in this experiment have to be connected to this network, if they need to communicate with one another. Each NodeMCU are assigned a unique local port number from 10001 and up, to avoid conflicts. If you have a different network, change the name in `ssid` and password in `pass` at the beginning of your espForth_44.ino program.

Click the Upload Button(→), the one with an arrow pointing to the right:

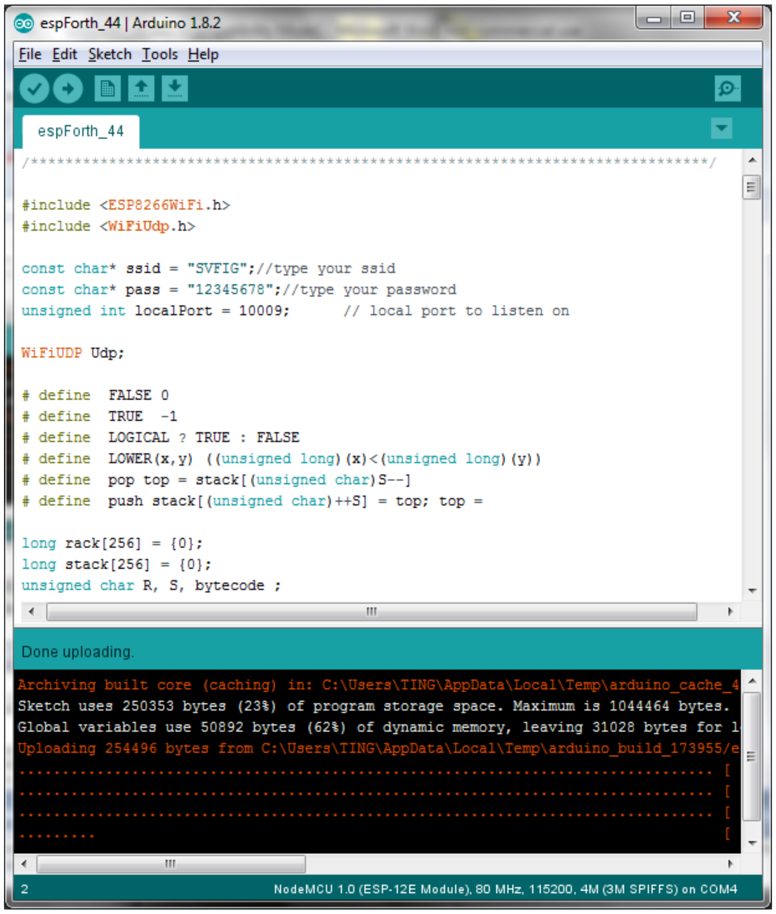


```
/*****************************************************************************/

#include <ESP8266WiFi.h>
#include <WiFiUdp.h>

const char* ssid = "SVFIG";//type your ssid
const char* pass = "12345678";//type your password
unsigned int localPort = 10009;      // local port to listen on

WiFiUDP Udp;

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

long rack[256] = {0};
long stack[256] = {0};
unsigned char R, S, bytecode ;
long* Pointer ;
long  P, IP, WP, top ;
char* cData ;
long long int d, n, m ;
```

```
Compiling sketch...
```

```
2                    NodeMCU 1.0 (ESP-12E Module), 80 MHz, 115200, 4M (3M SPIFFS) on COM4
```

It takes a few minutes for Arduino IDE to compile the code and then upload the binary image into the flash memory on NodeMCU Kit. Eventually, it will report ‘Done uploading’, and reports to you what it accomplished:



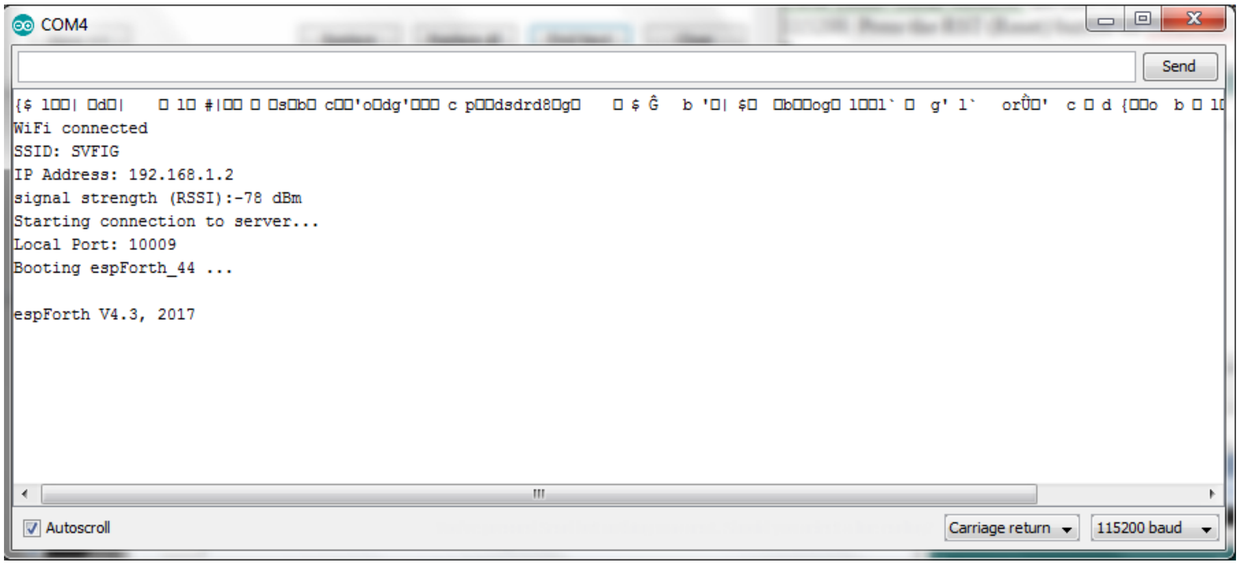
```
/*****************************************************************************/

#include <ESP8266WiFi.h>
#include <WiFiUdp.h>

const char* ssid = "SVFIG";//type your ssid
const char* pass = "12345678";//type your password
unsigned int localPort = 10009;      // local port to listen on

WiFiUDP Udp;

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

long rack[256] = {0};
long stack[256] = {0};
unsigned char R, S, bytecode ;
```

```
Done uploading.
Archiving built core (caching) in: C:\Users\TING\AppData\Local\Temp\arduino_cache_4
Sketch uses 250353 bytes (23%) of program storage space. Maximum is 1044464 bytes.
Global variables use 50892 bytes (62%) of dynamic memory, leaving 31028 bytes for l
Uploading 254496 bytes from C:\Users\TING\AppData\Local\Temp\arduino_build_173955/e
............................................................................ [
............................................................................ [
............................................................................ [
.........                                                                    [
```

2
NodeMCU 1.0 (ESP-12E Module), 80 MHz, 115200, 4M (3M SPIFFS) on COM4

Test espForth

Click Tools>Serial Monitor. Be sure to select the correct COM port, and set baud rate to 115200. Press the RST (Reset) button on NodeMCU, and you will see espForth signing in:

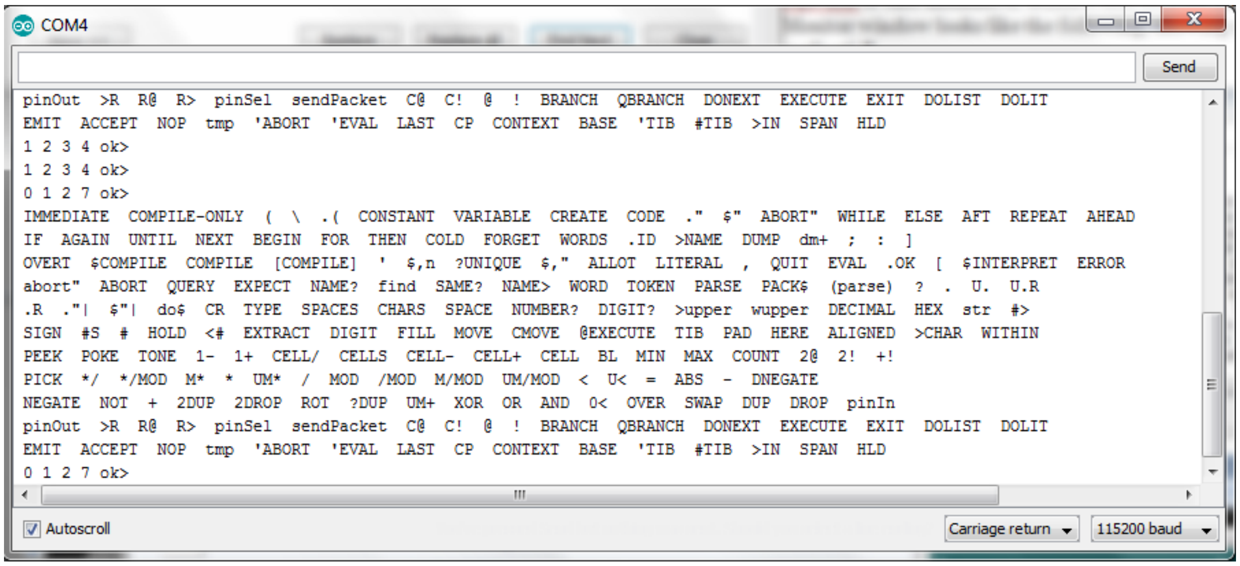


```
{$ l□□| □d□|    □ l□ #|□□ □ □s□b□ c□□'o□dg'□□□ c p□□dsdrd8□g□    □ $ Ĝ   b '□| $□  □b□□og□ l□□l` □  g' l`   orÛ□'  c □ d {□□o  b □ l□
WiFi connected
SSID: SVFIG
IP Address: 192.168.1.2
signal strength (RSSI):-78 dBm
Starting connection to server...
Local Port: 10009
Booting espForth_44 ...

espForth V4.3, 2017
```

Note the IP Address and the Local Port number. You will need these numbers to send UDP packets. Now you can type in the following Forth commands in the text box on the top of Serial Monitor window, (and click the Send button to the right), to exercise espForth system. After entering one line, press the Send button to the right of the text box.

```
1 2 3 4
+
WORDS
```

espForth is case insensitive. `WORDS` and `words` are the same. After `WORDS` is entered, Serial Monitor window looks like the following, showing all the Forth commands implemented in espForth:



```
pinOut  >R  R@  R>  pinSel  sendPacket  C@  C!  @  !  BRANCH  QBRANCH  DONEXT  EXECUTE  EXIT  DOLIST  DOLIT
EMIT  ACCEPT  NOP  tmp  'ABORT  'EVAL  LAST  CP  CONTEXT  BASE  'TIB  #TIB  >IN  SPAN  HLD
1 2 3 4 ok>
1 2 3 4 ok>
0 1 2 7 ok>
IMMEDIATE  COMPILE-ONLY  (  \  .(  CONSTANT  VARIABLE  CREATE  CODE  ."  $"  ABORT"  WHILE  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  [  $INTERPRET  ERROR
abort"  ABORT  QUERY  EXPECT  NAME?  find  SAME?  NAME>  WORD  TOKEN  PARSE  PACK$  (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
PEEK  POKE  TONE  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  UM+  XOR  OR  AND  0<  OVER  SWAP  DUP  DROP  pinIn
pinOut  >R  R@  R>  pinSel  sendPacket  C@  C!  @  !  BRANCH  QBRANCH  DONEXT  EXECUTE  EXIT  DOLIST  DOLIT
EMIT  ACCEPT  NOP  tmp  'ABORT  'EVAL  LAST  CP  CONTEXT  BASE  'TIB  #TIB  >IN  SPAN  HLD
0 1 2 7 ok>
```

