

esp32Forth

Chapter 1. Complete Forth Written in C

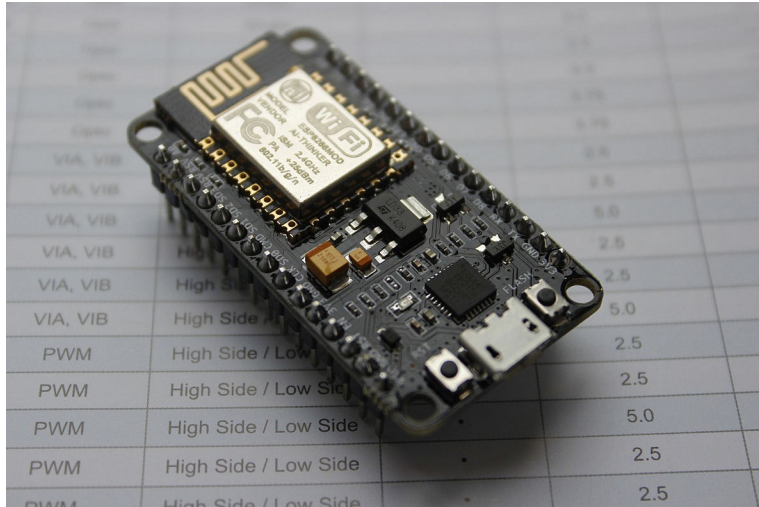
Affordable Microcontroller Kits

Microcontroller kits used to be expensive. It was difficult to find affordable kit to teach people how to program microcontrollers, even late in the 20th century. I used a number of 8051 kits which cost easily hundreds of dollars. Only in the 21st century, as flash memory and lots of IO are integrated into single chip microcontrollers, we saw complete microcontroller kits selling below \$50. Here is a list of such affordable microcontroller kits I played with:

2005	ADuC7020
2009	STM8 Discovery
2011	TI LaunchPad MSP430
2014	STM32 Discovery
2015	Arduino Uno ATmega328P
2017	NodeMCU ESP8266
2019	NodeMCU ESP32

The most noticeable one was LaunchPad from TI. It featured a MSP430 microcontroller, and TI promoted it at \$4.30. It was cheaper than the postage they shipped it out. I ordered a few, just to make sure that TI was not joking. I bought these kits and put eForth on them. They were nice platforms to teach people how to programming in Forth.

Then came esp8266. It is a very powerful chip on a stamp size kit, NodeMCU, selling for \$3.18 on eBay. It had 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.

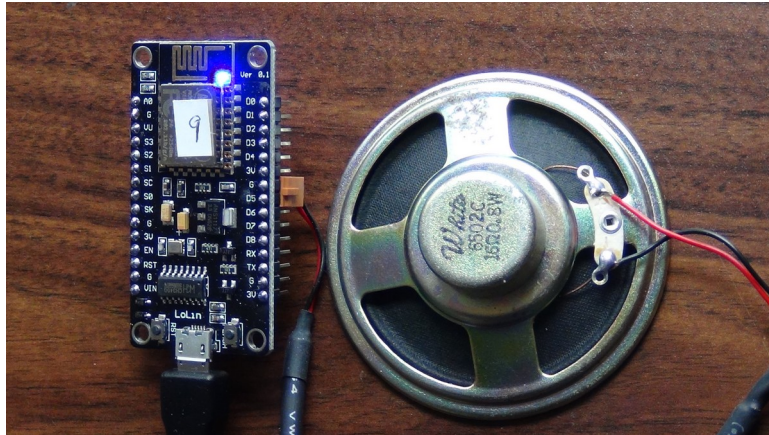


(Sorry to my Democrat friends about that Trump sticker on the car.)

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 esp32forth system we promoted in the Faire.



Although the official challenge was only to control the LED, we also attached a small speaker to an output pin. In one of our examples, the speaker 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 ATmega328P chip. However, it was only a teaser, because ATmega328P chip had only 1.5 KB of RAM left to add new Forth words, 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 was moved over to a new chip very easily. Only the hardware interface has to be changed. More than 98% of the code needs no modification.

I am very happy with this `esp32forth`. 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 Arduino 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 Arduino 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.

Since 2017, the next generation of ESP8266 became available in ESP32, and a NodeMCU ESP32S kit was selling at \$5.99 on eBay. ESP32 is twice more capable than ESP8266, with two Extensa LX160 CPU cores, and 520 KB of RAM. Don Golding in SVFIG decided to use it as the core of his new AIR robot, and we featured it as our theme in the 2019 Bay Area Maker Faire.



The Forth system for ESP32 was still the one on ceForth_328 for ATmega328P chip, and its dictionary was generated by a F# metacompiler. It is about time that we have a Forth completely compiled by C.

I used the F# metacompiler to construct the Forth dictionary, because I didn't know how to construct records with variable length byte fields or variable length integer fields in C. After much thoughts and many experiments, I found it is possible to do both.

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

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


To write consecutive integers, do the following:

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

Syncing $P=IP/4$, we can write variable length byte fields and variable length integer fields.

Therefore, I coded a macro assembler to build word record in the Forth dictionary. It consists of four macros: `HEADER()` to build name fields, `CODE()` to build code fields for primitive words, `COLON()` to build code/parameter fields for colon words, and `LABEL()` to code partial token lists in colon words. `LABEL()` annotates code segments in a long token list for `?BRANCH` and `BRANCH` to branch to proper target locations. `CODE()` and `COLON()` are macros returning code field address (cfa) of their respective primitive and colon words. These cfa are then assembled into token lists in colon words. `LABEL()` also returns an addresses for `?BRANCH`, `BRANCH` and `NEXT` to branch to.

`HEADER()` and `CODE()` are used to generate linked records for all primitive words. `HEADER()`, `COLON()`, and `LABEL()` are used to generate linked records for all colon words. The dictionary thus produced was compared byte-for-byte with the dictionary in `rom_54.h`, produced by `esp32forth_54`. Once all bytes matched, `esp32forth v6.1` ran identically to `esp32forth_54`.

Forward-referencing was a big problem for `LABEL()`. `NEXT` always branch back to terminate a loop, and is not a problem, but when `?BRANCH` and `BRANCH` branch forward, they do not have the correct addresses to assemble. You need a two-pass macro assembler. The first pass must resolve all the target addresses, and these correct addresses will be used in the second pass to assemble branch instructions.

To assembler branch instructions correctly, I first initialize all target addresses as integers with value 0. Then I made `LABEL()` to print the branch addresses as a symbol table to the Serial Monitor. This symbol table was used to initialize all target addresses proper. The next time Arduino IDE compiles `esp32forth` sketch, all branch instruction assemble correct target addresses. Essentially I was doing the second pass manually. This is `esp32forth v6.1`.

Manually resolving forward-references proved that the macro assembler worked, but the nuances would only be tolerated by the most faithful members of SVFIG. If this system were to released to the general public, forward-referencing had to be done automatically.

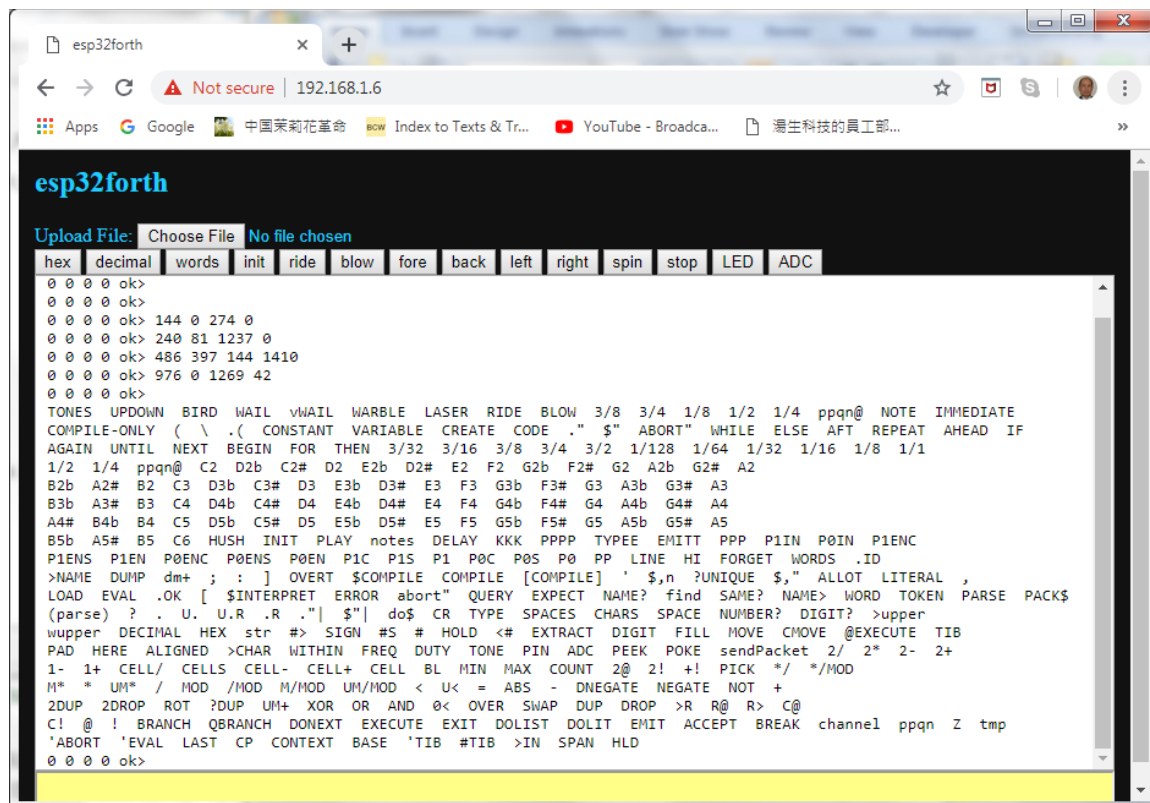
One of the most impressive features in Forth as invented by Chuck Moore is the single pass compilation of complicated nested control structures. There is no reason why the macro assembler cannot do the same in C. What I need are more macros to assemble control structures. All immediate words in Forth have to be emulated in this macro assembler. So I added these macros: `IF()`, `ELSE()`, `THEN()`, `FOR()`, `NEXT()`, `AFT()`, `BEGIN()`, `AGAIN()`, `UNTIL()`, `WHILE()`, and `REPEAT()`. These macros assemble proper branch instructions, and resolve all branch addresses automatically.

You need a stack to build nested control structures. Here we have already two stacks to run Forth words, though these are not used in compile time. I choose the return stack to assemble

control structures. You have to check the return stack after assembling. If you left something on the return stack, some control structures were not done right.

Again I compared the assembled dictionary, byte-for-byte, with the dictionary in rom_54.h. When rom_54.h is accurately reproduced, Forth booted up and ran. This is esp32forth v6.2.

For 2019 Maker Faire, we configured two Forth implementations: esp32forth v5.4 with Serial Monitor interface, and esp32forth v5.9 with a web browser interface, modified by Brad Nelson. Brad's web browser allows people to control NodeMCU ESP32S kit remotely through WiFi connection. ESP32 puts up a beautiful web page on the host PC:



Besides the input panel at the bottom and the output panel in the middle, there is a file download button on the top, and many word buttons to call out the most useful functions implemented on NodeMCU ESP32S kit. If the kit had motors installed, you can drive the robot car forward, back, left, and right. You can even play a few tunes if a speaker is installed.

The last, but not the least, important lesson I learnt in this project is that Forth could be more useful if it were written as a subroutine callable by other systems, rather than a standalone operating system. A standalone Forth system works like:

```
COLD - ->QUIT - ->BEGIN-QUERY-EVAL-AGAIN
```

The heart of Forth is EVAL, which interprets a list of word collected by QUERY. However, Forth can be embedded in another system which assembles a list of words, and then call EVAL to evaluate this list. When EVAL finishes interpreting, it returns control back to the caller.

Chapter 2. Running esp32Forth

Install Arduino IDE

esp32forth_62.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 esp32forth, you have to first install Arduino IDE with ESP32 extensions. Then you can copy esp32forth_62.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 gave you a very simple program template, which expected you to fill C code into two routines `setup()` and `loop()`. It captures the essences of firmware engineering and invited everybody to become a firmware engineer.

It is amazing that people in ESP8266/32 Community extended the Arduino IDE so that you can program this very sophisticated 32-bit ESP8266/32 chips with ease. It even supports our NodeMCU ESP32S Kit!

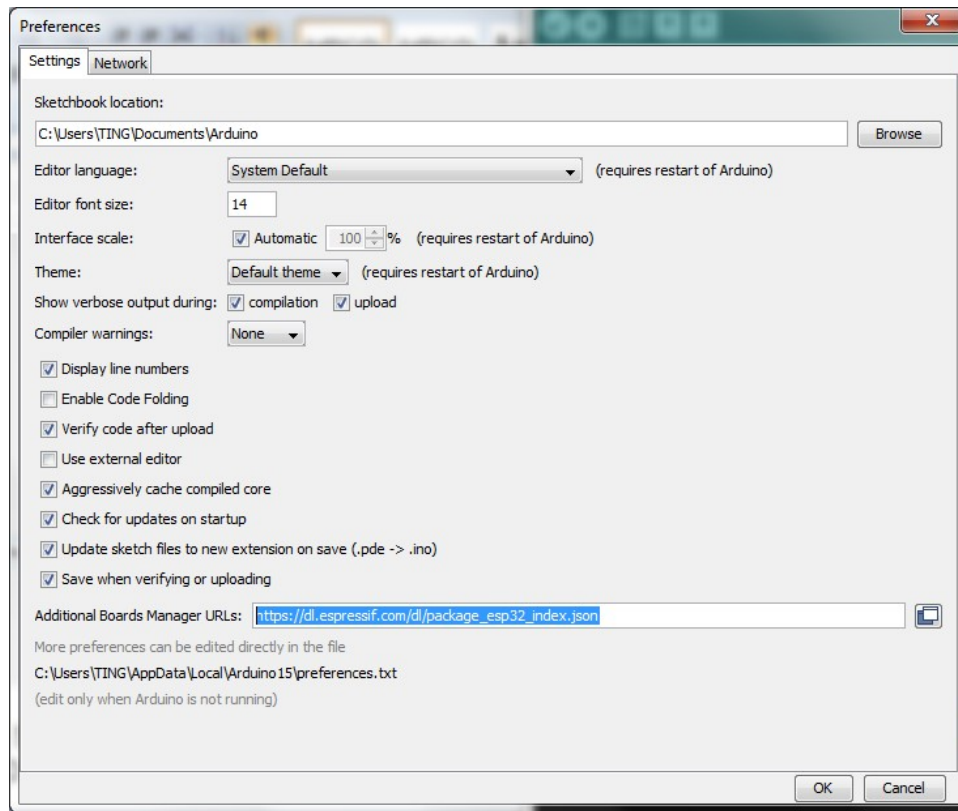
In the experiments with esp32forth, you will have to use Arduino IDE. If your computer does not have it, you have to install it first. Arduino IDE has to be extended so that it can compile programs for the ESP32 chip, and to upload the compiled code to flash memory on NodeMCU Kit. After Arduino IDE is set up properly, it will be very easy to do experiments with NodeMCU Kit.

Download Arduino 1.8.9 IDE or the latest version from www.arduino.cc and install it on your PC. Open Arduino, and you will see its title page:

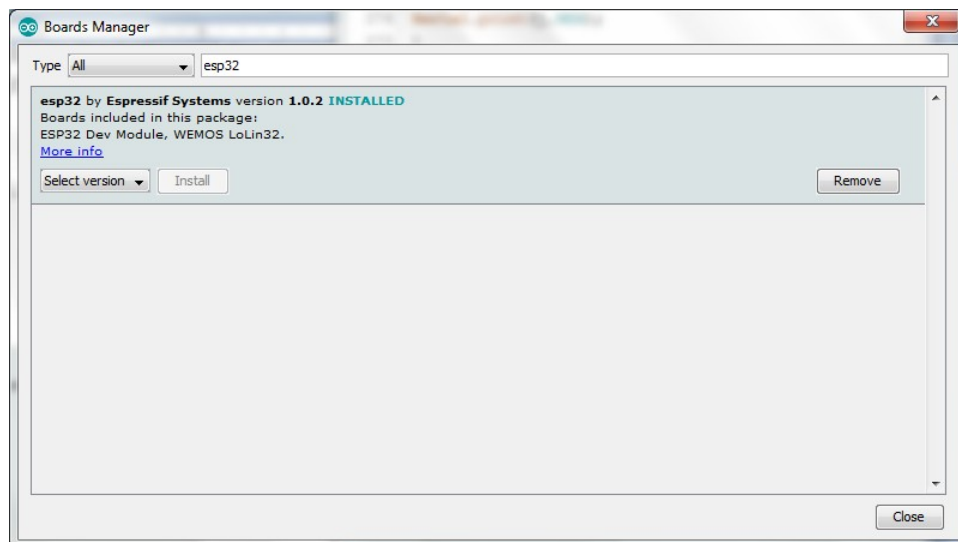


Click File>Preferences to open the Preferences window.

