

ceForth_328

Chen-Hanson Ting
Offete Enterprises
www.offete.com
November 2011

ceForth_328.pde is a free copy of ceForth_328 system for you to try on your Arduino Uno, or other compatible, board. For the source code of its metacompiler and complete documentation, purchase the ceForth_328 system from www.offete.com, Item# 2162, for \$25. The ceForth_328 system is in the public domain, and you can use it without restriction.

1. Introduction

Since 1990, I have been promoting a simple FORTH language model called eForth. This model consists of a kernel of 30 primitive FORTH commands which have to be implemented in machine instructions of a host microcontroller, and 190 compound FORTH commands constructed from the primitive commands and other compound commands. By isolating machine dependent commands from machine independent commands, the eForth model can be ported to any microcontroller very easily.

This FORTH system, ceForth_328 is derived from the cEF Version 1.0 system I wrote in C, which follows closely the original eForth model, with only 33 primitive. cEF 1.0 system was compiled by gcc in the cygwin environment. ceForth_328 is an eForth implementation for the ATmega328P microcontroller from Atmel as a sketch on the Arduino 0022 system. It can be compiled and uploaded to the Arduino Uno to give you a taste of FORTH. Because the limitations imposed by Arduino 0022, you can add only 1.5 KB of new commands to the RAM memory. It sounds like a severe limitation. However, because of the compactness of FORTH commands, you can still compile substantial applications into the small RAM memory. Another serious limitation is that you cannot save the application in the flash memory because Arduino 0022 system does not provide tools to write new code to the flash memory at run time.

If you really need to develop large applications and to have the complete control over the underlying microcontroller, you can use the native FORTH system I built for ATmega328P, the 328eForth system. In the 328eForth system, new FORTH commands are compiled directly into the flash memory, and you can make the full use of the 32 KB of flash memory, as well as the 2 KB RAM memory. The drawback of the 328eForth system is that you need a separated programming device, like AVRISP mkII, to upload it into the flash memory. In essence, 328eForth is not compatible with Arduino 0022. This is the reason why I developed this ceForth_328 system, which is basically a teaser introducing you to FORTH, and perhaps to the real FORTH implementation of 328eForth.

2. What Good is ceForth_328?

With the limitations I talked about above, you may ask: "Why should I bother with ceForth_328?"

Well, I have seen lots of discussions in the Arduino community on the Internet that people missed the PEEK and POKE functions in the BASIC language that were very popular in the early microprocessor days. PEEK allows you to examine the contents in a memory location, and POKE allows you to change them. They are very useful in debugging an application, especially if the IO registers are mapped into the memory space.

I give you PEEK and POKE in ceForth_328. Here are a few examples to show you what you can do with PEEK and POKE.

I assume that you have your Arduino Board set up and connected to your PC through the USB cable. I use an Arduino Uno. First bring up Arduino 0022, and click File/Open button, then select ceForth_328.pde file, wherever you last left it. Compile and upload it to Arduino Uno. The Arduino window looks like this:

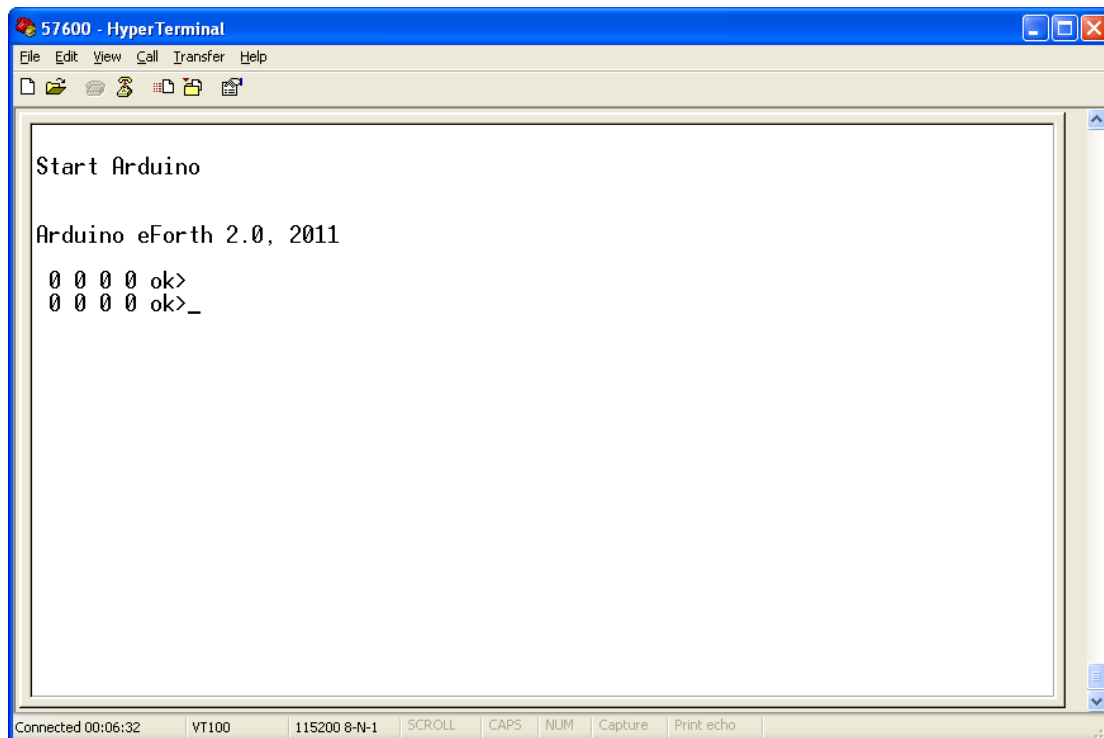


```
Arduino IDE - eforth_328 | Arduino 0022
File Edit Sketch Tools Help
eforth_328.pde
eforth_328 $
/* Arduino eForth_328, 2.0: For Atmega328 on arduino Uno
*****
/* Chen-Hanson Ting
/* eForth_328, 2.0 07nov11c
/* Compile new commands to RAM
/* eForth_328, 1.0 21sep11cht
/* Adopted from eForth_11.c
/* Compiled by Arduino as a sketch
/* Follow closely the original eForth model
/* Kernel has 32 primitives
/* code[] array must be filled with rom.mif produced by ceMETA328
/* @, !, C@ and C! access RAM memory
*****
#include <avr/pgmspace.h>

#define LOGICAL ? 0xFFFF: 0
#define LOWER(x,y) ((unsigned int)(x)<(unsigned int)(y))
#define pop top = *S--

Done uploading.
Binary sketch size: 13056 bytes (of a 32256 byte maximum)
6
```

