Initial PWM Test

Dennis Ruffer

18 May 2012

Abstract

These are some notes from my 1st experiences on Green Array's Evaluation Board (EVB001 rev 0.1.1). I will try to explain things precisely, so that others can follow after me. Although I have worked with earlier versions of Chuck's chip, I have not used the ColorForth IDE. This means that I am learning just about as much as most readers of this document and, as they recommend in the arrayForth User's Manual (DB004 Revised 12/23/11), I will be making small changes, saving often and testing every change.

Contents

1 Setup

I have already soldered all the extra parts to the board and bought some additional 1-up pins, so that I can connect female patch wires to just about anything on the board. I have executed the **`selftest`** and **`autotest`** commands from the Getting Started Application Note (AN004 rev 09/02/11), so I know my board is working. I have setup my Win7 computer to talk to the target chip on USB port C, so I don't have to worry too much about interfering with any other IDE features, nor do I have to actually use many of them.

2 Practical Example

The example that is delivered with the EVB001 uses the host chip, so it must be changed to work on the target chip. This is easy enough, just use the colorForth editor to change block 844 to use `c-com` rather than `a-com` and `c-bps` rather than `a-bps`. This is a good time to start learning the editor and there are ample tutorials for doing that, so I will not go into it here. Once done, you can follow Section 9 Practical Example of the User's Manual and see pin 600.17 light up on the target chip. Make sure you figure out how to **`save`** those changes, unless you really like to do these changes every time you boot. You will need to know this as we move forward.

Additionally, figure out how to backup the **`OkadWork.cf`** file frequently. Invariably, you will end up making some change that causes colorForth to no longer boot, as I just did. I didn't have a backup, so I had to type my code back in and get it back to the point I was working on. This served a valuable purpose to review the code and fix some typos, but it's painful, and you may not be writing down everything that you are doing as I am right now. So, do yourself a favor and backup your work!

3 Color Blindness

Before I start to put much more ColorForth code into this document, I should explain that I am red/green color blind. This means that I do not see, or react to color like most people do. 7% of males have this condition, as well as other people who do not perceive color the same way as others do. I use a program called eyePilot (Version 1.0.12 from Tenebraex) so I can figure out what colors ColorForth is using. I don't always need it, but frequently, `yellow` and **`green`** look far to much alike and on the block I will be using in a few moments, I see that the green component can have an RGB value of 192 or 255. I see that the User's manual explains that these are HEX numbers, but this does not make using colorForth any easier for me.

Additionally, tools that I tend to rely on in my programming, have not caught up with the use of color in source code. HTML editors are getting closer, and colorForth even has an HTML listing utility, but I have not found them good enough yet. Most explicitly, the use of Literate Programming that I will be using in this documentation can not be done with color attributes yet. Some day, the rest of us might catch up with where Chuck wants us to be, but at the moment, I am not there yet.

Therefore, for the rest of this document, I will be using an ASCII translation of the syntax used by colorForth. This makes the code look very similar to ANS Forth, but do not be mistaken, it will not run in any other version of Forth that I know about. Here is the translation matrix:

colorForth	ASCII translation
red	: red
	(white)
green	: ... green ... ;
yellow	[yellow]
magenta	:# magenta
blue	{ blue }
cyan	POSTPONE cyan
grey	< grey >

Numbers will be preceded by the base operator that they are in (e.g. D# or H#, but B# and O# can not be translated back).

Notice that cyan and grey doesn't show up properly under the colors that LyX uses and I pity the person who is reading this with a black and white ebook reader. Hopefully, I will be able to automate this translation someday, but today, I haven't found a way to import code into colorForth yet anyway.

4 PWM Test

The PWM Demo code in block 842 is a good example, that would be useful, with a few modifications, in some code that I want to port to the GA144. I also have need of a potentiometer, and from what I remember about previous generations of Chuck's chip, it shouldn't be too difficult to connect a pot to the PWM so that it controls the intensity of the LED that it is connected to.

4.1 PWM Node

Copy the PWM code and shadow from 842 to 852 so we can start modifying it to look like this:

3 ⟨*pwm* 3⟩≡

```
D# 852 code{
 ( pwm ) [ d# 500 node d# 0 org ] { br }

: pol < 00 > @b h# 8000 ( rw ) and if { cr }
 ( ... ) < 03 > ( ... ) right b! @b push ex { cr }
: rtn < 06 > ( ... ) io b! then < 08 > drop { br }

 ( cyc ie- ) h# 1FFFF and over . + -if { cr }
 ( ... ) < 0c > ( ... ) h# 20000 !b pol ; { cr }
 ( ... ) < 0f > then h# 10000 < 0 > !b pol ; { br }

: upd < 12 > ( xex- ) drop push drop h# 18000 { cr }
 ( ... ) < 14 > pop pop ( iex- ) rtn ; < 16 >
}block
```

This should be the shadow for the PWM:

4a ⟨*pwm-shadow* 4a⟩≡
```
D# 853 shadow{
 ( pwm for target node 500 ) { br }

: pol ( checks for ide inputs and calls ) right { cr }
 ( when noticed. )
: rtn ( is the return point from a ) right ( call ) { cr }
 ( and is used by ) [ upd ] ( as an re-entry point. )
: cyc ( begins the actual pwm code. )
: upd ( is the ide entry point for initial ) { cr }
 ( start or output update. )
}block
```

We also need to update block 200 to **`compile`** this new code. We are going to revisit this code every time we add another block of target code, so we will number each version:

4b ⟨*compile1* 4b⟩≡
```
D# 200 code{
 ( user f18 code ) [ reclaim ] { br }

 ( softsim example ) [ reclaim d# 0 node d# 1342 load ] { cr }
 ( practical example pwm code ) [ reclaim d# 842 load ] { cr }
 ( initial test of pwm code ) [ reclaim d# 852 load ]
}block
```

Now, when I load the PWM Demo in block 844 and **`run`** it, the LED connected to 500.17 glows dimly. Just like 600.17 does when I **`seed`** it with a 0. This implies that it is never picking up the 1800 that I put into block 852. Without the support of **`run`**, my new **`upd`** never gets called. There are certainly ways, like **`run`**, to execute **`upd`** manually, but the purpose of this exercise is to make this happen automatically, from the next node.

4.2 Feed Node

So, just what is being transferred to the target when **`seed`** is executed on the host? We can see that pol is looking at the **`right`** port and that it is using **`ex`** to execute the data that it receives. So, we can make a simple experiment in block 854 like this:

4c ⟨*feed* 4c⟩≡
```
D# 854 code{
 ( feed ) [ d# 501 node d# 0 org ] { br }

: feed right b! . . @p !b . . ( / ) h# 12 end { cr }
 ( ... ) begin end
}block
```

We need to go back to block 200 and load this block too, which I forgot, at first, and wondered why nothing changed.

