ARD101 Tutorial Conversion

April 14, 2014

Software Version	Date
0.0.0.2	04/14/14

Contents

1 OSEPP 101 Arduino Basics Starter Kit

This ARD101 project started when Fry's put this kit on sale for $35.99.

`http://osepp.com/products/kits/101-arduino-basic-starter-kit/`

It contains an UNO R3 Plus processor, which is compatible with, at least, Dr. Ting's 328eForth for Arduino.

`http://www.offete.com/328eForth.html`

It is also compatible with SwiftX AVR and probably with amForth, but I haven't gotten the latter working yet.

`http://www.forth.com/downloads/SwiftX-Eval-AVR-3.7.1-f4qbm8hnnrg5r42ko.exe`

`http://amforth.sourceforge.net/`

The Kit also contains all the parts needed to complete 7 different tutorials involving flashing LEDs, playing tones and reading voltages and GPIOs, which can provide a very basic comparison between the C and Forth programming languages. Unfortunately, while the Kit is complete for programming in C, you still need an additional AVR ISP programmer to work with Forth. While Atmel does sell one that is very mature, I decided to try to find one that was a little smaller. After a couple of misfires, I went back to the one recommended by Leon Wagner from Forth, Inc. at the February 2011 SVFIG meeting:

`http://www.forth.org/svfig/kk/02-2011.html`

`http://www.pololu.com/product/1300/`

1.1 Atmel Studio 6

The Kit does require it's own software. ARDUINO 1.0.5-r2 - 2014.01.08 is the latest version, but I ended up using ARDUINO 1.5.5-r2 BETA 2014.01.10 to support my Windows 8.1 computer.

`http://arduino.cc/en/Main/Software`

While SwiftX is self contained, amForth and eForth require a separate assembler and compiler system. The most recent is Atmel's Studio 6 (Version 6.1.2730 - Service Pack 2).

http://www.atmel.com/tools/ATMELSTUDIO.aspx

Getting all of this setup is a much more significant part of the problem than it needs to be, but then again, supporting all of the updates of all of the pieces is certainly an expensive proposition. I just wish more vendors considered it to be a priority. OSEPP is doing a better job than most, but even they are at the mercy of the Arduino open-source community, and Win8.1 is causing most companies issues with their driver security requirements.

http://preview.tinyurl.com/krnp7nv

Atmel has also gone through many variations to its compiler suite, and with this latest version, they chose to use Microsoft's Visual Studio as the IDE for their compiler. They are still stuck in the VS2010 version, but at least they have a reasonable update mechanism and they do provide the older version for the less adventurous. I personally believe in continuous integration, so I still am struggling to figure out how to get amForth to work. Dr. Ting's eForth was not too difficult, but the Pololu programmer instructions have not been updated to the latest menus in Studio 6. The instructions that say to "select Add STK500… from the Tools menu" should actually be to select Add target... and Select the STK500 tool. A minor but frustratingly significant difference.

After having to rebuild my computer and do a quick contract between jobs, I've gotten this setup again, and found the command line syntax to flash without loading the entire IDE. It is:

```
atprogram -t stk500 -c 6 -i ISP -d ATmega328P -v program -f target.hex
```

You may need to adjust some of the parameters for your environment.

1.1.1 328eForth v2.20

Once I finally got everything setup properly, I was able to backup the existing flash image, which is essential if you want to return to the Kit's original Tutorials. However, I lost that backup when I had to rebuild my computer, so I had to find out where the "official" image is called **ATmegaBOOT_168_atmega328_pro_8MHz.hex** and was at:

C:\Arduino\arduino-1.5.5-r2\hardware\arduino\avr\bootloaders\atmega

Dr. Ting's instructions also say to set the High Fuse byte to 0xD8, which I have so far, found to be unnecessary and the original setting of 0xDE (BOOTSZ = 256W_3F00) works the same. Once I flashed the **328eforth.hex** and connected the serial port to PuTTY, I got the following:

```
328eForth v2.20
ok
words
 VARIABLE CONSTANT CREATE IMMEDIATE : ] ; OVERT ." $" ABORT" WHILE ELSE AFT THEN
REPEAT IF AGAIN UNTIL NEXT FOR BEGIN LITERAL COMPILE [COMPILE] , IALLOT ALLOT
D- D+ D> > 2- 2+ 1- 1+ READ WRITE ERASE COLD WORDS .S IDUMP DUMP ' QUIT EVAL [
QUERY EXPECT NAME> WORD CHAR \ ( .( ? . U. U.R .R CR ITYPE TYPE SPACES SPACE KEY
NUMBER? DECIMAL HEX #> SIGN #S # HOLD <# FILL CMOVE @EXECUTE TIB PAD HERE ICOUNT
COUNT +! PICK DEPTH */ */MOD M* * UM* / MOD /MOD M/MOD UM/MOD WITHIN MIN MAX <
U< = ABS - DNEGATE NEGATE INVERT + 2DUP 2DROP ROT ?DUP BL 2/ 2* LAST DP CP
CONTEXT HLD 'EVAL 'TIB #TIB >IN SPAN TMP BASE 'BOOT UM+ XOR OR AND 0< OVER
SWAP DUP DROP >R R@ R> C@ C! FLUSH I! IC@ I@ @ ! EXIT EXECUTE EMIT ?KEYok
ok
```

So now, I can start translating the tutorials.