Open HyperTerminal, and configure it to 115,200 baud, 1 start bit, 8 data bits, 1 stop bit, no parity, and no flow control. You will see the following HyperTerminal console. If not, check to see if you have the right COM port settings. You can use other terminal emulator programs, and I assume that they behave similarly.



Press the Return key a couple of times, and the `ok>` messages echo on the console. Type in the following commands to exercise the `ceForth_328` system:

```
WORDS
1 2 3 4
+
*
: TEST1 CR ." HELLO, WORLD!" ;
TEST1
: TEST2 IF 1 ELSE 2 THEN . ;
1 TEST2
0 TEST2
: TEST3 10 FOR R@ . NEXT ;
TEST3
```

`ceForth_328` is case insensitive. You can type commands in either upper or lower case. Note that `ceForth_328` is in the hexadecimal base when it starts.

When you first get an Arduino Board, the first thing you do is turning that on-board LED on and off. The LED is connected to the Digital IO Line D13. With the following POKE commands, you can turn the LED on and off:

```
20 24 POKE
20 23 POKE
20 23 POKE
```

(After POKE, press Return key to send one line of commands to ATmega328P to be executed.)

The Digital I/O Line D13 on Arduino Uno is connected to bit-5 of GPIO Port B, PB-5. Port B is a general purpose I/O device which has the following registers:

Address	Register	Name	Function
\$23	PINB	Input register	Status of input pins
\$24	DDRB	Direction register	1: output; 0: input
\$25	PORTB	Data register	Output data, pull-up resistor

Setting a bit in DDRB register makes the corresponding pin an output pin. This is what the commands

```
20 24 POKE
```

do. Then, writing this bit in PINB register toggles the output pin. This is done by the commands

```
20 23 POKE
```

The command POKE takes two arguments in front of it: the first argument is one byte of data, and the second argument is the address of memory location where the byte data is deposited. Alternatively, you can POKE the PORTB register to set or clear the bit at PB-5, and respectively turn the LED on or off:

```
20 25 POKE
```

```
0 25 POKE
```

See? When you can poke the IO registers, you can control the ATmega328P chip directly, without writing a sketch.

Type the above commands. Press Enter key at the end of each line. You will see how ceForth_328 responds to your commands as show in the following console display:

```

57600 - HyperTerminal
File Edit View Call Transfer Help
Start Arduino
Arduino eForth 2.0, 2011
0 0 0 0 ok>
0 0 0 0 ok>20 24 POKE
0 0 0 0 ok>20 23 POKE
0 0 0 0 ok>20 23 POKE
0 0 0 0 ok>
0 0 0 0 ok>
0 0 0 0 ok>20 25 POKE
0 0 0 0 ok>0 25 POKE
0 0 0 0 ok>
0 0 0 0 ok>
0 0 0 0 ok>23 PEEK
0 0 0 10 ok>24 PEEK
0 0 10 20 ok>25 PEEK
0 10 20 0 ok>
0 10 20 0 ok>
0 10 20 0 ok>
0 10 20 0 ok>
0 10 20 0 ok>
0 10 20 0 ok>
0 10 20 0 ok>

```

The zero's to the left of the prompt ok> at the beginning of this session show the top 4 elements on the parameter stack in the FORTH Virtual Machine. After you

execute the commands

```
23 PEEK
```

the value returned from register 23 is pushed on the parameter stack as the number 10. Subsequently, the value returned by commands

```
24 PEEK
```

is 20, and that returned by

```
25 PEEK
```

is 0. You can use PEEK to see the contents of any memory location, including all the CPU registers, all the IO registers, all the RAM memory, and all the flash memory. You can use POKE to change the contents of registers and RAM memory. You cannot change the contents of flash memory, though.

Type in the above command lines to verify that you get the same results as show in the above console display.

Just these two commands PEEK and POKE, make it worthwhile for you to look at ceForth_328 seriously.

Now. Heed this warning!

Use the POKE command carefully. There is no protection against your poking into sensitive locations which might cause trouble. You have to stay away, absolutely and positively, from locations 0 to \$1F (decimal 0-31), because they map directly into the CPU registers. Only God knows what data are stored there, and they change dynamically. You are encouraged to poke into the IO registers from \$20 to \$FF, but you have to study carefully the AVR Family Data Book so that you know exactly what the consequences are, before you try it. If you do the poking correctly, you can make all the IO devices to do what you want. However, incorrectly poking the IO register may have no ill consequences, or may crash the system at the worst.

The C compiler uses RAM locations from \$100 to \$2FF. ceForth_328 uses the RAM locations from \$300 to \$8FF. If you did not compile new commands into the RAM memory, locations from \$380 to \$87F are free, and you can poke these locations without any problem.

