

Tao of Arduino

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

328eForth.hex is a free copy of 328eForth system for you to try on your Arduino Uno, or other compatible, board. For its AVR assembly source code and complete documentation, purchase the 328eForth system from www.offete.com, Item# 2159, for \$25. The 328eForth system is in the public domain, and you can use it without restriction.

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 commands constructed from the primitive commands and other compound commands. By isolating machine dependent commands from machine independent commands, the eForth model can be easily ported to any microcontroller. I ported this model to ATmega328P, and the result is the 328eForth system, which runs very nicely on Arduino Uno.

328eForth is optimized for performance. The number of primitive commands is expanded to 68. Commands which are used to build the FORTH system but rarely used by users are hidden so that you are not overwhelmed with unused commands. Only 151 commands are exposed to you. The source code is written in AVR assembly. The code is available so that you can modify it to suite your application. The entire system takes up 5,156 bytes in flash memory. leaving lots of room for your application. Hence, it is Tao of Arduino

Installing 328eForth

Get an Arduino Uno board for about \$29.

Get an Atmel AVR ISP mkII programmer for about \$34.

Download the AVR Studio 4 from Atmel web site:

http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725

Install AVR Studio 4. Do not connect the AVR ISP mkII until the software installation is complete. Studio 4 will install its driver, Jungo USB, for its AVR ISP mkII. The USB cable must not be connected until After the install in done.

Download the Arduino 0022 package at <http://www.arduino.cc/en/Main/Software>
Unzip and install. This should load the USB to COM simulator from FTDI. The drivers are located in \Arduino-0022\drivers\FTDI drivers\.

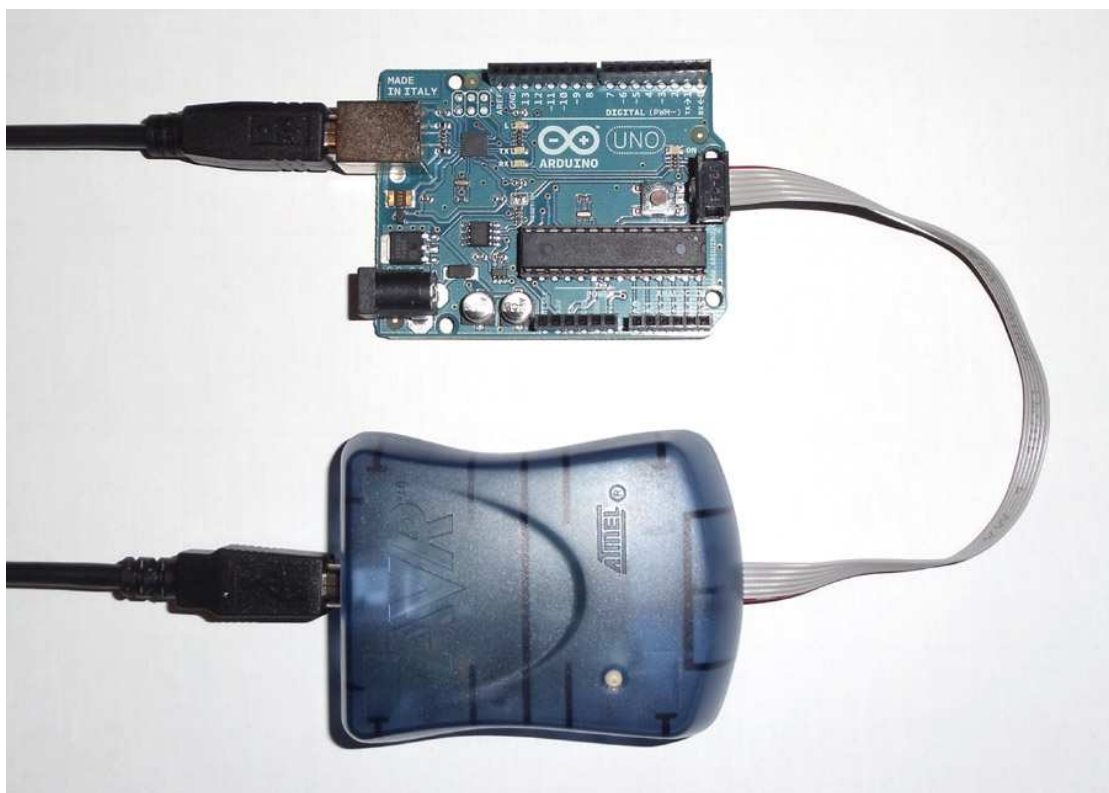
To check on these USB drivers, go to

Start\Control-Panel\System\Hardware\Device-Manager and you will see Jungo\AVR ISP mkII. Under Ports (Com & LPT), you will see Arduino Uno (COM X). Remember the COM port number X for use with HyperTerminal or RealTerm.

Download and install RealTerm from SourceForge (<http://realterm.sourceforge.net/>). HyperTerminal is standard in Windows under Start\Accessories\Communication\, and it works similarly.

Connect the AVR ISP mkII ribbon cable to the six pin ISP header on Arduino Uno. Red wire is #1 matching a tiny dot below the header, as shown in the picture below

Connect a USB cable from the AVR ISP mkII to the PC. Connect another USB cable from Arduino Uno to the PC. The AVR ISP MkII doesn't power the target so the Arduino USP is its power source. Check the serial connections as noted above.



With the cables connected, you should be able to invoke arduino.exe in the Arduino 0022 folder, and do all the wonderful things with Arduino sketches.