Now, try to turn the on-board LED on and off with these lines of commands:

```
1 2 pinSel
1 2 pinOut
0 2 pinOut
```

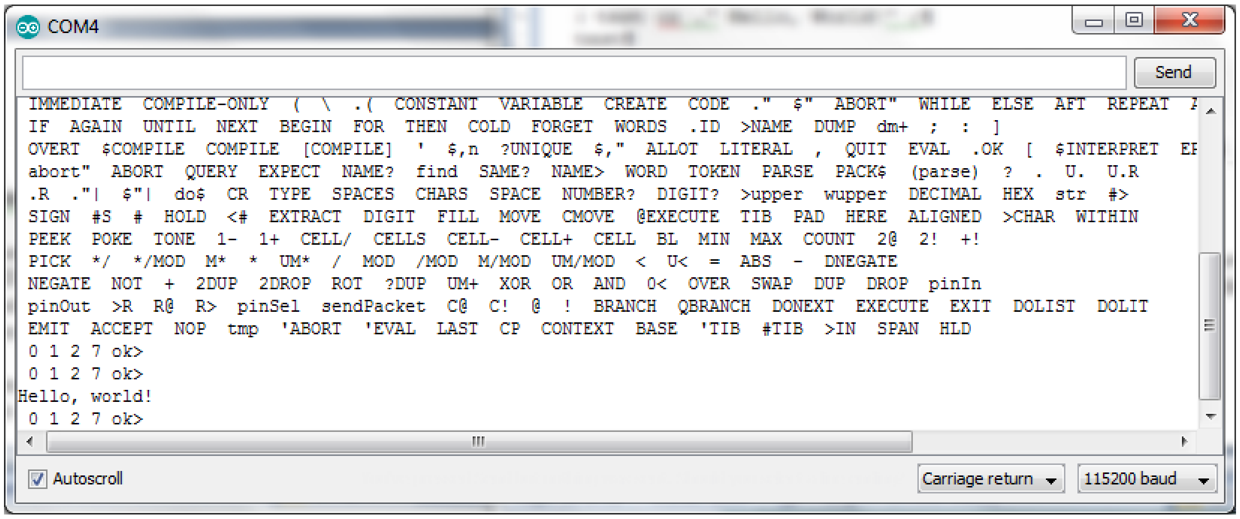
The commands ‘`1 2 pinSel`’ configure GPIO Pin 2 as an output pin. ‘`0 2 pinOut`’ commands turn the on-board LED on. ‘`1 2 pinOut`’ commands turns the LED off.

OK. You can turn the LED on and off, interactively, through USB-Serial cable. Now, try to do it remotely.

Before you go, do this universal experiment:

```
: test cr ." Hello, World!" ;
test
```

And you will see something like the following:



```
 IMMEDIATE  COMPILE-ONLY  (  \  .(  CONSTANT  VARIABLE  CREATE  CODE  ."  $"  ABORT"  WHILE  ELSE  AFT  REPEAT  A
 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  [  $INTERPRET  EF
 abort"  ABORT  QUERY  EXPECT  NAME?  find  SAME?  NAME>  WORD  TOKEN  PARSE  PACK$  (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
 PEEK  POKE  TONE  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  UM+  XOR  OR  AND  0<  OVER  SWAP  DUP  DROP  pinIn
 pinOut  >R  R@  R>  pinSel  sendPacket  C@  C!  @  !  BRANCH  QBRANCH  DONEXT  EXECUTE  EXIT  DOLIST  DOLIT
 EMIT  ACCEPT  NOP  tmp  'ABORT  'EVAL  LAST  CP  CONTEXT  BASE  'TIB  #TIB  >IN  SPAN  HLD
 0 1 2 7 ok>
 0 1 2 7 ok>
Hello, world!
 0 1 2 7 ok>
```

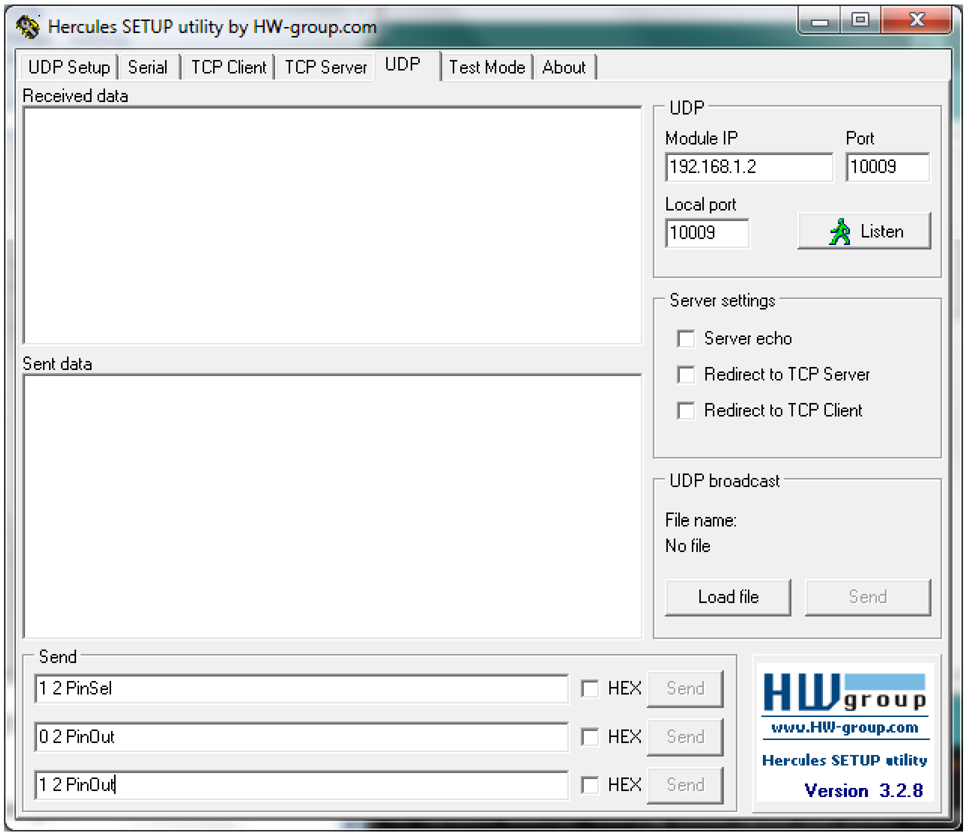
espForth is up and running.

espForth UDP Interface

To control the LED remotely over WiFi, the server on NodeMCU is set up to receive UDP packets, and acts on the payloads. We just want NodeMCU to turn its on-board LED on and off. To send packets you need a client on the same WiFi network. A client can be programmed to send many different packets, according to specific network protocols. Protocols can be very complicated, and for this experiment, you are required to learn only the simplest protocol, UDP, the User Datagram Protocol. It does not make sense to use complicated protocols to do such a simple job, just turning a LED on and off.

There are infinite ways to send UDP packets. We picked the Hercules SETUP Utility to do it.

Open Hercules.exe, which is a network communication utility. Select UDP. Enter IP address and Port number of the server that you set up on NodeMCU board:



UDP: Module IP 192.168.1.2, Port 10009, Local port 10009

Server settings: [ ] Server echo, [ ] Redirect to TCP Server, [ ] Redirect to TCP Client

UDP broadcast: File name: No file

Send:

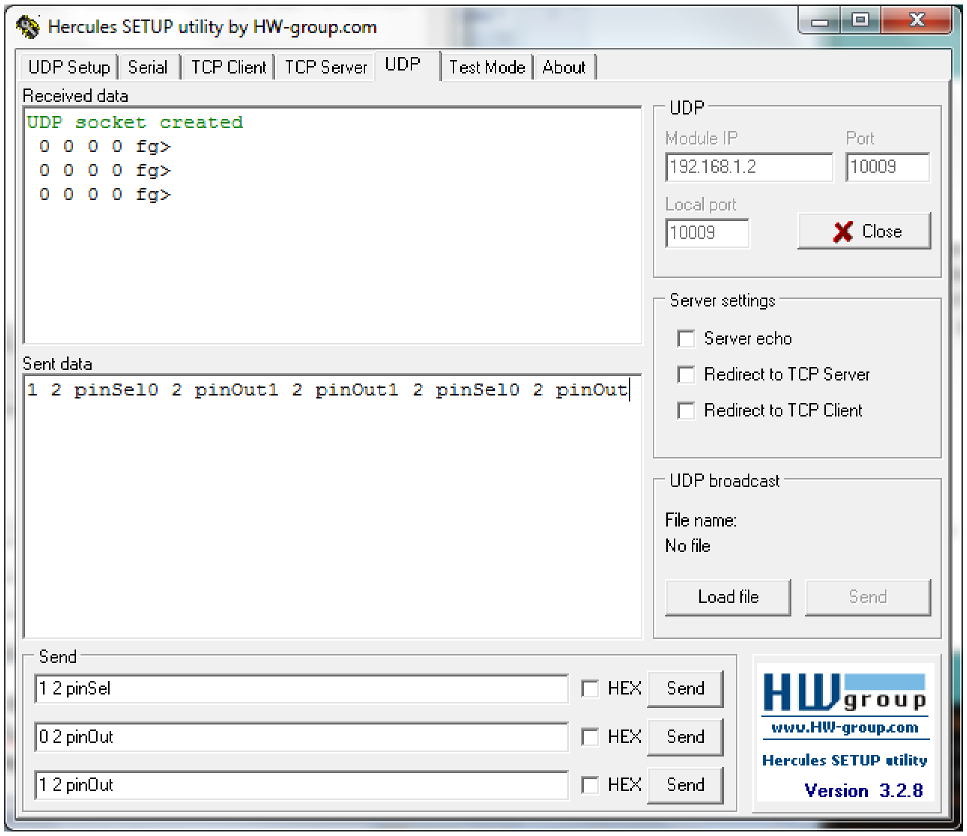
1 2 PinSel	[ ] HEX	Send
0 2 PinOut	[ ] HEX	Send
1 2 PinOut	[ ] HEX	Send

Press the Listen button to create a UDP socket which you can use to send UDP packets to NodeMCU.

At the bottom of UDP window, there are three text boxes. Enter three messages:

```
1 2 pinSel
0 2 pinOut
1 2 pinOut
```

into the text boxes.