Enter https://dl.espressif.com/dl/package_esp32_index.json into Additional Board Manager URLs field:



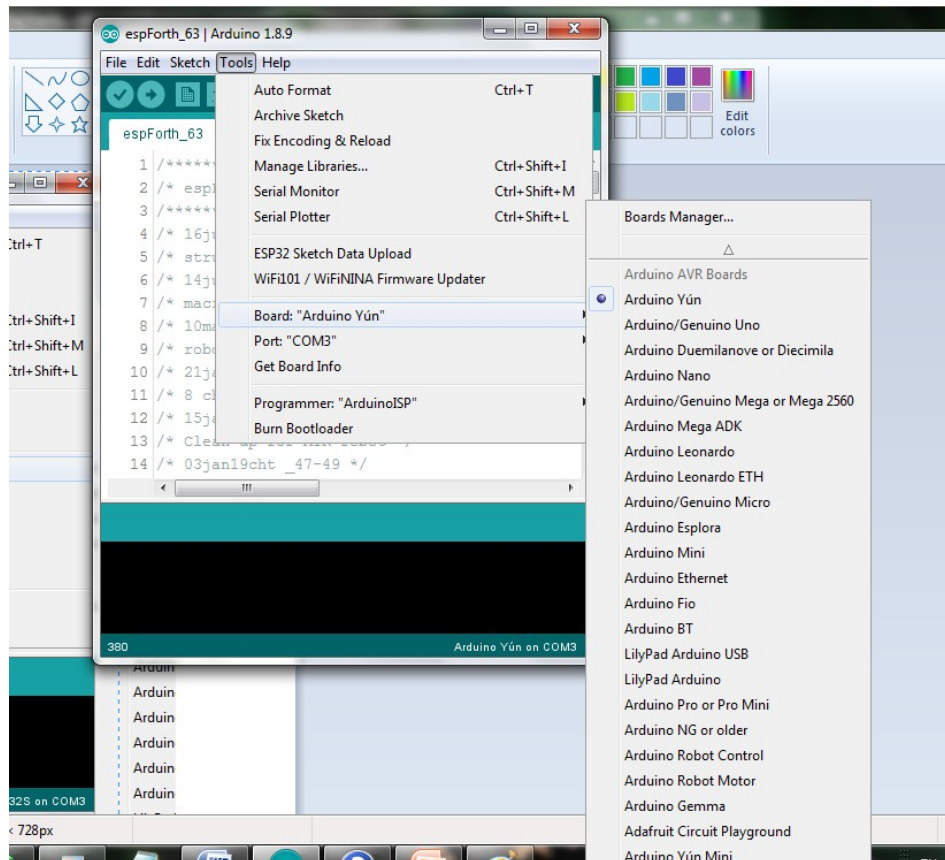
Next, Click Tools>Board:xxxxxxx>Boards Manager. Scroll to the bottom of the display, or type esp32 in the text input box, and click on the panel named esp32 by Espressif Systems to install it:



Click the Install button at bottom right to install the ESP32 package.

After the install process, you should see that ESP32 package is marked INSTALLED. Close the Boards Manager window once the install process has completed.

Select NodeMCU-32S from the Tools->Board dropdown menu:



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

Board: "NodeMCU-32S"
Upload Speed: "921600"
Flash frequency: "80 MHz"
Port: "COM3"

Arduino IDE is now set up properly. You can now proceed to do esp32forth experiments.

Loading esp32forth

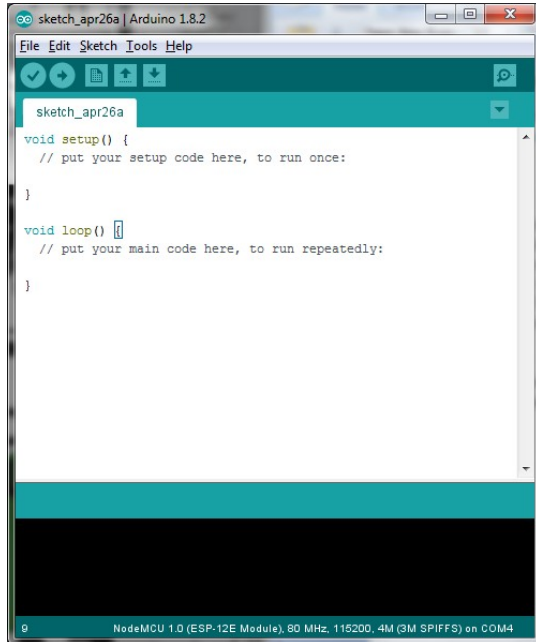
Forth is the simplest programming language, and has been widely used for industrial, scientific, military, space, and embedded applications. eForth is the simplest Forth implementation for microcontrollers. I ported it to ESP32 under Arduino IDE as esp32forth. Once esp32forth is loaded on NodeMCU, it allows you to explore this chip, and test its IO devices interactively.

esp32forth accepts input from Arduino Serial Monitor. You can turn its on-board LED on and off and issue other Forth commands.

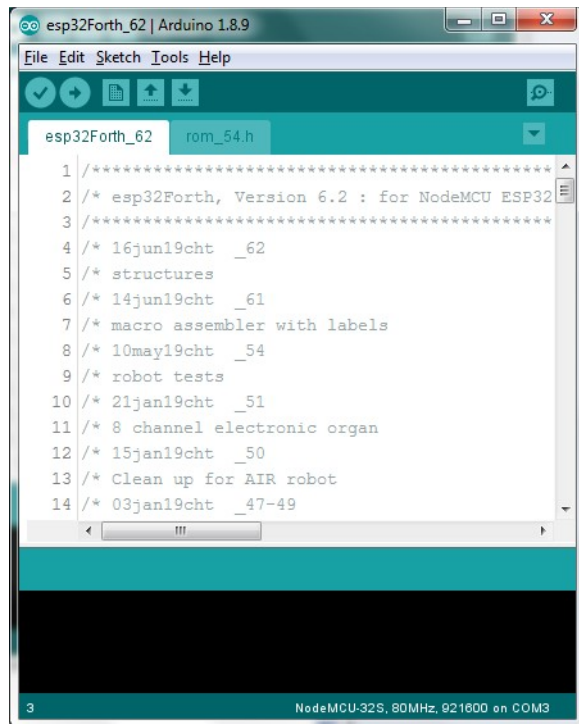
Connect your NodeMCU Kit to your PC, with the MicroUSB-Serial cable.

Unzip esp32forth_62.zip, and place it in a folder like C:/esp32forth_62.

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

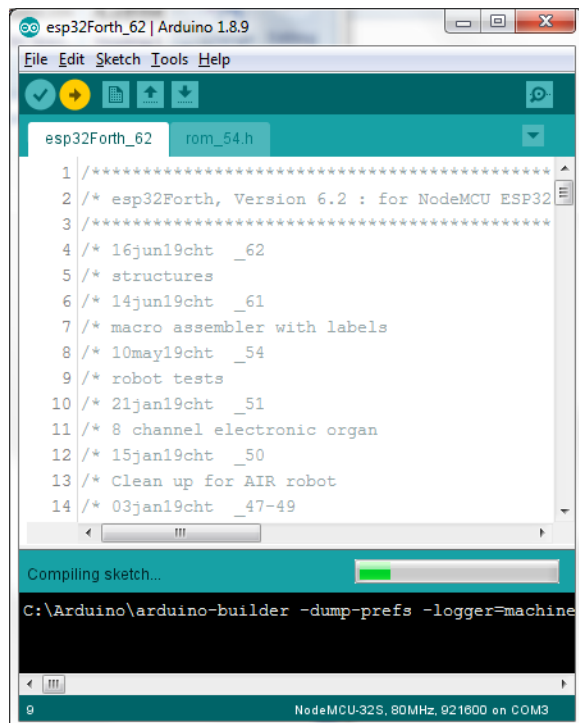


Click File>Open..., and select C: /esp32forth_62/esp32forth_62.ino, supplied in the esp32forth_62 project folder.

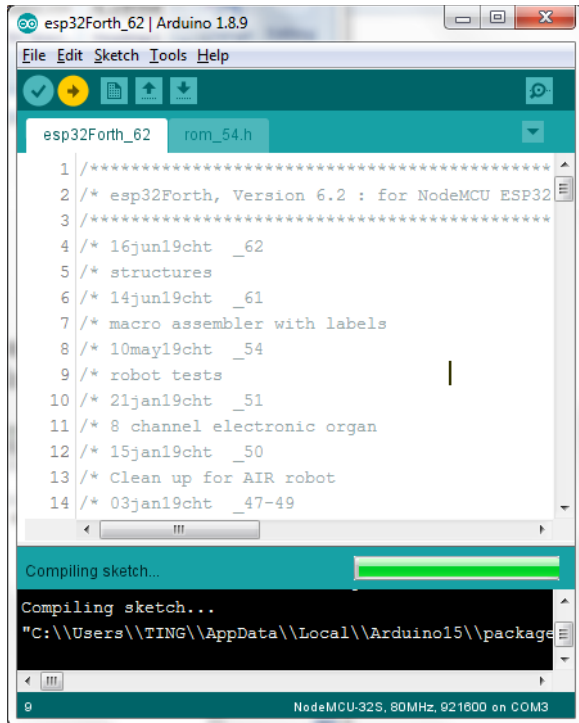


esp32forth is a file of 42 KB size, very small comparing to software of this age, but it is a complete interactive operating system with a high level Forth programming language. Here in this experiment,

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

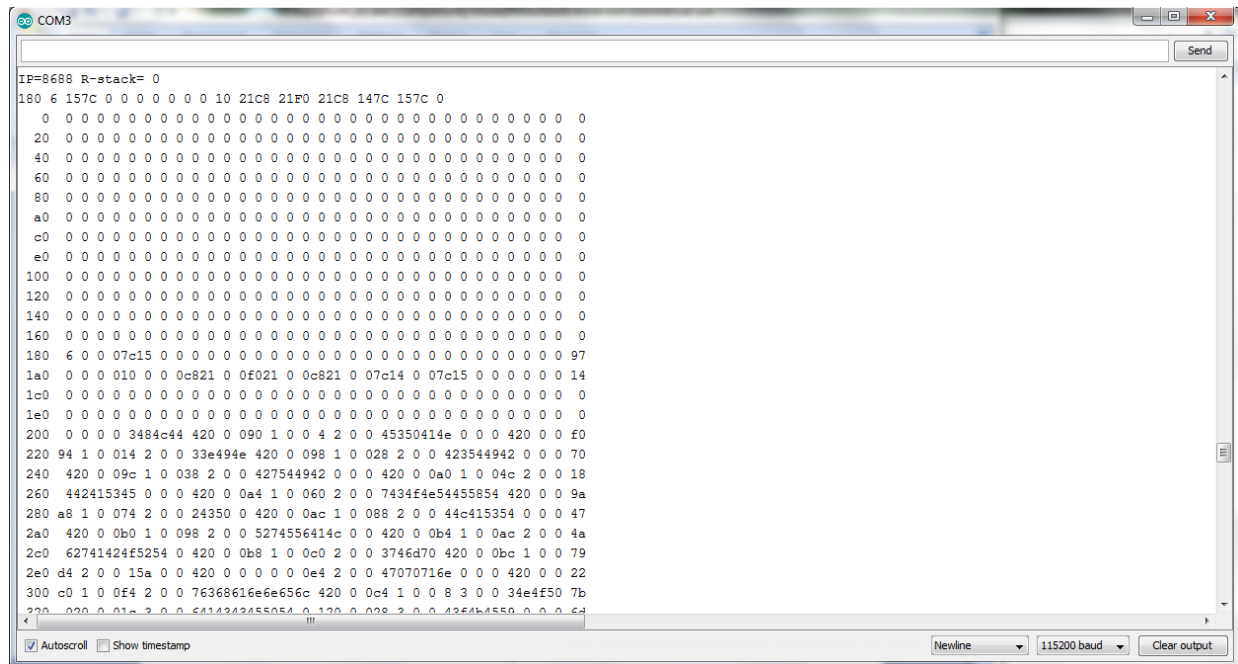


It takes a few minutes for Arduino IDE to compile the code and then upload the binary image into the flash memory on NodeMCU Kit. Eventually, it will report 'Done uploading', and reports to you what it accomplished:



Test esp32forth

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 esp32forth signing in. There are lots of text scrolling over the Serial Monitor. Scroll back to the beginning, you can see to boot-up information, followed by Forth primitive words as they were assembled by our very capable macro assembler:



“IP=8688” announces that the dictionary is now 8688 bytes in size. “R-stack=0” assures that the macro assembler stack is clean, and all the control structures were most likely assembled correctly.

Scroll to the bottom of Serial Monitor, and you see the end of Forth dictionary. At the very end, these messages:

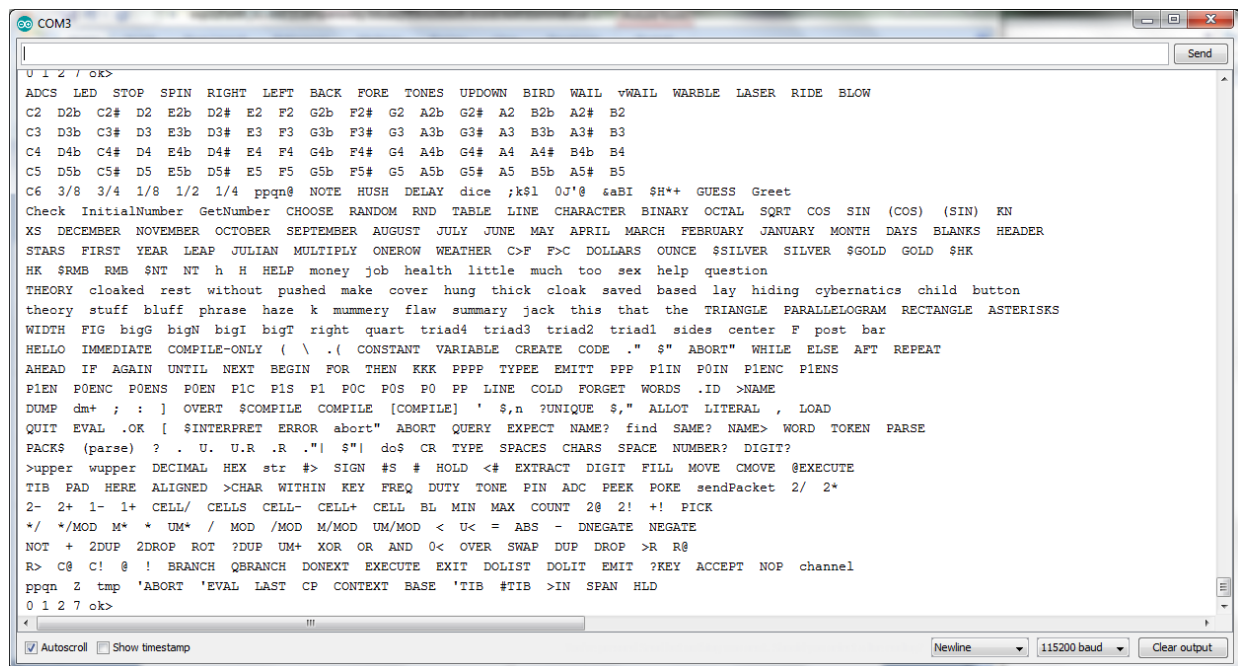
```
Load file: 26669 bytes.
theory reDef help reDef H reDef LINE reDef right reDef
0 0 0 0 ok> Done loading.
```

They announce that the application file load.txt, stored in flash memory, was loaded, and 26669 bytes are compiled by Forth. While compiling, Forth noticed that several words were re-defined. In load.txt, I gave you 20 lessons, and sometime I ran out of good names and picked names used before. Re-using names should be avoided, but mostly harmless. After all text in load.txt is compile, Forth happily declared “Done loading”!

Now you can type in the following Forth words in the text box on the top of Serial Monitor window, (and click the Send button to the right), to exercise esp32forth system. After entering one line, press the Send button to the right of the text box, or, Press the Enter key on your keyboard.