PEEK allows you to examine memory contents one byte at a time. I give you a much more powerful command DUMP to display 256 bytes of contiguous memory locations. DUMP takes one argument as an address, and displays the contents of the next 256 bytes in a nicely formatted table. For example, the commands

```
0 DUMP
```

displays the contents of all the CPU registers and all the IO registers, as show in the following:

```

57600 - HyperTerminal
File Edit View Call Transfer Help
0 10 20 0 ok>
0 10 20 0 ok>
0 10 20 0 ok>
0 10 20 0 ok>
0 10 20 0 ok>
0 10 20 0 ok>
0 10 20 0 ok>0 DUMP
 0 43 0 F7 FF A4 B7 BA F9 FF 6F FF FA FD 1 5 11 C_w_$7:y_o_z}__
10 40 F 4D 73 0 0 30 0 9 0 1A 0 68 20 43 13 @_yP_____h C_
20 62 62 62 10 20 0 0 0 0 3 0 0 35 62 62 62 bbb_____bbbb
30 62 62 62 62 62 6 7 7 62 62 23 0 0 0 0 0 b!b!_____b
40 0 FF 3 0 3 3 96 0 0 62 0 0 0 0 0 62 _____b____b
50 30 FF 62 0 0 0 62 0 62 62 62 0 62 F6 8 B5 0_b____b_bbb_bv_5
60 0 0 62 62 0 62 A6 62 0 0 14 0 0 0 1 0 _____4b_b&b____
70 0 20 62 21 62 62 62 14 0 0 87 0 0 62 0 0 _____b5b_b#b____b
80 1 3 0 62 BE 0 0 0 0 0 0 0 20 0 22 21 _____k_____b!b_
90 21 22 62 62 62 0 23 2 20 21 23 62 62 62 21 62 bbbbbbbbbbbb b_b
A0 62 62 62 62 62 62 62 62 62 62 0 21 21 62 62 21 bbbbbbbbbbb!b b#b
B0 1 4 1B 0 0 21 0 22 0 F8 FE FF 0 0 62 62 _____2_b_b_x____bb
C0 62 98 6 62 10 0 50 62 62 62 62 62 62 62 62 b__b__Pbbbbbbbbbb
D0 62 62 21 62 62 62 34 14 62 62 15 62 62 62 20 62 bbbbbbbbbbbb!b4b
E0 62 62 62 62 62 62 62 14 62 62 62 21 62 62 62 35 bbbbbbbbbbbbbbbb
F0 14 62 62 62 21 62 62 62 21 62 62 62 22 62 62 bbbbbbbbbbbbbbbb
0 10 20 0 ok>
0 10 20 0 ok>

```

Connected 04:32:41 VT100 115200 8-N-1 SCROLL CAPS NUM Capture Print echo

Here, you see the contents of the CPU registers from location 0 to \$1F, and those of the IO registers from \$20 to \$FF. The locations showing data of \$62 are those not implemented as IO registers. The locations showing data other than \$62 are generally valid registers. Poke them carefully after you study their functions in the AVR Family Data Book. ceForth_328 is the best companion of the AVR Family Data Book as you read about the ATmega328 microcontroller.

Another example is the commands
 900 DUMP
 and the results are shown in the following:

```

E0 62 62 62 62 62 62 62 14 62 62 62 21 62 62 62 35 bbbbbbbbbbbbbbbb
F0 14 62 62 62 21 62 62 62 21 62 62 62 22 62 62 62 bbbbbbbbbbbbbbbb
0 10 20 0 ok>
0 10 20 0 ok>
0 10 20 0 ok>
0 10 20 0 ok>900 DUMP
900 28 1B 0 0 0 0 0 0 0 80 8 10 0 54 16 ( _____ T_
910 0 0 E8 1C 20 3 E8 1C 2 17 0 0 0 0 0 0 _h_h_
920 0 0 3 74 6D 70 4 16 1E 3 22 9 3 48 4C 44 _tmp_ " HLD
930 4 16 2 3 2C 9 4 53 50 41 4E 0 4 16 4 3 _____ SPAN
940 36 9 3 3E 49 4E 4 16 6 3 42 9 4 23 54 49 6 >IN B #TI
950 42 0 4 16 8 3 4C 9 4 27 54 49 42 0 4 16 B L 'TIB
960 A 3 58 9 4 42 41 53 45 0 4 16 C 3 64 9 _X_BASE_d
970 5 27 45 56 41 4C 4 16 E 3 70 9 3 48 4C 44 _EVAL_p_HLD
980 4 16 10 3 7C 9 7 43 4F 4E 54 45 58 54 4 16 _CONTEXT
990 12 3 86 9 2 43 50 0 4 16 14 3 94 9 4 4C _____ CP
9A0 41 53 54 0 4 16 16 3 9E 9 6 27 41 42 4F 52 AST _ABOR
9B0 54 0 4 16 18 3 AA 9 2 44 50 0 4 16 1A 3 T * DP
9C0 B8 9 7 43 55 52 52 45 4E 54 4 16 1C 3 C2 9 8 CURRENT_B
9D0 3 42 59 45 1 0 D0 9 3 3F 52 58 2 16 D8 9 BYE_P ?RX_X
9E0 3 54 58 21 3 16 E0 9 3 21 49 4F 0 16 E8 9 TX! _ !IO_h
9F0 5 64 6F 4C 49 54 5 16 F0 9 4 45 58 49 54 0 _doLIT_p_EXIT
0 10 20 0 ok>
0 10 20 0 ok>_

```

It shows the first 256 bytes of the ceForth_328 dictionary with 19 complete records of FORTH commands. The data dumped in bytes may not make any sense to you at this point, but you should recognize the names of these commands in the ASCII dump on the right hand side of the display.

The dictionary covers the flash memory locations from \$900 to \$1C9B. You can POKE or DUMP this area at will. You cannot change the contents and POKE has no effect on them.

Do you like PEEK and POKE? As a matter of fact, PEEK and POKE are actually aliases of the commands C@ and C!, which are native FORTH command common to most FORTH systems. It is kind of cheating, but I hope you are a good sport.