5a ⟨*compile2* 5a⟩≡ 7c ▷

```
  D# 200 code{
   ( user f18 code ) [ reclaim ] { br }

   ( softsim example ) [ reclaim d# 0 node d# 1342 load ] { cr }
   ( practical example pwm code ) [ reclaim d# 842 load ] { cr }
   ( initial test of pwm code ) [ reclaim d# 852 2 loads ]
  }block
```

But still nothing is changing. I see that other code puts **`( / )`** between the local code and what is being picked up with **`@p`**, but that isn't helping either. Using **`so`**, I can see the decompile of the node, reminding me to put in the `.`'s to align the words. But wait, that's only on the simulator. What's on the target chip after all of this?

I'm using the IDE to start up the PWM Demo, so I used that to see that nothing had been loaded into nodes 500 and 501. In fact, node 501 is not even accessible on the 0th path that we are using and the directions are wrong if I use the 2nd path. I need to rethink this problem a bit.

4.3 PWM Demo Extended

You can find the IDE paths on block 120, and see that the 0th path goes around the exterior nodes in a counter-clockwise direction. For the sake of this experiment, that is good enough. Looking at the chip layout poster, node 500 is the mirror image of 600 and nodes 200 and 100 replicate this pattern. So I can just use those patterns to check out the 3 other LED connections:

Copy the PWM code and shadow from 842 to 852 so we can start modifying it to look like this:

5b ⟨*pwm1* 5b⟩≡

```
  D# 852 code{
   ( pwm ) [ d# 500 node d# 0 org ] { br }

  : pol < 00 > @b h# 200 ( uw ) and if { cr }
   ( ... ) < 03 > ( ... ) up b! @b push ex { cr }
  : rtn < 06 > ( ... ) io b! then < 08 > drop { br }

   ( cyc ie- ) h# 1FFFF and over . + -if { cr }
   ( ... ) < 0c > ( ... ) h# 20000 !b pol ; { cr }
   ( ... ) < 0f > then h# 10000 < 0 > !b pol ; { br }

  : upd < 12 > ( xex- ) drop push drop h# 1100 { cr }
   ( ... ) < 14 > pop pop ( iex- ) rtn ; < 16 >
  }block
```

This should be the shadow for the PWM1:

6a ⟨*pwm1-shadow* 6a⟩≡
```
D# 853 shadow{
 ( pwm for target node 500 ) { br }

: pol ( checks for ide inputs and calls ) up { cr }
 ( when noticed. )
: rtn ( is the return point from a ) up ( call ) { cr }
 ( and is used by ) [ upd ] ( as an re-entry point. )
: cyc ( begins the actual pwm code. )
: upd ( is the ide entry point for initial ) { cr }
 ( start or output update. )
}block
```

Copy the PWM code and shadow from 842 to 854 so we can start modifying it to look like this:

6b ⟨*pwm2* 6b⟩≡
```
D# 854 code{
 ( pwm ) [ d# 200 node d# 0 org ] { br }

: pol < 00 > @b h# 2000 ( dw ) and if { cr }
 ( ... ) < 03 > ( ... ) down b! @b push ex { cr }
: rtn < 06 > ( ... ) io b! then < 08 > drop { br }

 ( cyc ie- ) h# 1FFFF and over . + -if { cr }
 ( ... ) < 0c > ( ... ) h# 20000 !b pol ; { cr }
 ( ... ) < 0f > then h# 10000 < 0 > !b pol ; { br }

: upd < 12 > ( xex- ) drop push drop h# 8100 { cr }
 ( ... ) < 14 > pop pop ( iex- ) rtn ; < 16 >
}block
```

This should be the shadow for PWM2:

6c ⟨*pwm2-shadow* 6c⟩≡
```
D# 855 shadow{
 ( pwm for target node 200 ) { br }

: pol ( checks for ide inputs and calls ) down { cr }
 ( when noticed. )
: rtn ( is the return point from a ) down ( call ) { cr }
 ( and is used by ) [ upd ] ( as an re-entry point. )
: cyc ( begins the actual pwm code. )
: upd ( is the ide entry point for initial ) { cr }
 ( start or output update. )
}block
```

Copy the PWM1 code and shadow from 852 to 856 so we can start modifying it to look like this:

7a ⟨*pwm3* 7a⟩≡

```
D# 856 code{
 ( pwm ) [ d# 100 node d# 0 org ] { br }

: pol < 00 > @b h# 200 ( uw ) and if { cr }
 ( ... ) < 03 > ( ... ) up b! @b push ex { cr }
: rtn < 06 > ( ... ) io b! then < 08 > drop { br }

 ( cyc ie- ) h# 1FFFF and over . + -if { cr }
 ( ... ) < 0c > ( ... ) h# 20000 !b pol ; { cr }
 ( ... ) < 0f > then h# 10000 < 0 > !b pol ; { br }

: upd < 12 > ( xex- ) drop push drop h# 10100 { cr }
 ( ... ) < 14 > pop pop ( iex- ) rtn ; < 16 >
}block
```

This should be the shadow for the PWM3:

7b ⟨*pwm3-shadow* 7b⟩≡

```
D# 857 shadow{
 ( pwm for target node 100 ) { br }

: pol ( checks for ide inputs and calls ) up { cr }
 ( when noticed. )
: rtn ( is the return point from a ) up ( call ) { cr }
 ( and is used by ) [ upd ] ( as an re-entry point. )
: cyc ( begins the actual pwm code. )
: upd ( is the ide entry point for initial ) { cr }
 ( start or output update. )
}block
```

We need to go back to block 200 and load all 3 blocks.

7c ⟨*compile2* 5a⟩+≡ ◁5a

```
D# 200 code{
 ( user f18 code ) [ reclaim ] { br }

 ( softsim example ) [ reclaim d# 0 node d# 1342 load ] { cr }
 ( practical example pwm code ) [ reclaim d# 842 load ] { cr }
 ( initial test of pwm code ) [ reclaim d# 852 3 loads ]
}block
```

Then we need a way to load all of them at once, by modifying the template in block 846:

8 ⟨*loader1* 8⟩≡
```
D# 846 code{
 ( loader template ) [ host load ] { ,
 , }
: seed ( n ) h# 13 r! h# 12 call upd ; { ,
 , }
 [ loader load ] { , }
 ( using default ide paths ) { , }
 ( kill boots ) [ d# 0 d# 708 hook d# 0 -hook ] { ,
 , }
 ( setup application ) { ,
 ... } [ d# 600 +node d# 600 /ram d# 0 d# 1 /stack h# 12 /p ] { ,
 ... } [ d# 500 +node d# 500 /ram d# 0 d# 1 /stack h# 12 /p ] { ,
 ... } [ d# 200 +node d# 200 /ram d# 0 d# 1 /stack h# 12 /p ] { ,
 ... } [ d# 100 +node d# 100 /ram d# 0 d# 1 /stack h# 12 /p ] { ,
 , }
 ( visit whole chip ) [ d# 2 ship panel upd ?ram ]
}block
```

Be patient when loading this block. At first, I thought it had crashed, and changed the **ship** to use path 0, which works faster. However, path 2 just takes longer, because it visits every node on the chip, as the comment indicates. Once the **panel** does come up, you will see that all 4 LEDs reflect the values that we put into their **upd** routines. Everyone is executing, but when you **hook** into a specific node so that you change change its **seed**, the nodes that come before it stop executing so they can participate in the communication path. I haven't figured out how to let it know that those nodes really are still participating in path 0, by design. Someday, I may know how to do that, but for now, I can move on.