1.2 SwiftX for AVR

http://www.forth.com/embedded/eval-upgrade.html?MCU=AVR

Initially, I was stuck in the SwiftX AVR Target Reference Manual, Appendix A.1.1 Uno Board Overview. I was trying to get the RELOAD! command to work, but it would not work with the Pololu USB AVR Programmer. Once I got it working as an STK500 in Atmel Studio, and read further in Appendix D: Atmel STK500, I saw that this is the "normal" way to use this interface. So, now I can start including that system in the translation too.

```
SwiftX Evaluation AVR 3.7.0 01-Jan-2014
INCLUDE DEBUG
    Start     End     Size    Used   Unused    Type    Name
     0000    7FFF    32768    7308    25460    CDATA   FLASH
     0100    01FF      256      29      227    IDATA   IRAM
     0200    08FF     1792     421     1371    UDATA   URAM
TARGET READY
SwiftX/AVR Arduino Uno SOS Demo  ok
2 6 + . 8  ok
go
TARGET READY  ok
```

2 Tutorials

Starting from OSEPP's learning center, I have shortened the URL for each of the subsections below to fit on a printed page.

http://osepp.com/learning-centre/start-here/101-basic-starter-kit/

http://tinyurl.com/megobz3

I have also setup the hardware interfaces so that all 7 of the tutorials are connected at the same time. The only overlap that this creates is with the 7 segment LED. This just means that the speaker clicks during the LED tutorial, but otherwise, all of the I/O used in the tutorials functions properly

2.1 Tutorial 1: Loading the First Sketch

http://tinyurl.com/megobz3/tutorial-1-loading-the-first-sketch/

This tutorial looks to be well represented in the **`flasher.txt`** sample that is included with eForth. However, 1st you need to have a terminal emulator that can send text files with a 900 ms delay in between lines. 900 ms is probably way too slow, but the system can not handle no delay between lines. 200 ms seems about right. Unfortunately, PuTTY can not do this. Realterm is a reasonable substitute, but scrollback is an issue.

http://realterm.sourceforge.net/

I've always preferred the capabilities of HyperACCESS, which is the parent of Windows old HyperTerminal. It's expensive, but I've seen issues with just about any other terminal program and I don't recall ever finding an issue with HyerACCESS. They also still sell the original HyperTerminal Private Edition if you want something less expensive.

http://www.hilgraeve.com/hyperaccess-trial/

With decent scroll back capabilities, I was able to see that **`flasher.txt`** required **`io-core.txt`** which was not loading properly. Eventually, I figured out that I needed to load **`marker.txt`** 1st. Even Dr. Ting's documentation has a mistake there in that **`hello-world.txt`** also requires **`marker.txt`**, but in the end, the system can finally be considered to be functional.

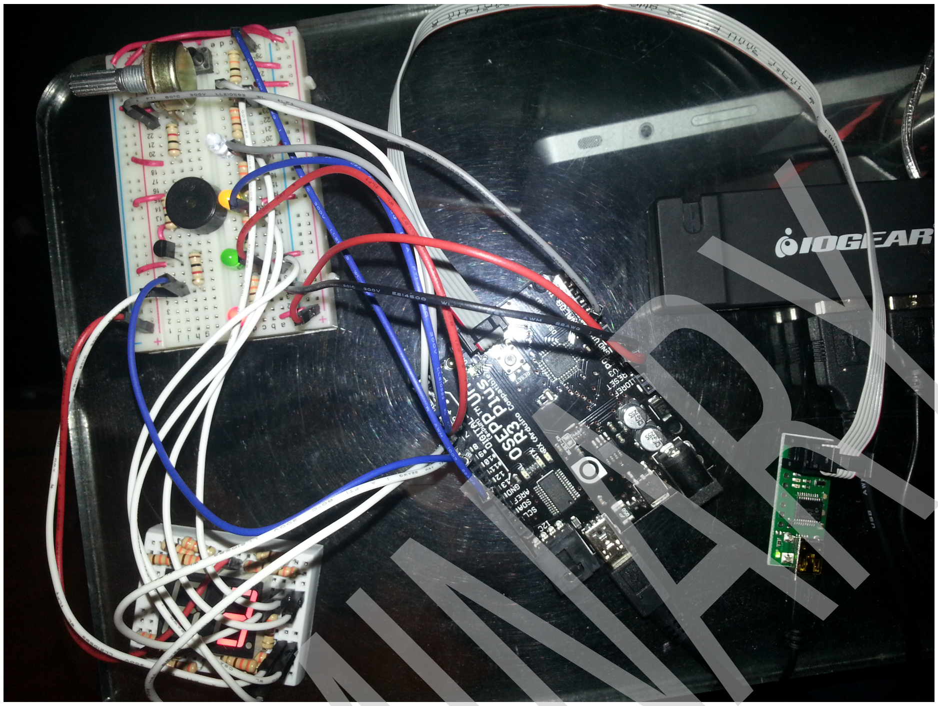

Figure 1:

```
1000 3 manyok
```

Note that the “ok” doesn’t output a space first, so the acknowledgment can be a little confusing.

2.1.1 Tutorial1

5 ⟨*Tutorial1.ino* 5⟩≡

```
/* Tutorial 1
Blink
Turns on an LED on for one second, then off for one second, repeatedly.
This example code is in the public domain.
*/
// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
int led = 13;
// the setup routine runs once when you press reset:
void setup() {
// initialize the digital pin as an output.
pinMode(led, OUTPUT);
}
// the loop routine runs over and over again forever:
void loop() {
```

```
digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
delay(1000);               // wait for a second
digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
delay(1000);               // wait for a second
}
```

2.1.2 flasher.txt

William F. Ragsdale had written these demo applications for Arduino with AmForth. Dr. Ting modified them so that they work properly under 328eForth.