You need AVR Studio 4. I used the AVR assembler in Studio 4 to assemble the 328eForth system, and then used AVR ISP mkII cable to upload 328eForth.hex to ATmega328P on Arduino Uno.

Start Atmel AVR Tools \AVR Studio 4\. Dismiss the Welcome window if it comes up. In the second row of icons you can see two icons that look like integrated circuits. Click on the left one labeled CON, and the Connection Dialog window appears. Check AVRISP mkII on the left and USB on the right. Then click Connect. You will then be taken to the AVRISP window. If not, click on the bug icon to the right labeled AVR.

On the AVRISP window, select Main page. In the Device and Signature Bytes panel, pick ATmega328P in the Device box. In the Programming and Target Settings panel, you will see that the ISP Frequency is set to 1 MHz. Click the Erase Device and Read Signature buttons to verify that you can erase the chip and read its signature bytes. If AVRISP failed to erase ATmega328P or read the signature bytes, click the Settings button, and lower the ISP frequency to probably 125 kHz.

Select the Lock Bits page. ATmega328P also has what's called lock bits and fuse bytes, which are used to configure the chip to behave properly according to your requirements. The lock bits protect sections of flash memory from inadvertent reading and writing operations. Select 0xFF for the lock bits to allow writing to the flash memory. Click Program button to program the lock bits.

Select the Fuses page. The fuse bytes configure CPU, memory, and I/O devices and select modes of operations for these components. Select 0xFD for the Extended Fuse byte, 0xD8 for the High Fuse byte, and 0xFF for the Low Fuse byte. Click Program button to program the fuse bytes.

You should now have the green power LED lit on the Arduino Uno. The green LED lit inside AVR ISP MkII case (shows USB OK) and the green LED lit (shows programmer cable OK) on the surface of the AVR ISP MkII.

Select the Program page. In the Device panel, check the box labeled "Erase device before flash programming." In the Flash panel, open and navigate to your 328eforth hex file. Click the Program button in the Flash panel. You will see a dialog at the lower left as the program is loaded.

You may now disconnect the AVR ISP mkII programmer. However I generally keep it connected in case I have to reload 328eForth.

Running 328eForth

After 328eForth is loaded through the AVRISP programmer, load HyperTerminal or Realterm, and you can now talk to 328eForth on Arduino Uno.

On the HyperTerminal console, pull down the Call menu and select Disconnect. Then, pull down the File menu and select Properties option. In the Connect Using dialog box, select the COM port you saw earlier in the USB device assignment. Click the Configuration button and a COMx Properties window pops up. Select 19,200 baud, 8 data bits, no parity, 1 stop bit, and no flow control. Then click OK button to dismiss the COMx Properties window.

In the main Properties window, click the Settings tab and then click the ASCII Setup button, and an ASCII Setup window pops up. Enter 900 in the Line Delay dialog box to insert 900 msec delay after sending each line of text. Later when you download source code files, you will need this end of line delay.

Click OK button to dismiss the ASCII Setup window. Click OK button in the main Properties window and dismiss this window also.

Now you are back to the HyperTerminal Console window. Pull down Call menu and select Call, and you will see the sign-on message generated by 328eForth:

```
328eForth v2.20
```

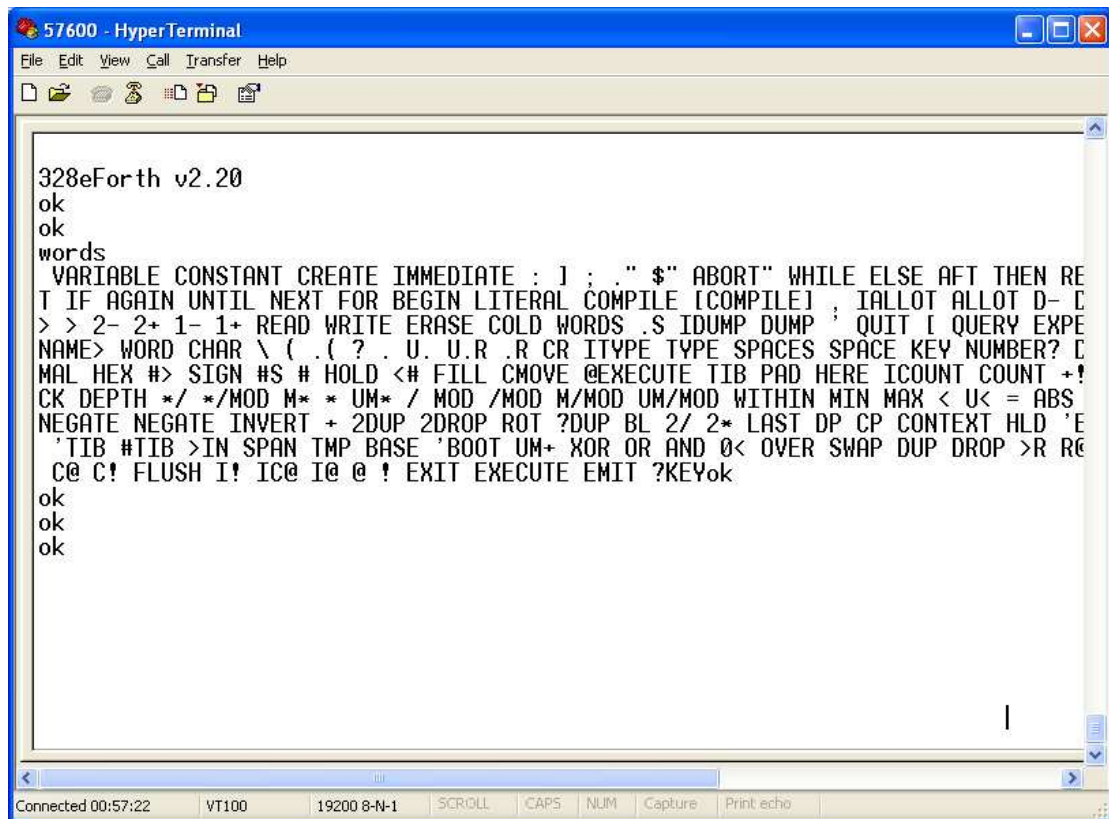
Hitting Return key several times, and you should see the two send/receive LEDs flash on Arduino Uno, and ok messages are displayed on the HyperTerminal console. You can now type in FORTH commands to interact with 328eForth.