4.4 Potentiometer Node

The analog nodes of the GA144 have both a DAC and an ADC, and I see that the **target** load block contains a definition for for setting the **dac**. Because the ADC drives a VCO, reading it is not so simple. Technically, you have to write to the **data** port, to stop the VCO before you read it, but if you try just reading it when **hook**ed to node 709 you will see that the returned values are not moving. According to the F18 Technical Reference manual (DB001 Revision 12 April 2011), you need to set the A/D Mode to High Impedance ADC input. You can try to figure out how to do an Alt write to the **i/o** port, or just send a 0 to the **dac** and see what happens. The **data** values start moving, which doesn't tell you which of the other modes it is in, but you do know that the counter is no longer disabled.

Now we need a little bit of target code to read the VCO difference after a fixed time interval, like this:

9a ⟨*pot1* 9a⟩≡
```
D# 858 code{
 ( potentiometer code ) [ d# 709 node d# 0 org ] { br }

: pol @b h# 8000 ( rw ) and if { cr }
 ( ... ) right b! @b push ex { cr }
 ( ... ) data b! then drop { br }

 ( read -v ) dup !b @b push { cr }
 ( ... ) h# 1FF for next { cr }
 ( ... ) dup !b @b pop - . + { cr }
 ( ... ) pol ;
}block
```

We again need to go back to block 200 and load this block too.

9b ⟨*compile4* 9b⟩≡
```
D# 200 code{
 ( user f18 code ) [ reclaim ] { br }

 ( softsim example ) [ reclaim d# 0 node d# 1342 load ] { cr }
 ( practical example pwm code ) [ reclaim d# 842 load ] { cr }
 ( initial test pwm code ) [ reclaim d# 852 3 loads ] { cr }
 ( initial potentiometer ) [ reclaim d# 858 load ]
}block
```

We also need to update the load block 846:

10a ⟨*loader2* 10a⟩≡ 12b ▷
```
D# 846 code{
 ( loader template ) [ host load ] { ,
 , }
: seed ( n ) h# 13 r! h# 12 call upd ; { ,
 , }
 [ loader load ] { , }
 ( using default ide paths ) { , }
 ( kill boots ) [ d# 0 d# 708 hook d# 0 -hook ] { ,
 , }
 ( setup application ) { ,
 ... } [ d# 600 +node d# 600 /ram d# 0 d# 1 /stack h# 12 /p ] { ,
 ... } [ d# 500 +node d# 500 /ram d# 0 d# 1 /stack h# 12 /p ] { ,
 ... } [ d# 200 +node d# 200 /ram d# 0 d# 1 /stack h# 12 /p ] { ,
 ... } [ d# 100 +node d# 100 /ram d# 0 d# 1 /stack h# 12 /p ] { ,
 ... } [ d# 709 +node d# 709 /ram d# 0 d# /p ] { ,
 , }
 ( visit whole chip ) [ d# 2 ship panel upd ?ram ]
}block
```

I forgot the **/p** at first, and wondered why nothing was changed, except the RAM dump. I also read the docs closer to learn that the VCO is a down counter. Now, I can tune the scaling so I produce a wide range of values. I see that it is noisy, but it should serve the purpose well enough for this exercise:

10b ⟨*pot2* 10b⟩≡
```
D# 858 code{
 ( potentiometer code ) [ d# 709 node d# 0 org ] { br }

: pol @b h# 8000 ( rw ) and if { cr }
 ( ... ) right b! @b push ex { cr }
 ( ... ) data b! then drop { br }

 ( read -v ) dup !b @b { cr }
 ( ... ) d# 500 for next dup !b @b - . + { cr }
 ( ... ) h# 3c900 . + d# 9 for 2* unext { cr }
 ( ... ) pol ;
}block
```

I can see a mistake here, **pol** should be looking at the left port and register **b** needs to point to **io** before returning to **pol**, but that causes this not to load? The LEDs light up properly, so the nodes have code and they are executing, but the panel never comes up until I unplug the USB cable. Leaving it until later.

Now, how am I going to get these values from node 709 all the way over to the PWM nodes?

4.5 Wire nodes

A wire node just passes data from one neighbor to another, so the code is trivial, as long as I ignore the IDE paths. Since the path I've chosen uses only interior nodes, this is the quickest thing to do right now. The tedious part is that there are 9 nodes on the horizontal and 1 node vertical for every row below 600 that I want to talk to. We'll start with targeting node 500, which would look like this:

11 ⟨*wire* 11⟩≡

```
 D# 860 code{
  ( wire code ) { , }
  [ d# 609 node d# 0 org reclaim ]
 : pass -d-- b! ---u a! begin @b ! end ; { , }
  [ d# 509 node d# 0 org reclaim ]
 : pass ---u b! r--- a! begin @b ! end ; { , }
  [ d# 508 node d# 0 org reclaim ]
 : pass r--- b! --l- a! begin @b ! end ; { , }
  [ d# 507 node d# 0 org reclaim ]
 : pass --l- b! r--- a! begin @b ! end ; { , }
  [ d# 506 node d# 0 org reclaim ]
 : pass r--- b! --l- a! begin @b ! end ; { , }
  [ d# 505 node d# 0 org reclaim ]
 : pass --l- b! r--- a! begin @b ! end ; { , }
  [ d# 504 node d# 0 org reclaim ]
 : pass r--- b! --l- a! begin @b ! end ; { , }
  [ d# 503 node d# 0 org reclaim ]
 : pass --l- b! r--- a! begin @b ! end ; { , }
  [ d# 502 node d# 0 org reclaim ]
 : pass r--- b! --l- a! begin @b ! end ; { , }
  [ d# 501 node d# 0 org reclaim ]
 : pass --l- b! r--- a! begin @b ! end ;
 }block
```

We again need to go back to block 200 and load this block too.