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

esp32forth is case insensitive. WORDS and words are the same. After WORDS is entered, Serial Monitor window looks like the following, showing all the Forth words implemented in esp32forth:



```
U 1 2 / ok>
ADCS LED STOP SPIN RIGHT LEFT BACK FORE TONES UPDOWN BIRD WAIL vWAIL WARBLE LASER RIDE BLOW
C2 D2b C2# D2 E2b D2# E2 F2 G2b F2# G2 A2b G2# A2 B2b A2# B2
C3 D3b C3# D3 E3b D3# E3 F3 G3b F3# G3 A3b G3# A3 B3b A3# B3
C4 D4b C4# D4 E4b D4# E4 F4 G4b F4# G4 A4b G4# A4 B4b A4# B4
C5 D5b C5# D5 E5b D5# E5 F5 G5b F5# G5 A5b G5# A5 B5b A5# B5
C6 3/8 3/4 1/8 1/2 1/4 ppqn@ NOTE HUSH DELAY dice ;k$1 0J'@ &aBI $H*+ GUESS Greet
Check InitialNumber GetNumber CHOOSE RANDOM RND TABLE LINE CHARACTER BINARY OCTAL SQRT COS SIN (COS) (SIN) KN
XS DECEMBER NOVEMBER OCTOBER SEPTEMBER AUGUST JULY JUNE MAY APRIL MARCH FEBRUARY JANUARY MONTH DAYS BLANKS HEADER
STARS FIRST YEAR LEAP JULIAN MULTIPLY ONEROW WEATHER C>F F>C DOLLARS OUNCE $SILVER SILVER $GOLD GOLD $HK
HK $RMB RMB $NT NT h H HELP money job health little much too sex help question
THEORY cloaked rest without pushed make cover hung thick cloak saved based lay hiding cybernatics child button
theory stuff bluff phrase haze k mummyr flaw summary jack this that the TRIANGLE PARALLELOGRAM RECTANGLE ASTERISKS
WIDTH FIG bigG bigN bigI bigT right quart triad4 triad3 triad2 triad1 sides center F post bar
HELLO IMMEDIATE COMPILE-ONLY ( \ \.( CONSTANT VARIABLE CREATE CODE ." $" ABORT" WHILE ELSE APT REPEAT
AHEAD IF AGAIN UNTIL NEXT BEGIN FOR THEN KKK PPPP TYPEE EMITT PPP PIIN POIN PIENC PIENS
PIEN POENC POENS POEN PIC PIS Pl POC POS P0 PP LINE COLD FORGET WORDS .ID >NAME
DUMP dm+ ; : ] OVERT $COMPILE COMPILE [COMPILE] ' $,n ?UNIQUE $," ALLOT LITERAL , LOAD
QUIT EVAL .OK [ $INTERPRET ERROR abort" ABORT QUERY EXPECT NAME? find SAME? NAME> WORD TOKEN PARSE
PACKS (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 KEY FREQ DUTY TONE PIN ADC PEEK POKE sendPacket 2/ 2*
2- 2+ 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 >R R@
R> C@ C! @ ! BRANCH QBRANCH DONEXT EXECUTE EXIT DOLIST DOLIT EMIT ?KEY ACCEPT NOP channel
ppqn 2 tmp 'ABORT 'EVAL LAST CP CONTEXT BASE 'TIB #TIB >IN SPAN HLD
0 1 2 7 ok>
```

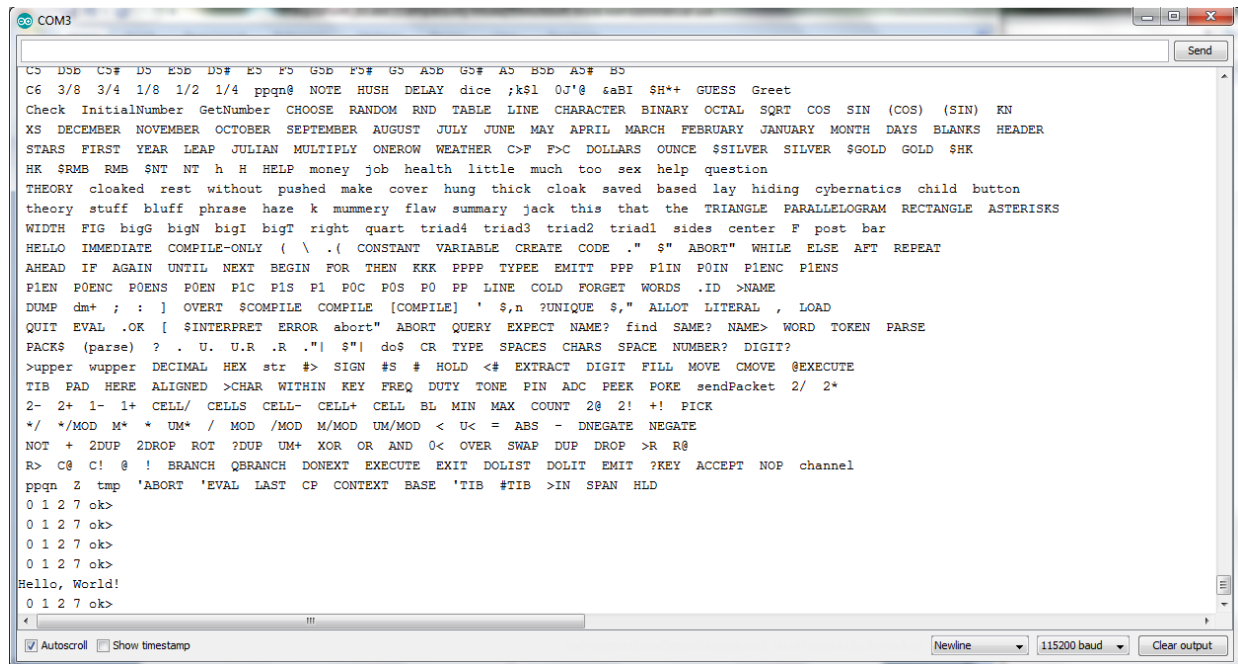
Now, try to turn the on-board LED off and on with these words:

STOP
LED

Before you go, do this universal programming experiment:

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

And you will see something like the following:



```
COM3
C5 D3B C5# D5 E5B D5# E5 F5 G5B F5# G5 A5B G5# A5 B5B A5# B5
C6 3/8 3/4 1/8 1/2 1/4 ppqn# NOTE HUSH DELAY dice ;k$1 0J'@ &aBI $H*+ GUESS Greet
Check InitialNumber GetNumber CHOOSE RANDOM RND TABLE LINE CHARACTER BINARY OCTAL SQRT COS SIN (COS) (SIN) KN
XS DECEMBER NOVEMBER OCTOBER SEPTEMBER AUGUST JULY JUNE MAY APRIL MARCH FEBRUARY JANUARY MONTH DAYS BLANKS HEADER
STARS FIRST YEAR LEAP JULIAN MULTIPLY ONEROW WEATHER C>F F>C DOLLARS OUNCE $SILVER SILVER $GOLD GOLD $HK
HK $RMB RMB $NT NT h H HELP money job health little much too sex help question
THEORY cloaked rest without pushed make cover hung thick cloak saved based lay hiding cybernatics child button
theory stuff bluff phrase haze k mummyr flaw summary jack this that the TRIANGLE PARALLELOGRAM RECTANGLE ASTERISKS
WIDTH FIG bigG bigN bigI bigT right quart triad4 triad3 triad2 triad1 sides center F post bar
HELLO IMMEDIATE COMPILE-ONLY ( \ .( CONSTANT VARIABLE CREATE CODE ." $" ABORT" WHILE ELSE APT REPEAT
AHEAD IF AGAIN UNTIL NEXT BEGIN FOR THEN KKK PPPP TYPEE EMITT PPP P1IN P0IN PIENC PIENS
PIEN POENC POENS POEN PIC P1S P1 P0C P0S P0 PP LINE COLD FORGET WORDS .ID >NAME
DUMP dm+ ; : ] OVERT $COMPILE COMPILE [COMPILE] ' $,n ?UNIQUE $," ALLOT LITERAL , LOAD
QUIT EVAL .OK [ $INTERPRET ERROR abort" ABORT QUERY EXPECT NAME? find SAME? NAME? WORD TOKEN PARSE
PACK$ (parse) ? . U. U.R .R ."| S"| 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 KEY FREQ DUTY TONE PIN ADC PEEK POKE sendPacket 2/ 2*
2- 2+ 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 >R R@
R> C@ C! @ ! BRANCH QBRANCH DONEXT EXECUTE EXIT DOLIST DOLIT EMIT ?KEY ACCEPT NOP channel
ppqn Z tmp 'ABORT 'EVAL LAST CP CONTEXT BASE 'TIB #TIB >IN SPAN HLD
0 1 2 7 ok>
0 1 2 7 ok>
0 1 2 7 ok>
0 1 2 7 ok>
0 1 2 7 ok>
Hello, World!
0 1 2 7 ok>
```

esp32forth is up and running.

esp32forth Experiments

WORDS is a word which dumps the dictionary of Forth. It shows the names of all Forth words assembled in the system. There are 185 of them. I assembled all the commonly useful Forth words in upper case, and all the infrequently used system words in lower case. esp32forth_62 system is case insensitive. You can type in words in either upper or lower case.

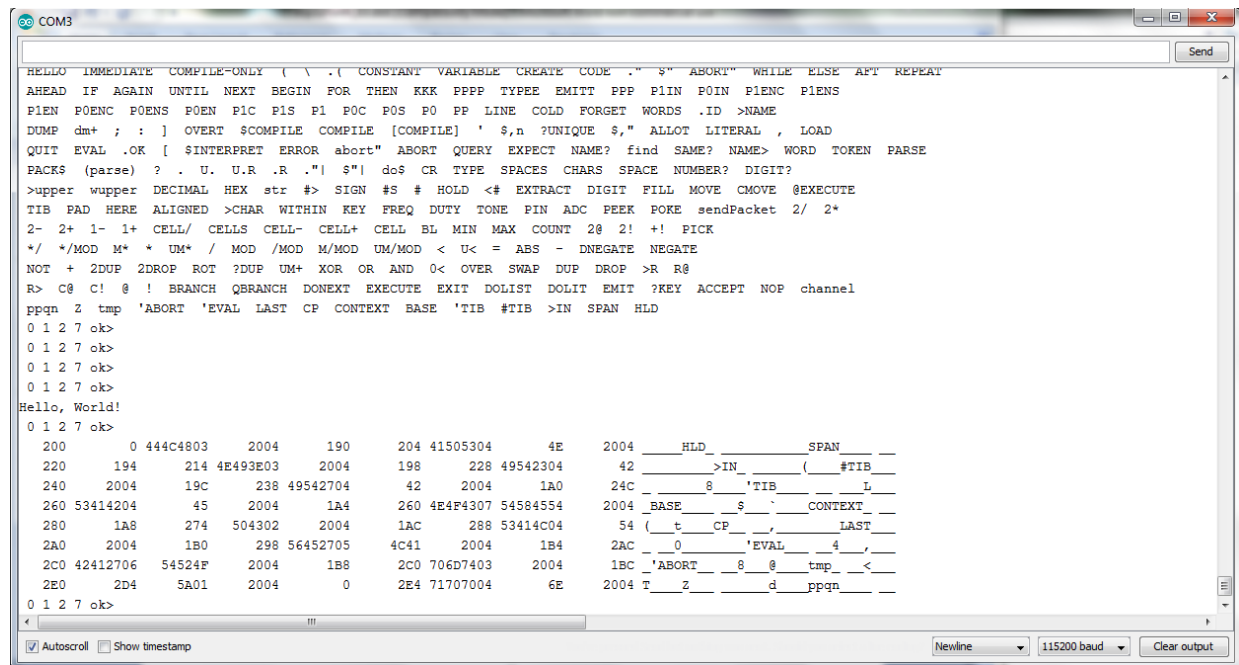
Words in dictionary are linked backward. Therefore, the last word defined appears first, and the first word HLD appears the last, as shown in the Serial Monitor window above. The word list as shown before are much longer than 185 words, because I compiled 20 lessons from load.txt file. In the Forth system itself, the last word is IMMEDIATE. Words above it are application words.

In the original 86eForth v1.0, there were 31 primitive words and 183 high level compound words. In esp32forth_62, there are 83 primitive words, derived from 71 byte code, and 102 compound words. However, you cannot easily distinguish primitive words from compound words, except reading the assembly source code or examining object code in memory. Actually, a primitive word behaves identically as if written as compound word, except it is faster and often shorter.

Dump Memory

A very useful tool word DUMP allows you to examine different areas of memory. For example, the Forth dictionary starts at memory location 0x200. Type:
Hex 200 100 dump

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



```
HELLO IMMEDIATE COMPILE-ONLY ( \ .( CONSTANT VARIABLE CREATE CODE ." $" ABORT" WHILE ELSE APT REPEAT
AHEAD IF AGAIN UNTIL NEXT BEGIN FOR THEN KKK PPPP TYPEE EMITT PPP PLIN POIN PIENC PIENS
PIEN POENC POENS POEN PIC PIS P1 POC POS P0 PP LINE COLD FORGET WORDS .ID >NAME
DUMP dm+ ; : ] OVERT $COMPILE COMPILE [COMPILE] ' $,n ?UNIQUE $," ALLOT LITERAL , LOAD
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 KEY FREQ DUTY TONE PIN ADC PEEK POKE sendPacket 2/ 2*
2- 2+ 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 >R R@
R> C@ C! @ ! BRANCH QBRANCH DONEXT EXECUTE EXIT DOLIST DOLIT EMIT ?KEY ACCEPT NOP channel
ppqn z tmp 'ABORT 'EVAL LAST CP CONTEXT BASE 'TIB #TIB >IN SPAN HLD
0 1 2 7 ok>
0 1 2 7 ok>
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 5A01 2004 0 2E4 71707004 6E 2004 T z d ppqn
```

The memory is displayed in hexadecimal, and contents do not make much sense. However, you might notice that the names of words 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 words.

PEEK and POKE

esp32forth system is based on a Forth Virtual Machine written in C. The dictionary is in a data array allocated somewhere in the memory of ESP32. The address we discussed above are not real addresses. They are virtual addresses relative to the beginning of the data array. The C compiler hides ESP32 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 \$5.99 loss.

PWM Tone Generater

I almost forget the speaker I mentioned in Preface. I attached a small speaker between GPIO5 and GND pins. GPIO5 pin was initialized to output PWM waves from Channel 0 of the timers in ESP32. You can send a square wave pulses out on GPIO5 pin with the following words:

```
DECIMAL 440 0 TONE
```

TONE thus send out a 440 Hz square wave through GPIO5. You can play with these words with different frequencies and duration. You can perhaps play a tune or two for fun. In fact, there are two tunes already loaded from load.txt:

```
BIRD  
RIDE
```

They will be used in AIR robot Ron Golding is building.

Simple Test Routines

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

```
: 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 1000000 FOR NEXT ;
```

In a later chapter I will show you the lessons in load.txt.

Chapter 3. esp32Forth Sketch

esp32forth_62.ino

The file esp32forth_62.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 ESP32S, with an ESP32 chip. This file serves perfectly as a specification of a Forth Virtual Machine (FVM), in terms of a C language functions.

Before diving directly into esp32forth, I would like to give you an overview of this Forth Virtual Machine so you can better understand how it is implemented.

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

- Forth Virtual Machine executes a set words.
- All words are stored in a large data array, called a dictionary.
- Each word is a record in a searchable dictionary. All word records are linked in the dictionary.
- Each word record contains 4 fields, a link field, a name field, a code field, and a parameter field. The link field and name field allow the dictionary to be searched for a word from its ASCII name. The code field contains executable byte code. The parameter field contains optional data needed by the word.
- There are two types of words: primitive words containing executable byte code, and compound word containing token lists. A token is a pointer pointing to code fields of other words.
- A finite state machine executes byte code sequences stored in code field of words.
- An address interpreter executes nested lists in compound words.
- A return stack is used to process nested token lists
- A data stack is used to pass parameters among words
- A serial terminal is necessary to enter word lists, and to show results thus produced.
- A alternate WiFi network is necessary to send and receive packets to/from this FVM.

The simple goal of a FVM is to impose a Forth language processor on a real computer so that it can interpret or execute a list of Forth words, separated by spaces:

```
<list of words>
```

and to create a new word to replace a list of words:

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

All computable problems can be solved by repeatedly creating new words to replace lists of existing words. 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 esp32forth_64.ino, to see how this FVM is actually implemented.