6 ⟨*flasher.txt* 6⟩≡

```
\ FLASHER.txt to Demo LED control               WFR 2011-01-27
( must have io-core.txt installed )
chop-flasher
marker chop-flasher  ( a forget point)
$23 value PortB  $26 value PortC  $29 value PortD
  5 value LED
: 1-cycle  ( ms_delay ---   flash LED on then off )
    PortB LED PoBiHi   dup ms   PortB LED PoBiLo   ms ;
: many  ( on_time flashes ---   produce controlled LED flashes)
    PortB LED PoBiOut ( set LED pin as output)
    for aft  dup 1-cycle then next drop ;
( use 'many' leading with on-time and # of flashes )
( end of flasher.txt )
flush
```

Note that in both C and Forth, there are many support routines that are not always listed. Knowing the environment is always key to your productivity and you can usually learn a lot by examining the sample source listings. You should also notice that the **`setup`** and **`loop`** functions need to be done explicitly in Forth. You can always assume that you need to do those steps, but how often and in what order is typically, an application specific requirement. Thus the use of parameters is more typical in Forth than infinite loops.

You should also notice that while the C routines deal with sequential pins which span multiple ports, the Forth routines deal with the port bits directly. The onboard LED is on pin 13, but it can also be referenced as bit 5 on Port B. The lower 8 pins are on Port D.

2.1.3 distress.f

An onboard LED example is also included in SwiftX AVR, but it is listed as Copyright 2001-2007 FORTH, Inc. You should look at it and execute it to make sure everything is working properly. However, I will not list it here to avoid any copyright infringement. It is interesting to note that this example puts the **`SOS`** distress code into a background **`BEACON`** task, which allows it to continue running while you continue to exercise the tether interface. This is extremely useful, but for compatibility, I will not use it for the rest of these tutorials. Instead, I will attempt to use the same code on all of the Forth systems.

However, there will be some differences, and I will need a modified version of the io-core.f support from eForth:

7 ⟨*io-core.f* 7⟩≡

```
\ Port Input Output for AmForth                    DaR 28Mar14
\ loaded as  io-core.f
\ Modified for 328eForth, 23mar11cht
\ Modified for SwiftX, daruffer@gmail.com

\ manually begin with chop-io entered

: mask ( bit# --- port_mask  convert bit to 8 bit mask)
    1 swap lshift  ;

: DDR  \ port --- port' adjust input port# to DDR
   1- ;

: Input \ port --- port adjust input port# to output
   2 - ;

: RegFrom \ Reg mask --- value  read masked bits from register
      \ To read all bits:  PortB true RegFrom -> value
    swap c@  and ;

: RegTo \ Reg mask new ---  write masked new into register
    over and >r   invert over c@ and   r> or  swap c! ;

: PoBiI/O  \ port bit direction --- configure bit in/out
   rot DDR   rot mask   rot RegTo

: PoBiOut  \ port bit --- configure as output
```

```
   true  PoBiI/O ;

 : PoBiRead  \ port bit --- value  read bit value from port
    swap Input swap mask RegFrom ;

 : PoBiHi  \ port bit --- set port bit 0..7 high
    mask true  RegTo ;

 : PoBiLo   \ port bit --- clear port bit 0..7 low
    mask false RegTo ;

 : PoBiIn   \ port bit --- configure as input,  no pull-up
   2dup false PoBiI/O   PoBiLo ( pullup inactive) ;

 : PoBiInPu \ port bit --- configure as input with pull-up
   2dup false PoBiI/O   PoBiHi ( pullup active) ;

 \ read bits from register  Reg#         select      RegFrom
 \ write bits to register   Reg#         select bits RegTo
 \ write 1-bit to register  Reg#   5 mask true       RegTo
 \ write 0-bit to register  Reg#   5 mask false      RegTo
 \ configure bits as output PortB DDR    select True RegTo
 \ write bits to output     PortB Output select bits RegTo
 \ configure bit as output  PortB     LED            PoBiOut
 \ bit as input with pullup PortB     Switch3        PoBiInPu
 \ read bit from port       PortB     Switch3        PoBiRead
 \ write 1-bit to port      PortB     LED            PoBiHi
 \ write 0-bit to port      PortB     LED            PoBiLo
 \ Note, when initializing a 16 bit register, TCNT1 etc. it
 \  must be written directly hi/lo not using RegTo.
 \  The proper form to clear is:    TCNT1hi false c!
 \                          then:    TCNT1lo false c!
 \

 \ end of io-core.txt
```

Note that the differences are:

1. The **`marker`** concepts don't really apply here.
2. The port names refer to the output port, rather than the input port. Thus, the adjustments are reversed.
3. The **`flush`** concept used by eForth also doesn't apply here.

2.2 Tutorial 2: Controlling Digital Outputs

http://tinyurl.com/megobz3/tutorial-2-controlling-digital-outputs/

500 3 cyclesok

2.2.1 Tutorial2

9 ⟨*Tutorial2.ino* 9⟩≡

```
/*
Tutorial 2 Digital Output
*/
int LED0  = 2;     // Use digital pin 2 to drive the white LED
int LED1  = 3;     // Use digital pin 3 to drive the yellow LED
int LED2  = 4;     // Use digital pin 4 to drive the green LED
int LED3  = 5;     // Use digital pin 5 to drive the red LED
void setup() {
  // initialize digital pin 2 to 5 as output:
  pinMode(LED0, OUTPUT);
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT);
}
void loop(){
  // Toggle each LED at a time with a 500ms delay
  digitalWrite(LED0, HIGH);
  delay(500);
  digitalWrite(LED0, LOW);
  delay(500);
  digitalWrite(LED1, HIGH);
  delay(500);
  digitalWrite(LED1, LOW);
  delay(500);
  digitalWrite(LED2, HIGH);
  delay(500);
  digitalWrite(LED2, LOW);
  delay(500);
  digitalWrite(LED3, HIGH);
  delay(500);
```

```
  digitalWrite(LED3, LOW);
  delay(500);
}
```