12a ⟨*compile5* 12a⟩≡
```
D# 200 code{
 ( user f18 code ) [ reclaim ] { br }

 ( softsim example ) [ reclaim d# 0 node d# 1342 load ] { cr }
 ( practical example pwm code ) [ reclaim d# 842 load ] { cr }
 ( initial test pwm code ) [ reclaim d# 852 3 loads ] { cr }
 ( initial potentiometer ) [ reclaim d# 858 load ] { cr }
 ( initial wire code ) [ reclaim d# 860 load ]
}block
```

We also need to update the load block 846:

12b ⟨*loader2* 10a⟩+≡ ◁10a 29▷
```
D# 846 code{
 ( loader template ) [ host load ] { ,
 , }
: seed ( n ) h# 13 r! h# 12 call upd ; { ,
 , }
 [ loader load d# 0 d# 708 hook d# 0 -hook ] { ,
 , }
 ( setup application ) { ,
 ... } [ d# 600 +node d# 600 /ram d# 0 d# 1 /stack h# 12 /p ] { ,
 ... } [ d# 500 +node d# 500 /ram d# 0 d# 1 /stack h# 12 /p ] { ,
 ... } [ d# 200 +node d# 200 /ram d# 0 d# 1 /stack h# 12 /p ] { ,
 ... } [ d# 100 +node d# 100 /ram d# 0 d# 1 /stack h# 12 /p ] { ,
 ... } [ d# 709 +node d# 709 /ram d# 0 /p ] { ,
 ... } [ d# 609 +node d# 609 /ram d# 0 /p ] { ,
 ... } [ d# 509 +node d# 509 /ram d# 0 /p ] { ,
 ... } [ d# 508 +node d# 508 /ram d# 0 /p ] { ,
 ... } [ d# 507 +node d# 507 /ram d# 0 /p ] { ,
 ... } [ d# 506 +node d# 506 /ram d# 0 /p ] { ,
 ... } [ d# 505 +node d# 505 /ram d# 0 /p ] { ,
 ... } [ d# 504 +node d# 504 /ram d# 0 /p ] { ,
 ... } [ d# 503 +node d# 503 /ram d# 0 /p ] { ,
 ... } [ d# 502 +node d# 502 /ram d# 0 /p ] { ,
 ... } [ d# 501 +node d# 501 /ram d# 0 /p ] { ,
 , }
 [ d# 2 ship panel upd ?ram ]
}block
```

Note that we removed some of the comments because the block is getting crowded. However, we've probably got enough nodes involved now to complete this experiment.

4.6 Connecting the Wires

Now we need to connect the **pwm** and the **pot** to the neighboring wire nodes that we just created:

13 ⟨*pwm4* 13⟩≡

```
D# 852 code{
 ( pwm ) [ d# 500 node d# 0 org ] { br }

: pol < 00 > @b h# 200 ( uw ) and if { cr }
 ( ... ) < 03 > ( ... ) up b! @b push ex { cr }
: rtn < 06 > ( ... ) io b! then < 08 > [ h# 16 ] end { cr }

: cyc < 09 > ( ie- ) h# 1FFFF and over . + -if { cr }
 ( ... ) < 0c > ( ... ) h# 20000 !b pol ; { cr }
 ( ... ) < 0f > then h# 10000 < 0 > !b pol ; { br }

: upd < 12 > ( xex- ) drop push drop h# 1100 { cr }
 ( ... ) < 14 > pop pop ( iex- ) rtn ; { br }

: chk < 16 > ( xex- ) drop push @b h# 8000 ( rw ) { cr }
 ( ... ) < 18 > and if drop drop right . { cr }
 ( ... ) < 1b > ( ... ) b! @b dup io b! then { cr }
 ( ... ) < 1e > drop pop ( ie- ) cyc ; < 20 >
}block
```

Note that I hacked this new definition **chk** into this code so that I could maintain the addresses used by **seed**, but I thought it might be useful to show how to do this trick. It's ugly, but sometimes, it's better than screwing with code that you've already finalized. Forward references would make this hack look better, but as with most other Forths, colorForth's one pass compiler doesn't allow them.

The **pot** code doesn't need this hack, because the **check** can be done at the end of the loop.

14 ⟨*pot3* 14⟩≡
```
 D# 858 code{
  ( potentiometer code ) [ d# 709 node d# 0 org ] { br }

 : pol @b h# 8000 ( rw ) and if { cr }
  ( ... ) right b! @b push ex { cr }
  ( ... ) data b! then drop { br }

  ( read -v ) dup !b @b { cr }
  ( ... ) d# 500 for next dup !b @b - . + { cr }
  ( ... ) h# 3c900 . + d# 9 for 2* unext { br }

  ( check ) io a! @ h# 4000 ( dr ) and if { cr }
  ( ... ) down a! over ! then { cr }
  ( ... ) drop pol ;
 }block
```

Note that I had to use the **a** register to get this to load and show the panel. This feels like the same problem I observed earlier and, since the **pot** is not controlling the **pwm**, maybe I have to figure out the root cause now.

5 Debugging

Now, the tedious job of debugging nodes that you can only see indirectly. The IDE relies on paths that it puts into the code, which is part of the problem I have observed earlier. The **pwm** code assumes that the 0th path is active, but the 2nd path is needed to fill the interior nodes, which block 846 uses. So, the **seed** word I put on 846 won't work properly.

The **softsim** (**so**) display has an easy way to disassemble code, and I noticed that the wire nodes were not correctly compiled. I saw that **reclaim** was used in block 366, where multiple nodes use the same names, but that didn't make them compile properly. I then tried switching **r---** to **right** and that compiled better, so I switched them all. You should learn from this mistake. Double check your code.

15 ⟨*wire1* 15⟩≡

```
D# 860 code{
 ( wire code ) { , }
 [ d# 609 node d# 0 org reclaim ]
: pass down b! up a! begin @b ! end ; { , }
 [ d# 509 node d# 0 org reclaim ]
: pass up b! right a! begin @b ! end ; { , }
 [ d# 508 node d# 0 org reclaim ]
: pass right b! left a! begin @b ! end ; { , }
 [ d# 507 node d# 0 org reclaim ]
: pass left b! right a! begin @b ! end ; { , }
 [ d# 506 node d# 0 org reclaim ]
: pass right b! left a! begin @b ! end ; { , }
 [ d# 505 node d# 0 org reclaim ]
: pass left b! right a! begin @b ! end ; { , }
 [ d# 504 node d# 0 org reclaim ]
: pass right b! left a! begin @b ! end ; { , }
 [ d# 503 node d# 0 org reclaim ]
: pass left b! right a! begin @b ! end ; { , }
 [ d# 502 node d# 0 org reclaim ]
: pass right b! left a! begin @b ! end ; { , }
 [ d# 501 node d# 0 org reclaim ]
: pass left b! right a! begin @b ! end ;
}block
```

To limit the scope of what I need to look at, I simply changed the **@b** in node 501 to a **h# 18000** to see if I could just pass that to the **pwm** in 500, but still no joy. Then again, I used **h# 1800** and didn't notice a change. When I actually used **h# 18000**, I can see that it is working. However, switching back to **@b** still didn't give me **pot** control of the **pwm**.