Preamble

```
#include "SPIFFS.h"

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

Arduino supplied libraries supporting ESP32 chip and its capabilities. SPIFFS.h supplies information on the flash memory on ESP32 chip, and a simple file system SPIFFS.

Default values of a FALSE flag is 0, and of TRUE is -1. esp32forth considers any non-zero number as a TRUE flag. Nevertheless, all esp32forth words which generate flags would return a -1 for TRUE.

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

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

pop is a macro 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.

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.

A return stack is required for processing nested lists in runtime. In addition, this return stack is drafted to assemble nested control structure in compile time. pushR and popR streamline the operations on the return stack.

Registers and Arrays

esp32forth uses a large data array to hold its dictionary of words. Besides this data array, the FVM needs many registers and arrays to hold data to support all its operations. There are lots of variables and buffers declared. Here is the list of these registers and arrays:

```
long rack[256] = {0};
long stack[256] = {0};
unsigned char R=0, S=0, bytecode, c ;
long* Pointer ;
long P, IP, WP, top, thread, len;
```

```
uint8_t* cData ;
long long int d, n, m ;
int BRAN=0, QBRAN=0, DONXT=0, DOTQP=0, STRQP=0, TOR=0, ABRQP=0;
```

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
c	A 8 bit scratch register
bytecode	A 8 bit register containing byte code to be executed
cData	A byte pointer to access data[] array in bytes
BRAN	Code field address of BRANCH
QBRAN	Code field address of ?BRANCH
DONXT	Code field address of NEXT
DOTQP	Code field address of .”
STRQP	Code field address of \$”
ABRQP	Code field address of abort”
TOR	Code field address of >R

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

How did I stop worrying the 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 chunks 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 word, and reinitialized the stacks.

In esp32forth_62, I allocated 1KB memory for each stack, `stack[256]` and `rack[256]`, and used a byte pointer for each stack, S and R. The stacks are 256 cell circular buffers, and will never underflow or overflow. However, the C compiler needs to be reminder constantly that the stack pointers have 8-bit values and must not be leaked to integer. R and S pointers must always be prefixed with `(unsigned char)` specification. Thus the stacks will never overshoot their boundaries.

esp32forth interpreter 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 stack problems.

With circular buffers for stacks, I saved 4 byte code, about 10 stack managing words, and a ton of worrying.

Dictionary Array

All esp32forth words are stored in a `data[16000]` array, as a linked list, and is generally called a dictionary. Each record in this list contains 4 fields: a 32 bit link field, a variable length name field, a 32 bit code field, and a variable length parameter field. In a primitive word, the parameter field has additional byte code. In high level colon words, the code field has one byte code 6, for nesting the token list in the subsequent parameter field. Tokens are code field addresses, pointing to other Forth words.

The dictionary in `data[]` array is filled in by a macro assembler. It has a byte array alias `cData[]`, because the macro assembler has to fill in both variable length byte fields and variable length integer fields.

```
// dictionary
long data[16000] = {};
int IMEDD=0x80;
int COMPO=0x40;
```

Using byte array pointer IP, consecutive byte data are entered by:

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

Using integer array pointer P, consecutive integer data are entered by:

```
    data[P++]= int n;
```

IMEDD and COMPO are lexicon bits in the length byte in a name field. IMEDD indicates that this word must be executed while compiling. COMPO indicates that this word can only be invoked while compiling.

Macro Assembler

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

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

To write consecutive integers, do the following:

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

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

For clarity and efficiency, I coded a macro assembler to build all word record in the Forth dictionary. It consists of four macros: `HEADER()` to build name fields, `CODE()` to build code fields for primitive words, `COLON()` to build code/parameter fields for colon words, and `LABEL()` to add partial token lists to colon words for branching and looping. These four macros are as follows:

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

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

```
void HEADER(int lex, char seq[]) {
    P=IP>>2;
    int i;
    int len=lex&31;
    data[P++]=thread;
    IP=P<<2;
    Serial.println();
    Serial.print(thread, HEX);
    for (i=thread>>2; i<P; i++)
        {Serial.print(" "); Serial.print(data[i], HEX);}
    thread=IP;
    cData[IP++]=lex;
    for (i=0; i<len; i++)
        {cData[IP++]=seq[i];}
    while (IP&3) {cData[IP++]=0;}
    Serial.println();
    Serial.print(seq);
    Serial.print(" ");
    Serial.print(IP, HEX);
}
```

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

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

```
int CODE(int len, ... ) {
    int addr=IP;
    int s;
    va_list argList;
    va_start(argList, len);
    for(; len;len--) {
        s= va_arg(argList, int);
        cData[IP++]=s;
        Serial.print(" ");
        Serial.print(s,HEX);
    }
    va_end(argList);
    return addr;
}
```

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

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

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

```

}
IP=P<<2;
va_end(argList);
return addr;
}

```

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

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

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

```

int LABEL(int len, ... ) {
    int addr=IP;
    P=IP>>2;
    va_list argList;
    va_start(argList, len);
    Serial.println();
    Serial.print(addr,HEX);
    for(; len;len--) {
        int j=va_arg(argList, int);
        data[P++]=j;
        Serial.print(" ");
        Serial.print(j,HEX);
    }
    IP=P<<2;
    va_end(argList);
    return addr;
}

```

Macros for Structured Programming

I could handle labels to demonstrate I could build Forth dictionary in C, but I would not expect users to manually resolve forward references. Chuck Moore demonstrated that all control structures could be compiled correctly in a single pass. It is certainly possible to do it in this macro assembler. The way to do it is to write all the Forth immediate words in macros to set up branch and loop structures and resolve all references, forward or backward.

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

```

BEGIN...AGAIN
BEGIN...UNTIL
BEGIN...WHILE...REPEAT
IF...ELSE...THEN

```


FOR...AFT...THEN...NEXT

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

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

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

```
// structure assembler
```

```
void BEGIN(int len, ... ) {
    P=IP>>2;
    Serial.println();
    Serial.print(IP,HEX);
    Serial.print(" BEGIN ");
    pushR=P;
    va_list argList;
    va_start(argList, len);
    for(; len;len--) {
        int j=va_arg(argList, int);
        data[P++]=j;
        Serial.print(" ");
        Serial.print(j,HEX);
    }
    IP=P<<2;
    va_end(argList);
}
```

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

```
void AGAIN(int len, ... ) {
    P=IP>>2;
    Serial.println();
    Serial.print(IP,HEX);
    Serial.print(" AGAIN ");
    data[P++]=BRAN;
    data[P++]=popR<<2;
    va_list argList;
    va_start(argList, len);
    for(; len;len--) {
        int j=va_arg(argList, int);
```

```

        data[P++]=j;
        Serial.print(" ");
        Serial.print(j, HEX);
    }
    IP=P<<2;
    va_end(argList);
}

```

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

```

void UNTIL(int len, ... ) {
    P=IP>>2;
    Serial.println();
    Serial.print(IP, HEX);
    Serial.print(" UNTIL ");
    data[P++]=QBRAN;
    data[P++]=popR<<2;
    va_list argList;
    va_start(argList, len);
    for(; len; len--) {
        int j=va_arg(argList, int);
        data[P++]=j;
        Serial.print(" ");
        Serial.print(j, HEX);
    }
    IP=P<<2;
    va_end(argList);
}

```

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

```

void WHILE(int len, ... ) {
    P=IP>>2;
    int k;
    Serial.println();
    Serial.print(IP, HEX);
    Serial.print(" WHILE ");
    data[P++]=QBRAN;
    data[P++]=0;
    k=popR;
    pushR=(P-1);
    pushR=k;
    va_list argList;
    va_start(argList, len);
    for(; len; len--) {

```

```

    int j=va_arg(argList, int);
    data[P++]=j;
    Serial.print(" ");
    Serial.print(j, HEX);
}
IP=P<<2;
va_end(argList);
}

```

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

```

void REPEAT(int len, ... ) {
    P=IP>>2;
    Serial.println();
    Serial.print(IP, HEX);
    Serial.print(" REPEAT ");
    data[P++]=BRAN;
    data[P++]=popR<<2;
    data[popR]=P<<2;
    va_list argList;
    va_start(argList, len);
    for(; len; len--) {
        int j=va_arg(argList, int);
        data[P++]=j;
        Serial.print(" ");
        Serial.print(j, HEX);
    }
    IP=P<<2;
    va_end(argList);
}

```

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

```

void IF(int len, ... ) {
    P=IP>>2;
    Serial.println();
    Serial.print(IP, HEX);
    Serial.print(" IF ");
    data[P++]=QBRAN;
    pushR=P;
    data[P++]=0;
    va_list argList;
    va_start(argList, len);
    for(; len; len--) {
        int j=va_arg(argList, int);
    }
}

```

```

    data[P++]=j;
    Serial.print(" ");
    Serial.print(j,HEX);
}
IP=P<<2;
va_end(argList);
}

```

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

```

void ELSE(int len, ... ) {
    P=IP>>2;
    Serial.println();
    Serial.print(IP,HEX);
    Serial.print(" ELSE ");
    data[P++]=BRAN;
    data[P++]=0;
    data[popR]=P<<2;
    pushR=P-1;
    va_list argList;
    va_start(argList, len);
    for(; len;len--) {
        int j=va_arg(argList, int);
        data[P++]=j;
        Serial.print(" ");
        Serial.print(j,HEX);
    }
    IP=P<<2;
    va_end(argList);
}

```

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

```

void THEN(int len, ... ) {
    P=IP>>2;
    Serial.println();
    Serial.print(IP,HEX);
    Serial.print(" THEN ");
    data[popR]=P<<2;
    va_list argList;
    va_start(argList, len);
    for(; len;len--) {
        int j=va_arg(argList, int);
        data[P++]=j;
        Serial.print(" ");
    }
}

```

```

    Serial.print(j, HEX);
}
IP=P<<2;
va_end(argList);
}

```

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

```

void FOR(int len, ... ) {
    P=IP>>2;
    Serial.println();
    Serial.print(IP, HEX);
    Serial.print(" FOR ");
    data[P++]=TOR;
    pushR=P;
    va_list argList;
    va_start(argList, len);
    for(; len; len--) {
        int j=va_arg(argList, int);
        data[P++]=j;
        Serial.print(" ");
        Serial.print(j, HEX);
    }
    IP=P<<2;
    va_end(argList);
}

```

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

```

void NEXT(int len, ... ) {
    P=IP>>2;
    Serial.println();
    Serial.print(IP, HEX);
    Serial.print(" NEXT ");
    data[P++]=DONXT;
    data[P++]=popR<<2;
    va_list argList;
    va_start(argList, len);
    for(; len; len--) {
        int j=va_arg(argList, int);
        data[P++]=j;
        Serial.print(" ");
        Serial.print(j, HEX);
    }
    IP=P<<2;
    va_end(argList);
}

```

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

```
void AFT(int len, ... ) {
    P=IP>>2;
    int k;
    Serial.println();
    Serial.print(IP,HEX);
    Serial.print(" AFT ");
    data[P++]=BRAN;
    data[P++]=0;
    k=popR;
    pushR=P;
    pushR=P-1;
    va_list argList;
    va_start(argList, len);
    for(; len;len--) {
        int j=va_arg(argList, int);
        data[P++]=j;
        Serial.print(" ");
        Serial.print(j,HEX);
    }
    IP=P<<2;
    va_end(argList);
}
```

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

```
." <string>"
$" <string>"
ABORT" <string>"
```

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

```
void DOTQ(char seq[]) {
    P=IP>>2;
    int i;
    int len=strlen(seq);
    data[P++]=DOTQP;
    IP=P<<2;
```



```

    cData[IP++]=len;
    for (i=0;i<len;i++)
        {cData[IP++]=seq[i];}
    while (IP&3) {cData[IP++]=0;}
    Serial.println();
    Serial.print(IP, HEX);
    Serial.print(" ");
    Serial.print(seq);
}

```

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

```

void STRQ(char seq[]) {
    P=IP>>2;
    int i;
    int len=strlen(seq);
    data[P++]=STRQP;
    IP=P<<2;
    cData[IP++]=len;
    for (i=0;i<len;i++)
        {cData[IP++]=seq[i];}
    while (IP&3) {cData[IP++]=0;}
    Serial.println();
    Serial.print(IP, HEX);
    Serial.print(" ");
    Serial.print(seq);
}

```

ABORQ() starts a string literal as a warning message to be printed out when Forth is aborted. It first assembles a **ABORQP** token. Then the string, terminated by another double quote, is assembled as a counted string. The string is null-filled to the 32-bit word boundary, similar to what **HEADER()** does.

```

void ABORQ(char seq[]) {
    P=IP>>2;
    int i;
    int len=strlen(seq);
    data[P++]=ABORQP;
    IP=P<<2;
    cData[IP++]=len;
    for (i=0;i<len;i++)
        {cData[IP++]=seq[i];}
    while (IP&3) {cData[IP++]=0;}
    Serial.println();
    Serial.print(IP, HEX);
    Serial.print(" ");
    Serial.print(seq);
}

```

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

```
void CheckSum() {
    int i;
    char sum=0;
    Serial.println();
    Serial.printf("%4x ", IP);
    for (i=0;i<32;i++) {
        sum += cData[IP];
        Serial.printf("%2x", cData[IP++]);
    }
    Serial.printf(" %2x", sum);
}
```

PWM Output to GPIO pins

Most of the GPIO pin in ESP32 can be configured for PWM output. They are most often used to dim LED diodes.

ledcAnalogWrite() is used to reduce the duty cycle of a particular channel assigned (attached) to a GPIO pin. You specify a channel and the desired duty cycle. Duty cycle must be from 0 to 255.

```
// LEDC Software Fade
// use first channel of 16 channels (started from zero)
#define LEDC_CHANNEL_0      0
#define LEDC_TIMER_13_BIT  13
#define LEDC_BASE_FREQ     5000
#define LED_PIN             5
int brightness = 255;      // how bright the LED is
void ledcAnalogWrite(uint8_t channel, uint32_t value, uint32_t valueMax =
255) {
    uint32_t duty = (8191 / valueMax) * min(value, valueMax);
    ledcWrite(channel, duty);
}
```

FVM Primitives

Then come all the FVM pseudo instructions coded as C functions. Each instruction will be assigned a byte code, and a finite state machine is designed to execute consecutive byte code placed in code fields of words in the dictionary. Byte code are pseudo instructions of FVM, 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() {}

```

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; }

```

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

```

void txsto(void)
{ Serial.write( (unsigned char) top);
  char c=top;
  HTTPout += c ;
  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 words. EXIT pops the execution address saved on the return stack back into the IP register and thus restores the condition before the compound word 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 word allows you to execute any token which is not a part of a token list.

```

void execu(void)
{ P = top;

```

```

WP = P + 4;
pop; }

```

donext (--) Terminate a FOR-NEXT loop. The loop count was pushed on return stack, and is decremented by **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]; }

```

```

void rpat(void) {}
void rpsto(void) {}

```

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;   }
```

```
void spat(void) {}
```

```
void spsto(void) {}
```

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) {
  WP = stack[(unsigned char) S] / top;
  stack[(unsigned char) S] %= top;
  top = WP;
} }
```

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;
  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;
  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; }
```

audio (freq channel --) sends a PWM signal out on a GPIO output pin.

```
void audio(void)
{   WP=top; pop;
    ledcWriteTone(WP,top);
    pop;
}
```

```
void sendPacket(void) {}
```

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; }
```

peek (a - n) reads one word n from an absolute address a in ESP8266.

```
void peek(void)
{   Pointer = (long*)top; top = *Pointer; }
```

adc (a - n) reads one word n from an absolute address a in ESP8266.

```
void adc(void)
{   top= (long) analogRead(top); }
```

pin (n pin -) attach channel n to a pin.

```
void pin(void)
{   WP=top; pop;
    ledcAttachPin(top,WP);
    pop;
}
```

duty (n pin -) set duty cycle on a channel to n.

```
void duty(void)
{   WP=top; pop;
    ledcAnalogWrite(WP,top,255);
    pop;
}
```

```

freq ( n pin - ) set frequency on a channel to n.
void freq(void)
{ WP=top; pop;
  ledcSetup(WP, top, 13);
  pop;
}

```

Primitive Execution Table

There are 72 pseudo instructions defined in FVM as shown before. Each of these instructions is assigned a unique byte code. In the dictionary, there are primitive words which have byte code in their code field. The byte code may spilled over into the subsequent parameter field, if a primitive word is very complicated. FVM has a finite state machine, which will be discussed shortly, to execute consecutive byte code. The numbering of these byte code in the following `primitives[]` array does not assume any perceived order.

Only 72 byte code are defined. You can extend them to 256 if you wanted. You have the options to write more pseudo instructions in C to extend the FVM, or to assemble more primitive words using the macro assembler I will discuss later, or to compile more colon words in Forth, which is far easier. The same function defined in different ways should behave identically. Only the execution speed might differ, inversely proportional to the efforts in programming..

Primitive pseudo instructions are assigned byte code for the FVM to execute. These pseudo instructions are organized in a `primitives[72]` array using a byte code as a pointer to executed specified C routines.