2.2.2 cycle.txt

10 ⟨*cycle.txt* 10⟩≡

```
\ cycle.txt to Demo multiple LED control      DaR 2014-02-16
chop-cycle
marker chop-cycle
\ Define LED port bits and flashing order
CREATE LEDS   4 2* 1 + allot \ Number of LEDS, then order
: \LEDS ( --- Initialize the RAM array )   4 LEDS C!
   LEDS count 2* PortD fill \ Overfill the Port addresses to save code
   2 LEDS 1 + C!  3 LEDS 3 + C!  4 LEDS 5 + C!  5 LEDS 7 + C! ;
: cycle ( time port bit --- flash LED on then off )
   2dup PoBiHi  rot dup ms  rot rot PoBiLo  ms ;
: cycles ( time cycles --- produce cycles of LED flashes )   \LEDS
   LEDS count for aft  count >r count r> PoBiOut  then next drop
   for aft  LEDS count for aft  count >r count >r over r> r>
         cycle  then next  drop  then next  drop ;
flush
```

Note a few principles here:

1. Look for patterns of doing things 3 or more times and factory them out. Each LED name was used 3 times, which leads to putting them into an array, which only needs to be referenced twice. This also allows a significant reduction in code size.

2. Be careful where things are compiled when systems have multiple address spaces. I had thought that `,` would work to create the table, but no, I had to resort to a much less elegant solution.

3. Still, the lack of elegance is at compile time and does not effect the run time behavior. That makes it much less objectionable.

4. Last minute uglyness is the requirement for a **`flush`** before **`LEDS`** can be referenced. Otherwise, the system would reboot while compiling this code. That's the risk for compile time initialization and thus, why it is now in a definition and called everytime **`cycles`** starts up. This gives it some runtime overhead, but saves compatibility issues with other systems.

5. Don't be afraid of passing multiple parameters. Up to 3 parameters are easily handled in Forth and even more can be handled with minimal difficulties. Watch for literals or fixed values that might change over time, like the LED parameters here in the **`cycle`** routine. The original **`1-cycle`** routine could have been written this way with some forethought.

6. Know when to stop factoring things out. I could have broken **`cycles`** down into, at least, 2 other words. However, again you should remember the rule of 3. I might use a similar pattern 1 more time in the next tutorial, but as with most test code, a 3rd time is unlikely.

2.2.3 cycle.f

11 ⟨*cycle.f* 11⟩≡

```
\ cycle.f to Demo multiple LED control      DaR 2014-03-29
\ Define LED port bits and flashing order
CREATE LEDS   4 2* 1 + allot \ Number of LEDS, then order
: \LEDS ( --- Initialize the RAM array )   4 LEDS C!
   LEDS count 2* PortD fill \ Overfill the Port addresses to save code
   2 LEDS 1 + C!  3 LEDS 3 + C!  4 LEDS 5 + C!  5 LEDS 7 + C! ;
: cycle ( time port bit --- flash LED on then off )
   2dup PoBiHi  rot dup ms  rot rot PoBiLo  ms ;
: cycles ( time cycles --- produce cycles of LED flashes )   \LEDS
   LEDS count 0 do  count >r count r> PoBiOut  loop  drop
   0 do  LEDS count 0 do  count >r count >r over r> r>
         cycle  loop  drop  loop  drop ;
```

Note that the differences are:

1. The **marker** and **flush** concepts don't really apply here.
2. The **for aft ... then next** structure is directly replaced with the more standard **0 do ... loop** structure.

2.3 Tutorial 3: Using Digital Input

http://tinyurl.com/megobz3/tutorial-3-using-digital-input/

2.3.1 Tutorial3

12 ⟨*Tutorial3.ino* 12⟩≡

```
/*
Tutorial 3 Digital Input
*/
const int buttonPin = 12;     // Use digital pin 12 for the button pin
int buttonState = 0;          // variable for storing the button status
void setup() {
  // initialize the pushbutton pin as an input:
  pinMode(buttonPin, INPUT);
  // initialize the serial port;
  Serial.begin(9600);  // start serial for output
}
void loop(){
  // read the state of the pushbutton value:
  buttonState = digitalRead(buttonPin);
  // Output button state
  Serial.print("The button state is ");
  Serial.println(buttonState);
  // Delay 1000ms
  delay(1000);
}
```

2.3.2 button.txt

13a ⟨*button.txt* 13a⟩≡

```
\ button.txt to Demo Digital input      DaR 2014-02-17
chop-button
marker chop-button
4 value buttonPin \ Use digital pin 12 for the button pin
: states ( --- read state of button )
   PortB buttonPin 2dup PoBiIn  PoBiRead
   begin  PortB buttonPin PoBiRead  2dup - if
         cr ." The button state is "  dup . swap
   then  drop  ?key until  drop ;
flush
```

A few more principles:

1. For testing words, like these, it is often convenient to just wait for a key press to terminate the loop. You have an interactive terminal loop running anyway. You might just as well use it. However, be warned that eForth appears to have a bug with **`until`**. The **`drop`**, or anything else after **`until`**, never executes. Not a big problem here, and I have reported it.
2. We also don't need to initialize the serial port because it is the terminal loop. I suspect that this may not always be the case.
3. Don't add things that you don't use. Note that **`buttonState`** is not needed in Forth, when the stack can hold the state.
4. Don't time a polled event if you don't need to. There's no need to report the state unless it changes.