3. Examples

ceForth_328 has about 1.5 KB of RAM memory free to compile new FORTH commands. It is not very big, but enough to compile substantial applications. Here I will show you a few examples to get you started. You can type these commands directly into the HylerTerminal Console. Or, you can type these commands into a text file through your favorite text editor, and download the file using the Transfer/Transfer-text-file tool in HyperTerminal.

3.1. Compiler Tests

When I implement a new FORTH system, there are a few new commands I use to test the system, and to verify that the compiler works correctly. Get the Auduino 0022 up and upload ceForth_328.pde. Then get the HyperTerminal up. Type in these commands as shown in the following:

```

57600 - Hyper Terminal
File Edit View Call Transfer Help
Start Arduino

Arduino eForth 2.0, 2011
( Tests of compiler 10nov11cht )
0 0 0 0 ok>: test1 1 2 3 4 5 ;
0 0 0 0 ok>: test2 if 1 else 2 then . ;
0 0 0 0 ok>: test3 10 for r@ . next ;
0 0 0 0 ok>: test4 cr ." Hello, world!" ;
0 0 0 0 ok>_

Connected 00:05:14 VT100 115200 8-N-1 SCROLL CAPS NUM Capture Print echo

```

Now, types these commands to test these commands:

```

test1
0 test2
1 test2
test3
test4

```

3.2. BLINK

Blink.pde is generally the first sketch people would try which they first get an Arduino Board. I showed you how to turn the D13 LED on and off in an earlier section. Here I will show you the FORTH program which blinks the LED. The commands are in the following box. Text in parentheses are comments and you can ignore them.

```

( Blink Line D13, 01nov11cht )
HEX
: MS ( n -- ) FOR AFT $40 FOR NEXT THEN NEXT ;
: BLINK 20 24 C! BEGIN 20 23 C! 400 MS ?KEY UNTIL ;

```

The MS command causes a delay. You give the number of milliseconds before MS. In the BLINK command, we first initialize the D13 line as an output port, and then fall into an infinite loop. In the loop, the LED is toggled, and there is a delay by the commands:

```
400 MS
```

400 in hexadecimal is 1024 in decimal. Therefore, the delay lasts about 1000 milliseconds. After that, ?KEY looks at the USART0 receiver. If there is no input character, the loop is repeated. If you hit any key on the keyboard, the loop will be terminated.

Type in the command `BLINK` with a Return, the D13 LED will blink. On for 1 second and off for 1 second, until you hit a key, and `ceForth_328` returns to the text interpreter, showing the `ok>` prompt.

3.3. TONE

This example allows you to generate a tone on the D6 digital output line. Why D6? Because D6 connects to one of the outputs from the Timer/Counter0 in ATmega328P. We will thus use Timer/Counter0 to produce a square wave on D6. If you connect one lead of a speaker or a buzzer to D6, and the other lead to the ground, you will hear a tone.

The commands to generate a tone are shown below. Backslashes are also comment commands, and you can ignore them and the texts following them.

```
( Tone generator, 09nov11cht)

HEX
: SETUP
  40 2A C! \ make OC0A (I/O Line 6, PD-6) an output pin
  42 44 C! \ toggle OC0A on compare match, select CTC mode
  FF 47 C! \ maximum count in OCR0A to compare
  3 45 C! \ select /64, prescaler=3, start counter
  ;
: PRESCALER ( 0-5 -- )
  45 C! ;
: TUNING ( c -- )
  47 c! ;
```

Then type in

```
SETUP
```

If you had a speaker connected to D6, you will hear a tone.

The Timer/counter0 has a prescaler which scales the master clock and uses the slowed oscillator to drive the counter. The command `PRESCALER` takes one argument from 0 to 5. Changing the prescaler, you will generate a different tone according to the following table:

Prescaler	Base Frequency
0	Stop oscillator
1	31.2 KHz
2	7.81 KHz
3	980 Hz
4	244 Hz
5	61 Hz

The command `TUNING` allows you to fine-tune the frequency of the tone more accurately. `TUNING` takes one argument from 0 to `FF`. A smaller argument

produces a higher pitch.

3.4. Servo Motors

ATmega328P has three timer/counters. Timer/Counter0 and Timer/Counter2 are 8 bit timer/counters, and Timer/Counter 1 is a 16-bit timer/counter. Timer/Counter1 is more complicated, naturally, but you can run it in the 8-bit mode, so that all three behave similarly. Each Timer/Counter has two outputs which can be programmed to generate two different PWM waves driving two servo motors. The commands are shown below:

```
( Servo Motors on Arduino Uno )
( Chen-Hanson Ting, 5/18/2011 )
( OC1A: $88, PB1, Pin 9 )
( OC1B: $8a, PB2, Pin 10 )
( OC2A: $b3, PB3, Pin 11 )
( OC2B: $b4, PD3, Pin 3 )
( OC0B: $47, PD5, Pin 5 )
( OC0A: $48, PD6, Pin 6 )
( Master clock 16 MHz, prescaler 1024 )
( 3 Counter/Timers, fast PWM mode, 8 bit counter )
( PWM wave frequency 60 Hz, period 16 ms )
( PWM control code: $10, 1 ms; $18, 1.5 ms; $20, 2 ms )

hex

: init-ports
  E 24 c! 68 2a c! \ output ports
  a3 44 c! 5 45 c! \ TCCR0A, TCCR0B
  18 47 C! 18 48 C! \ OCR0A, OCR0B
  a1 80 c! d 81 c! \ TCCR1A, TCCR1B
  18 88 c! 18 8a C! \ OCR1A, OCR1B
  a3 b0 c! 7 b1 c! \ TCCR2A, TCCR2B
  18 b3 C! 18 b4 C! \ OCR2A, OCR2B
  ;

: s1 ( n -- ) 88 c! ;
: s2 ( n -- ) 8a c! ;
: s3 ( n -- ) b3 c! ;
: s4 ( n -- ) b4 c! ;
: s5 ( n -- ) 47 c! ;
: s6 ( n -- ) 48 c! ;
```

The command `init-ports` is a bit complicated, and you have to read the three chapters in the AVR Family Data Book on Timer/Counter1, 2 and 3 to fully understand it. However, I just summarized the most important information on these timer/counters in the comment lines at the beginning of `Servo.txt` file shown above.