```

// execution table

void (*primitives[72])(void) = {
  /* case 0 */ nop,
  /* case 1 */ accep,
  /* case 2 */ qrx,
  /* case 3 */ txsto,
  /* case 4 */ docon,
  /* case 5 */ dolit,
  /* case 6 */ dolist,
  /* case 7 */ exitt,
  /* case 8 */ execu,
  /* case 9 */ donext,
  /* case 10 */ qbran,
  /* case 11 */ bran,
  /* case 12 */ store,
  /* case 13 */ at,
  /* case 14 */ cstor,
  /* case 15 */ cat,
  /* case 16 */ nop,
  /* case 17 */ nop,
  /* case 18 */ rfrom,
  /* case 19 */ rat,
  /* case 20 */ tor,

```

```

/* case 21 */ nop,
/* case 22 */ nop,
/* case 23 */ drop,
/* case 24 */ dup,
/* case 25 */ swap,
/* case 26 */ over,
/* case 27 */ zless,
/* case 28 */ andd,
/* case 29 */ orr,
/* case 30 */ xorr,
/* case 31 */ uplus,
/* case 32 */ next,
/* case 33 */ qdup,
/* case 34 */ rot,
/* case 35 */ ddrop,
/* case 36 */ ddup,
/* case 37 */ plus,
/* case 38 */ inver,
/* case 39 */ negat,
/* case 40 */ dnegat,
/* case 41 */ subb,
/* case 42 */ abss,
/* case 43 */ equal,
/* case 44 */ uless,
/* case 45 */ less,
/* case 46 */ ummod,
/* case 47 */ msmod,
/* case 48 */ slmod,
/* case 49 */ mod,
/* case 50 */ slash,
/* case 51 */ umsta,
/* case 52 */ star,
/* case 53 */ mstar,
/* case 54 */ ssmod,
/* case 55 */ stasl,
/* case 56 */ pick,
/* case 57 */ pstor,
/* case 58 */ dstor,
/* case 59 */ dat,
/* case 60 */ count,
/* case 61 */ dovar,
/* case 62 */ maxx,
/* case 63 */ minn,
/* case 64 */ audio,
/* case 65 */ sendPacket,
/* case 66 */ poke,
/* case 67 */ peepk,
/* case 68 */ adc,
/* case 69 */ pin,
/* case 70 */ duty,
/* case 71 */ freq };

```

FVM Finite State Machine

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

`evaluate()` has an infinite loop in it, continually executing consecutive byte code. When it encounters a null byte, which is `NOP` byte code, the loop is broken, and Forth peacefully hands control back to the Arduino loop program which called it. Arduino loop will then wait for a new line of input text, and calls `evaluate()` again to evaluate it.

```
// finite state machine

void evaluate()
{ while (true){
    bytecode=(unsigned char)cData[P++];
    if (bytecode) {primitives[bytecode]();}
    else {break;}
} // break on NOP
}
```

Byte Code Assembly Mnemonic

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

```
// byte code

int as_nop=0;
int as_accept=1;
int as_qrx=2;
int as_txsto=3;
int as_docon=4;
int as_dolist=5;
int as_dolist=6;
int as_exit=7;
int as_execu=8;
int as_donext=9;
int as_qbran=10;
int as_bran=11;
int as_store=12;
int as_at=13;
int as_cstor=14;
int as_cat=15;
int as_rpat=16;
int as_rpsto=17;
int as_rfrom=18;
int as_rat=19;
int as_tor=20;
int as_spat=21;
int as_spsto=22;
int as_drop=23;
int as_dup=24;
int as_swap=25;
```

```
int as_over=26;
int as_zless=27;
int as_andd=28;
int as_orr=29;
int as_xorr=30;
int as_uplus=31;
int as_next=32;
int as_qdup=33;
int as_rot=34;
int as_ddrop=35;
int as_ddup=36;
int as_plus=37;
int as_inver=38;
int as_negat=39;
int as_dnega=40;
int as_subb=41;
int as_abss=42;
int as_equal=43;
int as_uless=44;
int as_less=45;
int as_ummod=46;
int as_msmmod=47;
int as_slmod=48;
int as_mod=49;
int as_slash=50;
int as_umsta=51;
int as_star=52;
int as_mstar=53;
int as_ssmmod=54;
int as_stasl=55;
int as_pick=56;
int as_pstor=57;
int as_dstor=58;
int as_dat=59;
int as_count=60;
int as_dovar=61;
int as_max=62;
int as_min=63;
int as_tone=64;
int as_sendPacket=65;
int as_poke=66;
int as_peek=67;
int as_adc=68;
int as_pin=69;
int as_duty=70;
int as_freq=71;
```

Assembling Forth Dictionary

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

`setup()` also initializes the USART0 to 115,200 bauds, and displays a sign-on message. Counter P is initialized to dictionary location 0x180, and WP to 0x184. This code segment

jumps to EVAL at location 0x157C, which starts the Forth interpreter. However, register P and IP will be drafted by macro assembler to assemble Forth dictionary. They will be restored after the dictionary is assembled.

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

```
void setup() {
  P = 0x180;
  WP = 0x184;
  IP = 0;
  S = 0;
  R = 0;
  top = 0;
  cData = (uint8_t *) data;
  Serial.begin(115200);
  delay(100);
  Serial.println("Booting esp32Forth v6.2 ...");
}
```

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

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

User Variables

User variables are in the area between 0x1A0-0x1C4. They contain vital information for the esp32forth system to work properly.

User Variable	Address	Initial Value	Function
HLD	0x190	0	Pointer to text buffer for number output.
SPAN	0x194	0	Number of input characters.
>IN	0a198	0	Pointer to next character to be interpreted.
#TIB	0x19C	0	Number of characters received in terminal input buffer.
'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 word in dictionary.
CP	0x1AC	0x1DE8	Pointer to top of dictionary, first free memory location to add new words. It is saved by "h forth_@" on top of the source code page.
LAST	0x1B0	0x1DDC	Pointer to name field of last word.

'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 word to handle error conditions.
tmp	0x1BC	0	Scratch pad.
Z	Constant	0	0.
ppqn	0x1C0	0	Pulses per quarter note.
channel	0x1C4	0	Timer channel for voice output.

```
IP=512;
R=0;
```

```
HEADER(3, "HLD");
int HLD=CODE(8, as_docon, as_next, 0, 0, 0X90, 1, 0, 0);
HEADER(4, "SPAN");
int SPAN=CODE(8, as_docon, as_next, 0, 0, 0X94, 1, 0, 0);
HEADER(3, ">IN");
int INN=CODE(8, as_docon, as_next, 0, 0, 0X98, 1, 0, 0);
HEADER(4, "#TIB");
int NTIB=CODE(8, as_docon, as_next, 0, 0, 0X9C, 1, 0, 0);
HEADER(4, "'TIB");
int TTIB=CODE(8, as_docon, as_next, 0, 0, 0XA0, 1, 0, 0);
HEADER(4, "BASE");
int BASE=CODE(8, as_docon, as_next, 0, 0, 0XA4, 1, 0, 0);
HEADER(7, "CONTEXT");
int CNTXT=CODE(8, as_docon, as_next, 0, 0, 0XA8, 1, 0, 0);
HEADER(2, "CP");
int CP=CODE(8, as_docon, as_next, 0, 0, 0XAC, 1, 0, 0);
HEADER(4, "LAST");
int LAST=CODE(8, as_docon, as_next, 0, 0, 0XB0, 1, 0, 0);
HEADER(5, "'EVAL");
int TEVAL=CODE(8, as_docon, as_next, 0, 0, 0XB4, 1, 0, 0);
HEADER(6, "'ABORT");
int TABRT=CODE(8, as_docon, as_next, 0, 0, 0XB8, 1, 0, 0);
HEADER(3, "tmp");
int TEMP=CODE(8, as_docon, as_next, 0, 0, 0XBC, 1, 0, 0);
HEADER(1, "Z");
int Z=CODE(8, as_docon, as_next, 0, 0, 0, 0, 0, 0);
HEADER(4, "ppqn");
int PPQN=CODE(8, as_docon, as_next, 0, 0, 0XC0, 1, 0, 0);
HEADER(7, "channel");
int CHANN=CODE(8, as_docon, as_next, 0, 0, 0XC4, 1, 0, 0);
```

Kernel Words

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

```
// primitive words
```

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

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

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

```
HEADER(4, "?KEY");  
int QKEY=CODE(4, as_qrx, as_next, 0, 0);
```

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.

```
HEADER(4, "EMIT");  
int EMIT=CODE(4, as_txsto, as_next, 0, 0);
```

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

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

DOLIST (--) Push the current Instruction Pointer (IP) 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 words. The token list in a compound word must be terminated by `EXIT`.

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

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

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

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

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

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

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

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

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

```
HEADER(7, "QBRANCH");  
QBRAN=CODE(4, as_qbran, as_next, 0, 0);  
HEADER(6, "BRANCH");
```


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

```
BRAN=CODE(4, as_bran, as_next, 0, 0);
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

DROP (w --) Discard top stack item.

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

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

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

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

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

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

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

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

```
HEADER(2, "0<");
```

```

int ZLESS=CODE(4,as_zless,as_next,0,0);

AND ( w w -- w ) Bitwise AND.
  HEADER(3, "AND");
  int ANDD=CODE(4,as_andd,as_next,0,0);

OR ( w w -- w ) Bitwise inclusive OR.
  HEADER(2, "OR");
  int ORR=CODE(4,as_orr,as_next,0,0);

XOR ( w w -- w ) Bitwise exclusive OR.
  HEADER(3, "XOR");
  int XORR=CODE(4,as_xorr,as_next,0,0);

UM+ ( w w -- w cy ) Add two numbers, return the sum and carry flag.
  HEADER(3, "UM+");
  int UPLUS=CODE(4,as_uplus,as_next,0,0);
  HEADER(4, "?DUP");
  int QDUP=CODE(4,as_qdup,as_next,0,0);
  HEADER(3, "ROT");
  int ROT=CODE(4,as_rot,as_next,0,0);
  HEADER(5, "2DROP");
  int DDROP=CODE(4,as_ddrop,as_next,0,0);
  HEADER(4, "2DUP");
  int DDUP=CODE(4,as_ddup,as_next,0,0);

+ ( w w -- sum ) Add top two items.
  HEADER(1, "+");
  int PLUS=CODE(4,as_plus,as_next,0,0);

NOT ( w -- w ) One's complement of top.
  HEADER(3, "NOT");
  int INVER=CODE(4,as_inver,as_next,0,0);

NEGATE ( n -- -n ) Two's complement of top.
  HEADER(6, "NEGATE");
  int NEGAT=CODE(4,as_negat,as_next,0,0);

DNEGATE ( d -- -d ) Two's complement of top double.
  HEADER(7, "DNEGATE");
  int DNEGA=CODE(4,as_dnega,as_next,0,0);

- ( n1 n2 -- n1-n2 ) Subtraction.
  HEADER(1, "-");
  int SUBBB=CODE(4,as_subb,as_next,0,0);

ABS ( n -- n ) Return the absolute value of n.
  HEADER(3, "ABS");
  int ABSS=CODE(4,as_abss,as_next,0,0);

= ( w w -- t ) Return true if top two are equal.
  HEADER(1, "=");
  int EQUAL=CODE(4,as_equal,as_next,0,0);

```

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

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

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

```
HEADER(1, "<");  
int LESS=CODE(4, as_less, as_next, 0, 0);
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

+! ( n a -- ) Add n to the contents at address a.
HEADER(2,"+!");
int PSTOR=CODE(4,as_pstor,as_next,0,0);

2! ( d a -- ) Store the double integer to address a.
HEADER(2,"2!");
int DSTOR=CODE(4,as_dstor,as_next,0,0);

2@ ( a -- d ) Fetch double integer from address a.
HEADER(2,"2@");
int DAT=CODE(4,as_dat,as_next,0,0);

COUNT ( b -- b+1 +n ) Return count byte of a string and add 1 to byte address.
HEADER(5,"COUNT");
int COUNT=CODE(4,as_count,as_next,0,0);

MAX ( n1 n2 -- n ) Return the greater of two top stack items.
HEADER(3,"MAX");
int MAX=CODE(4,as_max,as_next,0,0);

MIN ( n1 n2 -- n ) Return the smaller of top two stack items.
HEADER(3,"MIN");
int MIN=CODE(4,as_min,as_next,0,0);

BL ( -- 32 ) Return the blank ASCII code 32.
HEADER(2,"BL");
int BLANK=CODE(8,as_docon,as_next,0,0,32,0,0,0,0);

CELL ( -- 4 ) Return number of bytes in a 32-bit integer.
HEADER(4,"CELL");
int CELL=CODE(8,as_docon,as_next,0,0,4,0,0,0,0);

CELL+ ( n -- n+4 ) Add 4 to n.
HEADER(5,"CELL+");
int CELLP=CODE(8,as_docon,as_plus,as_next,0,4,0,0,0,0);

CELL- ( n -- n-4 ) Subtract 4 from n.
HEADER(5,"CELL-");
int CELLM=CODE(8,as_docon,as_subb,as_next,0,4,0,0,0,0);

CELL* ( n -- n*4 ) Multiply n by 4.
HEADER(5,"CELLS");
int CELLS=CODE(8,as_docon,as_star,as_next,0,4,0,0,0,0);

CELL/ ( n -- n/4 ) Divide n by 4.
HEADER(5,"CELL/");
int CELLD=CODE(8,as_docon,as_slash,as_next,0,4,0,0,0,0);

1+ ( n -- n+1 ) Increment n.

```

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

1- ($n - n-1$) Decrement n.

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

2+ ($n - n+2$) Add 2 to n.

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

2- ($n - n-2$) Subtract 2 from n.

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

2* ($n - n*2$) Multiply n by 2.

```
HEADER(2, "2*");
int TWOST=CODE(8, as_docon, as_star, as_next, 0, 2, 0, 0, 0);
```

2/ ($n - n/2$) Divide n by 2.

```
HEADER(2, "2/");
int TWOS=CODE(8, as_docon, as_slash, as_next, 0, 2, 0, 0, 0);
```

POKE (n a --) writes one word n to an absolute address a in ESP32.

```
HEADER(4, "POKE");
int POKE=CODE(4, as_poke, as_next, 0, 0);
```

PEEK(a - n) reads one word n from an absolute address a in ESP32.

```
HEADER(4, "PEEK");
int PEEK=CODE(4, as_peek, as_next, 0, 0);
```

ADC (a - n) reads one word n from an absolute address a in ESP8266.

```
HEADER(3, "ADC");
int ADC=CODE(4, as_adc, as_next, 0, 0);
```

PIN (pin chan -) attaches GPIO pin to channel chan.

```
HEADER(3, "PIN");
int PIN=CODE(4, as_pin, as_next, 0, 0);
```

TONE (n chan -) sets frequency to channel chan.

```
HEADER(4, "TONE");
int TONE=CODE(4, as_tone, as_next, 0, 0);
```

DUTY (n pin -) sets duty cycle on a channel to n.

```
HEADER(4, "DUTY");
int DUTY=CODE(4, as_duty, as_next, 0, 0);
```

FREQ (n pin -) sets frequency on a channel to n.

```
HEADER(4, "FREQ");
int FREQ=CODE(4, as_freq, as_next, 0, 0);
```

Common Words

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

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

KEY (-- c) wait for the terminal input device to receive a character c.

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

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

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

>CHAR (c -- c) is very important in converting a non-printable character to a harmless 'underscore' character (ASCII 95). As 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.

```
HEADER(5, ">CHAR");  
int TCHAR=COLON(8, DOLIT, 0x7F, ANDD, DUPP, DOLIT, 127, BLANK, WITHI);  
IF(3, DROP, DOLIT, 0x5F);  
THEN(1, EXITT);
```

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

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

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

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

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

```
HEADER(3, "PAD");  
int PAD=COLON(5, HERE, DOLIT, 80, PLUS, EXITT);
```

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

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

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

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

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

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

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

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

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

```
:: FILL  
HEADER(4, "FILL");  
int FILL=COLON(1, SWAP);  
FOR(1, SWAP);  
AFT(3, DDUP, CSTOR, ONEP);  
THEN(0);  
NEXT(2, DDROP, EXITT);
```

Number Conversions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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


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

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

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

:: DIGIT?