328eForth is case insensitive. You can type commands in either upper or lower case. Note that 32eForth is in the hexadecimal base when it starts.

Type the following commands to test the 328eForth system. Ignore the text after the \ command, which starts a comment string until end of line. Terminate each line with a Return key.

```
words
100 dump
200 idump
7000 idump
```

After bring up 328eForth, type words and you will see a list of eForth commands on the HyperTerminal console:



```
57600 - HyperTerminal
File Edit View Call Transfer Help
328eForth v2.20
ok
ok
words
VARIABLE CONSTANT CREATE IMMEDIATE ; ] ; ." $" ABORT" WHILE ELSE AFT THEN RE
T IF AGAIN UNTIL NEXT FOR BEGIN LITERAL COMPILE [COMPILE] ; IALLOT ALLOT D- C
> > 2- 2+ 1- 1+ READ WRITE ERASE COLD WORDS .S IDUMP DUMP ; QUIT [ QUERY EXPE
NAME> WORD CHAR \ ( . ( ? . U. U.R .R CR ITYPE TYPE SPACES SPACE KEY NUMBER? [
MAL HEX #> SIGN #S # HOLD <# FILL CMOVE @EXECUTE TIB PAD HERE ICOUNT COUNT +!
CK DEPTH */ */MOD M* * UM* / MOD /MOD M/MOD UM/MOD WITHIN MIN MAX < U< = ABS
NEGATE NEGATE INVERT + 2DUP 2DROP ROT ?DUP BL 2/ 2* LAST DP CP CONTEXT HLD 'E
'TIB #TIB >IN SPAN TMP BASE 'BOOT UM+ XOR OR AND 0< OVER SWAP DUP DROP >R R@
C@ C! FLUSH I! IC@ I@ @ ! EXIT EXECUTE EMIT ?KEYok
ok
ok
ok
```

HyperTerminal breaks up a word at the right margin of the window console. You will have to read across lines to see whole words. There are 151 FORTH commands visible in 328eForth system. There are actually about 200 eForth commands, but many of them are hidden, without link and name fields. These hidden commands are

needed to implement the 328eForth system, but are not useful in normal programming. These 151 visible commands are documented in the Appendix for your reference.

Warning:

After you defined a new FORTH command with the : command, always type a FLUSH command before executing the new command. When a new command is defined, it is stored in a RAM buffer. FLUSH writes the new commands in the RAM buffer to the main flash memory, where they can be executed. If the RAM buffer is not flushed, executing new commands in the RAM buffer will crash the FORTH system. You will have to reboot or reload 328eForth system through the AVRISP programmer cable.