Six servo motors are connected to Digital lines D3, D5, D6, D9, D10, and D11. D3, D5, and D6 are driven by three lines in Port PD as PD3, PD5 and PD6, respectively.

D9, D10, and D11 are driven by three lines in Port PB as PB1, PB2 and PB3, respectively. The commands in init-ports

```
E 24 c! 68 2a c! \ output ports
assigned these 6 lines as output lines.
```

Relevant IO registers, their addresses, and their basic functions are summarized in the following table:

Register	Timer/Counter0	Timer/Counter1	Timer/Counter2	Function
TCCRnA	44	80	B0	Timer control register A
TCCRnB	45	81	B1	Timer control register B
OCRnA	47	88	B3	Output compare register A
OCRnB	48	8A	B4	Output compare register B

To drive a servo motor you give it a PWM wave at 50 Hz, with the turn-on period varying from 1 ms to 2 ms. This range is controlled by writing a value from \$10 to \$20 into the corresponding output compare register. An initial value of \$18 written into the output compare registers sets the servo motors at their mid points. The commands S1 to S6 allow you to change the set points of these 6 motors.

If you examine the output lines with an oscilloscope, you will see that the output PWM waves have a frequency of 60 Hz instead of the required frequency of 50 Hz. This is due to the fact that the ATmega328P is driven by a 16 MHz crystal clock, and 60 Hz comes out the prescalers naturally. If you want to drive servos at exactly 50 Hz, you can use one timer/counter to drive a second one and tune the first timer/counter accurately for 50 Hz operation. But then, you could only drive 3 servo motors. However, most servo motors do not really care about the base frequency of the PWM waves, and 60 Hz works just fine.

3.5. Traffic Controller

A traffic controller is my favorite demo application. I often challenge people to write the simplest and the most efficient program to control traffic lights at a highway intersection. In each of the north, south, east and west directions, I place two sensors to sense forward and left-turn cars, and 4 lights to indicate go, left-turn, caution, and stop signals. On the Arduino Boards, there are not enough output lines to drive 16 traffic signals, so I give the north and south directions the same 4 signals, and the east and west directions another 4 signals.

The commands are shown below:

```
( Traffic Controller on Arduino Uno )
( Chen-Hanson Ting, 5/10/2011      )
( Switches: PC: 0, N; 1, NL; 2, S; 3, SL; 4, W; 5, WL )
(           PB: 2: E; 3, EL )
( LEDs:     PD: 2, nsG; 3, nsY; 4, nsR, 5, nsL; 6, ewG; 7,ewY )
(           PB: 0, ewR; 1,ewL )

hex
```

```

: init-ports
  fc 2a c! 3 24 c!      \ output ports
  3f 28 c! c 25 c! ;   \ input ports, pullup resistors

: seconds for aft 100 for 100 for next next then next ;

: lights ( n -- )
  dup 2b c!      \ PD outputs
  100 / C or 25 c! ; \ PB outputs, maintain pullups

: switches ( -- n )
  23 c@ 100 *      \ PB inputs
  26 c@ or        \ PC inputs
  dup cr . ;

: N-S begin 104 lights 5 seconds
  switches c3a and if 108 lights 2 seconds then
  switches a and if 130 lights 3 seconds then
  switches c30 and until
  ;

: E-W begin 50 lights 5 seconds
  switches 82f and if 90 lights 2 seconds then
  switches 820 and if 310 lights 3 seconds then
  switches f and until
  ;

: go init-ports
  begin N-S E-W ?key until drop ;

```

I am very proud of this program, as I have revised it several times and now it is in its best shape.

Port	IO Line	IO Device	Function
D2	PD2	Green LED	North-South Go
D3	PD3	Yellow LED	North-South Caution
D4	PD4	Red LED	North-South Stop
D5	PD5	Green LED	North-South Left-Turn
D6	PD6	Green LED	East-west Go
D7	PD7	Yellow LED	East-west Caution
D8	PB0	Red LED	East-west Stop
D9	PB1	Green LED	East-west Left-Turn
A0	PC0	Switch	North Forward
A1	PC1	Switch	North Left-Turn
A2	PC2	Switch	South Forward
A3	PC3	Switch	South Left-Turn
A4	PC4	Switch	West Forward
A5	PC5	Switch	West Left-Turn
D10	PB2	Switch	East Forward

D11	PB3	Switch	East Left-Turn
-----	-----	--------	----------------

Commands are explained in the following table:

Command	Function
init-ports	Initialize the three IO ports PA, PC and PD properly. The input ports do not have to be initialized, except that their pull-up resistors are activated for the proper operation of the external switches. It is very satisfying that the ATmega328P can drive LED's directly with its output lines without current limiting resistors, and that it has optional pull-up resistors to simplify input circuitry. The actual layout of the traffic controller is therefore extremely simple.
seconds	Delay a number of seconds.
lights	From a 16 bit value, turn on/off 8 LEDs. The lower byte controls PD port, and the upper byte controls PB port.
switches	From a 16 bit value, read 8 switches. The lower byte reads PC port, and the upper byte controls PB port.
N-S	A loop managing north-south traffic. If either forward switches in the north or south direction are active, turn on North-South Go LED for 5 seconds. Next, if there are activity in other directions, turn on North-South Stop and Caution LEDs for 2 seconds. Then, if either left-turn switches in the north or south direction are active, turn on North-South Stop and Left-turn LED for 3 seconds. Then, if there are activity in the East-West direction, turn off North-South Caution LED's and exit this command so that E-W command has a chance to run. Otherwise, repeat N-S loop.
E-W	A loop managing east-west traffic. If either forward switches in the east or west direction are active, turn on East-West Go LED for 5 seconds. Next, if there are activity in other directions, turn on East-West Caution LED for 2 seconds. Then, if either left-turn switches in the east or west direction are active, turn on East-West Stop and Left-turn LEDs for 3 seconds. Then, if there are activity in the East-West direction, turn off East-west Caution LEDs and exit this command so that N-S command has a chance to run. Otherwise, repeat E-W loop.
go	Initialize IO ports and enter a loop repeating N-S and E-W commands. Exit this loop if the user hit any key on the keyboard.