Hercules SETUP utility by HW-group.com
UDP Setup | Serial | TCP Client | TCP Server | UDP | Test Mode | About

Received data

```
UDP socket created
 0 0 0 0 fg>
 0 0 0 0 fg>
 0 0 0 0 fg>
```

Sent data

```
1 2 pinSel0 2 pinOut1 2 pinOut1 2 pinSel0 2 pinOut
```

UDP: Module IP 192.168.1.2, Port 10009, Local port 10009

Server settings: Server echo ☐, Redirect to TCP Server ☐, Redirect to TCP Client ☐

UDP broadcast: File name: No file

Send:
- 1 2 pinSel — HEX ☐
- 0 2 pinOut — HEX ☐
- 1 2 pinOut — HEX ☐

Close
Load file
Send
HW group
www.HW-group.com
Hercules SETUP utility
Version 3.2.8

Pressing a Send button to the left of a text box, you send the corresponding UDP packet from Hercules to NodeMCU. You can see that the LED on NodeMCU is turned on and off, with the `pinOut` packets.

espForth Experiments

`WORDS` is a command which dumps the dictionary of eForth. It shows the names of all Forth commands assembled in the system. There are 185 of them. I assembled all the commonly useful Forth commands in upper case, and all the infrequently used system commands in lower case. espForth_44 system is case insensitive. You can type in commands in either upper or lower case.

Commands in dictionary are linked backward. Therefore, the last command `IMMEDIATE` appears first, and the first command `HLD` appears the last, as shown in the console window above.

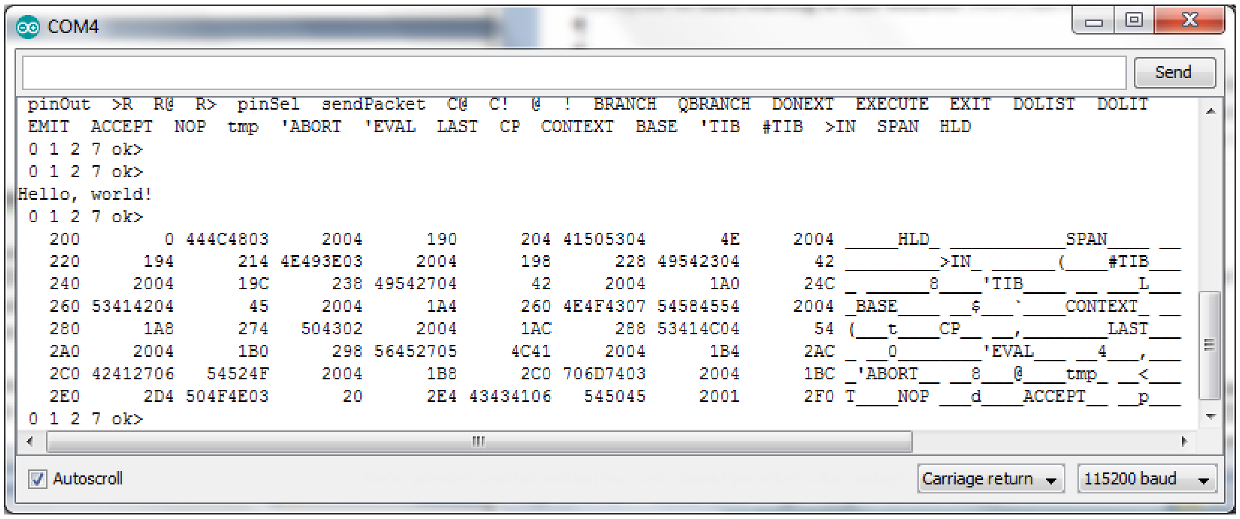
In the original 86eForth v1.0, there were 31 primitive commands and 183 high level compound commands. In espForth_44, there are 83 primitive commands, derived from 67 byte code, and 102 compound commands. However, you cannot easily distinguish primitive commands from compound commands, except reading the assembly source code or examining object code in memory. Actually, a primitive command behaves identically as if written as compound command, except it is faster and often shorter.

Dump Memory

A very useful tool command `DUMP` allows you to examine different areas of memory. For example, the Forth dictionary starts at memory location 0x200. Type:

```
200 100 dump
```

and memory from 0x200 to 0x2FF is dumped on the console:



```
 pinOut  >R  R@  R>  pinSel  sendPacket  C@  C!  @  !  BRANCH  QBRANCH  DONEXT  EXECUTE  EXIT  DOLIST  DOLIT
 EMIT  ACCEPT  NOP  tmp  'ABORT  'EVAL  LAST  CP  CONTEXT  BASE  'TIB  #TIB  >IN  SPAN  HLD
 0 1 2 7 ok>
 0 1 2 7 ok>
Hello, world!
 0 1 2 7 ok>
   200        0 444C4803     2004      190      204 41505304       4E     2004 _____HLD_ ___________SPAN____ __
   220      194      214 4E493E03     2004      198      228 49542304       42 ___________>IN_ _______(____#TIB___
   240     2004      19C      238 49542704       42     2004      1A0      24C _ _______8____'TIB____ __ ___L___
   260 53414204       45     2004      1A4      260 4E4F4307 54584554     2004 _BASE____ __$___`____CONTEXT_ __
   280      1A8      274   504302     2004      1AC      288 53414C04       54 (___t____CP__ __,_______LAST___
   2A0     2004      1B0      298 56452705     4C41     2004      1B4      2AC _ __0_________'EVAL___ __4___,___
   2C0 42412706   54524F     2004      1B8      2C0 706D7403     2004      1BC _'ABORT__ __8___@____tmp_ __<___
   2E0      2D4 504F4E03       20      2E4 43434106   545045     2001      2F0 T____NOP ___d____ACCEPT__ __p___
 0 1 2 7 ok>
```

The memory is displayed in hexadecimal, and contents do not make much sense. However, you might notice that the names of commands `HLD`, `SPAN`, `>IN`, `#TIB`, etc. appear in the ASCII dump on the right hand side. Looking closely, you might be able to identify the link fields, name fields, code fields, and parameter fields in these commands.

PEEK and POKE

espForth system is based on a Virtual Forth Machine written in C. The dictionary is in a data array allocated somewhere in the memory of ESP8266. The address we discussed above are not real addresses. They are addresses relative to the beginning of the data array. The C compiler hides ESP8266 from you for your own good. If you messed up data in real memory, the computer might crash.

As a Forth programmer in heart, I like to know where things are, especially the IO devices. It is my computer. Why am I not allowed to mess around with the IO devices? I am sure that I can make more IO devices run faster, if I have all the design information.

OK. I give you PEEK and POKE, to look at real memories, and real IO devices. PEEK takes an absolute 32-bit address and replaces it with the 32-bit word stored in this absolute address. POKE takes a 32-bit data and a 32-bit address, and stores that data into the address.

Have fun with PEEK to see what's stored where. But, be careful with POKE. Store wrong thing in places might do permanent damages to your computer. Well, it is only a $3 loss.

PWM Tone Generater

I almost forget the speaker I mentioned in Preface. I attached a small speaker between D5 and GND pins. The pins are next to each other. For some strange reasons, Arduino assigned D5 pin to GPIO Port 14. You can send a square wave pulses out on D5 pin with the following commands:

```
DECIMAL 14 440 1000 TONE
```

`TONE` configures GPIO Port 14 as a PWM output port, and sets the frequency of PWM wave at 440 Hz. The duration of this pulse train is 1000 ms, or 1 second. You can play with these commands with different frequencies and duration. You can perhaps play a tune or two for fun.

Test Programs

When I implemented a eForth system, I always tested it by compiling and executing the following new commands:

```
: TEST1 1 2 3 4 ;
: TEST2 IF 1 ELSE 2 THEN . ;
: TEST3 10 FOR R@ . NEXT ;
: TEST4 10 BEGIN DUP WHILE DUP . 1- REPEAT ;
: TEST5 100000 FOR NEXT ;
```

Chapter 2. espForth Virtual Forth Engine

espForth_44.ino

The file espForth_44.ino is a sketch (program) which can be compiled by Arduino IDE, and the resulting program image is uploaded to a WiFi kit like NodeMCU, with an ESP8266 chip. This file serves perfectly as a specification of a Virtual Forth Engine, in terms of a C language functions.

Before diving directly into espForth, I would like to give you an overview of this (VFM) so you can better understand how it is implemented.

I think the following topics are the most important in understanding Forth:

- VFM executes a set of commands.
- All commands are stored in a large data array, called a dictionary.
- Each command is a record in a searchable dictionary. All command 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 data needed by the command.
- There are two types of commands: primitive commands containing executable byte code, and compound command containing token lists. A token is a pointer pointing to code fields of other commands.
- A finite state machine executes byte code sequences stored in code field of primitive commands.
- An address interpreter executes nested lists in compound commands.
- A return stack is used to process nested token lists
- A data stack is used to pass parameters among commands
- A serial terminal is necessary to enter command lines, and to show results thus produced.
- A WiFi network is necessary to send and receive UDP packets to/from this VFM, in parallel with the serial terminal.

The simple goals of a VFM are to impose a Forth language processor on a real computer so that it can interpret or execute a list of Forth commands, separated by spaces:

```
<list of commands>
```

and to create a new command to replace a list of commands:

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

All computable problems can be solved by repeatedly creating new commands to replace lists of existing commands. It is also the simplest and most efficient way to explore solution spaces far and wide, and to arrive at optimized solutions.

Here let's read the source code in espForth_44.ino, to see how this VFM is actually implemented.

Preamble

```
#include <ESP8266WiFi.h>
#include <WiFiUdp.h>
WiFiUDP Udp;
const char* ssid = "SVFIG";//type your ssid
const char* pass = "12345678";//type your password
unsigned int localPort = 10009;      // local port to listen on
# define  FALSE 0
# define  TRUE  -1
# define  LOGICAL ? TRUE : FALSE
# define  LOWER(x,y) ((unsigned long)(x)<(unsigned long)(y))
# define  pop top = stack[(unsigned char)S--]
# define  push  stack[(unsigned char)++S] = top; top =
```

Arduino supplied libraries supporting ESP8266 chip and its WiFi capabilities. Information needed by the C compiler are in the ESP8266WiFi.h header file. For WiFi communication, I choose UDP protocol, which is the simplest way for computers to communicate with one another over a WiFi network. espForth is a proof-of-concept implementation. Once you master UDP protocol, other protocols will be straightforward, though more complicated. The UDP utilities are included in WiFiUdp.h header file.

An object `Udp` is created with the command `WiFiUDP`. Henceforth, all UDP functions are called with `Udp` object prefix.

espForth needs a WiFi network for communication with other computers and devices. For our "IoT for Fun!" WiFi Worshop in the 2017 Bay Area Maker Faire, I used an independent router with its `ssid` as "SVFIG" and its password as "12345678." You will have to set up your own router, or to use your existing router your PC is connected to. Use a large number like 10009 for the `localPort` for your NodeMCU Kit, so that it does not conflict with more popular `localPorts`, which were generally assigned low numbers.

Default values of a `FALSE` flag is 0, and of `TRUE` is -1. espForth considers any non-zero number as a `TRUE` flag. Nevertheless, all espForth commands which generate flags would return a -1 for `TRUE`.