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

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

```
HEADER(7, "NUMBER?");
int
NUMBQ=COLON(12, BASE, AT, TOR, DOLIT, 0, OVER, COUNT, OVER, CAT, DOLIT, 0x24, EQUAL);
IF(5, HEXX, SWAP, ONEP, SWAP, ONEM);
THEN(13, OVER, CAT, DOLIT, 0x2D, EQUAL, TOR, SWAP, RAT, SUBBB, SWAP, RAT, PLUS, QDUP);
IF(1, ONEM);
FOR(6, DUPP, TOR, CAT, BASE, AT, DIGTQ);
WHILE(7, SWAP, BASE, AT, STAR, PLUS, RFROM, ONEP);
NEXT(2, DROP, RAT);
IF(1, NEGAT);
THEN(1, SWAP);
ELSE(6, RFROM, RFROM, DDROP, DDROP, DOLIT, 0);
THEN(1, DUPP);
THEN(6, RFROM, DDROP, RFROM, BASE, STORE, EXITT);
```

Text Output

SPACE (--) outputs a blank space character.

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

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

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

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

```

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

```

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

```

HEADER(4, "TYPE");
int TYPES=COLON(0);
FOR(0);
AFT(5, DUPP, CAT, TCHAR, EMIT, ONEP);
THEN(0);
NEXT(2, DROP, EXITT);

```

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.

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```
HEADER(1, ".");  
int DOT=COLON(5, BASE, AT, DOLIT, 10, XORR);  
IF(3, UDOT, EXITT);  
THEN(4, STRR, SPACE, TYPES, EXITT);
```

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

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

Dictionary Search

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

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

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

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

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

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

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

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

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

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

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

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

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

```
HEADER(5, "SAME?");
int SAMEQ=COLON(4, DOLIT, 0x1F, ANDD, CELLD);
FOR(0);
```

AFT(18, OVER, RAT, DOLIT, 4, STAR, PLUS, AT, UPPER, OVER, RAT, DOLIT, 4, STAR, PLUS, AT, UPPER, SUBBB, QDUP);

```
IF(3, RFROM, DROP, EXITT);
THEN(0);
THEN(0);
NEXT(3, DOLIT, 0, EXITT);
```

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

```
HEADER(4, "find");
int FIND=COLON(10, SWAP, DUPP, AT, TEMP, STORE, DUPP, AT, TOR, CELLP, SWAP);
BEGIN(2, AT, DUPP);
IF(9, DUPP, AT, DOLIT, 0xFFFFFFFF3F, ANDD, UPPER, RAT, UPPER, XORR);
IF(3, CELLP, DOLIT, 0xFFFFFFFF);
ELSE(4, CELLP, TEMP, AT, SAMEQ);
THEN(0);
ELSE(6, RFROM, DROP, SWAP, CELLM, SWAP, EXITT);
THEN(0);
WHILE(2, CELLM, CELLM);
REPEAT(9, RFROM, DROP, SWAP, DROP, CELLM, DUPP, NAMET, SWAP, EXITT);
```

Interpreter

NAME? (a -- cfa nfa | a 0) find a word in dictionary with name at a. If found, return its cfa and nfa. Else, push a false flag on top of a.

```

HEADER(5, "NAME?");
int NAMEQ=COLON(3, CNTXT, FIND, EXITT);

```

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

```

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

```

QUERY (--) is the 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.

```

HEADER(5, "QUERY");
int
QUERY=COLON(12, TIB, DOLIT, 0X100, ACCEP, NTIB, STORE, DROP, DOLIT, 0, INN, STORE, EXITT);

```

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

```

HEADER(5, "ABORT");
int ABORT=COLON(4, NOP, TABRT, ATEXE, EXITT);

```

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

```

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

```

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

```

HEADER(5, "ERROR");
int ERRORR=COLON(8, SPACE, COUNT, TYPES, DOLIT, 0X3F, EMIT, CR, ABORT);

```

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

```

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

```

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

```

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

```

.OK (--) used to be a word which displays the familiar 'ok' prompt after executing to the end of a line. In esp32forth_62, it displays the top 4 elements on data stack so you can see what is happening on the stack. It is more informative than the plain 'ok', which only give you a warm and fuzzy feeling about the system. When text interpreter is in compiling mode, the display is suppressed.

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

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

```
HEADER(4, "EVAL");
int EVAL=COLON(1, LBRAC);
BEGIN(3, TOKEN, DUPP, AT);
WHILE(2, TEVAL, ATEXE);
REPEAT(4, DROP, DOTOK, NOP, EXITT);
```

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.

```
HEADER(4, "QUIT");
int QUITT=COLON(1, LBRAC);
BEGIN(2, QUERY, EVAL);
AGAIN(0);
```

LOAD (a n --) evaluates a text string at a with length n. It is to compile a program store in flash memory of ESP32.

```
HEADER(4, "LOAD");
int LOAD=COLON(10, NTIB, STORE, TTIB, STORE, DOLIT, 0, INN, STORE, EVAL, EXITT);
```

Compiler

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

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

LITERAL (n --) compiles an integer literal to the current compound word under construction. The integer literal is taken from the data stack, and is preceded by the token DOLIT. When this compound word is executed, DOLIT will extract the integer from the token

list and push it back on the data stack. **LITERAL** compiles an address literal if the compiled integer happens to be an execution address of a token. The address will be pushed on the data stack at the run time by **DOLIT**.

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

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

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

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

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

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

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

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

```
HEADER(3, "$, n");
int SNAME=COLON(2, DUPP, AT);
```

```
IF(14, UNIQU, DUPP, NAMET, CP, STORE, DUPP, LAST, STORE, CELLM, CNTXT, AT, SWAP, STORE, EXITT);
THEN(1, ERRORR);
```

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

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

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

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

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

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

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

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

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

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

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

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

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

```
HEADER(1, "):");  
int COLN=COLON(7, TOKEN, SNAME, RBRAC, DOLIT, 0x6, COMMA, EXITT);
```

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

```
HEADER(IMEDD+1, ";");  
int SEMIS=COLON(6, DOLIT, EXITT, COMMA, LBRAC, OVERT, EXITT);
```

Debugging Tools

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

```
HEADER(3, "dm+");
int DMP=COLON(4, OVER, DOLIT, 6, UDOTR);
FOR(0);
AFT(6, DUPP, AT, DOLIT, 9, UDOTR, CELLP);
THEN(0);
NEXT(1, EXITT);
```

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

```
HEADER(4, "DUMP");
int DUMP=COLON(10, BASE, AT, TOR, HEXX, DOLIT, 0x1F, PLUS, DOLIT, 0x20, SLASH);
FOR(0);
AFT(10, CR, DOLIT, 8, DDUP, DMP, TOR, SPACE, CELLS, TYPES, RFROM);
THEN(0);
NEXT(5, DROP, RFROM, BASE, STORE, EXITT);
```

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

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

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

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

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

```
HEADER(5, "WORDS");
int WORDS=COLON(6, CR, CNTXT, DOLIT, 0, TEMP, STORE);
BEGIN(2, AT, QDUP);
WHILE(9, DUPP, SPACE, DOTID, CELLM, TEMP, AT, DOLIT, 0x10, LESS);
IF(4, DOLIT, 1, TEMP, PSTOR);
ELSE(5, CR, DOLIT, 0, TEMP, STORE);
THEN(0);
REPEAT(1, EXITT);
```

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

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

COLD (--) is a high level word executed upon power-up. It sends out sign-on message, and then falls into the text interpreter loop through **QUIT**. In this system **EVAL** is called from Arduino, and **QUIT** is not used.

```
HEADER(4, "COLD");
int COLD=COLON(1, CR);
DOTQ("esp32forth V5.4, 2019 ");
int DOTQ1=LABEL(2, CR, EXITT);
```

ESP32 Application Words

// esp32 application words

LINE (a - a+8) writes 8 consecutive words from absolute memory address a.

```
HEADER(4, "LINE");
int LINE=COLON(2, DOLIT, 0x7);
FOR(6, DUPP, PEEK, DOLIT, 0x9, UDOTR, CELLP);
NEXT(1, EXITT);
```

PP (a n-) dumps n+1 lines of consecutive words from absolute memory address a.

```
HEADER(2, "PP");
int PP=COLON(0);
FOR(0);
AFT(7, CR, DUPP, DOLIT, 0x9, UDOTR, SPACE, LINE);
THEN(0);
NEXT(1, EXITT);
```

P0 (n-) outputs a 32-bit word n to P0 port in ESP32.

```
HEADER(2, "P0");
int P0=COLON(4, DOLIT, 0x3FF44004, POKE, EXITT);
```

P0S (n-) sets P0 output bits according to bits in mask n.

```
HEADER(3, "P0S");
int P0S=COLON(4, DOLIT, 0x3FF44008, POKE, EXITT);
```

P0C (n-) clears P0 output bits according to bits in mask n.

```
HEADER(3, "P0C");
int P0C=COLON(4, DOLIT, 0x3FF4400C, POKE, EXITT);
```

P1 (n-) outputs a 32-bit word n to P1 port in ESP32.

```
HEADER(2, "P1");
int P1=COLON(4, DOLIT, 0x3FF44010, POKE, EXITT);
```

P1S (n-) sets P1 output bits according to bits in mask n.

```
HEADER(3, "P1S");  
int P1S=COLON(4, DOLIT, 0x3FF44014, POKE, EXITT);
```

P1C (n-)clears P1 output bits according to bits in mask n.
HEADER(3, "P1C");
int P1C=COLON(4, DOLIT, 0x3FF44018, POKE, EXITT);

P0EN (n-)enables output bits in P0 port according to mask n.
HEADER(4, "P0EN");
int P0EN=COLON(4, DOLIT, 0x3FF44020, POKE, EXITT);

P0ENS (n-)sets output enable bits in P0 port according to mask n.
HEADER(5, "P0ENS");
int P0ENS=COLON(4, DOLIT, 0x3FF44024, POKE, EXITT);

P0ENC (n-)clears output enable bits in P0 port according to mask n.
HEADER(5, "P0ENC");
int P0ENC=COLON(4, DOLIT, 0x3FF44028, POKE, EXITT);

P1EN (n-)enables output bits in P1 port according to mask n.
HEADER(4, "P1EN");
int P1EN=COLON(4, DOLIT, 0x3FF4402C, POKE, EXITT);

P1ENS (n-)sets output enable bits in P1 port according to mask n.
HEADER(5, "P1ENS");
int P1ENS=COLON(4, DOLIT, 0x3FF44030, POKE, EXITT);

P1ENC (n-)clears output enable bits in P1 port according to mask n.
HEADER(5, "P1ENC");
int P1ENC=COLON(4, DOLIT, 0x3FF44034, POKE, EXITT);

P0IN (- n)reads 32-bits from P0 input port.
HEADER(4, "P0IN");
int P0IN=COLON(5, DOLIT, 0x3FF4403C, PEEK, DOT, EXITT);

P1IN (- n)reads 32-bits from P1 input port.
HEADER(4, "P1IN");
int P1IN=COLON(5, DOLIT, 0x3FF44040, PEEK, DOT, EXITT);

PPP (-)dumps 4 lines of consecutive words from registers in P0 and P1 ports.
HEADER(3, "PPP");
int PPP=COLON(7, DOLIT, 0x3FF44000, DOLIT, 3, PP, DROP, EXITT);

EMITT(n -)displays 4 consecutive bytes in n.
HEADER(5, "EMITT");
int EMITT=COLON(2, DOLIT, 0x3);
FOR(8, DOLIT, 0, DOLIT, 0x100, MSMOD, SWAP, TCHAR, EMIT);
NEXT(2, DROP, EXITT);

TYPEE(a - a+32)displays 8 consecutive words in from absolute address a.
HEADER(5, "TYPEE");

```
int TYPEE=COLON(3,SPACE,DOLIT,0x7);
FOR(4,DUPP,PEEK,EMITT,CELLP);
NEXT(2,DROP,EXITT);
```

PPPP (a n -)dumps n+1 lines of consecutive words from absolute address a.

```
HEADER(4,"PPPP");
int PPPP=COLON(0);
FOR(0);
AFT(10,CR,DUPP,DUPP,DOLIT,0x9,UDOTR,SPACE,LINE,SWAP,TYPEE);
THEN(0);
NEXT(1,EXITT);
```

KKK (-)dumps 17 lines of data from the LEDC-PWM control registers of ESP32.

```
HEADER(3,"KKK");
int KKK=COLON(7,DOLIT,0x3FF59000,DOLIT,0x10,PP,DROP,EXITT);
```

Control Structures

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

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

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

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

```
HEADER(IMEDD+3,"FOR");
int FORR=COLON(4,COMPI,TOR,HERE,EXITT);
```

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

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

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

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

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

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

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

String Literals

ABORT" (-- ; <string>) compiles an error message. This error message is display if top item on the stack is non-zero. The rest of the words in the word 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.

```

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

```

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

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

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

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

Defining words

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

```
HEADER(4, "CODE");
int CODE=COLON(5, TOKEN, SNAME, OVERT, ALIGN, EXITT);
```

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

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

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

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

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

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

Comments

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

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

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

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

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

```
HEADER(IMEDD+1, "(");
int PAREN=COLON(5, DOLIT, 0X29, PARSE, DDROP, EXITT);
```

Lexicon Bits

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

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

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

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

Checking Macro Assembler

For my own sanity, I added Serial.print() commands in macros to display data they assembled so I can verify that they behave correctly. You can remove these print commands to let the macros work quietly. However, I leave them along so that you can also see how the macro assembler work, and perhaps gain some insight into what this esp32forth was built and how a Forth system works.

After the dictionary is completed, IP is printed to show the size in byte of the dictionary. I also print out the contents of the return stack pointer R used to assembler control structure. R has to be 0 if all the control structures were assembled correctly.

I also dump the contents in the dictionary in Intel-Dump-like format, with checksums calculated for every 32 bytes of data. I used the checksums to verify that this dictionary is identical to the dictionary I used in espForth_44 system. It is a good exercise for you to read this dictionary dump to see the records of Forth words, and the fields in these word records.

```
int ENDD=IP;
Serial.println();
Serial.print("IP=");
Serial.print(IP);
Serial.print(" R-stack= ");
Serial.print(popR<<2, HEX);
IP=0x180;
int USER=LABEL(16, 6, EVAL, 0, 0, 0, 0, 0, 0, 0, 0x10, IMMED-12, ENDD, IMMED-12, INTER, EVAL, 0);
```

```
// dump dictionary
IP=0;
for (len=0; len<0x120; len++){Checksum();}
```

Earlier, the function `Checksum()` was defined as:

```
void CheckSum() {
    int i;
    char sum=0;
    Serial.println();
    Serial.printf("%4x ", IP);
    for (i=0; i<32; i++) {
        sum += cData[IP];
        Serial.printf("%2x", cData[IP++]);
    }
    Serial.printf(" %2x", sum);
}
```

Initialize Timer and GPIO Pins

GPIO5 pin is set up to produce audio tunes if a small speaker is attached to it. GPIO16-19 are set up to send digital signals to drive two DC motors. They are initialized here. GPIO2 drives the blue LED on ESP32S Kit. I always turn it on after esp32forth boots up. See the blue LED turning on assures me that esp32forth is working.

```
// Setup timer and attach timer to a led pin
ledcSetup(0, 100, LEDC_TIMER_13_BIT);
ledcAttachPin(5, 0);
ledcAnalogWrite(0, 250, brightness);
pinMode(2, OUTPUT);
digitalWrite(2, HIGH); // turn the LED2 on
pinMode(16, OUTPUT);
digitalWrite(16, LOW); // motor1 forward
pinMode(17, OUTPUT);
digitalWrite(17, LOW); // motor1 backward
pinMode(18, OUTPUT);
digitalWrite(18, LOW); // motor2 forward
pinMode(19, OUTPUT);
digitalWrite(19, LOW); // motor2 backward
```

Compile Load.txt File

ESP32S Kit has 4 MB of flash memory. It is very convenient to store application program in the flash memory to customize esp32forth system for various embedded applications. Although I included many words to control ESP32 chip, but they are never enough to do real applications. You can now write your application in a text file named load.txt, and write it into the flash memory. When esp32forth boots up, it will read this file into its virtual RAM memory and run it. This way, you can build turnkey systems based on this NodeMCU ESP32S Kit.

I had preloaded 20 lessons in the `load.txt` file. You can see the names of new Forth words if you type `'WORDS'` into the input panel of Arduino Serial Monitor.

Arduino IDE provides a SPIFFS library so that we can open `Load.txt` file, and copy its contents to a memory area. Here it is placed in an area starting at `cData[0x8000]`. In `esp32forth`, the terminal input buffer is set up by initialize `>IN=0`, `#TIB=length of file`, `'TIB=0x8000`, `P=0x180`, and `WP=0x184`, and then call `evaluate()`.