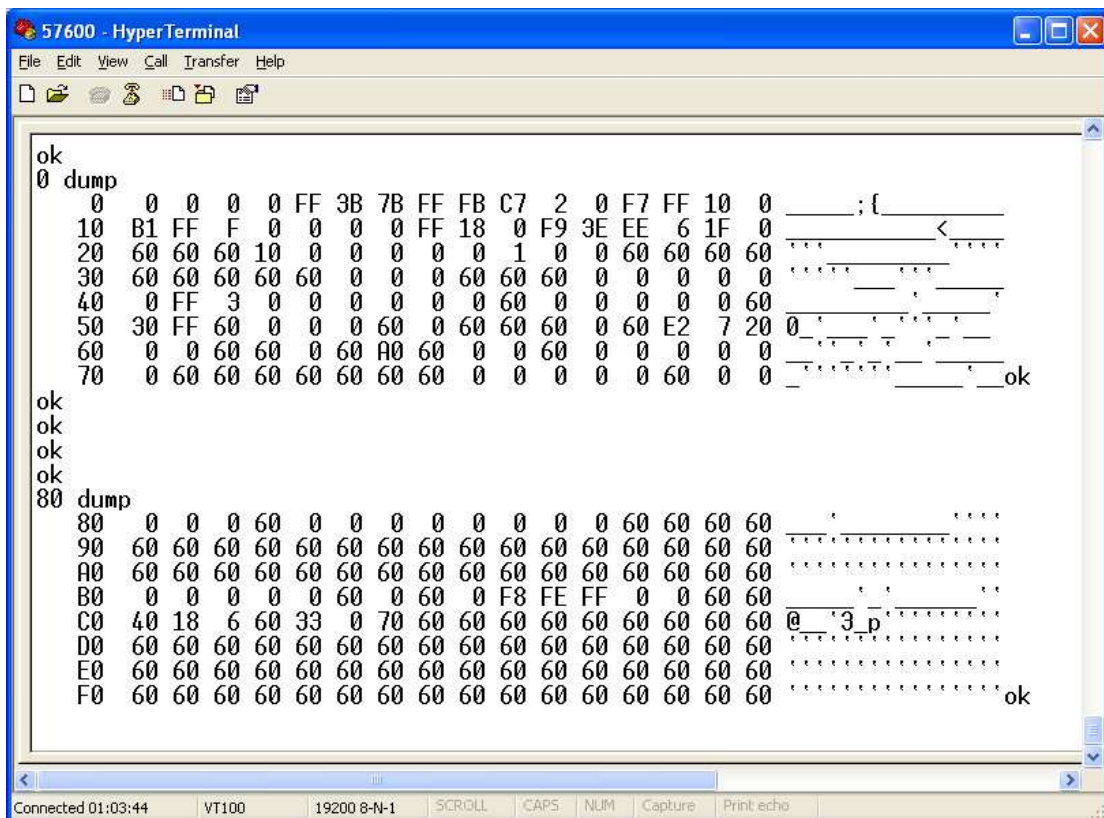
More Exercises

What can eForth do over and above Arduino 0022?

One quick answer I can give you is to ask you typing in the following command:

```
0 DUMP
80 DUMP
```

and you will see the following display in the HyperTerminal console:



In this display, you see the RAM memory of ATmega328 from location 0 to location \$FF. If you had read the ATmega Microcontroller Data Book, you would know that the first 32 bytes are 32 registers in ATmega CPU, the next 64 bytes from \$20 to \$5F are the I/O registers, and the last 160 bytes from \$60 to \$FF are Extended I/O registers. Many of these registers are not implemented as physical devices, and they

show up containing \$60. Actual I/O registers show their actual contents.

Even better, you can change the contents of the CPU and I/O registers! As CPU registers are used by the 328eForth system, I do not recommend your changing them without knowing exactly what you are doing. You can easily crash the system if you advertently change some of the critical CPU registers. However, there is no better way to understand the I/O devices in ATmega328 than to study the register definitions and functions of the bits in these registers, and to change these bits while observing the signals coming out of the corresponding I/O pins.

Once you understand the control, status, and data registers in an I/O device, you can write a short FORTH command to exercise this device the way you eventually will use it. This command to test the device will grow to be the basis of your application.

In the following sections, I will show you how to change some of the I/O registers directly with C! commands, to operate these I/O devices. You need that thick 566 page ATmega Microcontroller Data Book opened on your computer, and read the register definitions to follow the discussions. I will show you addresses of the I/O registers, but you will have to look up the definitions of bits in these registers to go along. It is difficult at first to read register addresses and contents in hex, but I hope you will get used to them. It will be very rewarding when you see what these bits actually do and produce results you can observe visually.

The best way to wade through this thick Data Book is to test the devices interactively with 328eForth.

USART

The first device I will discuss is the serial USART0 port, because it is the only I/O device used by 328eForth. It has the following set of registers:

Address	Register	Name	Function	Initial Value
\$C0	UCSR0A	Control and status register A	Status of transmitter and receiver	--
\$C1	UCSR0B	Control and status register B	Interrupt, enabling, data format	\$18
\$C2	UCSR0C	Control and status register C	Mode select, start, stop, parity	\$6
\$C4	UBRR0L	Baud rate register low	Baud rate divisor, low byte	\$33
\$C5	UBRR0H	Baud rate register high	Baud rate divisor, high byte	0
\$C6	UDR0	Data register	Transmitted or received data	--

UCSR0A reports the current status of the USART0 and UDR0 contains transmitted or received data. These registers change dynamically and do not require initialization. UCSR0B/C selects 1 start bit, 8 data bits, 1 stop bit, no parity and no flow control. The UBRR0L/H registers set USART0 up to run at 19,200 baud. Do not change these 4 registers unless you know what you are doing. If you mess up these registers,

Arduino Uno will not talk to the HyperTerminal and you have to reach for the reset button.

Read Atmega Data Book to learn what each bit in UCSR0B/C is doing. You can understand these bits better when you are actually looking at them on the HyperTerminal console.

One easy experiment you can do is to change the baud rate register UBBR0L from \$33 to \$66 by typing the following commands:

```
$66 $C4 C!
```

If you are in hexadecimal mode, you do not have to type the \$ prefix before numbers:

```
HEX 66 C4 C!
```

The baud rate is changed from 19,200 baud to 9,600 baud. Now, HyperTerminal stops talking to Arduino. Pull down Call menu and select Disconnect option. Pull down File menu and select Properties option. In the Properties window, change the baud rate to 9,600 baud. Connect the phone line, and Arduino will talk to HyperTerminal at 9,600 baud.

Type '0 DUMP' commands, and you will see that contents of UBRR0L register at C4 is changed to 66.

Type the following commands to get back to 19,200 baud:

```
33 C4 C!
```

Change the baud rate of HyperTerminal back to 19,200 baud so Arduino will talk again.

Always return HyperTerminal to 19,200 baud. Otherwise, when Arduino Uno is reset and reverts to 19,200 baud, you would spend a long time wondering why Arduino does not talk to the HyperTerminal. It could cause a panic, if you forget that they are using a different baud rate.