The next step, then, is to move the literal back one node at a time. 502 works, but reloads do not always work. Sometimes hitting reset helps. Sometimes I have to pull the USB cable. This worked until I got back to node 609. Then the literal made the load not finish, but the LEDs say that the literal got through. So, whatever is wrong, it's got to be in node 709.

So, I might be able to use not being able to reload as a test. If I reset register **b** to **io**, rather than **data** and jump to **pol** at the end of that line, I can reload as many times as I want to. I can switch all of the other access to use register **a**, but that makes it not reload again. So, let's just try moving the jump back to **pol** further into the code, until it doesn't reload. It stops reloading when the jump is after the **next**.

It's interesting that when I change the **next** to a **unext**, it reloads fine. Since **next** is about 2 times slower than **unext**, maybe there's a timing limit that the loop is exceeding. Dropping the loop count from **500** to **200** and leaving the **next** in also fixes the reload problem. However, the data has gone away, so I have to back out the other changes. Then recalibrate the scaling.

However, the data is no longer changing with potentiometer position. I can usually reload with a loop count of 430, but the longer it runs, the slower it gets, so 400 should be safe. Block 546 has code for basic analog checks, and I see that it writes to io with the mode. It also fixes the subtract properly, so it's worth showing here:

16 ⟨*pot4* 16⟩≡
```
D# 858 code{
 ( potentiometer code ) [ d# 709 node d# 0 org ] { br }

: adc ( mode ) h# 0 < h# 2000 h# 4000 h# 6000 > !b { br }

: pol @b h# 8000 ( rw ) and if { cr }
 ( ... ) right b! @b push ex { cr }
 ( ... ) data b! then drop { br }

 ( read -v ) dup !b @b dup d# 400 for next { cr }
 ( ... ) !b @b - d# 1 . + . + h# 1FFFF and { cr }
 ( ... h# 3c900 . + d# 9 for 2* unext ) { br }

 ( check ) io b! @b h# 4000 ( dr ) and if { cr }
 ( ... ) down b! over !b then { cr }
 ( ... ) drop io b! pol ;
}block
```

Note also here that I have simply commented out the scaling portion. In colorForth, that is extremely easy by simply changing the color of the words to white using your left pinky finger. However, I see that the **`pot`** values are still not moving as I adjust the potentiometer, and that the LED is not reflecting the **`h# 1fd55`** values that I am getting. Going back to the earlier experiment, of forcing a value into the **`wire`** nodes, I see that changing 609 still causes the system to not reload, but that changing 509 works fine. So, the code in 709 is still not right. Even with a loop count of 200, it still won't reload without the read in 609. Further, a forced value in 609 (in addition to a read from 709) does not make it out to the LED. But wait, adding the literal, without removing the **`@b`** doesn't even work in node 501, even after adding an extra noop so the jump doesn't fall in slot 3.

With my deadline to get this working approaching, and frustration setting in, I sent this document and cf file to Green Arrays, asking for help. Then I walked away for a bit, which I have found to always be the best debugging aid. When I came back, I accepted the fact that a rewrite of the pot node was necessary. The IDE can't tolerate the delay in between VCO readings, so it's time to fix that. As I did that, I also realized that the algorithm needs to scale the results automatically, since every code change changes the scale factors. Then, as I went to sleep, I realized that the data passing problem was caused by the read flags being inverted from the writes. This morning I proved that is true, but I still need to finish the rewrite.

I had emailed a follow up to Green Arrays when I saw some pot movement in an early rewrite and I received a response from Greg Bailey, President of Green Arrays last night, with assurance that he would have time later in the week, if I still needed help. Despite the fact that I have known Greg for many years, and I know that Green Arrays is manned by a small core of dedicated volunteers, the fact that they are willing to help is reassuring. Sometimes, this assurance makes all the difference.

5.1 Potentiometer redesign

The goals of this redesign are to only read the ADC once around each polling loop and to scale the results automatically. My initial findings toward the 1st goal show that the polling overhead provides sufficient time between readings and that the potentiometer I am using is giving me less than 16 counts of difference. I may have to investigate why I am getting such a small range, at some point, but for now, I can take advantage of this knowledge in the scaling algorithm.

17 ⟨*pot5* 17⟩≡
```
 D# 858 code{
  ( potentiometer code ) [ d# 709 node d# 0 org ] { br }

 : adc ( mode ) h# 0 !b { cr }
  ( ... initial ) data a! dup ! @ push { cr }
  ( ... reduction ) d# 0

 : pol @b h# 8000 ( rw ) and if { cr }
  ( ... ) right b! @b push ex { cr }
  ( ... ) io b! then drop { br }

 : read ( r-vr ) data a! dup ! @ dup a! { cr }
  ( ... ) - d# 1 . + pop . + h# 1ffff and { cr }
  ( ... -rd ) over over - push . + ( -rf ) -if { cr }
  ( adjust lo ) drop drop pop dup push d# 3 . + d# 2 { cr }
  ( ... ) then dup - d# 18 . + ( -rfo ) -if { cr }
  ( adjust hi ) drop drop drop pop dup push { cr }
  ( ... ... ) d# 16 . + d# 15 d# 0 { cr }
  ( ... ) then drop pop drop a push if { cr }
  ( ... ... ) h# 3ffff . + push 1 begin 2* unext { cr }
  ( swap ) then a! push a pop ( -vr ) { br }

 : check ( vr-vr ) io b! @b - h# 4000 ( dr ) and if { cr }
  ( ... ... ) down a! push over ! pop { cr }
  ( ... ) then drop pol ; <39>
 }block
```

Now, you will notice that the **read** routine is unreadable, if you will forgive the pun, and I have to admit that it is also impossible to debug in this form. There are ways to mitigate this problem, but I'm out of time and patience. So, knowing that the stack is, at least, functioning properly, let me just rewrite the logic in pseudo code and inject the literals manually.

The loop that runs around this node doesn't factor out any subroutines, at the moment, but it does assume that the previous **adc** reading is on top of the return stack and that a **reduction** value (**r**) is on top of the data stack. The **read** routine may update **r**, but its main purpose is to add a new pot value (**v**) to the stack. You may notice the **swap** equivalent at the end of the read routine to keep **r** on top of the stack.