2.3.3 button.f

13b ⟨*button.f* 13b⟩≡

```
\ button.f to Demo Digital input      DaR 2014-03-29
4 value buttonPin \ Use digital pin 12 for the button pin
: states ( --- read state of button )
   PortB buttonPin 2dup PoBiIn  PoBiRead
   begin  PortB buttonPin PoBiRead
      2dup = while  nip  repeat
   cr ." The button state is " . drop ;
```

Note that the differences are:

1. The **`marker`** and **`flush`** concepts don't really apply here.
2. Since I don't have serial port support in SwiftX, I only loop until the button state changes.

2.4 Tutorial 4: An LED Game

http://tinyurl.com/megobz3/tutorial-4-an-led-game/

2.4.1 Tutorial4

14 ⟨*Tutorial4.ino* 14⟩≡

```
/*
  Tutorial 4 Digital Input and Output Game
  In this game, the LED will loop from white, yellow, green, red
  then back to white.  The goal is to press the push button at the exact
  moment when the green LED is ON. Each time you got it right, the LED
  will speed up and the difficulty will increase.
*/
int currentLED = 2;
int delayValue = 200;
void setup() {
  // initialize digital pin 12 as input;
  pinMode(12, INPUT);   // button input
  // initialize digital pin 2 to 5 as output:
  pinMode(2, OUTPUT);   // white LED
  pinMode(3, OUTPUT);   // yellow LED
  pinMode(4, OUTPUT);   // green LED
  pinMode(5, OUTPUT);   // red LED
}
int checkInput() {
  if (digitalRead(12) == 0) {
    return 1;
  } else {
    return 0;
  }
}
void loop(){
  // Check if the button is press at the right moment
  if (digitalRead(12) == 0) {
    if (currentLED == 4) {
       // Blink the correct (green) LED
       digitalWrite(4, HIGH);
       delay(200);
       digitalWrite(4, LOW);
       delay(200);
       digitalWrite(4, HIGH);
       delay(200);
       digitalWrite(4, LOW);
       delay(200);
       // Speed up the LEDs
       delayValue = delayValue - 20;
    } else {
       // Blink the wrong LED
       digitalWrite(currentLED, HIGH);
```

```
        delay(200);
        digitalWrite(currentLED, LOW);
        delay(200);
        digitalWrite(currentLED, HIGH);
        delay(200);
        digitalWrite(currentLED, LOW);
        delay(200);
      }
    }
    // Loop LED from white > yellow > green > red
    digitalWrite(currentLED, HIGH);
    delay(delayValue);
    digitalWrite(currentLED, LOW);
    delay(delayValue);
    currentLED = currentLED + 1;
    if (currentLED > 5) {
      currentLED = 2;
    }
  }
```

2.4.2 game.txt

15 ⟨*game.txt* 15⟩≡

```
\ game.txt Digital Input and Output Game      DaR 2014-02-20
chop-game
marker chop-game
variable delayValue
: game ( --- cycles LEDs and check button presses )   \LEDS
   LEDS count for aft  count >r count r> PoBiOut  then next drop
   PortB buttonPin PoBiIn  200 delayValue !
   begin  LEDS count for aft  count >r count r>
         2dup delayValue @ rot rot cycle
         PortB buttonPin PoBiRead 0 = if
            rot dup LEDS count 1 - 2* + = if
               delayValue @ 20 - dup 0 = if
                  ." You win!" 2drop drop exit
               then  delayValue !
            then  rot rot
            2dup 200 rot rot cycle
            2dup 200 rot rot cycle
         then
         2drop  ?key if
            drop exit
         then
   then next  drop  again ;
flush
```

Things to note here:

1. A pointer can easily serve as an index. You just have to use something a little less opaque than a number for comparison. Typically, that comparison value can be computed, which is certainly a requirement for using this technique.
2. The use of multiple **exit**s with an endless **again** loop is common in Forth and not something that should be frowned upon as it is with other languages.
3. Unfortunately, this technique appears to also have an issue, like **until** does, as was discussed earlier. In this case, the chip reboots as soon as a key is pressed, or when you win. In the later case, the message doesn't even get a chance to finish.

2.4.3 game.f

16 ⟨*game.f* 16⟩≡

```
\ game.f Digital Input and Output Game      DaR 2014-03-29
variable delayValue
: game ( --- cycles LEDs and check button presses )   \LEDS
   LEDS count 0 do  count >r count r> PoBiOut  loop  drop
   PortB buttonPin PoBiIn  200 delayValue !
   begin  LEDS count 0 do  count >r count r>
         2dup delayValue @ rot rot cycle
         PortB buttonPin PoBiRead 0 = if
            rot dup LEDS count 1 - 2* + = if
               delayValue @ 20 - dup 0 = if
                  ." You win!" 2drop drop exit
               then  delayValue !
            then  rot rot
            2dup 200 rot rot cycle
            2dup 200 rot rot cycle
         then  2drop
      loop  drop
   again ;
```

Note that the differences are:

1. The **marker** and **flush** concepts don't really apply here.
2. The **for aft ... then next** structure is directly replaced with the more standard **0 do ... loop** structure.
3. Since I don't have serial port support in SwiftX, I loop until the CPU is reset.
4. It appears that the eForth reboot also effects this system, but I'm not sure how pervasive it is yet.