GPIO Port B

The Digital I/O Line 13 on Arduino Uno is connected to bit-5 of GPIO Port B, PB-5. Port B as a general purpose I/O device has the following registers:

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

Setting a bit in DDRB register makes the corresponding pin an output pin. Then, writing this bit in PORTB register sends it to the output pin. It is very easy to turn the LED connected to Line 13 on and off by the following commands:

```
HEX  
20 24 C! \ make Line 13 an output pin
```

```

20 25 C!    \ turn Line 13 LED on
0  25 C!    \ turn Line 13 LED off

```

If you read ATmega Data Book.. carefully, you will find that when a pin is set up as an output pin, writing a 1 to that bit in PINB register will toggle this output pin. Try the following commands and you can verify this function:

```

20 23 C!    \ toggle Line 13 LED
20 23 C!    \ toggle Line 13 LED

```

Type `0 DUMP` commands and you can see the current state of these registers as you turn the LED on and off.

Now, we can replicate what the BLINK sketch example does in Arduino 0022. Here is the program in FORTH:

```

: MS ( n -- ) FOR AFT $1CB FOR NEXT THEN NEXT ;
: BLINK 20 24 C! BEGIN 20 23 C! 400 MS AGAIN ;
FLUSH
BLINK

```

400 in hexadecimal equals to 1024 in decimal. \$400 MS will cause a delay of about 1 second. Execute BLINK will cause the Line 13 LED to blink forever.

BLINK sketch is the first program every Arduino user runs. It gives the user a warm and fuzzy feeling that he is making Arduino Uno do something significant. However, the above FORTH BLINK program is the silliest program a FORTH programmer can ever write. It is an infinite loop you cannot get out, unless you push the reset button or pull the power plug off. The ATmega328 microcontroller is not made to run BLINK. It is much more powerful and much more intelligent than just turning a stupid LED on and off.

A thoughtful FORTH programmer would write this program instead:

```

: BLINK 20 24 C! BEGIN 20 23 C! 400 MS ?KEY UNTIL DROP ;
FLUSH
BLINK

```

This program will blink the LED forever as the earlier one. But, when you are tired of looking at this stupid LED, you can stop it by pressing any key on the keyboard. You can exit the loop. Now, you can type in other commands to the 328eForth system, and do other useful things.

If a bit in DDRB is cleared to 0, the corresponding pin becomes an input pin. Initially this input pin is tri-stated. If you set the corresponding bit in PORTB register to a 1, this input pin will be pulled to Vcc by an internal pull-up resistor. This pull-up resistor is very useful and it simplifies the external circuitry of many input devices. For example, you can connect this input pin to a push-button switch with its other terminal grounded. If the switch is open, you will read a high on the input pin, because of the pull-up resistor. If the switch is closed, you will read a low on the input pin.

Try this on Digital I/O Line 8, which is connected to Bit-0 in Port B. Type the

following commands to test the switch:

```

0 24 C!      \ make all Port B pins input
1 25 C!      \ turn on pull-up resistor for Line 8
23 C@ .      \ read PINB port and show its contents
23 C@ .      \ repeat with switch on and off

```

Timer/Counter0 and Tone Generator

ATmega328 has three very powerful, and hence complicated, timer/counters. They can be used as timers, counters, pulse width modulators, and square wave generators. Timer/Counter0 and Timer/Counter2 have an 8 bit counter registers, and Timer/counter1 has 16 bit counter register. Timer/Counter0 had the following registers:

Address	Register	Name	Function	Initial Value
\$44	TCCR0A	Control register A	Mode select	0
\$45	TCCR0B	Control register B	Clock select	0
\$46	TCNT0	Count register	Counter value	0
\$47	OCR0A	Output compare register A	Compare value. When equal to TCNT0, generate output on OC0A	0
\$48	OCR0B	Output compare register B	Compare value. When equal to TCNT0, generate output on OC0B	0

The bits in Control Registers A and B are complicated, and you have to read the ATmega Data Book. to understand them. To run Timer/Counter0 as a free run counter, set it up in the CTC (Clear Timer on Compare Match) mode. Store a value in OCR0A register to specify the period of the audio tone. Connect a speaker to Digital I/O Line 6, which is on Bit 6 in Port D, PD-6, and is toggles by the output compare signal of OC0A. Here are the commands you have to type:

```

HEX
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

```

You will hear a tone from the speaker, if everything is set up correctly. To turn off the speaker, type:

```

0 45 C!      \ prescaler=0, no clock to timer/counter0

```

Storing a value from 1 to 5 into TCCR0B changes the prescaler between the master clock and Timer/Counter0. Each step in the prescaler increases the prescaler divisor by a factor of 4 or 8, and you can hear the tone pitch changes drastically. To make smaller changes in the tone pitch, change the value in OCR0A register at location \$47.

Arduino Uno has a master clock of 16 MHz. With a /64 prescaler, the clock to Timer/Counter0 is 250 KHz. With a divisor of 255 in OCR0A register, the pitch we

get from OCR0A is about 490 Hz. You can play with the prescaler and the value in OCR0A to get different pitches.

Now, let us try to run Timer/Counter0 as a PMW (Pulse Width Modulator) device. Remove the speaker from Digital I/O Line 6, and connect an LED to it. The anode pin (long leg) is connected to Line 6, and the cathode (short leg) is connected to ground. Type in the following commands:

```
HEX
40 2A C! \ I/O Line 6 is set up as an output pin
83 44 C! \ TCCR0A, select fast, non-inverting PWM
mode
80 47 C! \ set OCR0A to mid-range
3 45 C! \ prescaler=3, start PWM
```

The LED will be turn on to medium brightness. Reduce the brightness by typing:

```
10 47 C! \ decrease LED brightness
```

Increase the brightness by typing:

```
F0 47 C! \ increase LED brightness
```

Now change to the fast, inverting PWM mode:

```
C3 44 C! \ inverting PWM mode
```

PWM output is now inverted. Storing a bigger value in OCR0A reduces LED brightness. Storing a smaller value in OCR0A increases LED brightness.