The basic processing in **read** gets a new **adc** value, saving it to replace the previous value on the return stack. It computes the delta between the 2 values by adding the negation of the new reading. Sometimes, you can get away with just using the inversion (**-**), but for the sake of precision, I am adding **1** to produce the true negation. It then reduces the delta by **r** to give a factored value (**f**) and adjusts for variations so we can use this as a power of 2 for the value (**v**) we will pass on to the **pwm** node.

We need to allow for some noise, so if the new delta (**x**) is less than the old delta (**y**), the **adjust lo** portion needs to replace the old delta with the new, and reduce it by 2 counts. Since we are again using the negation, we can combine the 2 negations into 1 sequence as follows:

$-(x-2)$ = $-x+2$ = x – 1 . + 2 . + = x – 3 . +

If the new delta (**x**) is greater than the old delta (**y**) by more than 17, then we won't be able to use it as a power of 2 and the **adjust hi** portion needs to replace the old delta with the new, reduced by 15. Again, this simplifies as follows:

$-(x-15)$ = $-x+15$ = x – 1 . + 15 . + = x – 16 . +

This can then be factored further to put the negate (**-**) before both **-if** conditionals.

The same trick can be used when comparing with 17, as follows:

$x>17$ = $17<x$ = $17-x<0$ = $-x+17<0$ = x – 1 . + 17 . + = x – 18 . +

However, even after all of these mathematical gymnastics, the values are not settling down. The only thing I can think of is the fact that I have not balanced the false conditionals. Each of the **if** cases do enough processing to throw the counts off on the next cycle. The readings seem to indicate that the values are bouncing from one end to the other, which I would expect, as I think about this flaw in my logic.

Some of you may also notice the grey number at the end of this block. This is a hex number that indicates the amount of memory used in this node. Since we only have 64 words, and only 6 words left, I will have to split it up into multiple blocks to do this balancing. However, we have now seen just about the limit to what can be put into one node. Obviously, I can't even recommending doing this much.

6 Balancing act

Now, let's see if I can balance everything that needs to be done before the Maker Faire this Saturday. It is 9 pm Tuesday night and I have to setup the SVFIG table at 9 am Saturday morning. We'll see how much I can get done.

Since I have no more room in node 709, I have to move the scaling logic out, which also means that I don't need to balance the conditionals, since they no longer influence the **adc** readings. The **read** algorithm becomes much less complex:

19 ⟨*pot6* 19⟩≡
```
D# 858 code{
  ( potentiometer code ) [ d# 709 node d# 0 org ] { br }
: adc ( mode ) h# 0 !b { cr }
  ( ... initial ) data a! dup ! @ push { br }

: pol @b h# 8000 ( rw ) and if { cr }
  ( ... ) right b! @b push ex { cr }
  ( ... ) io b! then drop { br }

: read ( -v ) data a! dup ! @ dup a! { cr }
  ( ... ) - d# 1 . + pop . + h# 1ffff and { cr }
  ( ... ) a push { br }

: check ( v-v ) io b! @b - h# 4000 ( dr ) and if { cr }
  ( ... ... ) down a! over ! { cr }
  ( ... ) then drop pol ; <20>
}block
```

Then, the scaling algorithm can stand on its own. At first, I put it into node 609, but then I realized than I can't see what it is doing there. The LED is now a steady bright, but does not respond to the **`pot`**.

20 ⟨*scaler* 20⟩≡

```
D# 862 code{
 ( scaling code ) [ d# 609 node d# 0 org ] { br }

: adc ( -r ) down a! @ { br }

: pol @b h# 800 ( lw ) and if { cr }
 ( ... ) left b! @b push ex { cr }
 ( ... ) io b! then drop { br }

: read ( r-vr ) @ over over - push . + ( -rf ) -if { cr }
 ( adjust lo ) drop drop pop dup push d# 3 . + d# 2 { cr }
 ( ... ) then dup - d# 18 . + ( -rfo ) -if { cr }
 ( adjust hi ) drop drop drop pop dup push { cr }
 ( ... ... ) d# 16 . + d# 15 d# 0 { cr }
 ( ... ) then drop pop drop if { cr }
 ( ... ... ) h# 3ffff . + push 1 begin 2* unext { cr }
 ( swap ) then a! push a pop ( -vr ) { br }

: check ( vr-vr ) io b! @b - h# 400 ( ur ) and if { cr }
 ( ... ... ) up a! push over ! pop { cr }
 ( ... ) then drop pol ; <2e>
}block
```

I commented out the code for node 609 in block 860 and changed block 200 to load this block.

21 ⟨*compile6* 21⟩≡

```
D# 200 code{
 ( user f18 code ) [ reclaim ] { br }

 ( softsim example ) [ reclaim d# 0 node d# 1342 load ] { cr }
 ( practical example pwm code ) [ reclaim d# 842 load ] { cr }
 ( initial test pwm code ) [ reclaim d# 852 3 loads ] { cr }
 ( initial potentiometer ) [ reclaim d# 858 load ] { cr }
 ( initial wire code ) [ reclaim d# 860 load ]
 ( initial scaler ) [ reclaim d# 862 load ]
}block
```

After changing node 100 to **pol** the **down** port, node 200 to **pol** the **right** port and node 500 to **pol** the **down** port, I can examine them and use **seed** to change their LED intensities. I left node 600 alone, so that it could still work with the original IDE demo in block 844. Yet, node 609 is still unreachable, because I did not put the **pol** routine into the **wire** nodes.

On the other hand, it no longer matters where the **scaler** is located along the wire path. I can easily move it over to node 501, where I should be able to examine it with the IDE. I also noticed a problem that would certainly prevent it from working properly, so crossing my fingers for the next tests.

Well, I think I stepped backwards, somehow. Not only can I not **hook** into node 501, but my **adc** measurement are bouncing worse than I've seen before. I'm getting every other reading toggling all the way up to bit 7. Arg! If I don't load my new scaler, the **adc** settles right down.

But of course, you can't have two nodes polling for each other to do something. The **pot** polls to send data to its neighbor, but the **pwm** polls to get data from its neighbor. The scaler in node 609 can't poll the **pot** and it can't poll the **pwm** in node 501. It has to be rewritten.