2.5 Tutorial 5: Building Voltage Meter

http://tinyurl.com/megobz3/tutorial-5-building-voltage-meter/

2.5.1 Tutorial5

17a ⟨*Tutorial5.ino* 17a⟩≡

```
/*
  Tutorial 5: Volt Meter
*/
int sensorPin = A0;    // select the analog input pin
int sensorValue = 0;   // variable to store the value coming from the sensor
float sensorVoltage = 0; // variable to store the voltage coming from the sensor
void setup() {
  Serial.begin(9600);  // start serial for output
}
void loop() {
  // Read the value from the analog input pin
  // A value of 1023 = 5V, a value of 0 = 0V
  int sensorValue = analogRead(sensorPin);
  // Convert sensor value to voltage
  float sensorVoltage= sensorValue*(5.0/1023.0);
  // print sensor value
  Serial.print("The voltage is ");
  Serial.println(sensorVoltage);
  // delay by 1000 milliseconds:
  delay(1000);
}
```

2.5.2 volts.f

17b ⟨*volts.f* 17b⟩≡

```
\ volts.f to Demo Analog input      DaR 2014-03-29
0 value analogPin \ Use analog pin 0 for the voltage pin
: volts ( --- read state of pot )
   PortC analogPin PoBiInPu
   $40 ADMux c!  $C3 ADCSra c!  ADCL @
   cr ." The voltage is "  500 1023 */
   0 <# # # $2E hold # #> type space ;
```

Notes for the SwiftX version only:

1. Although the eForth manual does have a section regarding analog inputs, I chose to use the SwiftX model after losing my work in a computer rebuild.
2. While the C version used floating point operators, this is often overkill for embedded systems. Fixed point math allows you to use the faster integer math operators and you simply need to keep track of where the decimal point is. Typically, the only place where this information is needed is when the value is displayed (see the **`$2E hold`** above). You do need to be concerned with the range of the value, which is why the */ operator uses a double-cell intermediate result.
3. Typically, you also would want to separate the voltage conversion from the display output, but for simplicity, I did not do that here.

2.6 Tutorial 6: Using Buzzer to Play a Melody

http://tinyurl.com/megobz3/tutorial-6-using-buzzer-to-play-a-melody/

2.6.1 Tutorial6a

18 ⟨*Tutorial6a.ino* 18⟩≡

```
/*   Tutorial 6a: Simple Scale Sweep */
int buzzerPin = 8;    // Using digital pin 8
#define NOTE_C6  1047
#define NOTE_D6  1175
#define NOTE_E6  1319
#define NOTE_F6  1397
#define NOTE_G6  1568
#define NOTE_A6  1760
#define NOTE_B6  1976
#define NOTE_C7  2093
void setup() {
  // nothing to setup
}
void loop() {
  //tone(pin, frequency, duration)
  tone(buzzerPin, NOTE_C6, 500);
  delay(500);
  tone(buzzerPin, NOTE_D6, 500);
  delay(500);
  tone(buzzerPin, NOTE_E6, 500);
  delay(500);
  tone(buzzerPin, NOTE_F6, 500);
  delay(500);
  tone(buzzerPin, NOTE_G6, 500);
  delay(500);
  tone(buzzerPin, NOTE_A6, 500);
  delay(500);
  tone(buzzerPin, NOTE_B6, 500);
  delay(500);
  tone(buzzerPin, NOTE_C7, 500);
  delay(500);
}
```

2.6.2 tone.f

The eForth system ha this tone generation code, which uses pin 6 on PortD, rather than pin 0 on PortB. So far, I have not figured out how to generate a tone on PortB, so I have simply switched the buzzer over to the bit used by eForth. This really is so much easier than figuring out the code.

19 ⟨*tone.f* 19⟩≡

```
\ Audio tone generator                        30Mar14
\ Modified for 328eForth, 23mar11cht
\ Modified for SwiftX by daruffer@gmail.com

\ Must have io-core.f installed

6 value Tone-out \ PortD bit 6

binary

: setup-osc \ prescale limit  ---  limit 1..255, prescale 1..5
   PortD   Tone-out                  PoBiOut \ setup output pin
   OCR0A   true      rot ( limit   ) RegTo
   TCCR0A  11000011  01000010        RegTo   \ CTC mode
           00000101  min    0  max    \ form TCCR0B prescale
   TCCR0B  00001111 rot ( prescale ) RegTo ;  \ and tone on

: tone-off \  ---  end output tone setting prescale to zeros
   TCCR0B  00000111  false RegTo ;

decimal

78 value Limit   4 value Prescale  \ 400 Hz tone parameters

: ud/mod ( ud1 n -- rem ud2 ) >R 0 R@ UM/MOD R> SWAP >R UM/MOD R> ;
: Hertz   \ frequency --- load Limit and Prescale
    $1200 $7A ( 8000000. ) rot ud/mod  \ total scale as rem double-quot
    dup            if ( >16 bits)  1024  5 else
    over $C000 and if ( >14 bits)   256  4 else
    over $F800 and if ( >10 bits)    64  3 else
    over $FF00 and if ( > 8 bits)     8  2 else   1  1
                      then then then then
      to Prescale   um/mod to Limit  drop drop ( two remainders ) ;

: tone-on    \ ---  begin tone from fixed presets
     Prescale  Limit  setup-osc ;

: note  \ duration ---    generate timed tone for duration msec.
     tone-on  ms  tone-off ;
```

```
\ End of tone.f
```

Notes for the SwiftX version only:

1. The **`marker`** and **`flush`** concepts don't really apply here.
2. The SwiftX system defines all of the ports as compile time **`EQU`** constants, so I do not need to define them here.
3. I have not defined a way to compile the musical notes yet, but the Ring Tone Text Transfer Language looks interesting. See: `http://www.srtware.com/index.php?/ringtones/rtttlformat.php`