`LOGICAL` is the macro command 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[(unsigned 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[(unsigned char)++S] = top; top =
```

Registers and Arrays

espForth 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};
unsigned char R, S, bytecode ;
long* Pointer ;
long  P, IP, WP, top ;
char* cData ;
long long int d, n, m ;
```

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

Register/Array	Functions
P	Program counter, pointing to pseudo instructions in data[].
IP	Interpreter pointer for address interpreter
WP	Scratch register, usually pointing to parameter field
Rack[]	Return stack, a 256 cell circular buffer
Stack[]	Data stack, a 256 cell circular buffer
R	One byte return stack pointer
S	One byte data stack pointer
top	Cached top element of the data stack
data[]	An array containing Forth dictionary
bytecode	A 8 bit register containing byte code to be executed
cCode	A byte pointer to access data[] array in bytes
d,m,n	Scratch register for 64 bit integers

Dictionary Array

All espForth 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 pointers 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 next chapter of this manual. This dictionary is saved in a file rom_43.h. It is copied into the array `data`[].

```
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 operations coded as C functions. Each function will be assigned a byte code, and a finite state machine is designed to execute byte code placed in code fields of primitive commands in the dictionary. Byte code are pseudo instructions of VFM, like machine instructions in a real computer.

`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;
  WP = P+4;  }
```

`accept` (b u1 -- b u2) Accept u1 characters to b. u2 returned is the actual count of characters received.

```
void accep()
/* UDP accept */
{ while (Udp.parsePacket()==0 && Serial.available()==0) {};
  int len;
  while (!Udp.available()==0) {
    len = Udp.read(cData, top); }
  while (!Serial.available()==0) {
    len = Serial.readBytes(cData, top); }
  if (len > 0) {
    cData[len] = 0; }
  top = len;
  }
```

`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)
  { while (Serial.available() == 0) {};
    push Serial.read();  push -1; }
```

`audio` (pin freq ms --) sends a PWM signal out on a GPIO output pin. Call Arduuino IO library function `tone(pin freq ms)` to do the work. Freq is in Hz, ms is in milliseconds.

```
void audio(void)
{  cell ms=top; pop;
   cell freq=top; pop;
   cell pin=top; pop;
  tone(pin,freq,ms);
}
```

`emit` (c --) Send a character to the serial terminal, and the UDP output buffer. The UDP packet will be send out when CR is executed.

```
void emit(void)
{ Serial.write( (unsigned char) top);
    Udp.write((char) top);
  pop; }
```

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

```
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 the 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[(unsigned char)++R] = IP;
  IP = WP;
  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[(unsigned char)R--];
  next(); }
```

`execu` (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.

```
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 `nonext`. 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[(unsigned char)R]) {
    rack[(unsigned char)R] -= 1 ;
    IP = data[IP>>2];
  } else { IP += 4;  (unsigned char)R-- ;  }
  next(); }
```

`qbran` (f --) Test top element 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. `CODE`

```
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[(unsigned 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] = (unsigned char) stack[(unsigned 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];  }
```

`sendPacket`(--) Send an UDP packet out to WiFi network. It is executed only by CR command.

```
void sendPacket(void)
{  Udp.endPacket();
   Udp.beginPacket(Udp.remoteIP(), Udp.remotePort()); }
```

`pinSel`(1/0 pin --) Select a GPIO pin for output of input. A 1 is for output, and a 0 is for input.

```
void pinSel(void)
{   WP=top; pop;
    pinMode(WP,top);
    pop; }
```

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

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

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

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

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

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

`pinOut`(1/0 pin --) Set or reset a GPIO output pin. 1 to pull high, and 0 to pull low.

```
void pinOut(void)
{   WP=top; pop;
    digitalWrite(WP,top);
    pop; }
```

`pinIn`(pin – n) Read an GPIO input pin.

```
void pinIn(void)
{   top = digitalRead(top); }
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`nop` (--) No operation.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
void slash(void)
{ top = (top) ? stack[(unsigned char) S--] / top : (stack[(unsigned 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 = (signed long long int)top;
  m = (signed long long int)stack[(unsigned char) S];
  m *= d;
  top = (signed long)(m >> 32);
  stack[(unsigned char) S] = (signed long)m; }
```

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

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

`pstor` (n a --) Add n to the contents at address a.
```
void pstor(void)
{   data[top>>2] += stack[(unsigned char)S--], pop;  }
```

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

`dovar`(-- a) Return address of the next cell in current token list
```
void dovar(void)
{   push WP; }
```

`maxx` (n1 n2 -- n) Return the greater of two top stack items.
```
void maxx(void)
{   if (top < stack[(unsigned char)S]) pop;
    else (unsigned char)S--; }
```

`minn` (n1 n2 -- n) Return the smaller of top two stack items.
```
void minn(void)
{   if (top < stack[(unsigned char)S]) (unsigned char)S--;
    else pop; }
```

`poke` (n a --) writes one word n to an absolute address a in ESP8266.
```
void poke(void)
{    Pointer = (long*)top; *Pointer = stack[(unsigned char)S--];
  pop;  }
```

`peeek` (a – n) reads one word n from an absolute address a in ESP8266.
```
void peeek(void)
{   Pointer = (long*)top; top = *Pointer;  }
```

Byte Code Array

There are 67 functions defined in VFM as shown before. Each of these functions are 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 spilled over into the subsequent parameter field, if a primitive command is very complicated. VFM has a finite state machine, which will be discussed shortly, to execute byte code sequences. 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 byte code, 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 might differ, inversely proportional to the efforts in programming..

```
void (*primitives[67])(void) = {
    /* case 0 */ nop,
    /* case 1 */ accep,
    /* case 2 */ qrx,
    /* case 3 */ emit,
    /* 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 */ sendPacket,
    /* case 17 */ pinSel,
    /* case 18 */ rfrom,
    /* case 19 */ rat,
    /* case 20 */ tor,
    /* case 21 */ pinOut,
    /* case 22 */ pinIn,
    /* 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 */ maxx,
    /* case 63 */ minn,
    /* case 64 */ audio,
    /* case 65 */ poke,
    /* case 66 */ peeek };
```

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 < 67) { primitives[code](); }
    else {
    Serial.print ("\n Illegal code= ") ;
    Serial.print (code) ;
    Serial.print ("  Address= ") ;
    Serial.print ( P ) ;
  } }
```

VFM initialization

To start the VFM running, some of the registers have to be initialized in the `setup()` function required by Arduino IDE:

```
void setup()
{
  P = 0x180;
  WP = 0x184;
  IP = 0;
  S = 0;
  R = 0;
  top = 0;
  cData = (char *) data;
```

`setup`() also initializes the USART0 to 115,200 bauds, and displays a sign-on message. It then initialize the WiFi radio, and sets it up to receive and transmit UDP message. The program counter P is initialized to a code segment stored in the dictionary location 0x180. This code segment jumps COLD at location 0x1A38, which starts the Forth interpreter.

```
  Serial.begin(115200);
  delay(100);
// attempt to connect to Wifi network:
  WiFi.begin(ssid, pass);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");
  Serial.println("WiFi connected");
  Serial.print("SSID: ");
  Serial.println(WiFi.SSID());
  IPAddress ip = WiFi.localIP();
  Serial.print("IP Address: ");
  Serial.println(ip);
  long rssi = WiFi.RSSI();
  Serial.print("signal strength (RSSI):");
  Serial.print(rssi);
  Serial.println(" dBm");
  Serial.println("Starting connection to server...");
  // if you get a connection, report back via serial:
  Udp.begin(localPort);
  Serial.print("Local Port: ");
  Serial.println(localPort);
}
```

VFM Byte Code Sequencer

At the very end of espForth_44.ino file, and we have the familiar Arduino function of `loop`(). `loop`() is essentially a byte code sequencer in FVM to execute byte code or pseudo instructions stored in the `data`[] array. It is simply:

```
void loop() {
  while (TRUE) {
    bytecode = (unsigned char)cData[P++];
    execute(bytecode);
  } }
```

This much simpler byte sequencer replaced the earlier Finite State Machine which reads a 32 bit word from dictionary, decodes the byte code in it, and then execute the byte code in sequence. After the last byte code is executed, it reads the next word and processes it, etc. It is much easier to read and execute byte code directly.

Chapter 3. Metacompilation of the espForth

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 espForth, 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 sketch on Arduino IDE, and causes NodeMCU Kit to run like a Forth machine.

In this Chapter, I will discuss how the F# metacompiler construct the Forth dictionary used in espForth_44.ino. The discussion follows the source files in their loading sequence.

The immediate goal of espForth 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 byte code assembly commands, which assembles byte code into the code fields of primitive commands in target dictionary. Examples are:

```
0 INST nop,
1 INST accept,   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 ACCEPT accept, 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 @ ;;
```

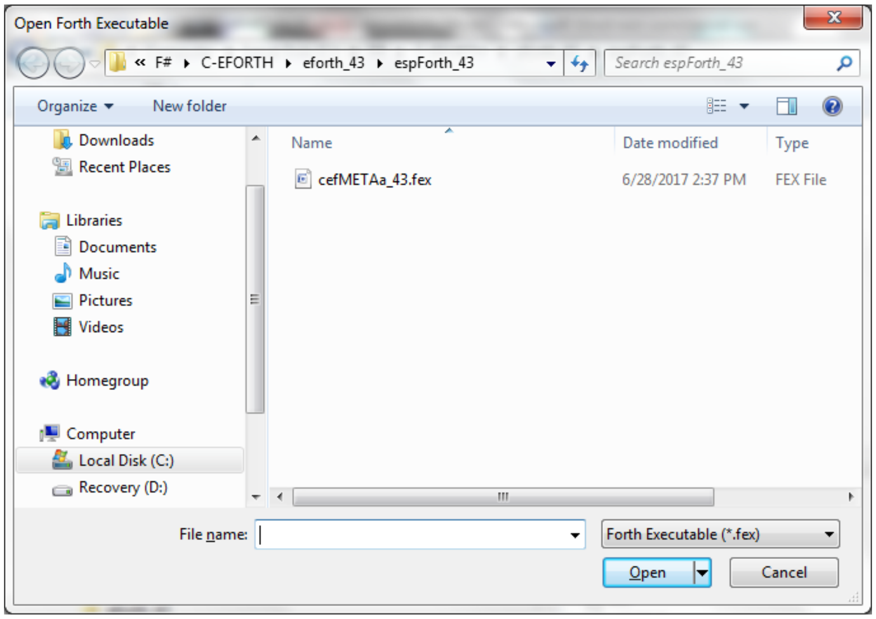
`CODE` and `::` (colon-colon) also mark the code field addresses of the new commands they created. When these commands are later executed, they compile their respective code field addresses into the token list under construction.

`INST,` `CODE` and `::` (colon-colon) are defined in cefASMa_43.f.

Metacompiling espForth

All source code of the espForth eForth system is contained in the espForth_44.zip file. F# system and its Windows utilities are also included here.