If you have an oscilloscope, you can watch the PWM waveforms. Then, you will really appreciate the ease in using 328eForth to control your hardware.

You can change the PWM to the phase correct mode by typing:

```
81 44 C! \ non-inverting phase correct PWM mode
or,
C1 44 C! \ inverting phase correct PWM mode
```

Changing the count value in OCR0A and the prescaler in TCCR0B, you can experiment with Timer/Counter0 to you heart's delight. You need an oscilloscope to see the waveforms, and preferably some servo motors to really see the PWM output doing real work.

The base frequency of the fast PWM oscillator is:

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

Timer/Counter1

Timer/Counter1 has a 16-bit counter which offers wider dynamic range and higher

accuracy in timing/counting. It is also more complicated than Timer/Counter0 and 2. Nevertheless, their operations are very similar. The registers and their functions in Timer/Counter1 are as follows:

Address	Register	Name	Function	Initial Value
\$80	TCCR1A	Control register A	Mode select	0
\$81	TCCR1B	Control register B	Mode and clock select	0
\$84	TCNT1L	Count register Low	Counter value low byte	0
\$85	TCNT1H	Count register High	Counter value high byte	0
\$88	OCR1AL	Output compare register A Low	Compare low byte. When equal to TCNT1, generate output on OC1A	0
\$89	OCR1AH	Output compare register A High	Compare high byte.	0

You first clear TCCR1A to set up Timer/Counter1 in the normal counting mode. To time an event, you clear the 16-bit counter TCNT1 and store a prescaler value into TCCR1B to start the counter. After the event, clear TCCR1B to stop the counter. Then, read the accumulated counts in TCNT1 counter.

Type in the following commands:

```

HEX
0 80 C!      \ clear TCCR1A to set up normal counting mode
0 84 !       \ clear 16-bit counter TCNT1
5 81 C! 100 MS 0 81 C!
              \ time '100 MS' commands
84 ?        \ read counts in TCNT1 counter

```

Let us stay in hexadecimal base, and 100 MS delays for 256 milliseconds. 400 MS delays for 1.024 seconds. My experiments show that 0 MS takes \$220 counts, 100 MS takes 1262 counts, and 400 MS takes \$4322 counts. They look right to me.

With a prescaler of 5, Timer/Counter1 overflows at about 4 seconds, while Time/Counter0 would overflow at about 16 milliseconds. To generate waves at 1 Hz range, you have to use Timer/Counter1. We can blink a LED at 1 second periods using Timer/Counter1, if we connect a LED to the compare output pin OC1A, which is the Digital I/O Line 9, or PB-1 port.

```

HEX
2 24 C!      \ set DDRB PB-1 (Line 9) as output pin
40 80 C!     \ set Timer/Counter1 to CTC mode
8000 88 !    \ init OCR1A compare register to a value
B 81 C!     \ CTC mode, prescaler=3, start wave

```

Changing the prescaler/mode value in TCCR1B changes the frequency of the output wave. The frequency and value in TCCR1B are shown as follows:

TCCR1B Value	Prescaler	Divisor	Frequency
9	1	1	244 Hz

A	2	8	30.5 Hz
B	3	64	3.75 Hz
C	4	256	0.96 Hz
D	5	1024	0.24 Hz

ADC

Analog to Digital Converter is the most interesting, and probably the most complicated device in a microcontroller. In ATmega328P chip, we have 6 channels of ADC to read analog signals from external circuits, making it extremely useful for real applications looking at real analog signals. From a programmer's point of view, its ADC is not very complicated, and we only have to worry about the following 5 registers:

Address	Register	Name	Function	Initial Value
\$78	ADCL	Data register Low	Data low byte	0
\$79	ADCH	Data register High	Date high byte	0
\$7A	ADCSRA	Control register A	Control, status, and prescaler bits	0
\$7B	ADCSRB	Control register B	Auto-triggering source	0
\$7C	ADMUX	Multiplexer selection register	Voltage reference and multiplexer section	0

ATmega328P has an internal temperature sensor, connected to Channel 8 of the ADC device. In addition, the internal 1.1 V reference voltage is connected to Channel 14, and a ground is connected to Channel 15. These Channels are very useful in testing the ADC.

Using 5 V power for reference and measuring the internal 1.1 V source, set up the ADMUX register and start the conversion this way:

```

HEX
4E 7C C! \ 4 selects 5 V reference; E selects Channel
14
C3 7A C! \ C enables/starts ADC; 3 selects /8
prescaler
78 ? \ display results, nominally $E0

```

For reasons I do not understand, a prescaler less than 3 would not start ADC conversion in this mode of operation. The following commands measures the ground on Channel 15:

```

4F 7C C! \ 5 V reference; ground input on Channel 15
C3 7A C! \ start conversion
78 ? \ display results, 0

```

The temperature sensor is connected to Channel 8, and it is recommended in ATmega Data Book to read it with the internal 1.1 V source for reference. Type the following commands:

```

C8 7C C! \ C selects 1.1 V reference; 8 selects
;temperature sensor

```

```

C3 7A C!      \ start conversion
78 ?         \ display results, nominally $160