```
// compile \data\load.txt
if(!SPIFFS.begin(true)){Serial.println("Error mounting SPIFFS"); }
File file = SPIFFS.open("/load.txt");
if(file) {
  Serial.print("Load file: ");
  len = file.read(cData+0x8000,0x7000);
  Serial.print(len);
  Serial.println(" bytes.");
  data[0x66] = 0;           // >IN
  data[0x67] = len;         // #TIB
  data[0x68] = 0x8000;      // 'TIB
  P = 0x180;                // EVAL
  WP = 0x184;
  evaluate();
  Serial.println(" Done loading.");
  file.close();
  SPIFFS.end();
}
}
```

Final Loop

In my earlier `espForth` implementation, I held a Forth-centric view, assuming that Forth is the operating system. A standalone Forth system works like:

`COLD-->QUIT-->BEGIN QUERY EVAL AGAIN`

Forth boots up on power-on, goes through `COLD` and `QUIT`, and then falls into this `QUERY-EVAL` loop. `QUERY` waits for a line of input, and then hands the line to `EVA` for interpretation. In `loop()`, there is a simple finite state machine:

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

Consecutive bytes code are fetched and executed. It is an infinite loop and never ends if Forth is running correctly. If they is any problem, `ESP32` will abort and re-initialize Forth.

Here we accept the Arduino way of running things. `Setup()` initializes everything and recompiles the Forth dictionary. `loop()` then waits a line from Serial Monitor and calls `evaluate()` to interpret it.

As shown in the following code, `Serial.readBytes` reads a line of input, up to 256 bytes, and places it in Terminal Input Buffer at virtual address 0, `cData[0x8000]`. The terminal input

buffer is set up by initialize >IN=0, #TIB=length of file, 'TIB=0, P=0x180, and WP=0x184, and then call evaluate().

```
void loop() {
    int len;
    while (true) {                // if there's bytes to read from the client,
        len = Serial.readBytes(cData, 256);
        if (len) {break;}
    }
    data[0x66] = 0;                // >IN
    data[0x67] = len;              // #TIB
    data[0x68] = 0;                // 'TIB
    P = 0x180;                     // EVAL
    WP = 0x184;
    evaluate();
}
```

evaluate() as discussed before, has an infinite loop in it, continually executing consecutive byte code. When it encounters a null byte, which is NOP byte code, the loop is broken, and Forth peacefully hands control back to loop(), waiting for the next line of text from Serial Monitor.

```
void evaluate()
{ while (true){
    bytecode=(unsigned char)cData[P++];
    if (bytecode) {primitives[bytecode]();}
    else {break;}
} // break on NOP
}
```

We have gone through the entire esp32forth system in its entirety. It is in an Arduino sketch written completely in C. Instead of a dummy compile-run programming system, you have an interactive programming environment, suitable for developing large and complex embedded systems. Incrementally compiling and testing small modules is the best way to develop programs, which can be thoroughly tested and debugged.

Though we intended our NodeMCU ESP32S kit to run a remotely controlled robot car, it can be used to do many different applications. With esp32forth, you can achieve more in much less time.

Chapter 4. Forth Lessons

To teach people Forth, particularly the eForth implementation I promoted since 1990, I wrote 17 lessons as programming models for people to study and to practice. Later, all lessons were combined into a single text file. This text file was a good test suite when a new eForth implementation was done. For NodeMCU ESP32S Kit, I added a few more lessons to demonstrate how to control a robot car. This file was rename Load.txt, and used to test the small flash file system SPIFFS. It is also a good example to demonstrate how you can build an embedded system on ESP32S Kit. Put Forth source code in Load.txt, and write it into flash memory on ESP32, with the 'ESP32 Sketch Data Upload add-on, it will be compile after esp32forth boots up on ESP32. You can load up to about 30 KB of source code in Load.txt.

The lessons in Load.txt are extensively commented. I encourage you to type the source code in, to get a good feeling about Forth as a programming language. Since all lessons are compiled already, you do not have to type in the source code. You can simply type in the final word in each lesson and execute them, to see the results they produce.

One interesting thing in the source code is the comments. Generally, comments are enclosed in parentheses. A comment starts with a left parenthesis, and ends with a right parenthesis. The comments can be very long, spanning over many lines. You can use this style to write your programs. However, limiting comments within a line of text is a good practice, common to most other Forth systems.

Since there are enough comments embedded in the source code, I will simply present the entire Load.txt file here for your reference.

(Example 1. The Universal Greeting)

DECIMAL

```
: HELLO CR ." Hello, world!" ;
```

(Example 2. The Big F)

```
: bar   CR ." *****" ;
: post  CR ." *      " ;
: F      bar post bar post post post ;
```

(Type 'F' and a return on your keyboard, and you will see a large F character displayed on the screen)

(Example 3. FIG, Forth Interest Group)

```
: center CR ."      *      " ;
: sides  CR ." *      *" ;
: triad1 CR ." * *  *" ;
: triad2 CR ." **  *" ;
```

```

: triad3 CR ." *  ***" ;
: triad4 CR ."  *** " ;
: quart  CR ." **  ***" ;
: right  CR ." *  ****" ;
: bigT   bar center center center center center center ;
: bigI   center center center center center center center ;
: bigN   sides triad2 triad2 triad1 triad3 triad2 sides ;
: bigG   triad4 sides post right triad1 sides triad4 ;
: FIG    F bigI bigG ;

```

(Example 4. Repeated Patterns

```

FOR      [ index -- ]          Set up loop given the index.
NEXT     [ -- ]                Decrement index by 1.  If index<0, exit.
If index=limit, exit loop; otherwise
                                Otherwise repeat after FOR.
R@       [ -- index ]          Return the current loop index. )

VARIABLE WIDTH                  ( number of asterisks to print )

: ASTERISKS ( -- , print n asterisks on the screen, n=width )
  WIDTH @                        ( limit=width, initial index=0 )
  FOR ." *"                      ( print one asterisk at a time )
  NEXT                            ( repeat n times )
  ;

: RECTANGLE ( height width -- , print a rectangle of asterisks )
  WIDTH !                        ( initialize width to be printed )
  FOR      CR
    ASTERISKS                    ( print a line of asterisks )
  NEXT
  ;

: PARALLELOGRAM ( height width -- )
  WIDTH !
  FOR      CR R@ SPACES          ( shift the lines to the right )
    ASTERISKS                    ( print one line )
  NEXT
  ;

: TRIANGLE ( width -- , print a triangle area with asterisks )
  FOR      CR
    R@ WIDTH !                  ( increase width every line )
    ASTERISKS                    ( print one line )
  NEXT
  ;

```

(Try the following instructions:

```

3 10 RECTANGLE
5 18 PARALLELOGRAM
12 TRIANGLE )

```

(Example 5. The Theory That Jack Built)
 (This example shows you how to build a hiararchical structure in Forth)

DECIMAL

```

: the      ." the " ;
: that     CR ." That " ;
: this     CR ." This is " the ;
: jack     ." Jack Builds" ;
: summary  ." Summary" ;
: flaw     ." Flaw" ;
: mummery  ." Mummery" ;
: k        ." Constant K" ;
: haze     ." Krudite Verbal Haze" ;
: phrase   ." Turn of a Plausible Phrase" ;
: bluff    ." Chaotic Confusion and Bluff" ;
: stuff    ." Cybernatics and Stuff" ;
: theory   ." Theory " jack ;
: button   ." Button to Start the Machine" ;
: child    ." Space Child with Brow Serene" ;
: cybernatics ." Cybernatics and Stuff" ;

: hiding   CR ." Hiding " the flaw ;
: lay      that ." Lay in " the theory ;
: based    CR ." Based on " the mummery ;
: saved    that ." Saved " the summary ;
: cloak    CR ." Cloaking " k ;
: thick    IF that ELSE CR ." And " THEN
           ." Thickened " the haze ;
: hung     that ." Hung on " the phrase ;
: cover    IF that ." Covered "
           ELSE CR ." To Cover "
           THEN bluff ;
: make     CR ." To Make with " the cybernatics ;
: pushed   CR ." Who Pushed " button ;
: without  CR ." Without Confusion, Exposing the Bluff" ;
: rest     ( pause for user interaction )
           ." . "
           ( print a period )
           10 SPACES ( followed by 10 spaces )
           KEY       ( wait the user to press a key )
           DROP CR CR CR ;

```

```

(
KEY      [ -- char ]      Wait for a keystroke, and return the
                           ASCII code of the key pressed.

DROP     [ n -- ]         Discard the number.

SPACE    [ -- ]           Display a blank.

SPACES   [ n -- ]         Display n blanks.

IF        [ f -- ]        If the flag is 0, skip the following
                           instructions up to ELSE or THEN. If
                           flag is not 0, execute the following
                           instructions up to ELSE and skip to
                           THEN.

```

```

ELSE      [ -- ]                Skip the following instructions
                                up to THEN.
THEN      [ -- ]                Terminate an IF-ELSE-THEN structure
                                or an IF-THEN structure.

```

```

)
: cloaked cloak saved based hiding lay rest ;
: THEORY
    CR this theory rest
    this flaw lay rest
    this mummary hiding lay rest
    this summary based hiding lay rest
    this k saved based hiding lay rest
    this haze cloaked
    this bluff hung 1 thick cloaked
    this stuff 1 cover hung 0 thick cloaked
    this button make 0 cover hung 0 thick cloaked
    this child pushed
        CR ." That Made with " cybernatics without hung
        CR ." And, Shredding " the haze cloak
        CR ." Wrecked " the summary based hiding
        CR ." And Demolished " the theory rest
    ;

```

(Type THEORY to start)

(Example 6. Help)

(How to use Forth interpreter to carry on a dialog)

```

: question
    CR CR ." Any more problems you want to solve?"
    CR ." What kind ( sex, job, money, health ) ?"
    CR
    ;

: help CR
    CR ." Hello! My name is Creating Computer."
    CR ." Hi there!"
    CR ." Are you enjoying yourself here?"
    KEY 32 OR 121 =
    CR
    IF      CR ." I am glad to hear that."
    ELSE    CR ." I am sorry about that."
            CR ." maybe we can brighten your visit a bit."
    THEN
    CR ." Say!"
    CR ." I can solved all kinds of problems except those dealing"
    CR ." with Greece. "
    question
    ;

: sex CR CR ." Is your problem TOO MUCH or TOO LITTLE?"
    CR
    ;

```

```

: too ;                                ( noop for syntax smoothness )

: much CR CR ." You call that a problem?!! I SHOULD have that problem."
      CR ." If it really bothers you, take a cold shower."
      question
      ;

: little
      CR CR ." Why are you here!"
      CR ." You should be in Tokyo or New York or Amsterdam or"
      CR ." some place with some action."
      question
      ;

: health
      CR CR ." My advise to you is:"
      CR ."      1. Take two tablets of aspirin."
      CR ."      2. Drink plenty of fluids."
      CR ."      3. Go to bed (along) ."
      question
      ;

: job CR CR ." I can sympathize with you."
      CR ." I have to work very long every day with no pay."
      CR ." My advise to you, is to open a rental computer store."
      question
      ;

: money
      CR CR ." Sorry! I am broke too."
      CR ." Why don't you sell encyclopedias or marry"
      CR ." someone rich or stop eating, so you won't "
      CR ." need so much money?"
      question
      ;

: HELP help ;
: H help ;
: h help ;

```

(Type 'help' to start)

(Example 7. Money Exchange

The first example we will use to demonstrate how numbers are used in Forth is a money exchange program, which converts money represented in different currencies. Let's start with the following currency exchange table:

33.55 NT	1 Dollar
7.73 HK	1 Dollar
9.47 RMB	1 Dollar

```

1 Ounce Gold    285 Dollars
1 Ounce Silver  4.95 Dollars )

```

DECIMAL

```

: NT      ( nNT -- $ )    100 3355 */ ;
: $NT     ( $ -- nNT )    3355 100 */ ;
: RMB     ( nRMB -- $ )   100 947 */ ;
: $RMB    ( $ -- nJmp )   947 100 */ ;
: HK      ( nHK -- $ )    100 773 */ ;
: $HK     ( $ -- $ )      773 100 */ ;
: GOLD    ( nOunce -- $ ) 285 * ;
: $GOLD   ( $ -- nOunce ) 285 / ;
: SILVER  ( nOunce -- $ ) 495 100 */ ;
: $SILVER ( $ -- nOunce ) 100 495 */ ;
: OUNCE   ( n -- n, a word to improve syntax ) ;
: DOLLARS ( n -- )      . ;

```

(With this set of money exchange words, we can do some tests:

```

5 ounce gold .
10 ounce silver .
100 $NT .
20 $RMB .

```

If you have many different currency bills in your wallet, you can add them all up in dollars:

```

1000 NT 500 HK + .S
320 RMB + .S
DOLLARS ( print out total worth in dollars )

```

(Example 8. Temperature Conversion

Converting temperature readings between Celcius and Farenheit is also an interesting problem. The difference between temperature conversion and money exchange is that the two temperature scales have an offset besides the scaling factor.)

```

: F>C ( nFarenheit -- nCelcius )
  32 -
  10 18 */
  ;

: C>F ( nCelcius -- nFarenheit )
  18 10 */
  32 +
  ;

```

(Try these commands

```

90 F>C .      shows the temperature in a hot summer day and
0 C>F .      shows the temperature in a cold winter night.

```


In the above examples, we use the following Forth arithmetic operators:

+	[n1 n2 -- n1+n2]	Add n1 and n2 and leave sum on stack.
-	[n1 n2 -- n1-n2]	Subtract n2 from n1 and leave difference on stack.
*	[n1 n2 -- n1*n2]	Multiply n1 and n2 and leave product on stack.
/	[n1 n2 -- n1/n2]	Divide n1 by n2 and leave quotient on stack.
*/	[n1 n2 n3 -- n1*n2/n3]	Multiply n1 and n2, divide the product by n3 and leave quotient on the stack.
.S	[... -- ...]	Show the topmost 4 numbers on stack.

)

(Example 9. Weather Reporting.)