Unzip file espForth_44.zip and put all the files into a folder named “espForth_44”. Start F# by double clicking F#.exe in the espForth_44 folder, and a file select window is opened:

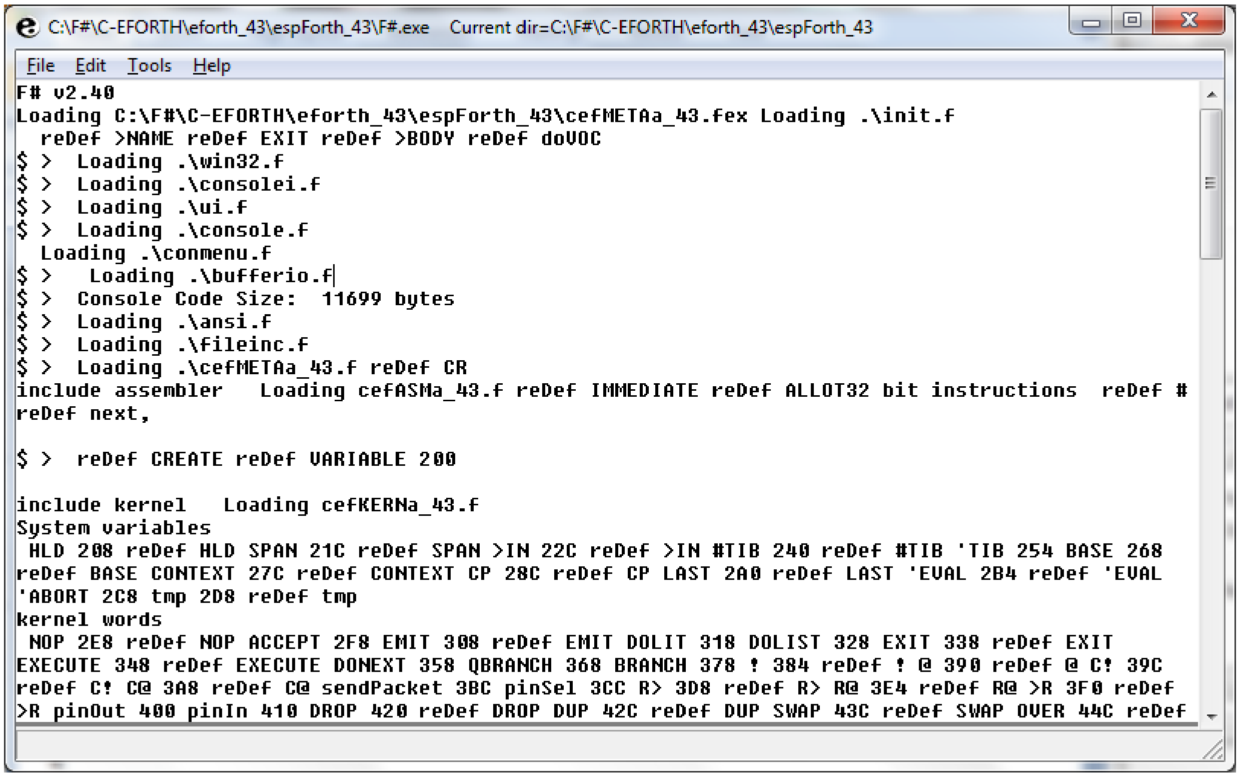


Name	Date modified	Type
cefMETAa_43.fex	6/28/2017 2:37 PM	FEX File

File name:
Forth Executable (*.fex)

F# organizes projects using .fex files. Each .fex file loads all the files needed in a project. Metacompiler of espForth_43 is contained in the ceMETAa_43.fex. Click Open button to run it.

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


C:\F#\C-EFORTH\eforth_43\espForth_43\F#.exe   Current dir=C:\F#\C-EFORTH\eforth_43\espForth_43
File Edit Tools Help

```
F# v2.40
Loading C:\F#\C-EFORTH\eforth_43\espForth_43\cefMETAa_43.fex Loading .\init.f
  reDef >NAME reDef EXIT reDef >BODY reDef doVOC
$ >  Loading .\win32.f
$ >  Loading .\consolei.f
$ >  Loading .\ui.f
$ >  Loading .\console.f
  Loading .\conmenu.f
$ >   Loading .\bufferio.f
$ >  Console Code Size:  11699 bytes
$ >  Loading .\ansi.f
$ >  Loading .\fileinc.f
$ >  Loading .\cefMETAa_43.f reDef CR
include assembler   Loading cefASMa_43.f reDef IMMEDIATE reDef ALLOT32 bit instructions  reDef #
reDef next,

$ >  reDef CREATE reDef VARIABLE 200

include kernel   Loading cefKERNa_43.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 ACCEPT 2F8 EMIT 308 reDef EMIT DOLIT 318 DOLIST 328 EXIT 338 reDef EXIT
EXECUTE 348 reDef EXECUTE DONEXT 358 QBRANCH 368 BRANCH 378 ! 384 reDef ! @ 390 reDef @ C! 39C
reDef C! C@ 3A8 reDef C@ sendPacket 3BC pinSel 3CC R> 3D8 reDef R> R@ 3E4 reDef R@ >R 3F0 reDef
>R pinOut 400 pinIn 410 DROP 420 reDef DROP DUP 42C reDef DUP SWAP 43C reDef SWAP OVER 44C reDef
```

The first number of lines show a number of files loaded. The last file loaded is cefMETAa_43.f, which loads in the metacompiler, and start building espForth dictionary. The contents of ceMETAa_43.fex are as following:

```
 ( espForth_43, 28jun17cht )
\ 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_43.f

cr .( Version FIX 12SEP09CHT )
```

Files loaded by ceMETAa_43.fex are for these purposes:

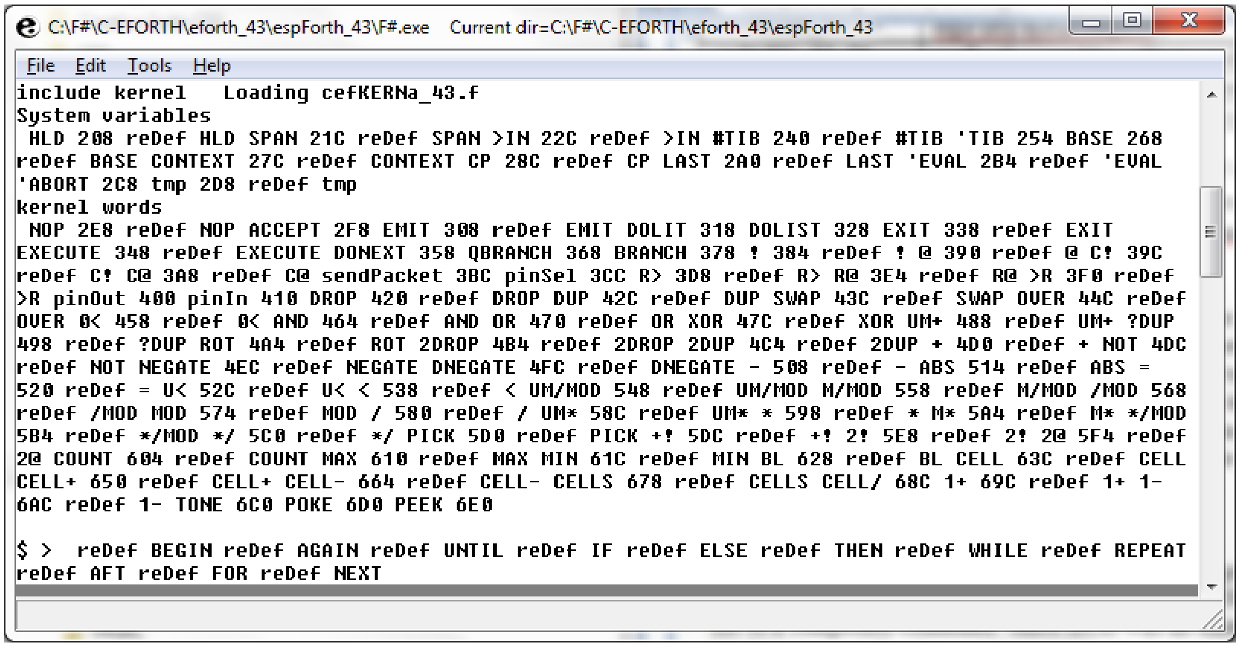
`init.f`	Extend F# core to manage vocabulary and constants
`win32.f`	WinAPI interface and CallBack
`consolei.f`	API constants and prototypes
`ui.f`	Windows user interface
`console.f`	Create main console window
`conmenu.f`	Windows console and menu
`bufferio.f`	Buffered console input and output
`ansi.f`	Add words for ANSI Forth compatibility
`fileinc.f`	Windows file interface API
`cefMETAa_43.f`	espForth metacompiler
`cefASMa_43.f`	espForth assembler of byte code pseudo instructions
`cefKERNa_43.f`	Compile espForth primitive commands
`cEFa_43.f`	Compile espForth compound commands

cefMETAa_43.f allocates a large data array `ram` in F# memory, to host the dictionary target to ESP8266 chip. The contents in the `ram` array will be saved as a text file rom_43.h, which will eventually be copied into the `data`[] array in Arduino sketch espForth_44.ino to run espForth on NodeMCU.

When cefKERNa_43.f is being loaded, new primitive commands are added to the `ram` array dictionary. F# and espForth 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:

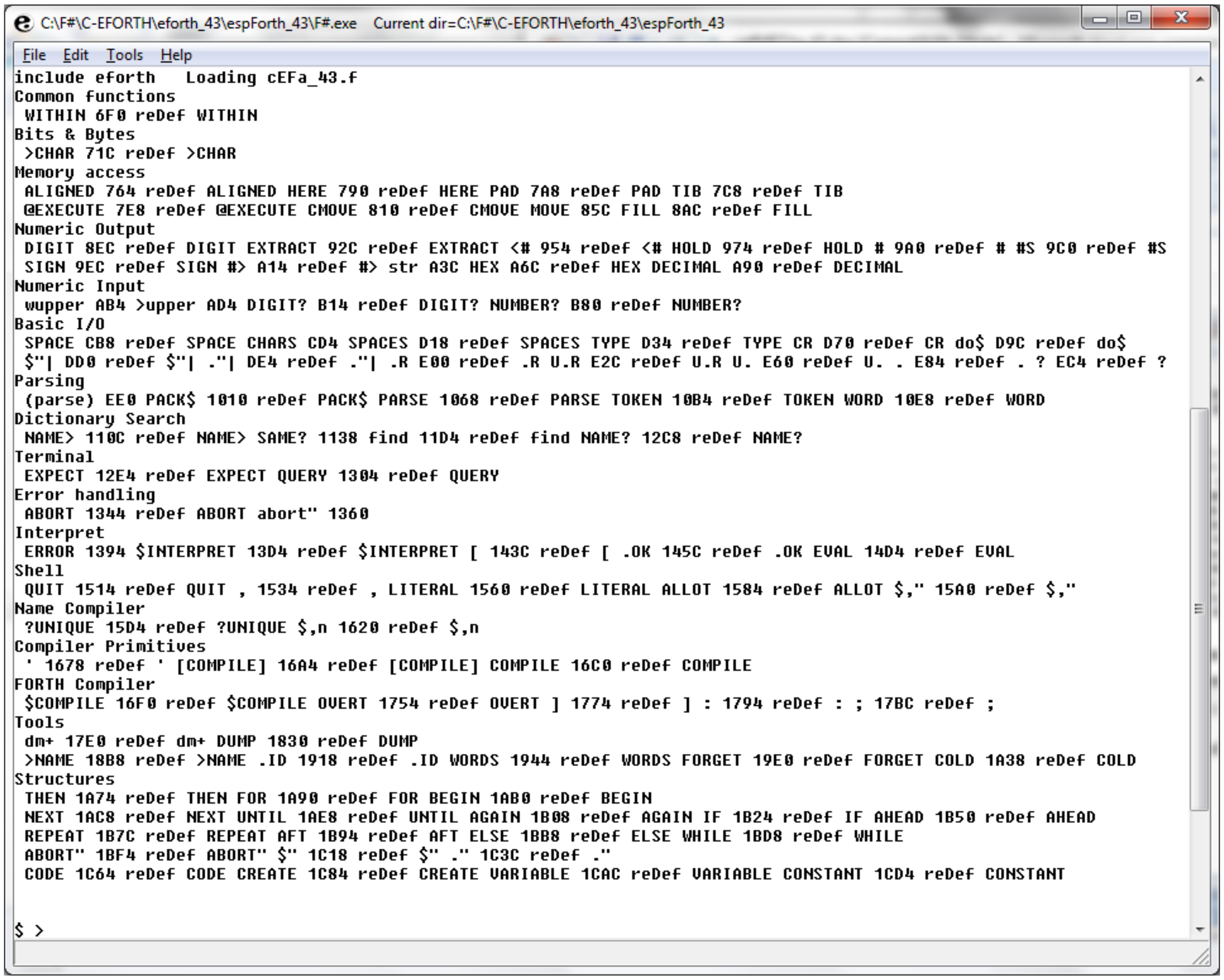
```
      HLD 208 redef HLD
```

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\eforth_43\espForth_43\F#.exe   Current dir=C:\F#\C-EFORTH\eforth_43\espForth_43
File Edit Tools Help

```
include kernel   Loading cefKERNa_43.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 ACCEPT 2F8 EMIT 308 reDef EMIT DOLIT 318 DOLIST 328 EXIT 338 reDef EXIT
EXECUTE 348 reDef EXECUTE DONEXT 358 QBRANCH 368 BRANCH 378 ! 384 reDef ! @ 390 reDef @ C! 39C
reDef C! C@ 3A8 reDef C@ sendPacket 3BC pinSel 3CC R> 3D8 reDef R> R@ 3E4 reDef R@ >R 3F0 reDef
>R pinOut 400 pinIn 410 DROP 420 reDef DROP DUP 42C reDef DUP SWAP 43C reDef SWAP OVER 44C reDef
OVER 0< 458 reDef 0< AND 464 reDef AND OR 470 reDef OR XOR 47C reDef XOR UM+ 488 reDef UM+ ?DUP
498 reDef ?DUP ROT 4A4 reDef ROT 2DROP 4B4 reDef 2DROP 2DUP 4C4 reDef 2DUP + 4D0 reDef + NOT 4DC
reDef NOT NEGATE 4EC reDef NEGATE DNEGATE 4FC reDef DNEGATE - 508 reDef - ABS 514 reDef ABS =
520 reDef = U< 52C reDef U< < 538 reDef < UM/MOD 548 reDef UM/MOD M/MOD 558 reDef M/MOD /MOD 568
reDef /MOD MOD 574 reDef MOD / 580 reDef / UM* 58C reDef UM* * 598 reDef * M* 5A4 reDef M* */MOD
5B4 reDef */MOD */ 5C0 reDef */ PICK 5D0 reDef PICK +! 5DC reDef +! 2! 5E8 reDef 2! 2@ 5F4 reDef
2@ COUNT 604 reDef COUNT MAX 610 reDef MAX MIN 61C reDef MIN BL 628 reDef BL CELL 63C reDef CELL
CELL+ 650 reDef CELL+ CELL- 664 reDef CELL- CELLS 678 reDef CELLS CELL/ 68C 1+ 69C reDef 1+ 1-
6AC reDef 1- TONE 6C0 POKE 6D0 PEEK 6E0

$ >  reDef BEGIN reDef AGAIN reDef UNTIL reDef IF reDef ELSE reDef THEN reDef WHILE reDef REPEAT
reDef AFT reDef FOR reDef NEXT
```

Then, cEFa_43.f file is loaded, and all the compound commands are compiled.


C:\F#\C-EFORTH\eforth_43\espForth_43\F#.exe   Current dir=C:\F#\C-EFORTH\eforth_43\espForth_43
File Edit Tools Help

```
include eforth   Loading cEFa_43.f
Common functions
 WITHIN 6F0 reDef WITHIN
Bits & Bytes
 >CHAR 71C reDef >CHAR
Memory access
 ALIGNED 764 reDef ALIGNED HERE 790 reDef HERE PAD 7A8 reDef PAD TIB 7C8 reDef TIB
 @EXECUTE 7E8 reDef @EXECUTE CMOVE 810 reDef CMOVE MOVE 85C FILL 8AC reDef FILL
Numeric Output
 DIGIT 8EC reDef DIGIT EXTRACT 92C reDef EXTRACT <# 954 reDef <# HOLD 974 reDef HOLD # 9A0 reDef # #S 9C0 reDef #S
 SIGN 9EC reDef SIGN #> A14 reDef #> str A3C HEX A6C reDef HEX DECIMAL A90 reDef DECIMAL
Numeric Input
 wupper AB4 >upper AD4 DIGIT? B14 reDef DIGIT? NUMBER? B80 reDef NUMBER?
Basic I/O
 SPACE CB8 reDef SPACE CHARS CD4 SPACES D18 reDef SPACES TYPE D34 reDef TYPE CR D70 reDef CR do$ D9C reDef do$
 $"| DD0 reDef $"| ."| DE4 reDef ."| .R E00 reDef .R U.R E2C reDef U.R U. E60 reDef U. . E84 reDef . ? EC4 reDef ?
Parsing
 (parse) EE0 PACK$ 1010 reDef PACK$ PARSE 1068 reDef PARSE TOKEN 10B4 reDef TOKEN WORD 10E8 reDef WORD
Dictionary Search
 NAME> 110C reDef NAME> SAME? 1138 find 11D4 reDef find NAME? 12C8 reDef NAME?
Terminal
 EXPECT 12E4 reDef EXPECT QUERY 1304 reDef QUERY
Error handling
 ABORT 1344 reDef ABORT abort" 1360
Interpret
 ERROR 1394 $INTERPRET 13D4 reDef $INTERPRET [ 143C reDef [ .OK 145C reDef .OK EVAL 14D4 reDef EVAL
Shell
 QUIT 1514 reDef QUIT , 1534 reDef , LITERAL 1560 reDef LITERAL ALLOT 1584 reDef ALLOT $," 15A0 reDef $,"
Name Compiler
 ?UNIQUE 15D4 reDef ?UNIQUE $,n 1620 reDef $,n
Compiler Primitives
 ' 1678 reDef ' [COMPILE] 16A4 reDef [COMPILE] COMPILE 16C0 reDef COMPILE
FORTH Compiler
 $COMPILE 16F0 reDef $COMPILE OVERT 1754 reDef OVERT ] 1774 reDef ] : 1794 reDef : ; 17BC reDef ;
Tools
 dm+ 17E0 reDef dm+ DUMP 1830 reDef DUMP
 >NAME 18B8 reDef >NAME .ID 1918 reDef .ID WORDS 1944 reDef WORDS FORGET 19E0 reDef FORGET COLD 1A38 reDef COLD
Structures
 THEN 1A74 reDef THEN FOR 1A90 reDef FOR BEGIN 1AB0 reDef BEGIN
 NEXT 1AC8 reDef NEXT UNTIL 1AE8 reDef UNTIL AGAIN 1B08 reDef AGAIN IF 1B24 reDef IF AHEAD 1B50 reDef AHEAD
 REPEAT 1B7C reDef REPEAT AFT 1B94 reDef AFT ELSE 1BB8 reDef ELSE WHILE 1BD8 reDef WHILE
 ABORT" 1BF4 reDef ABORT" $" 1C18 reDef $" ." 1C3C reDef ."
 CODE 1C64 reDef CODE CREATE 1C84 reDef CREATE VARIABLE 1CAC reDef VARIABLE CONSTANT 1CD4 reDef CONSTANT


$ >
```

Finally, the system variable a compile, the entire target dictionary is finished, and is written out in the rom_43.h file.

Copy the contents of rom_43.h into the data[4096] array in espForth_44.ino and you can build and test espForth_44 on a NodeMCU Kit.

Metacompiler cefMETAa_43.f

Metacompiler is a term used by Forth programmers to describe the process of building 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 espForth metacompiler is contained in file cefMETAa_43.f. It allocates a data array ram, and deposits records of primitive commands and compound commands to build a dictionary for espForth. The Virtual Forth Machine (VFM) was programmed in espForth_43.ino in 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_43.f loads source code from three other files to do specific jobs:

`cefASMa_43.f`	espForth assembler
`cefKERNa_43.f`	Assemble primitive commands
`cEFa_43.f`	Compile compound commands

Source code in cefMETAa_43.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 NOP 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 compile tokens or assemble byte code into the target dictionary. There are numerous occasions where 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, we can still exercise its original behavior by invoking the alternate name.

For example, “+” is a Forth command that adds the top two numbers on the data stack in the F# system. Then in the cefKERNa_43.f file, a new “DUP” command is defined to assemble a dup, instruction in the target espForth 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 ;
```

The espForth executes commands and accesses data in the dictionary, range 0-1FFF. In F# we allocate a 32k byte memory array, “`ram`”, to hold the espForth target dictionary. This array contains code and data to be copied into espForth data[] arry, to be executed on the ESP8266 chip.

`RAM` (-- a) Memory array in F# for the espForth target dictionary. It has a logical base address of 0 for the espForth. 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 espForth.

```
: 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  ;
```

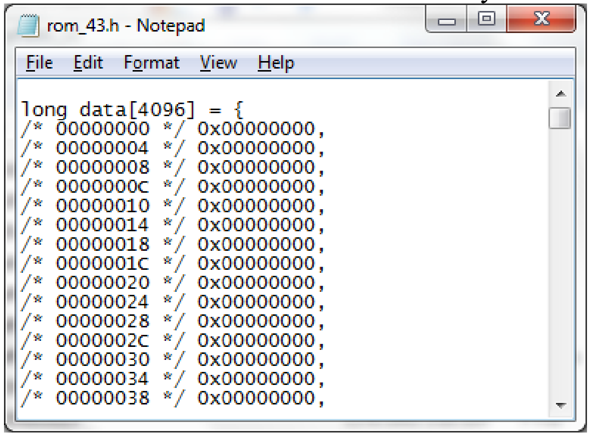
`SHOW` (a – a+128) Display 128 words in target from address “a”. It also returns a+128 to “show” the next block of 128 words.

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

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

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

The espForth metacompiler builds a target dictionary for the ESP8266 chip in “ram. This dictionary eventually will be imported to the espForth_44.ino so that this dictionary will be incorporated in espForth. Arduino 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:



```
long data[4096] = {
/* 00000000 */ 0x00000000,
/* 00000004 */ 0x00000000,
/* 00000008 */ 0x00000000,
/* 0000000C */ 0x00000000,
/* 00000010 */ 0x00000000,
/* 00000014 */ 0x00000000,
/* 00000018 */ 0x00000000,
/* 0000001C */ 0x00000000,
/* 00000020 */ 0x00000000,
/* 00000024 */ 0x00000000,
/* 00000028 */ 0x00000000,
/* 0000002C */ 0x00000000,
/* 00000030 */ 0x00000000,
/* 00000034 */ 0x00000000,
/* 00000038 */ 0x00000000,
```

This dictionary is written to a text file rom_43.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_43.h for writing.