```

If you connect an external analog signal source to A0 pin, then type the following commands to read its analog value:

```

1 27 C!      \ setup A0 as input pin, which is on PC-0 port
1 28 C!      \ turn on pull-up resistor on A0 pin
40 7C C!     \ setup reference and multiplexer inputs
C3 7A C!     \ start conversion
78 ?         \ display results

```

Build a Turnkey Application

In the FORTH parley, 'Turnkey' means the re-configuration of a FORTH system so that when power is applied and the system boots up, it initializes all the hardware devices in the system and start to execute the application it was designed to run. In 328eForth, you write lots of new commands. These commands are used to build more power commands until the last command looks like this:

```
: APPL SETUP BEGIN READ-INPUT SEND-OUTPUTS AGAIN ;
```

To turnkey this application so that it executes APPL command on booting-up, type the following commands:

```

' APPL 'BOOT ! \ store address of APPL in variable 'BOOT
$100 ERASE      \ erase flash
$100 $100 WRITE \ save RAM $100-17F to flash $100-17F

```

Now, the Atmega328 has the 328eForth system with the complete application saved to the flash memory. When the Arduino Uno board is reset or powered up, APPL will run.

Actually, after APPL command is compiled, all FORTH commands are already stored in the flash memory, but all the variables are still in RAM. Assuming that necessary data in RAM that have to be saved are between RAM locations from \$100 to \$17F, the WRITE commands above save them all to the flash memory from \$100 to \$17F. When 328eForth boots up, it automatically copies this page from flash to RAM, and APPL will start with all the necessary data in RAM.

Conclusions

What I give you in 5156 bytes, is a programming language, an interactive operating system, and all the debugging tools to develop applications on the Arduino Uno, for the Arduino Uno. The complete source code of 328eForth.asm contains 54 Kbytes, comparing to 232 MB hogged by Arduino 0022. It is an organic system, which can grow to accommodate any application that the ATmega328P microcontroller can host. It allows you to read all its CPU and I/O registers, and all its data and program memories. It also allows you to change the I/O registers and memories, and to add new commands to the flash memory. By adding new commands, you extend the 328eForth system and build a new system for your application.

In 328eForth, I try to reduce the FORTH language to its bare minimum, so that you

can learn this programming language quickly, and to use it to do useful work. ATmega328, like all the newer microcontrollers available now, contains many powerful and complicated I/O devices, and it takes the ATmega Data Book 566 pages to explain them. With 328eForth, you can examine all the I/O registers and modify them to make the I/O devices work the way you want them to work. There is no better way to study the ATmega Data Book than to read the book along with 328eForth, modifying the I/O registers and observe what the I/O devices do. 328eForth is a worthy companion to the ATmega Data Book.

Arduino Uno board is an excellent platform for FORTH. FORTH allows you to develop substantial applications quickly and produce high quality code. You write commands in small modules which can be (and must be) tested exhaustively. Fully tested commands can then be used to build more powerful commands at higher conceptual levels, until the last command, which becomes the application. This last command can be used to configure a turnkey system, so that it will be executed when the system boots up. You can do all these things with 328eForth on Arduino Uno.

Appendix 32eForth Commands

Stack Comments:

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

Stack Abbreviations of Number Types

flag	Boolean flag, either 0 or -1
char	ASCII character or a byte
n	16 bit number
addr	16 bit address
d	32 bit number

Stack Manipulation Commands

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

Arithmetic Commands

+	(n1 n2 -- n3)	Add n1 and n2.
-	(n1 n2 -- n3)	Subtract n2 from n1 (n1-n2=n3).
*	(n1 n2 -- n3)	Multiply. n3=n1*n2
/	(n1 n2 -- n3)	Division, signed (n3= n1/n2).
1+	(n -- n+1)	Increment n.
1-	(n -- n-1)	Decrement n.
2+	(n -- n+2)	Add two to n.
2-	(n -- n-2)	Subtract two from n.
2*	(n -- n*2)	Logic left shift.
2/	(n -- n/2)	Logic right shift.
UM+	(n1 n2 -- nd)	Unsigned addition, double precision result.
UM*	(n1 n2 -- nd)	Unsigned multiply, double precision result.
M*	(n n -- d)	Signed multiply. Return double product.
UM/MOD	(nd n1 -- mod quot)	Unsigned division with double precision dividend.
M/MOD	(d n -- mod quot)	Signed floored divide of double by single. Return mod and quotient.
MOD	(n1 n2 -- mod)	Modulus, signed (remainder of n1/n2).
/MOD	(n1 n2 -- mod quot)	Division with both remainder and quotient.
*/MOD	(n1 n2 n3 -- n4 n5)	Multiply and then divide (n1*n2/n3)
*/	(n1 n2 n3 -- n4)	Like */MOD, but with quotient only.
ABS	(n1 -- n2)	If n1 is negative, n2 is its two's complement.
NEGATE	(n1 -- n2)	Two's complement.
MAX	(n1 n2 -- n3)	n3 is the larger of n1 and n2.
MIN	(n1 n2 -- n3)	n3 is the smaller of n1 and n2.
WITHIN	(n1 n2 n3 -- flag)	Return true if n1 is within range of n2 and n3. (n2 <= n1 < n3)
DNEGATE	(d1 -- d2)	Negate double number. Two's complement.
D+	(d1 d2 -- d3)	Add double numbers.
D-	(d1 d2 -- d3)	Subtract double numbers.
D-	(d1 d2 -- d3)	Subtract double numbers.