2.6.3 scale.f

20 ⟨*scale.f* 20⟩≡

```
\ Play musical scale                         13Apr14

CREATE scale \ sequence of notes, pauses and times
   1047 , 500 , 0 , 500 , 1175 , 500 , 0 , 500 ,
   1319 , 500 , 0 , 500 , 1397 , 500 , 0 , 500 ,
   1568 , 500 , 0 , 500 , 1760 , 500 , 0 , 500 ,
   1976 , 500 , 0 , 500 , 2093 , 500 , 0 , 500 ,
      0 ,    0 ,  \ Null terminators

: notes ( a -- ) \ Play sequence of notes, pauses and times
   begin  dup 2@  2dup or while  ?dup if
         Hertz note  else  ms
      then  2 cells +
   repeat  drop ;
```

Notes for the SwiftX version only:

1. The only purpose here is to define **notes** to process the sequence of notes. The sequences of notes are not things that need to be created within the target. These, even define tables are pretty much pointless.
2. However, this did reveal that SwiftX is using timer 0 to support the **ms** and **counter ... timer** routines. Once **tone-on** is executed **counter** no longer changes and **ms** hangs until the board is reset.

2.6.4 Tutorial6b

21 ⟨*pitches.h* 21⟩≡

```
/*************************************************
 * Public Constants
 *************************************************/
#define NOTE_B0  31
#define NOTE_C1  33
#define NOTE_CS1 35
#define NOTE_D1  37
#define NOTE_DS1 39
#define NOTE_E1  41
#define NOTE_F1  44
#define NOTE_FS1 46
#define NOTE_G1  49
#define NOTE_GS1 52
#define NOTE_A1  55
#define NOTE_AS1 58
#define NOTE_B1  62
#define NOTE_C2  65
#define NOTE_CS2 69
#define NOTE_D2  73
#define NOTE_DS2 78
#define NOTE_E2  82
#define NOTE_F2  87
#define NOTE_FS2 93
#define NOTE_G2  98
#define NOTE_GS2 104
#define NOTE_A2  110
#define NOTE_AS2 117
#define NOTE_B2  123
#define NOTE_C3  131
#define NOTE_CS3 139
#define NOTE_D3  147
#define NOTE_DS3 156
#define NOTE_E3  165
#define NOTE_F3  175
#define NOTE_FS3 185
#define NOTE_G3  196
```

```
#define NOTE_GS3 208
#define NOTE_A3  220
#define NOTE_AS3 233
#define NOTE_B3  247
#define NOTE_C4  262
#define NOTE_CS4 277
#define NOTE_D4  294
#define NOTE_DS4 311
#define NOTE_E4  330
#define NOTE_F4  349
#define NOTE_FS4 370
#define NOTE_G4  392
#define NOTE_GS4 415
#define NOTE_A4  440
#define NOTE_AS4 466
#define NOTE_B4  494
#define NOTE_C5  523
#define NOTE_CS5 554
#define NOTE_D5  587
#define NOTE_DS5 622
#define NOTE_E5  659
#define NOTE_F5  698
#define NOTE_FS5 740
#define NOTE_G5  784
#define NOTE_GS5 831
#define NOTE_A5  880
#define NOTE_AS5 932
#define NOTE_B5  988
#define NOTE_C6  1047
#define NOTE_CS6 1109
#define NOTE_D6  1175
#define NOTE_DS6 1245
#define NOTE_E6  1319
#define NOTE_F6  1397
#define NOTE_FS6 1480
#define NOTE_G6  1568
#define NOTE_GS6 1661
#define NOTE_A6  1760
#define NOTE_AS6 1865
#define NOTE_B6  1976
#define NOTE_C7  2093
#define NOTE_CS7 2217
#define NOTE_D7  2349
#define NOTE_DS7 2489
#define NOTE_E7  2637
#define NOTE_F7  2794
```

```
#define NOTE_FS7 2960
#define NOTE_G7  3136
#define NOTE_GS7 3322
#define NOTE_A7  3520
#define NOTE_AS7 3729
#define NOTE_B7  3951
#define NOTE_C8  4186
#define NOTE_CS8 4435
#define NOTE_D8  4699
#define NOTE_DS8 4978
```

23 ⟨*Tutorial6b.ino* 23⟩≡

```
/* Tutorial 6b: Playing an Melody */
#include "pitches.h"
// notes in the melody:
int melody[] = {    NOTE_C4, NOTE_G3,NOTE_G3, NOTE_A3, NOTE_G3,0, NOTE_B3, NOTE_C4};
// note durations: 4 = quarter note, 8 = eighth note, etc.:
int noteDurations[] = {4, 8, 8, 4,4,4,4,4 };
void setup() {
  // iterate over the notes of the melody:
  for (int thisNote = 0; thisNote < 8; thisNote++) {
    // to calculate the note duration, take one second
    // divided by the note type.
    //e.g. quarter note = 1000 / 4, eighth note = 1000/8, etc.
    int noteDuration = 1000/noteDurations[thisNote];
    tone(8, melody[thisNote],noteDuration);
    // to distinguish the notes, set a minimum time between them.
    // the note's duration + 30% seems to work well:
    int pauseBetweenNotes = noteDuration * 1.30;
    delay(pauseBetweenNotes);
    // stop the tone playing:
    noTone(8);
  }
}
void loop() {
  // no need to repeat the melody.
}
```