```
: open-mif-file
   Z" rom_43.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 Arduino 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 espForth dictionary from memory array “ram” to the rom_43.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_43.h file.

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

`write-mif-file` (--) open rom_43.h file, write a header, write data, write trailer, and then closes the file. rom_43.h containing 4K words of the espForth dictionary.

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

The espForth metacompiler continues to load the byte code assembler in cefASM_43.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 espForth.

```
CR .( include assembler )
FLOAD cefASMa_43.f
```

After the assembler is built, we are ready to build the kernel part of espForth dictionary. All primitive commands are assembled. The kernel starts at location 0x200, leaving rooms for the Terminal Input Buffer TIB in the area 0-0x17F. System variables from 0x180-0x1FF.

```
$200 ORG
CR .( include kernel )
FLOAD cefKERNa_43.f
```

With the kernel in place, high level compound commands are compiled immediately after the kernel, by loading cEFa_43.f. The top espForth dictionary is at 0x1DAC 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_43.f
H forth_@
```

ceASMa_43.f, cdKERNa_43.f, and cEFa_43.f files will be discussed in separate chapters. Finally, several system variables must be initialized properly so that the Forth interpreter can work properly on boot.

When ESP8266 boots up, the IP register is initialized to 0x180, so we have to have some valid byte code at this. The espForth boot up routine is the command `COLD`. Therefore, in memory location 0x180, we assemble a `dolist, COLD` instruction.

```
CRR
$180 ORG
dolist, aanew
COLD
```

System variables are in the area between 0x1A0-0x1BF. They contain vital information for the espForth system to work properly. Only the following system variables have to be initialized:

```
$1A0 ORG $0                                  #, \ TIB
$1A4 ORG $10                                 #, \ BASE
$1A8 ORG lastH forth_@                       #, \ CONTEXT
$1AC ORG                                     #, \ CP
$1B0 ORG lastH forth_@                       #, \ LAST
$1B4 ORG forth_' $INTERPRET >body forth_@ #, \ 'EVAL
$1B8 ORG forth_' QUIT >body forth_@       #, \ 'ABORT
$1BC ORG 0                                   #, \ tmp
```

System Variable	Address	Initial Value	Function
`'TIB`	`0x1A0`	`0`	Pointer to Terminal Input Buffer.
`BASE`	`0x1A4`	`0x10`	Number base for hexadecimal numeric conversions.
`CONTEXT`	`0a1A8`	`0x1DDC`	Pointer to name field of last command in dictionary.
`CP`	`0x1AC`	`0x1DE8`	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`	`0x1B0`	`0x1DDC`	Pointer to name field of last command.
`'EVAL`	`0x1B4`	`0x13D4`	Execution vector of text interpreter, initialized to point to $`INTERPRET`. It may be changed to point to $`COMPILE` in compiling mode.
`'ABORT`	`0x1B8`	`0x1514`	Pointer to `QUIT` command to handle error conditions.
`tmp`	`0x1BC`	`0`	Scratch pad.

At last, write the contents of dictionary to `rom_43.h`.

```
write-mif-file
```

Done.

Chapter 4. espForth Assembler

Byte Code Assembler

The cefASM_43.f file contains a byte code assembler for espForth. 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.

espForth has 32-bit long words and 8-bit byte code. Byte code 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 optimizing assembler very simple and very efficient.

The espForth 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 assemble first build a header, with a link field and a name field. After that, the assembler simply pack consecutive bytes with byte code until the primitive commands is completed.

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

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

`COMPILE-ONLY ( -- )` Patch Bit 6 in first word 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 word 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 slot 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 6-bit machine instruction from a 32-bit instruction pattern. 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 primitive assembler and compiler. The espForth assembler is an extension of this primitive 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 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 ;
```

`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.

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

espForth Byte Code

Here are all the byte code assembly commands to be used to assemble byte code in primitive commands to build espForth kernel.

```
decimal
\ 0 INST nop,
1 INST accept,   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 sendPacket,
17 INST pinSel, 18 INST rfrom, 19 INST rat,   20 INST tor,
21 INST pinOut, 22 INST pinIn, 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,   64 INST audio,
65 INST poke,   66 INST peek,
```

Command Headers

In espForth, 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 built all the tools to compile new commands into the target dictionary, which will eventually run ESP8266 chip. 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 header for a 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 espForth 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> @ #, ;
```

`:: ( -- )` Create a new compound command in espForth 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> @ #, ;
```

Chapter 5. espForth Kernel

The kernel of espForth_44 is defined in file cefKERNa_43.f. The byte code it refers to are defined in espForth_44.ino.

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 espForth_44 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 commands, there are 4 byte code in the code field. The parameter field may have more byte code. A byte code sequence is 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 espForth_44, `next`() is defined as:

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

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 = WP;
  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 = (long) 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 espForth_44, 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 espForth_44 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      190 docon, next, #,       \ scratch
```

`SPAN` (-- a) Hold a character count received by `EXPECT`.
```
CODE SPAN     194 docon, next, #,       \ #chars input by EXPECT
```

`>IN` (-- a) Hold the current character pointer while parsing input stream.
```
CODE >IN      198 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     19C docon, next, #,       \ #chars in the input buffer
```

`'TIB` (-- a) Hold the current address of terminal input buffer.
```
CODE 'TIB     1A0 docon, next, #,       \ ptr to TIB
```

`BASE` (-- a) Store the current radix base for numeric I/O. Default to 0x10.
```
CODE BASE     1A4 docon, next, #,       \ number base
```

`CONTEXT` (-- a) Return a pointer to the last name field in Forth dictionary.
```
CODE CONTEXT  1A8 docon, next, #,       \ first search vocabulary
```

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

```
CODE CP       1AC 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     1B0 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    1B4 docon, next, #,       \ interpret/compile vector
```

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

```
CODE 'ABORT   1B8 docon, next, #,       \ ptr to error handler
```

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

```
CODE tmp      1BC docon, next, #,       \ ptr to converted # string
```

System Interface Commands

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

```
CODE ACCEPT accept, next,
void accep()
/* UDP accept */
{ while (Udp.parsePacket()==0 && Serial.available()==0) {};
  int len;
  while (!Udp.available()==0) {
    len = Udp.read(cData, top); }
  while (!Serial.available()==0) {
    len = Serial.readBytes(cData, top); }
  if (len > 0) {
    cData[len] = 0; }
  top = len;
  }
```

`EMIT` (c --) Send a character to the serial terminal, and the UDP output buffer. The UDP packet will be send out when CR is executed.

```
CODE EMIT txsto, next,
void txsto(void)
{ Serial.write( (unsigned char) top);
    Udp.write((char) top);
  pop; }
```

`sendPacket`() Send an UDP packet out to WiFi network. It is executed only by CR command.

```
CODE sendPacket sendPacket, next,
void sendPacket(void)
{  Udp.endPacket();
```

```
   Udp.beginPacket(Udp.remoteIP(), Udp.remotePort()); }
```

`pinSel`(1/0 pin --) Select a GPIO pin for output of input. A 1 is for output, and a 0 is for input.

```
CODE pinSel pinSel, next,
void pinSel(void) { w=top; pop; pinMode(w,top); pop; }
```

`pinOut`(1/0 pin --) Set or reset a GPIO output pin. 1 to pull high, and 0 to pull low.

```
CODE pinOut pinOut, next,
void pinOut(void) { w=top; pop; digitalWrite(w,top); pop; }
```

`pinIn`(pin – n) Read an GPIO input pin.

```
CODE pinIn pinIn, next,
void pinIn(void) { top = digitalRead(top); }
```

`audio` (pin freq ms --) sends a PWM signal out on a GPIO output pin. Call Arduuino IO library function `tone`(`pin freq ms`) to do the work. Freq is in Hz, ms is in milliseconds.

```
CODE TONE audio, next,
void audio(void)
{  cell ms=top; pop;
   cell freq=top; pop;
   cell pin=top; pop;
  tone(pin,freq,ms);
}
```

`POKE` (n a --) writes one word n to an absolute address a in ESP8266.

```
CODE POKE poke, next,
void poke(void) { Pointer = (cell*)top; *Pointer = stack[(unsigned
char)S--]; pop; }
```

`PEEK` (a – n) reads one word n from an absolute address a in ESP8266.

```
CODE PEEK peek, next,
void peeek(void) { Pointer = (cell*)top; top = *Pointer; }
```

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(); }
```

`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[(unsigned char)++R] = IP; IP = P; 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 = (cell) rack[(unsigned 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) { rack[(unsigned char)++R] = IP; P = top;  pop; jump(); }
```

`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[(unsigned char)R]) { rack[(unsigned char)R] -= 1 ; IP =
data[IP>>2]; }
    else { IP += 4;  (unsigned char)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[(unsigned 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] = (unsigned char) stack[(unsigned 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 = (cell) cData[top]; }
```

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

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

`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[(unsigned 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[(unsigned 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[(unsigned char)++S] = top; }
```

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

```
CODE SWAP swap, next,
void swap(void) { w = top; top = stack[(unsigned char)S];
     stack[(unsigned char)S] = w;  }
```

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

```
CODE OVER over, next,
void over(void) { push stack[(unsigned 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[(unsigned char)++S] = top ; }
```

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

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

`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[(unsigned char)(S-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[(unsigned char)S--] == top) LOGICAL; }
```

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

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

< (n1 n2 -- t) Signed compare of top two items.
```
CODE < less, next,
void less(void) { top = (stack[(unsigned 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[(unsigned char)S--]; }
```

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

XOR (w w -- w) Bitwise exclusive OR.
```
CODE XOR xorr, next,
void xorr(void) { top ^= stack[(unsigned char)S--]; }
```

MAX (n1 n2 -- n) Return the greater of two top stack items.
```
CODE MAX max, next,
void maxx(void)
{   if (top < stack[(unsigned char)S]) pop;
    else (unsigned char)S--; }
```

MIN (n1 n2 -- n) Return the smaller of top two stack items.
```
CODE MIN min, next,
void minn(void)
{   if (top < stack[(unsigned char)S]) (unsigned 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[(unsigned char)S] += top;
    top = LOWER(stack[(unsigned char)S], top);  }
```