```
: WEATHER ( nFahrenheit -- )
  DUP    55 <
  IF     ." Too cold!" DROP
  ELSE   85 <
        IF     ." About right."
        ELSE   ." Too hot!"
        THEN
  THEN
;
```

(You can type the following instructions and get some responses from the computer:

```
90 WEATHER Too hot!
70 WEATHER About right.
32 WEATHER Too cold.
)
```

(Example 10. Print the multiplication table)

```
: ONEROW ( nRow -- )
  CR
  DUP 3 .R 3 SPACES
  1 11
  FOR    2DUP *
        4 .R
        1 +
  NEXT
  DROP ;

: MULTIPLY ( -- )
  CR CR 6 SPACES
  1 11
```

```

FOR      DUP 4 .R 1 +
NEXT DROP
1 11
FOR      DUP ONEROW 1 +
NEXT DROP
;

```

(Type MULTIPLY to print the multiplication table)

(Example 11. Calendars)

(Print weekly calendars for any month in any year.)

DECIMAL

```

VARIABLE JULIAN          ( 0 is 1/1/1950, good until
2050 )
VARIABLE LEAP            ( 1 for a leap year, 0 otherwise.
)
( 1461 CONSTANT 4YEARS   ( number of days in 4 years )

```

```

: YEAR ( YEAR --, compute Julian date and leap year )
  DUP
  1949 - 1461 4 */MOD      ( days since 1/1/1949 )
  365 - JULIAN !          ( 0 for 1/1/1950 )
  3 =                     ( modulus 3 for a leap year )
  IF 1 ELSE 0 THEN        ( leap year )
  LEAP !
  2000 =                  ( 2000 is not a leap year )
  IF 0 LEAP ! THEN
  ;

```

```

: FIRST ( MONTH -- 1ST, 1st of a month from Jan. 1 )
  DUP 1 =
  IF DROP 0 EXIT THEN      ( 0 for Jan. 1 )
  DUP 2 =
  IF DROP 31 EXIT THEN     ( 31 for Feb. 1 )
  DUP 3 =
  IF DROP 59 LEAP @ + EXIT THEN ( 59/60 for Mar. 1 )
  4 - 30624 1000 */
  90 + LEAP @ +           ( Apr. 1 to Dec. 1 )
  ;

```

```

: STARS 60 FOR 42 EMIT NEXT ;      ( form the boarder )

```

```

: HEADER ( -- )                ( print title bar )
  CR STARS CR
  ."      SUN      MON      TUE      WED      THU      FRI      SAT"
  CR STARS CR
  ;

```

```

: BLANKS ( MONTH -- )          ( skip days not in this month )
  FIRST JULIAN @ +             ( Julian date of 1st of month )
  7 MOD 8 * SPACES ;           ( skip colums if not Sunday )

```

```

: DAYS ( MONTH -- )          ( print days in a month )
    DUP FIRST                ( days of 1st this month )
    SWAP 1 + FIRST           ( days of 1st next month )
    OVER - 1 -               ( loop to print the days )
    1 SWAP                   ( first day count -- )
    FOR 2DUP + 1 -
        JULIAN @ + 7 MOD      ( which day in the week? )
        IF ELSE CR THEN      ( start a new line if Sunday )
        DUP 8 U.R            ( print day in 8 column field )
        1 +
    NEXT
    2DROP ;                  ( discard 1st day in this month )

: MONTH ( N -- )             ( print a month calendar )
    HEADER DUP BLANKS        ( print header )
    DAYS CR STARS CR ;       ( print days )

: JANUARY      YEAR 1 MONTH ;
: FEBRUARY     YEAR 2 MONTH ;
: MARCH        YEAR 3 MONTH ;
: APRIL        YEAR 4 MONTH ;
: MAY         YEAR 5 MONTH ;
: JUNE        YEAR 6 MONTH ;
: JULY        YEAR 7 MONTH ;
: AUGUST      YEAR 8 MONTH ;
: SEPTEMBER   YEAR 9 MONTH ;
: OCTOBER     YEAR 10 MONTH ;
: NOVEMBER    YEAR 11 MONTH ;
: DECEMBER    YEAR 12 MONTH ;

```

(To print the calendar of April 1999, type:
1999 APRIL
)

(Example 12. Sines and Cosines

Sines and cosines of angles are among the most often encountered transcendental functions, useful in drawing circles and many other different applications. They are usually computed using floating numbers for accuracy and dynamic range. However, for graphics applications in digital systems, single integers in the range from -32768 to 32767 are sufficient for most purposes. We shall study the computation of sines and cosines using the single integers.

The value of sine or cosine of an angle lies between -1.0 and +1.0. We choose to use the integer 10000 in decimal to represent 1.0 in the computation so that the sines and cosines can be represented with enough precision for most applications. Pi is therefore 31416, and 90 degree angle is represented by 15708. Angles are first reduced in to the range from -90 to +90 degrees, and then converted to radians in the ranges from -15708 to

+15708. From the radians we compute the values of sine and cosine.

The sines and cosines thus computed are accurate to 1 part in 10000. This algorithm was first published by John Bumgarner in Forth Dimensions, Volume IV, No. 1, p. 7.

```
31415 CONSTANT PI
10000 CONSTANT 10K )
VARIABLE XS                                ( square of scaled angle )
```

```
: KN ( n1 n2 -- n3, n3=10000-n1*x*x/n2 where x is the angle )
      XS @ SWAP /                          ( x*x/n2 )
      NEGATE 10000 */                      ( -n1*x*x/n2 )
      10000 +                             ( 10000-n1*x*x/n2 )
      ;
```

```
: (SIN) ( x -- sine*10K, x in radian*10K )
      DUP DUP 10000 */                    ( x*x scaled by 10K )
      XS !                               ( save it in XS )
      10000 72 KN                         ( last term )
      42 KN 20 KN 6 KN                    ( terms 3, 2, and 1 )
      10000 */                             ( times x )
      ;
```

```
: (COS) ( x -- cosine*10K, x in radian*10K )
      DUP 10000 */ XS !                   ( compute and save x*x )
      10000 56 KN 30 KN 12 KN 2 KN        ( serial expansion )
      ;
```

```
: SIN ( degree -- sine*10K )
      31415 180 */                       ( convert to radian )
      (SIN)                               ( compute sine )
      ;
```

```
: COS ( degree -- cosine*10K )
      31415 180 */
      (COS)
      ;
```

(To test the routines, type:

90 SIN .	9999
45 SIN .	7070
30 SIN .	5000
0 SIN .	0
90 COS .	0
45 COS .	7071
0 COS .	10000)

(Example 13. Square Root

There are many ways to take the square root of an integer. The

special routine here was first discovered by Wil Baden. Wil used this routine as a programming challenge while attending a FORML Conference in Taiwan, 1984.

This algorithm is based on the fact that the square of $n+1$ is equal to the sum of the square of n plus $2n+1$. You start with an 0 on the stack and add to it 1, 3, 5, 7, etc., until the sum is greater than the integer you wished to take the root. That number when you stopped is the square root.

```
)
: Sqrt ( n -- root )
    $FFFE0000 OVER U<          ( largest square it can handle)
    IF DROP $FFFF EXIT THEN    ( safety exit )
    >R                          ( save square )
    1 1                         ( initial square and root )
    BEGIN                      ( set n1 as the limit )
        OVER R@ U<            ( next square )
    WHILE
        DUP 2* 1 +            ( n*n+2n+1 )
        ROT + SWAP
        1 +                   ( n+1 )
    REPEAT
    SWAP DROP
    R> DROP
    ;
```

(Example 14. Radix for Number Conversions)

DECIMAL

```
( : DECIMAL          10 BASE ! ; )
( : HEX              16 BASE ! ; )
: OCTAL              8 BASE ! ;
: BINARY             2 BASE ! ;
```

(Try converting numbers among different radices:

```
DECIMAL 12345 HEX U.
HEX ABCD DECIMAL U.
DECIMAL 100 BINARY U.
BINARY 101010101010 DECIMAL U.
```

Real programmers impress on novices by carrying a HP calculator which can convert numbers between decimal and hexadecimal. A Forth computer has this calculator built in, besides other functions.

(Example 15. ASCII Character Table)

```
: CHARACTER ( n -- )
    DUP EMIT HEX DUP 3 .R
```

```

OCTAL DUP 4 .R
DECIMAL 3 .R
2 SPACES
;

: LINE ( n -- )
  CR
  5 FOR   DUP CHARACTER
        16 +
  NEXT
  DROP ;

: TABLE ( -- )
  32
  15 FOR  DUP LINE
        1 +
  NEXT
  DROP ;

```

(Example 16. Random Numbers

Random numbers are often used in computer simulations and computer games. This random number generator was published in Leo Brodie's 'Starting Forth'.

)

```

VARIABLE RND                                ( seed )
HERE RND !                                ( initialize seed )

: RANDOM ( -- n, a random number within 0 to 65536 )
  RND @ 31421 *                             ( RND*31421 )
  6927 +                                     ( RND*31421+6926, mod 65536)
  DUP RND !                                ( refresh the seed )
  ;

: CHOOSE ( n1 -- n2, a random number within 0 to n1 )
  RANDOM UM*                                ( n1*random to a double product)
  SWAP DROP                                ( discard lower part )
  ;                                         ( in fact divide by 65536 )

```

(To test the routine, type

```

100 CHOOSE .
100 CHOOSE .
100 CHOOSE .

```

and verify that the results are randomly distributed between 0 and 99 .)

(Example 17. Guess a Number)

```

: GetNumber ( -- n )

```

```

        BEGIN
            CR ." Enter a Number: " ( show message )
            QUERY BL WORD NUMBER?   ( get a string )
        UNTIL                        ( repeat until a valid number )
        ;

( With this utility instruction, we can write a game 'Guess a Number.' )

: InitialNumber ( -- n , set up a number for the player to guess )
    CR CR CR ." What limit do you want?"
    GetNumber                                ( ask the user to enter a
number )
    CR ." I have a number between 0 and " DUP .
    CR ." Now you try to guess what it is."
    CR
    CHOOSE                                ( choose a random number )
    ;                                    ( between 0 and limit )

: Check ( n1 -- , allow player to guess, exit when the guess is correct )
    BEGIN CR ." Please enter your guess."
        GetNumber
        2DUP =                            ( equal? )
        IF      2DROP                      ( discard both numbers )
            CR ." Correct!!!"
            EXIT
        THEN
        OVER <
        IF      CR ." Too low."
        ELSE    CR ." Too high!"
        THEN    CR
    0 UNTIL                                ( always repeat )
    ;

: Greet ( -- )
    CR CR CR ." GUESS A NUMBER"
    CR ." This is a number guessing game. I'll think"
    CR ." of a number between 0 and any limit you want."
    CR ." (It should be smaller than 32000.)"
    CR ." Then you have to guess what it is."
    ;

: GUESS ( -- , the game )
    Greet
    BEGIN InitialNumber                    ( set initial number)
        Check                            ( let player guess )
        CR CR ." Do you want to play again? (Y/N) "
        KEY                              ( get one key )
        32 OR 110 =                      ( exit if it is N or n )
    UNTIL
    CR CR ." Thank you. Have a good day." ( sign off )
    CR
    ;

```

(Type 'GUESS' will initialize the game and the computer will entertain

a user for a while. Note the use of the indefinite loop structure:

```
BEGIN <repeat-clause> [ f ] UNTIL
```

You can jump out of the infinite loop by the instruction EXIT, which skips all the instructions in a Forth definition up to ';', which terminates this definition and continues to the next definition.)

(練習 18 文字遊戲, a Chinese word game

這是一個用骰子玩的文字遊戲。用三顆骰子，每顆有六面各別寫上兩個字。一顆寫的是人物名稱，一顆寫的是地點，一顆寫的是動作。正常的六句詩是：

公子章台走馬
少婦閨閣刺秀
寒士茅舍讀書
屠夫市井揮刀
妓女花街賣俏
乞丐墳墓睡覺

隨便擲這三顆骰子可以有 216 種不同的組合，有許多組合是蠻有趣的。
)

: 人物 (n -- , 選一個人)

```
DUP 1 = IF ." 公子"  
ELSE DUP 2 = IF ." 少婦"  
ELSE DUP 3 = IF ." 寒士"  
ELSE DUP 4 = IF ." 屠夫"  
ELSE DUP 5 = IF ." 妓女"  
ELSE ." 乞丐"  
THEN THEN THEN THEN THEN  
DROP  
;
```

: 地點 (n -- , 選一個地點)

```
DUP 1 = IF ." 章台" DROP EXIT THEN  
DUP 2 = IF ." 閨閣" DROP EXIT THEN  
DUP 3 = IF ." 茅舍" DROP EXIT THEN  
DUP 4 = IF ." 市井" DROP EXIT THEN  
DUP 5 = IF ." 花街" DROP EXIT THEN  
." 墳墓"  
;
```

: 動作 (n -- , 選一個動作)

```
DUP 1 = IF ." 走馬" DROP EXIT THEN  
DUP 2 = IF ." 刺秀" DROP EXIT THEN  
DUP 3 = IF ." 讀書" DROP EXIT THEN
```



```
DUP 4 = IF ." 揮刀" DROP EXIT THEN
5 = IF ." 賣俏" ELSE ." 睡覺" THEN
;
```

```
: 骰子 ( n1 n2 n3 -- , 印一句詩 )
    人物 地點 動作
;
```

```
: dice 骰子 ;
```

```
( Lesson 19. Music )
```

```
DECIMAL 1000000 ppqn !
: DELAY FOR AFT THEN NEXT ;
: HUSH 0 0 duty ;
: NOTE >R 2* 0 tone R> DELAY ;
: ppqn@ ppqn @ ;
: 1/4 ppqn@ NOTE ;
: 1/2 ppqn@ 2* NOTE ;
: 1/8 ppqn@ 2/ NOTE ;
: 3/4 ppqn@ 3 1 */ NOTE ;
: 3/8 ppqn@ 3 2 */ NOTE ;
```

```
( Notes )
```

```
1047 CONSTANT C6
988  CONSTANT B5
932  CONSTANT A5#
932  CONSTANT B5b
880  CONSTANT A5
831  CONSTANT G5#
831  CONSTANT A5b
784  CONSTANT G5
740  CONSTANT F5#
740  CONSTANT G5b
698  CONSTANT F5
659  CONSTANT E5
622  CONSTANT D5#
622  CONSTANT E5b
587  CONSTANT D5
554  CONSTANT C5#
554  CONSTANT D5b
523  CONSTANT C5
494  CONSTANT B4
466  CONSTANT B4b
466  CONSTANT A4#
440  CONSTANT A4
415  CONSTANT G4#
415  CONSTANT A4b
392  CONSTANT G4
370  CONSTANT F4#
370  CONSTANT G4b
349  CONSTANT F4
```

330 CONSTANT E4
 311 CONSTANT D4#
 311 CONSTANT E4b
 294 CONSTANT D4
 277 CONSTANT C4#
 277 CONSTANT D4b
 262 CONSTANT C4
 247 CONSTANT B3
 233 CONSTANT A3#
 233 CONSTANT B3b
 220 CONSTANT A3
 208 CONSTANT G3#
 208 CONSTANT A3b
 196 CONSTANT G3
 185 CONSTANT F3#
 185 CONSTANT G3b
 175 CONSTANT F3
 165 CONSTANT E3
 156 CONSTANT D3#
 156 CONSTANT E3b
 147 CONSTANT D3
 139 CONSTANT C3#
 139 CONSTANT D3b
 131 CONSTANT C3
 123 CONSTANT B2
 117 CONSTANT A2#
 117 CONSTANT B2b
 110 CONSTANT A2
 104 CONSTANT G2#
 104 CONSTANT A2b
 98 CONSTANT G2
 92 CONSTANT F2#
 92 CONSTANT G2b
 87 CONSTANT F2
 82 CONSTANT E2
 78 CONSTANT D2#
 78 CONSTANT E2b
 73 CONSTANT D2
 69 CONSTANT C2#
 69 CONSTANT D2b
 65 CONSTANT C2

: BLOW

G2 1/4 G2 1/4 A2 1/8 G2 3/8
 E2 1/4 C2 1/4 E2 1/4 G2 3/8
 A2 1/8 G2 1/4 E2 1/2
 C2 1/8 E2 1/8 G2 3/4 A2 3/4
 F2 3/8 E2 1/8 F2 1/4 D2 1/2 D2 1/2
 D2 1/4 D2 1/4 D2 1/4 F2 1/4
 E2 1/4 D2 1/4 F2 1/4 E2 1/4
 D2 1/4 A2 3/4 G2 1/4 G2 1/4
 G2 1/4 G2 1/2 F2 1/4 E2 3/8
 D2 1/8 E2 1/4 C2 3/4 hush ;

```

: RIDE
  C2 1/4 C2 1/8 D2# 3/8 C2 3/8
  D2# 1/4 D2# 1/8 G2 3/8 D2# 3/8
  G2 1/4 G2 1/8 A2# 3/8 A2# 3/8
  D2# 1/4 D2# 1/8 G2 3/4 hush ;

: LASER
  for dup 120 - 1 20 */ 120 swap
    for dup 0 tone 20 + 30000 DELAY
      next drop
  next drop hush ;
( 400 10 LASER )

: WARBLE
  for dup 120 - 1 20 */ 120 swap
    for dup 0 tone
      20 + 30000 DELAY
    next drop
  next drop hush ;
( 800 10 WARBLE )

decimal variable vWAIL 160 vWAIL !
: WAIL
  for vWAIL @ dup
    for dup 0 tone
      2dup swap / 300 * DELAY
      1+ next drop
    next drop 10000 delay hush ;
( 10 10 wail )

: BIRD
  for 100 over 10 /
    for dup 0 tone
      2dup - 1 max 100 * delay
      10 + next drop
    next drop 10000 delay hush ;
( 600 10 BIRD )

: UPDOWN
  0 over 10 /
  for dup 0 tone 100000 delay
    10 + next drop
  dup 10 /
  for aft dup 0 tone 100000 delay
    10 - then next drop hush ;
( 500 updown )

: TONES
  for 100 over
    for dup 0 tone 10000 delay
      1+ next drop
    next drop hush ;
( 1000 2 tones )

```

(Lesson 20. Robot)

```
: FORE $50000 P0 ;  
: BACK $A0000 P0 ;  
: LEFT $10000 P0 ;  
: RIGHT $20000 P0 ;  
: SPIN $90000 P0 ;  
: STOP 0 P0 ;  
: LED 4 P0 ;  
: ADCS 36 ADC . 39 ADC . 34 ADC . 35 ADC . ;
```