2.7 Tutorial 7: Counting Down with a 7 Segment LED

http://tinyurl.com/megobz3/tutorial-7-counting-down-with-a-7-segment-led/

2.7.1 Tutorial7

24 ⟨*Tutorial7.ino* 24⟩≡

```
//  Tutorial 7: 7 Segment LED
//
// Define the LED digit patterns, from 0 - 9
// Note that these patterns are for common anode displays
// 0 = LED on, 1 = LED off:
// Digital pin: 2,3,4,5,6,7,8
//              a,b,c,d,e,f,g
byte seven_seg_digits[10][7] = { { 0,0,0,0,0,0,1 },  // = 0
                                 { 1,0,0,1,1,1,1 },  // = 1
                                 { 0,0,1,0,0,1,0 },  // = 2
                                 { 0,0,0,0,1,1,0 },  // = 3
                                 { 1,0,0,1,1,0,0 },  // = 4
                                 { 0,1,0,0,1,0,0 },  // = 5
                                 { 0,1,0,0,0,0,0 },  // = 6
                                 { 0,0,0,1,1,1,1 },  // = 7
                                 { 0,0,0,0,0,0,0 },  // = 8
                                 { 0,0,0,1,1,0,0 }   // = 9
                               };
void setup() {
  pinMode(2, OUTPUT);
  pinMode(3, OUTPUT);
  pinMode(4, OUTPUT);
  pinMode(5, OUTPUT);
  pinMode(6, OUTPUT);
  pinMode(7, OUTPUT);
  pinMode(8, OUTPUT); }
void sevenSegWrite(byte digit) {
  byte pin = 2;
  for (byte segCount = 0; segCount < 7; ++segCount) {
    digitalWrite(pin, seven_seg_digits[digit][segCount]);
    ++pin;
  }
}
void loop() {
  for (byte count = 10; count > 0; --count) {
   delay(1000);
   sevenSegWrite(count - 1);
  }
  delay(3000);
```

```
}
```

2.7.2 count.f

25 ⟨*count.f* 25⟩≡

```
\ count.f to Demo 7-segment LED control      DaR 2014-04-08
\ Define LED segment port bits and segment order
CREATE LEDS   7 2* 1 + allot \ Number of segments, then order
: \LEDS ( --- Initialize the RAM array )   7 LEDS C!
   LEDS count 2* PortD fill \ Overfill the Port addresses to save code
   2 LEDS 1 + C!  3 LEDS 3 + C!  4 LEDS 5 + C!  5 LEDS 7 + C!
   6 LEDS 9 + C!  7 LEDS 11 + C!  0 LEDS 13 + C!
   PortB LEDS 14 + C! ; \ Last segment on PortB
CREATE SEGS \ Try flash based array on SwiftX
   0 c, 0 c, 0 c, 0 c, 0 c, 0 c, 1 c, \ = 0
   1 c, 0 c, 0 c, 1 c, 1 c, 1 c, 1 c, \ = 1
   0 c, 0 c, 1 c, 0 c, 0 c, 1 c, 0 c, \ = 2
   0 c, 0 c, 0 c, 0 c, 1 c, 1 c, 0 c, \ = 3
   1 c, 0 c, 0 c, 1 c, 1 c, 0 c, 0 c, \ = 4
   0 c, 1 c, 0 c, 0 c, 1 c, 0 c, 0 c, \ = 5
   0 c, 1 c, 0 c, 0 c, 0 c, 0 c, 0 c, \ = 6
   0 c, 0 c, 0 c, 1 c, 1 c, 1 c, 1 c, \ = 7
   0 c, 0 c, 0 c, 0 c, 0 c, 0 c, 0 c, \ = 8
   0 c, 0 c, 0 c, 1 c, 1 c, 0 c, 0 c, \ = 9
: sevenSegWrite ( digit --- turn on the segments for the given digit )
   LEDS count  rot over * SEGS +  -rot 0 do
      swap count >r  swap count >r count r>  r> if
         PoBiHi  else  PoBiLo
      then
   loop  2drop ;
: counts ( time cycles --- produce cycles of digits )   \LEDS
   LEDS count 0 do  count >r count r> PoBiOut  loop  drop
   0 do  10 dup 0 do  dup i - 1- sevenSegWrite  over ms
   loop  drop  loop  drop ;
```

Notes for the SwiftX version only:

1. Although I used the same RAM based array as I did in the **`cycle`** tutorial, I was able to use a “proper” flash based array in SwiftX.

3 Document Processing

A script for converting this document to PDF form:

26a ⟨*final* 26a⟩≡
```
lyx -e literate $1.lyx
noweave -delay -index "$1.nw" > "$1.tex"
pdflatex $1 latex=pdflatex
```

For marking the output with a PRELIMINARY watermark:

26b ⟨*preliminary* 26b⟩≡
```
echo "Make $1 PDF release notes..."
./final $1
pdftk $1.pdf stamp Preliminary.pdf output out.pdf
rm $1.pdf
mv out.pdf $1.pdf
```

Each of these scripts can be pulled out manually given the default * script defined below.

26c ⟨* 26c⟩≡
```
echo "Extract script $2 from $1.lyx..."
rm -f $1.nw
lyx -e literate $1.lyx
notangle -t4 -R$2 $1.nw > $2
chmod a+x $2
```

Once that script is pulled out and named extract, the following script can pull out all of the other scripts:

26d ⟨*extract-all* 26d⟩≡
```
echo "Extract all scripts..."
sedArgs="s/\(.*\)\.idx:.*entry{\(.*\)|hyper.*/\1 \2/g"
find . -type f -name \*idx |\
xargs grep "indexentry" |\
sed -e "$sedArgs" |\
xargs -n 2 ./extract
```

Index