This program is simple because I realized that it is a Finite State Machine with two major states, which are coded as N-S and S-W commands. There are three minor states in either major states and they are sequenced through under the appropriate conditions. You can treat it as a 6-state Finite State Machine, but the transition rules would be much more complicated.

4. Conclusion

I can bore you to death with more examples, but this seems a good point to stop. What I want to show you is that within the confines of Arduino 0022, it is possible to build a FORTH environment to let people explore this simple yet powerful

programming language. Although the small RAM memory in ATmega328P limits the number of new commands you can add to the FORTH system, and Arduino 0022 does not allow you to save the commands ceForth_328 compiles, it is a useful environment for you to explore this interesting microcontroller while you are reading the huge 566 page AVR Family Data Book.

PEEK and POKE are aliases of the native FORTH commands C@ and C!. They clearly demonstrate the power and the usefulness of FORTH as a programming language.

ATmega328P is a much more powerful microcontroller than what Arduino 0022 allows it to be. The roots of Arduino 0022 are in the UNIX operating system and in the C programming Language. I admire the developers of Arduino in simplifying the operating system and the language to the point that you are presented with only two routines:

```
setup( );  
loop( )
```

Most of the complications in the operating system and in the language are hidden from you so that you can go immediately doing useful things. However, the operating system and the language still insulate you from the underlying microcontroller, and prevent you from exploit the microcontroller to its full capacity.

FORTH is an operating system and a programming language which are transparent between you and the microcontroller you own. At the very low end, it allows you to push the microcontroller to the bare metal, giving you complete control over the registers, the IO devices and the memory. At the other end, it allows you to express you programming intentions at the highest conceptual level, in building nested lists to arbitrary depth, much like LISP albeit simpler, easier and without the irritating parentheses.

The ceForth_328 system is a teaser to give you some hands-on experience with FORTH on an Arduino Board. It introduces you to a real FORTH system 328eForth which give you access to the entire ATmega328P microcontroller, and allows you to build complete turnkey applications for Arduino Boards and even for bare ATmega328P chips. I hope to convince you that there is a better way to develop turnkey applications than Arduino 0022.

The Arduino 0022 system comes in a zipped file of 87,587 KB. It expands to fill 245 MB on your hard disk. You really don't know what's happening behind your back when you compile a sketch in Arduino 0022. It always amazes me that the results uploaded to the Arduino Uno Board actually works. It is a long and tedious task to learn about all the library routines provided in the Arduino 0022 system. Very often, it is difficult to find utilities and tools that you need to do your job. The huge Arduino community helps, but only to an extend. You are on your own in the end.

In contrast, the assembly source code of 328eForth system has only 54,472 bytes. This is 1/500th the size of the Arduino 0022 system, and it is within a single person's intelligence. However, this 54 KB of source code, describe a complete operating system , a programming language and a whole bunch of tools embedded inside a

microcontroller, independent of a host computer or a supporting operating system. It give you complete freedom in developing your specific applications.

Last but not least, actually, ATmega328P and the AVR family of microcontrollers, in my humble opinion, are great chips but of very poor design. Most microcontroller designers really don't know what they are doing. They just throw things together and called them microcontrollers. Not much thought were really put into the architecture, the instruction sets, and the peripheral devices. There were very few visions behind the microcontroller designs. And, hardware designers really do not understand software. They just throw the chip over the fence, and let software engineers make things work. On this side of the fence, software designers really do not understand software either, and they build clumsy, bulky, inefficient systems, plagued with bugs. So, we get a mess. Microcontrollers can be designed simpler and better, if the designers really understand hardware and software. In this respect, probably you should look at my 32-bit FORTH microcontroller design in eP32. But, that's a different story.