22 ⟨*scaler1* 22⟩≡
```
D# 862 code{
 ( scaling code ) [ d# 501 node d# 0 org ] { br }

 ( reduction ) d# 0 { br }

: pol @b h# 8000 ( rw ) and if { cr }
 ( ... ) right b! @b push ex { cr }
 ( ... ) io b! then drop { br }

: check ( -n ) @b - h# 800 ( lw ) and if { cr }
 ( ... ... ) left a! push @ { br }

: read ( rn-vr ) over over - push . + ( -rf ) -if { cr }
 ( adjust lo ) drop drop pop dup push d# 3 . + d# 2 { cr }
 ( ... ) then dup - d# 18 . + ( -rfo ) -if { cr }
 ( adjust hi ) drop drop drop pop dup push { cr }
 ( ... ... ) d# 16 . + d# 15 d# 0 { cr }
 ( ... ) then drop pop drop if { cr }
 ( ... ... ) h# 3ffff . + push 1 begin 2* unext { cr }
 ( swap ) then a! push a pop ( -vr ) { br }

 ( ... ... ) right a! over ! pop { cr }
 ( ... ) then drop pol ; <2e>
}block
```

Ah, this produces some success. I can **hook** into node 501, but it doesn't appear to be doing anything. Perhaps this is because **hook** takes over, which would explain why all except node 600's LED has turned off. At least I get a snapshot of its stack, which looks ok. This is the problem with the IDE. It's like Schrödinger's cat. Using it affects the way the nodes work.

7 Softsim integration

Let's see if we can get this working in softsim (so) to help figure out what is going on:

23 ⟨*softsim* 23⟩≡

```
D# 216 code{
 ( softsim configuration ) { , }
 { , }
 ( spi boot testbed d# 1244 d# 2 loads ) { , }
 ( sync boot testbed 'addr,len' d# 1230 load ) { , }
 { , }
 ( smtm ) [ d# 0 +node d# 0 /ram d# 0 /p ] { ,
 ... } [ d# 600 +node d# 600 /ram d# 0 d# 1 /stack h# 12 /p ] { ,
 ... } [ d# 500 +node d# 500 /ram d# 0 d# 1 /stack h# 12 /p ] { ,
 ... } [ d# 200 +node d# 200 /ram d# 0 d# 1 /stack h# 12 /p ] { ,
 ... } ( d# 100 +node d# 100 /ram d# 0 d# 1 /stack h# 12 /p ) { ,
 ... } [ d# 709 +node d# 709 /ram d# 0 /p ] { ,
 ... } [ d# 609 +node d# 609 /ram d# 0 /p ] { ,
 ... } [ d# 509 +node d# 509 /ram d# 0 /p ] { ,
 ... } [ d# 508 +node d# 508 /ram d# 0 /p ] { ,
 ... } [ d# 507 +node d# 507 /ram d# 0 /p ] { ,
 ... } [ d# 506 +node d# 506 /ram d# 0 /p ] { ,
 ... } [ d# 505 +node d# 505 /ram d# 0 /p ] { ,
 ... } [ d# 504 +node d# 504 /ram d# 0 /p ] { ,
 ... } [ d# 503 +node d# 503 /ram d# 0 /p ] { ,
 ... } [ d# 502 +node d# 502 /ram d# 0 /p ] { ,
 ... } [ d# 501 +node d# 501 /ram d# 0 /p ] { ,
 ( /command test ) [ d# 400 +node d# 0 /ram h# 25 /a h# 12 /b ] { , }
 [ d# 9 d# 8 d# 7 d# 6 d# 5 d# 4 d# 3 d# 2 d# 1 h# 12345 d# 10 /stack h# A9 /p ] { , }
 { , }
 ( rom write test d# 200 +node h# 13 /p ) { , }
 { , }
 ( d# 0 h# 32 d# 103 break ) { , }
 ( d# 0 h# be d# 300 break )
}block
```

That makes for an over full block, and I kept node 100 commented out to not interfere with the crawler demo, but now I can see that the **`scaler`** is delivering the 1st 2 values to the **`pwm`**, but then the **`wire`** nodes start backing up. I forgot to remove the **`-`** in **`check`** when I changed it from looking for reads to writes. That now makes the simulator behave, but I'm back to getting the 8 bit variations in **`adc`** values. The simulator also showed me that I can certainly afford to slow down the **`adc`** readings, since the wire is still being filled up with them. So, let's see what happens if I throw some code back into the **`pot`**.

First, let me try just balancing the conditionals:

24 ⟨*pot7* 24⟩≡

```
D# 858 code{
 ( potentiometer code ) [ d# 709 node d# 0 org ] { br }

: pol @b h# 8000 ( rw ) and if { cr }
 ( ... ... ) right b! @b push ex io b! drop ; { cr }
 ( ... ) then . . drop ; { br }

: check ( v-v ) io b! @b - h# 4000 ( dr ) and if { cr }
 ( ... ... ) down a! drop ! ; { cr }
 ( ... ) then . . d ; { br }

: adc < 12 > ( mode ) h# 0 !b { cr }
 ( ... initial ) data a! dup ! @ push { br }

: read ( -v ) pol data a! dup ! @ dup a! { cr }
 ( ... ) - d# 1 . + pop . + h# 1ffff and { cr }
 ( ... ) a push check read ; <23>
}block
```

This just gets us started and after changing the start address in both of the load blocks, I can test it and see that I still have some adjusting to do. For **pol**, we have an time when executing the stream, but we can go through the exercise for what we do know. Using so, I can find the instruction layout:

word	0	1	2	3	fetch
@p b! @b .	5000	1400	3500	1400	3200
push ex	1400	5200			
@p b! drop ;	5000	1400	1400	5200	

That makes 34,100 pS. The duplicated drop before each ; fit within the instruction words and allow me to use + in slot 3 to consume the numbers I add with @p (one of the slow instructions). So, I can balance the time with the following:

word	0	1	2	3	fetch
@p + @p +	5000	1400	5000	1400	3200
@p + . .	5000	1400	1400	1400	1200
. drop ;	1400	1400	5200		

That gives me 34,400 pS which is probably ok since we are not covering the **ex** in **pol** anyway, but not for **check**, where I have the following to balance:

word	0	1	2	3	fetch
@p a! over .	5000	1400	1400	1400	1200
! drop ;	3500	1400	5200		

That is 20,500 pS, which is going to need this:

word	0	1	2	3	fetch
@p + drop .	5000	1400	1400	1400	1200
@b drop ;	3500	1400	5200		

Note how both sides here are balanced, but not as efficient as they could be without having to balance them. Its interesting that now the variations have a range of about 4 or 5. It got slightly better by adding a @b + to the **pol** routine, but it's difficult to measure or adjust it much closer. Now I get sporatic movement on the LED, which might just be good enough.