Logic and Comparison Commands

AND	(n1 n2 -- n3)	Logical bit-wise AND.
OR	(n1 n2 -- n3)	Logical bit-wise OR.
XOR	(n1 n2 -- n3)	Logical bit-wise exclusive OR.
INVERT	(n1 -- n2)	Bit-wise one's complement.
0<	(n -- flag)	True if n is negative.
U<	(n1 n2 -- flag)	True if n1 less than n2. Unsigned compare.
<	(n1 n2 -- flag)	True if n1 less than n2.
=	(n1 n2 -- flag)	True if n1 equals n2.
>	(n1 n2 -- flag)	True if n1 greater than n2.
D>	(d1 d2 -- flag)	True if d1 greater than d2.

RAM Memory Commands

@	(addr -- n)	Replace addr by number at addr.
C@	(addr -- char)	Fetch least-significant byte only.
!	(n addr --)	Store n at addr.
C!	(char addr --)	Store least-significant byte only.
+	(n addr --)	Add n to number at addr.
COUNT	(addr1 -- addr+1 char)	Move string count from memory onto stack.
ALLOT	(n --)	Add n bytes to the RAM pointer DP.
HERE	(-- addr)	Address of next available RAM memory location.
PAD	(-- addr)	Address of a scratch area of at least 64 bytes.
TIB	(-- addr)	Address of terminal input buffer.
CMOVE	(addr1 addr2 n --)	Move n bytes starting at memory addr1 to addr2.
FILL	(addr n char --)	Fill n bytes of memory at addr with char.

Flash Memory Commands

I@	(addr -- n)	Replace addr by number at flash memory addr.
IC@	(addr -- char)	Fetch a byte from flash memory addr.
I!	(n addr --)	Store n at flash memory addr.
ICOUNT	(addr1 -- addr+1 char)	Move string count from flash memory onto stack.
IALLLOT	(n --)	Add n bytes to the flash memory pointer CP.
ITYPE	(addr n --)	Display a string of n characters in flash starting at address addr.
READ	(addr1 addr2 --)	Read 128 bytes from flash memory addr1 to RAM memory addr2.
WRITE	(addr1 addr2 --)	Write 128 bytes from RAM memory addr1 to flash memory addr2.
ERASE	(addr --)	Erase an 128 byte page in flash memory at addr.
FLUSH	(--)	Write modified flash buffers back to flash memory.

System Variables

'BOOT	(-- addr)	Contain address of application command to boot.
BASE	(-- addr)	Contain radix for number conversion
TMP	(-- addr)	Temporary scratch pad
SPAN	(-- addr)	Contain actual number of characters received by EXPECT
>IN	(-- addr)	Contain character offset into the input stream buffer.
#TIB	(-- addr)	Contain current length of terminal input buffer (TIB).
'TIB	(-- addr)	Contain current address of terminal input buffer (TIB)
'EVAL	(-- addr)	Contain interpreter or compiler to evaluate a command.
HLD	(-- addr)	Contain pointer to numeric string under construction.
CONTEXT	(-- addr)	Contain name field address of last command in dictionary
CP	(-- addr)	Contain first free address in flash memory
DP	(-- addr)	Contain first free address in RAM memory
LAST	(-- addr)	Contain name field address of command under compilation

Terminal Input-Output Commands

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

Compiler and Interpreter Commands

:<name>	(--)	Begin a colon definition of <name>.
;	(--)	Terminate execution of a colon definition.
CREATE <name>	(--)	Dictionary entry with no parameter field space reserved.
VARIABLE E <name>	(--)	Defines a variable. At run-time, <name> leaves its address.
CONSTANT T <name>	(n --)	Defines a constant. At run-time, n is left on the stack.
,	(n --)	Compile n to the dictionary in flash memory
IMMEDIATE	(--)	Cause last-defined command to execute even within a colon definition.
COMPILE <name>	(--)	<name> is compiled to dictionary.
[COMPILE <name>	(--)	Immediate command <name> is compiled to dictionary.
LITERAL	(n --)	Compile literal number n. At run-time, n is pushed on the stack.
[(--)	Switch from compilation to interpretation.
]	(--)	Switch from interpretation to compilation.
WORD<text>	(char -- addr)	Get the char delimited string <text> from the input stream and leave as a counted string at addr.
(comment)	(--)	Ignore comment text.
\ comment	(--)	Ignore comment till end of line.
." <text>"	(--)	Compile <text> message. At run-time display text message.
.(<text>)	(--)	Display <text> from the input stream.
\$(<text>)"	(-- addr)	Compile <text> message. At run-time return its address.
ABORT" <text>"	(flag --)	Compile <text> message. At run-time display message and abort if flag is true. Otherwise, ignore message and continue..
COLD	(--)	Start eForth system.
QUIT	(--)	Return to interpret mode, clear data and return stacks.
QUERY	(--)	Accept input stream to terminal input buffer.
NAME>	(addr1 -- addr2)	Traverse name field at addr1 and return code field address

		addr2.
NUMBER?	(addr -- n -1 addr 0)	Convert a number string to integer. Push a flag on tos.
EXECUTE	(addr --)	Execute command definition at addr.
@EXECUTE	(addr --)	Execute command definition whose execution address is in addr.
EXIT	(--)	Terminate execution of a colon definition.

Compiler Structure Commands

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

Utility Commands

'<name>	(-- addr)	Look up <name> in the dictionary. Return execution address.
WORDS	(--)	Display all eForth commands
DUMP	(addr --)	Dump 128 bytes of RAM memory starting from addr.
IDUMP	(addr --)	Dump 128 bytes of flash memory starting from addr.
.S	(--)	Dump the parameter stack.