Appendix eForth_328 Commands

-	(n1 n2 -- n3)	Subtract n2 from n1 (n1-n2=n3).
'<name>	(-- addr)	Find <name> and leave its address.
!	(n addr --)	Store n to addr.
!IO	(--)	Initialize the serial I/O devices.
#	(n -- n/base)	Convert next digit of n and add it to output string .
#>	(n -- addr n1)	Terminate numeric conversion, leaving addr and count n1.
#S	(n --)	Convert all significant digits in n to output string.
#TIB	(-- addr)	Return address of variable storing number of characters received in terminal input buffer.
\$" <string>	(-- addr)	Compile a string literal. Return its address at run time.
S"	(-- addr)	Return address of following string literal at run time.
\$.n	(addr --)	Build a new dictionary header using the string at addr.
\$COMPILE	(addr --)	Compile string at addr to dictionary as a token or literal.
\$INTERPRET	(addr --)	Interpret string a addr. Execute it or convert it to a number.
(<text>)	(--)	Ignore comment text.
(parse)	(addr n char -- addr n delta)	Scan string delimited by char. Return found string and its offset delta.
*	(n1 n2 -- n3)	Signed multiply. Leave product.
*/	(n1 n2 n3 -- n4)	Signed multiply and divide. Leave quotient of (n1*n2)/n3.
*/MOD	(n1 n2 n3 -- n4)	Signed multiply and divide. Leave remainder of (n1*n2)/n3.
,	(n --)	Add n to parameter field of the most recently defined word.
.	(n --)	Display signed number with a trailing blank.
." <text>"	(--)	Compile <text> message. At run-time display text message.
."!	(--)	Display following string literal as a text message.
.(<text>)	(--)	Display <text> received from the input stream.
.ID	(addr --)	Display name of a command at addr.
.OK	(--)	Display ok> message.
.R	(n n1 --)	Display n right justified in a field of n1 character width.
/	(n1 n2 -- quot)	Signed division. Leave quotient of n1/n2.
/MOD	(n1 n2 -- rem quot)	Signed division. Leave quotient and remainder of n1/n2.
: <name>	(--)	Begin a compound command of <name>.
;	(--)	Terminate a compound command.
?	(addr --)	Display contents in addr.
?DUP	(n -- n n 0)	Duplicate top of stack if it is not a 0.
?KEY	(-- char T F)	Return input character and true, or a false if no input.
?RX	(-- char T F)	Return input character and true, or a false if no input.
?UNIQUE	(addr --)	Display a "reDef" message if addr is an existing command.
@	(addr -- n)	Replace addr by number fetched from addr.
[(--)	Switch from compilation to interpretation.
[COMPILE] <name>	(--)	Compile the word <name> in the input stream as an token.
\ <text>	(--)	Ignore text till end of line.
]	(--)	Switch from interpretation to compilation.
^H	(bot eot cur -- bot eot cur)	Backspace. Backup the cursor by one character.
+	(n1 n2 -- n3)	Add n1 and n2.
+!	(n addr --)	Add n to number at addr.
<	(n1 n2 -- flag)	True if n1 less than n2.
<#	(--)	Start numeric output conversion.
<MARK	(-- addr)	Push current program address on stack.
<RESOLVE	(addr --)	Compile addr to dictionary.
=	(n1 n2 -- flag)	True if n1 equals n2.
>	(n1 n2 -- flag)	True if n1 greater than n2.
>CHAR	(n -- char)	Convert n to a printable character char. Non-printable character

		is converted to an underscore character.
>IN	(-- addr)	Return address of a variable pointing to current character being interpreted.
>MARK	(-- addr)	Compile 0 to dictionary. Push its address on stack
>NAME	(ca -- na)	Convert a code field address to a name field address.
>R	(n --)	Pop top and push it on return stack.
>RESOLVE	(addr --)	Store address of current program word in addr.
>UPPER	(addr --)	Convert a count string at addr to upper case.
0<	(n -- flag)	True if n is negative.
0=	(n -- flag)	True if n is 0.
1-	(n -- n-1)	Decrement top.
1+	(n -- n+1)	Increment top.
2-	(n -- n-2)	Decrement top by 2.
2!	(d addr --)	Store a double integer to addr.
2*	(n -- 2n)	Multiply top by 2.
2/	(n -- n/2)	Divide top by 2.
2@	(addr -- d)	Fetch a double integer from addr.
2+	(n -- n+2)	Increment top by 2
2DROP	(d --)	Pop two numbers off stack.
2DUP	(d -- d d)	Duplicate a double integer on stack.
ABORT	(--)	Clean up stack and jump to address in 'ABORT.
'ABORT	(-- addr)	Return address to handle error condition.
abort"	(flag --)	If flag is true, display following message and ABORT.
ABS	(n -- u)	Return absolute value of top.
accept	(addr n -- addr n1)	Accept n characters to buffer at addr. Replace n with actual count n1
AFT	(--)	Branch to THEN to skip a branch in FOR-NEXT loop.
AHEAD	(--)	Branch forward to address in next word.
ALIGNED	(n -- n1)	Adjust n to the word boundary.
ALLOT	(+n --)	Add +n bytes to parameter field of the most recently word.
AND	(n1 n2 -- n3)	Logical bit-wise AND.
BASE	(-- addr)	Contain radix for numeric conversion.
BEGIN	(--)	Start an indefinite loop.
BL	(-- 32)	Push 32 on stack.
BRANCH	(flag --)	Branch to address in next program word if flag is 0.
C!	(n addr --)	Store a byte to addr.
C@	(addr -- n)	Fetch a byte from addr.
CHAR <string>	(-- char)	Push first character in the following text string.
CHARS	(n char --)	Send n characters char to the output device.
CMOVE	(addr addr1 n --)	Copy n bytes starting at addr to memory starting at addr1.
CODE <name>	(--)	Start a new primitive command.
COLD	(--)	Initialize FORTH system and start text interpreter.
COMPILE <name>	(--)	Retrieve address of the following command and compile it as a token.
CONSTANT <name>	(n --)	Define a constant. At run-time, n is pushed on the stack.
CONTEXT	(-- addr)	Return address of a variable pointing to name field of last word in dictionary.
COUNT	(addr -- addr+1 n)	Replace addr with address and count of a count string.
CP	(-- addr)	Return address of a variable pointing to first free space on dictionary.
CR	(--)	Display a new line. Carriage return and line feed.
CREATE <name>	(--)	Define an array. At run-time, its address is left on the stack.
DECIMAL	(--)	Set number base to decimal.