+ (w w -- sum) Add top two items.
```
CODE + plus, next,
void plus(void) { top += stack[(unsigned 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[(unsigned 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[(unsigned char) S]);
  n = (long long int)((unsigned long)stack[(unsigned char) (S - 1)]);
  n += m << 32; pop;
  top = (unsigned long)(n / d);
  stack[(unsigned 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[(unsigned char) S]);
  n = (signed long long int)((signed long)stack[(unsigned char) S - 1]);
  n += m << 32; pop;
  top = (signed long)(n / d);
  stack[(unsigned 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) {
    w = stack[(unsigned char) S] / top;
    stack[(unsigned char) S] %= top;
    top = w; } }
```

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

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

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

```
CODE / slash, next,
void slash(void)
{ top = (top) ? stack[(unsigned char) S--] / top : (stack[(unsigned 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[(unsigned char) S];
  m *= d;
  top = (unsigned long)(m >> 32);
  stack[(unsigned char) S] = (unsigned long)m; }
```

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

```
CODE * star, next,
void star(void) { top *= stack[(unsigned 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[(unsigned char) S];
  m *= d;
  top = (signed long)(m >> 32);
  stack[(unsigned 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[(unsigned char) S];
  n = (signed long long int)stack[(unsigned char) (S - 1)];
  n += m << 32; pop;
  top = (signed long)(n / d);
  stack[(unsigned 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[(unsigned char) S];
  n = (signed long long int)stack[(unsigned 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, #,
```

Control Structure Commands

Above we have all the primitive commands in the kernel of espForth. 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 BRANCH instruction to "begin".
```
: AGAIN ( a -- ) BRANCH #, ;
```

`UNTIL` Terminate a begin-until loop if top is not zero.
```
: 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 BRANCH 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. espForth Compound Commands

The dictionary of espForth 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_43.f. They are discussed in their loading order. The loading order is very important in the espForth metacompiler, because forward referencing is not allowed. All assembling and compiling processes are accomplished in a single pass.

espForth 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_43.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 redeined in the metacompiler so they can construct control structures properly in the target dictionary. The only exception is the handling of literals. An integer encountered by metacompiler would remain of the data stack. If you intended to compiler it as a literal in the 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_43.f read identically like regular Forth code, except that integer literals have to be handled explicitly.

Common Commands

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

```
:: 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 ! ;;
```

Character Output

`SPACE` outputs a blank space character.

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

`CHARS` outputs n characters c.

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

`SPACES` outputs 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 DUP C@ >CHAR EMIT 1+ THEN NEXT DROP ;;
```

`CR` outputs a carriage-return and a line-feed. Prior output characters are accumulated in a UDP packet buffer. This packet is sent out by sendPacket.

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

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

```
:: 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

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

```
:: [ ( -- )  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 espForth_44, 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 not leave. It uses QUERY to accept a line of text from the terminal and then let EVAL parse out the tokens and execute them. After a line is processed, it displays the top of data stack and wait for the next line of text. When an error occurred during execution, it displays the command which caused the error with an error message. After the error is reported, it re-initializes the system by jumping to ABORT. Because the behavior of EVAL can be changed by storing either $INTERPRET or $COMPILE into 'EVAL, QUIT exhibits the dual nature of a text interpreter and a compiler.

```
:: 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 +! ;;
:: $," ( -- ) ( 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 ;;
```

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

```
:: $,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 paren)` 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 paren)` is now invoked to turn the text interpreter into a compiler, which will compile the following words in the input stream to a token list in the code dictionary. The new compound command is terminated by ;, which compiles an EXIT to terminate the token list, and executes `[ (left paren)` 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 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` 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` finds the name field address of a token from its code field address. If the token does not exist in the dictionary, it returns a false flag. `>NAME` is the mirror image of the 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 acertain 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` 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 a high level word executed upon power-up. It sends out sign-on message, and then falls into the text interpreter loop through `QUIT`.

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

Control Structures

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

`AFT` 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--Aa) 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-quot) 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 creates a new constant, initialized to the value on top of stack.
```
:: CONSTANT CODE $2004  LIT , , ;;
```

Immediate Commands

.((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 @ +! ;;
```

Chapter 7. 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 `loop()` routine:

```
void loop() {
    phase = clk & 7;
    switch(phase) {
      case 0: fetch_decode(); yield(); 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();
    }
    clk += 1;
}
```

In espForth, 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:

```
void loop() {
  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 espForth 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 accommodates both design equally well. No modification in ceForth dictionary is necessary.

Stacks

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, the data stack and the return stacks are open to you, and most of the times, the data stack becomes the focus of your attention. Both stacks have to work perfectly. There is no margin of error.

With stacks implemented in memory of finite size, 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 sequence of Forth words, it always check the stack pointer. If the stack point 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 seems to be happy. I checked often the 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 need not be reset. The system would restart with the present pointers.

In espForth_44, 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 needs to be reminder constantly that the stack pointers have 8-bit values and must not be leaked to integer or long number. R and S pointers must always be prefixed with `(unsigned char)` specification. I struggled with data stack underflow conditions for half a year, until I found that the stack pointers tended to overshot the byte boundary in my back.

espForth 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, and relieve 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 change CP to point to another memory location, like the target dictionary array we allocated for a target system, then we can 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 tart 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 observed without great efforts.

After the metacompiler finishes building the target dictionary, it is useless for any other purposes. Close F# and use the rom_43.h file to build espForth on Arduino IDE.

Byte Code

I use byte code to bridge the Virtual Forth Machine and the eForth system. The dictionary is stored is 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 could allow. In espForth, 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 that 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 espForth design.

Socket Programming

As far as WiFi is concerned, I started at ground zero. I had no idea what these terms meant: access point, client side programming, server side programming, etc. I heard of TCP/IP, HTTP, HTTPS, but really did not know what they were good for.

Playing with NodeMCU, I saw this example on Arduino IDE, WebLED.ino, to turn its LED on and off. It was on almost every website tutorial talking about ESP8266:

```
#include <ESP8266WiFi.h>
const char* ssid = "SVFIG";
const char* password = "12345678";
int ledPin = 2; // GPIO2 of ESP8266
WiFiServer server(80);
void setup() {
  Serial.begin(115200);
  pinMode(ledPin, OUTPUT);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) { delay(500); Serial.print("."); }
  server.begin();
}
void loop() {
  WiFiClient client = server.available();
  if (!client) { return; }
  while(!client.available()){ delay(1); }
  String request = client.readStringUntil('\r');
  client.flush();
  int value = HIGH;
  if (request.indexOf("/LED=ON") != -1) {
    digitalWrite(ledPin, LOW); value = LOW; }
  if (request.indexOf("/LED=OFF") != -1){
    digitalWrite(ledPin, HIGH); value = HIGH; }
  client.println("HTTP/1.1 200 OK");
  client.println("Content-Type: text/html");
  client.println(""); //  do not forget this one
  client.println("<!DOCTYPE HTML>");
  client.println("<html>");
  client.print("Led pin is now: ");
  if(value == HIGH) { client.print("On"); }
  else { client.print("Off"); }
  client.println("<br><br>");
  client.println("Click <a href=\"/LED=ON\">here</a> turn the LED on pin 2
ON<br>");
```

```
  client.println("Click <a href=\"/LED=OFF\">here</a> turn the LED on pin 2 
OFF<br>");
  client.println("</html>");
  delay(1);
}
```

It created a Web client by opening a http webpage with two buttons LED=ON, and LED=OFF. When you click one of the buttons, the client sent back a messages with either 'LED=ON' or 'LED=OFF' string embedded. The server examined the message and turned the LED on or off accordingly.

I was fairly confused by this example. Does my server have to manage a client to communicate with myself? I like to send an arbitrary message or command to my server and order it to do something I want to do. I didn't even know what server and client were. But, I knew what I wanted. I liked to have a WiFi network to replace the serial cable to send commands to my computer, and receive responses from it.

I googled WiFi, and checked out all the WiFi books for dummies, and kept myself confused for some months. Then I saw Auduino had another example wifiSoftAP.ino and tried it. It showed that you could turned NodeMCU Kit into a Soft Access Point. It meant that you could build a local network with NodeMCU. That was interesting.

Amid random searches on Google, I hit a pdf book on Socket Programming from IBM, of all the companies. Sockets made lots of sense, and much of the fog and clouds started to lift. After I learnt how to configure sockets for UDP protocol, all my problems were solved. UDP was all I needed. Never mind TCP.

UDP is all I need, because espForth receives commands in packets, and it sends out responses after commands are executed. If there were errors in transmission, espForth would let me know.

Everything worked out fine, until we went to Maker Faire. Things worked while we set up the benches and workstations. When the crowd moved in, many students just could not turn the LED on and off over WiFi, because network traffic was so intense, even we had our own local router. We had to give away kits, when students demonstrated that they had controlled the LED through serial cable.

Serial Monitor and UDP Packets

To communication with espForth over WiFi, I substantially modified the serial IO design in espForth. Original eForth Model assumed a serial IO system sending and receiving ASCII characters. However, WiFi communication generally assumes sending and receiving packets, a sequence of characters. To make espForth receiving packets, The IO commands are actually significantly simplified. Instead of relying on `KEY` and `?KEY` to receive characters, I use `ACCEPT` to receive packets, and all the input commands below `ACCEPT` are eliminated.

eForth provides a line editor so you can edit your input line by backing up and erasing mistyped characters. In WiFi, a client sends packets of characters, and the client always gives you the opportunity to edit the packet before you send it out.

If I could receive a packet directly into the Terminal Input Buffer, then `ACCEPT` would simple wait for the arrival of a packet and return with the number of characters received.

Where should I place the Terminal Input Buffer? It seems that the best place is the beginner of `data[]` array. Forth historical reasons, I leave 512 bytes empty at the beginning of the dictionary. Many microcontrollers use this space for reset and interrupt vectors. When `data[]` is used as a byte array, it is referenced as `cData[]`.

`ACCEPT` waits for the serial monitor or the UDP receiver to send a packet of characters. If either gets a packets, `ACCEPT` returns with a character count. Then the text interpreter scans the characters and interprets them.

`accept` (b u1 -- b u2) Accept u1 characters to b. u2 returned is the actual count of characters received.

```
void accep()
/* UDP accept */
{ while (Udp.parsePacket()==0 && Serial.available()==0) {};
  int len;
  while (!Udp.available()==0) {
    len = Udp.read(cData, top); }
  while (!Serial.available()==0) {
    len = Serial.readBytes(cData, top); }
  if (len > 0) {
    cData[len] = 0; }
  top = len;
  }
```

On the transmitter side, `EMIT` send a character to the serial terminal and the UDP transmitter. However, the UDP transmitter only adds the character to its output buffer. The whole output packet is only transmitted when espForth executes `CR` command. It does not make sense to ship each character out in a separate UDP packet. The transmitter involves the following commands:

`sendPacket`(--) Send an UDP packet out to WiFi network. It is executed only by `CR` command.

```
CODE sendPacket sendPacket, next,
void sendPacket(void)
{  Udp.endPacket();
   Udp.beginPacket(Udp.remoteIP(), Udp.remotePort()); }
```

`CR` outputs a carriage-return and a line-feed. Prior output characters are accumulated in a UDP packet buffer. This packet is sent out by `sendPacket`.

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