DIAGNOSE	(--)	Exercise all primitive commands for debugging.
DIGIT	(n -- char)	Convert digit u to a character.
DNEGATE	(d -- d1)	Negate a double integer on stack.
do\$	(-- addr)	Return the address of the following compiled string.
doCON	(-- n)	Return contents of next program word.
doLIST	(--)	Start processing a new nested list.
doLIT	(-- n)	Push an inline literal.
doNEXT	(--)	Terminate a single index loop.
doVAR	(-- addr)	Return address of next program word.
DROP	(n --)	Discard top of stack.
DUMP	(addr n --)	Dump n bytes of memory starting from addr.
DUP	(n1 -- n2)	Duplicate top of stack.
ELSE	(--)	Terminate <true> clause, continue after the THEN.
EMIT	(char --)	Initialize the serial I/O devices.
ERASE	(addr n --)	Clear a n byte array at addr
ERROR	(addr --)	Display error message at addr and jump to ABORT.
EVAL	(--)	Interpret input stream in terminal input buffer.
'EVAL	(-- addr)	Return address of variable containing \$INTERPRET or \$COMPILE.
EXECUTE	(addr --)	Execute the command at addr.
EXIT	(--)	Terminate execution of current compound command.
EXPECT	(addr n --)	Accept n characters into buffer at addr.
EXTRACT	(n base -- n/base n1)	Extract the least significant digit n1 from n. n is divided by base.
FILL	(addr n char --)	Fill an array at address with n characters char.
find	(a va -- ca na a 0)	Search dictionary at va for a string at a. Return ca and na if succeeded, else return a and 0.
FOR	(n --)	Setup loop. Repeat loop until limit n is decremented to 0.
FORGET <name>	(--)	Delete command <name> and all words added afterwards.
HERE	(-- addr)	Address of next available dictionary location.
HLD	(-- addr)	Return address of a variable pointing to next converted digit.
HOLD	(char --)	Add character char to the number string under conversion.
IF	(flag --)	If flag is zero, branches forward to <false> or after THEN.
IMMEDIATE	(--)	Set immediate bit in name field of last command added.
KEY	(-- char)	Get an ASCII character from the keyboard. Does not echo.
kTAP	(bot eot cur char -- bot eot cur)	Process a control character, CR or backspace.
LAST	(-- char)	Get an ASCII character from the keyboard. Does not echo.
LITERAL	(n --)	Compile number n. At run-time, n is pushed on the stack.
M*	(n1 n2 -- d)	Multiply n1 and n2. Return double integer product.
M/MOD	(d n -- mod quot)	Divide double integer d by n1. Return remainder and quotient.
MAX	(n1 n2 -- n3)	n3 is the larger of n1 and n2.
MIN	(n1 n2 -- n3)	n3 is the smaller of n1 and n2.
MOD	(n1 n2 -- mod)	Signed divide. Leave remainder of n1/n2.
NAME?	(addr -- ca na a F)	Search dictionary for name at addr. Return code field address and name field address if a command is found, else push a false.
NAME>	(na -- ca)	Convert a name field address to a code field address.
NEGATE	(n1 -- n2)	Two's complement.
NEXT	(--)	Decrement index and repeat loop until index is less than 0
NOT	(n1 -- n2)	Bit-wise one's complement.
NUMBER?	(addr -- n T addr F)	Convert a string at addr to an integer and push a true flag. If it is not a number, push a false flag.
OR	(n1 n2 -- n3)	Logical bit-wise OR.
OVER	(n1 n2 -- n1 n2 n1)	Make copy of second item on stack.
OVERT	(--)	Change CONTEXT to add a new command to dictionary.
PACK\$	(addr n-- addr1)	Copy a string at addr with length n, to a count string at addr1.

PAD	(-- addr)	Return address of a scratch pad area.
PARSE	(char -- addr n)	Parse terminal input buffer for a string terminated by char. Return its address and length.
PEEK	(addr -- n)	Fetch a byte from addr.
POKE	(n addr --)	Store a byte to addr.
QBRANCH	(flag --)	Branch to address in next word if flag is zero.
QUERY	(-- addr)	Leave address of a scratch area of at least 84 bytes.
QUIT	(--)	Return to terminal, no stack change, no message.
R@	(-- n)	Copy top of return stack on stack.
R>	(-- n)	Pop top of return stack and push it on stack.
REPEAT	(--)	Unconditional backward branch to BEGIN.
ROT	(n1 n2 n3 -- n2 n3 n1)	Rotate third item to top. "rote"
SAME?	(addr1 addr2 n -- aadr1 addr2 flag)	Compare two strings at addr1 and addr2 for n bytes. If string1>string2, returns a positive integer. If string1<string2, return a negative integer. If strings are identical, return a 0.
SEE <name>	(--)	Decompile the word <name>.
SIGN	(n --)	If n is negative, add a - sign to the number output string.
SPACE	(--)	Display a space.
str	(n -- addr n1)	Convert signed integer n to a numeric output string at addr, length n1.
SPACES	(n --)	Display n spaces.
SWAP	(n1 n2 -- n2 n1)	Exchange top two stack items.
TAP	(bot eot cur char -- bot eot cur)	Accept and echo a character and bump the cursor.
THEN	(--)	Terminate the IF-ELSE structure.
TIB	(-- addr)	Push address of terminal input buffer.
TIB	(-- addr)	Return address of variable pointing to terminal input buffer.
tmp	(-- addr)	Return address of a temporary variable.
TOKEN	(-- addr)	Parse next string delimited by space into a word buffer 2 bytes above the top of dictionary.
TX!	(char --)	Send character c to the output device.
TYPE	(addr +n --)	Display a string of +n characters starting at address addr.
U.	(n --)	Display unsigned number with trailing blank.
U.R	(n n1 --)	Display unsigned number n right justified in a field of n1 characters.
U<	(n1 n2 -- flag)	Unsigned compare. Return true if n1<n2.
UM*	(n1 n2 -- d)	Unsigned multiply. Return double integer product.
UM/MOD	(d n -- mod quot)	Unsigned divide. Return remainder and quotient.
UM+	(n1 n2 -- d)	Unsigned add. Return double integer sum.
UNTIL	(flag --)	Repeat <loop-body> until the flag is non-zero.
UPPER	(char -- char1)	Convert a character to upper case.
VARIABLE <name>	(--)	Define a variable. At run-time, <name> leaves its address.
WHILE	(flag --)	Repeat <loop-body> and <>true> clause while the flag is non-zero.
WITHIN	(n1 n2 n3 -- flag)	Return true flag if n1<=n3<n2. Else, return false flag.
WORD <text>	(char -- addr)	Get the char delimited string <text> from the input stream and leave as a counted string at addr.
WORDS	(--)	Display all commands in the dictionary.
XOR	(n1 n2 -- n3)	Logical bit-wise exclusive OR.