8 Smoothing buffer

I am way too much of a perfectionist to leave good enough alone, so let's see how quickly I can throw in a buffer to average the variation in the values. As I watch the values through the IDE, which, of course, changes their values, I see that they only get up to about **h# 3ac** when the **pot** is turned up all the way. That means I can safely sum 16 or 32 values without overflowing. So, taking the next wire node, I can turn it into an averaging buffer like this:

25 ⟨*avg* 25⟩≡
```
D# 864 code{
 ( averaging buffer code ) [ d# 502 node d# 0 org ]
: pass h# 30 a! d# 15 for { , }
 ( ... ... ) right b! @b !+ { , }
 ( ... ... ) a push h# 30 a! @+ { , }
 ( ... ... ) d# 14 for @+ + 2/ unext { , }
 ( ... ... ) pop a! left b! !b { , }
 ( ... ) next pass ; < 11 >
}block
```

Once I remembered to scale the sum back down before shipping it out, it appeared to work pretty good, except when I used **`watch`** to monitor the **`adc`** valued. I could also see that I had room for 32 values, which makes it work better under **`watch`**, but not completely. Again, good enough.

9 Conclusion

This is obviously, way too much code to just replace a wire between the LED and the potentiometer, but I've learned a lot about the GA144 and the GreenArray's IDE. I would now do this differently, which is the fate of most throw away code, and I should look at the voltage feeding the potentiometer, to see if I can give it more range. However, I can close this tutorial knowing that I completed what I set out to do.

Most of the source blocks have been touched in the last thrashing, so here they all are in their final form:

26 ⟨*final-compile* 26⟩≡

```
D# 200 code{
  ( user f18 code ) [ reclaim ] { br }

  ( softsim example ) [ reclaim d# 0 node d# 1342 load ] { cr }
  ( practical example pwm code ) [ reclaim d# 842 load ] { cr }
  ( initial test pwm code ) [ reclaim d# 852 d# 3 loads ] { cr }
  ( initial potentiometer ) [ reclaim d# 858 load ] { cr }
  ( initial wire code ) [ reclaim d# 860 load ] { cr }
  ( initial scaler ) [ reclaim d# 862 load ] { cr }
  ( averaging code ) [ reclaim d# 864 load ]
}block
```

27 ⟨*final-softsim* 27⟩≡

```
D# 216 code{
 ( softsim configuration ) { ,
 , }
 ( spi boot testbed 1244 2 loads ) { , }
 ( sync boot testbed 'addr,len' 1230 load ) { ,
 , }
 ( smtm ) [ d# 0 +node d# 0 /ram d# 0 /p ] { ,
 ... } [ d# 600 +node d# 600 /ram d# 0 d# 1 /stack h# 12 /p ] { ,
 ... } [ d# 500 +node d# 500 /ram d# 0 d# 1 /stack h# 12 /p ] { ,
 ... } [ d# 200 +node d# 200 /ram d# 0 d# 1 /stack h# 12 /p ] { ,
 ... } ( d# 100 +node d# 100 /ram d# 0 d# 1 /stack h# 12 /p ) { ,
 ... } [ d# 709 +node d# 709 /ram h# 19 /p ] { ,
 ... } [ d# 609 +node d# 609 /ram d# 0 /p ] { ,
 ... } [ d# 509 +node d# 509 /ram d# 0 /p ] { ,
 ... } [ d# 508 +node d# 508 /ram d# 0 /p ] { ,
 ... } [ d# 507 +node d# 507 /ram d# 0 /p ] { ,
 ... } [ d# 506 +node d# 506 /ram d# 0 /p ] { ,
 ... } [ d# 505 +node d# 505 /ram d# 0 /p ] { ,
 ... } [ d# 504 +node d# 504 /ram d# 0 /p ] { ,
 ... } [ d# 503 +node d# 503 /ram d# 0 /p ] { ,
 ... } [ d# 502 +node d# 502 /ram d# 0 /p ] { ,
 ... } [ d# 501 +node d# 501 /ram d# 0 /p ] { , }
 ( /command test ) [ d# 400 +node d# 0 /ram h# 25 /a h# 12 /b ] { , }
 [ d# 9 d# 8 d# 7 d# 6 d# 5 d# 4 d# 3 d# 2 d# 1 h# 12345 d# 10 /stack h# A9 /p ] { ,
 , }
 ( rom write test { 0000190F } +node { 0000027F } /p ) { ,
 , }
 ( d# 0 h# 32 d# 103 break ) { , }
 ( d# 0 h# be d# 300 break )
}block
```

28a ⟨*original-pwm* 28a⟩≡
```
D# 842 code{
 ( pwm demo ) [ d# 600 node d# 0 org ] { br }

: pol < 00 > @b h# 2000 ( dw ) and if { cr }
 ( ... ) < 03 > ( ... ) down b! @b push ex { cr }
: rtn < 06 > ( ... ) io b! then < 08 > drop { br }

 ( cyc ie- ) h# 1FFFF and over . + -if { cr }
 ( ... ) < 0c > ( ... ) h# 20000 !b pol ; { cr }
 ( ... ) < 0f > then h# 10000 ( h# 0 ) !b pol ; { br }

: upd < 12 > ( xex- ) drop push drop h# 100 { cr }
 ( ... ) < 14 > pop pop ( iex- ) rtn ; < 16 >
}block
```

28b ⟨*final-example* 28b⟩≡
```
D# 844 code{
 ( demo ide boot ) [ empty compile serial load ] { br }

 ( customize ) [ -canon d# 0 fh orgn ! ] { cr }
 [ c-com sport ! c-bps bps ! !nam ] { br }

: seed ( n ) h# 13 r! h# 12 call upd ; { br }

: run talk d# 0 d# 600 hook d# 0 d# 64 d# 600 boot { indent }
   upd ?ram panel d# 0 lit h# 00018000 seed ;
}block
```

29 ⟨*loader2* 10a⟩+≡ ◁ 12b

```
 D# 846 code{
  ( loader template ) [ target load ] { ,
  , }
 : seed ( n ) h# 13 r! h# 12 call upd ; { ,
  , }
  [ loader load d# 0 d# 708 hook d# 0 -hook ] { ,
  , }
  ( setup application ) { ,
  ... } [ d# 600 +node d# 600 /ram d# 0 d# 1 /stack h# 12 /p ] { ,
  ... } [ d# 500 +node d# 500 /ram d# 0 d# 1 /stack h# 12 /p ] { ,
  ... } [ d# 200 +node d# 200 /ram d# 0 d# 1 /stack h# 12 /p ] { ,
  ... } [ d# 100 +node d# 100 /ram d# 0 d# 1 /stack h# 12 /p ] { ,
  ... } [ d# 709 +node d# 709 /ram h# 19 /p ] { ,
  ... } [ d# 609 +node d# 609 /ram d# 0 /p ] { ,
  ... } [ d# 509 +node d# 509 /ram d# 0 /p ] { ,
  ... } [ d# 508 +node d# 508 /ram d# 0 /p ] { ,
  ... } [ d# 507 +node d# 507 /ram d# 0 /p ] { ,
  ... } [ d# 506 +node d# 506 /ram d# 0 /p ] { ,
  ... } [ d# 505 +node d# 505 /ram d# 0 /p ] { ,
  ... } [ d# 504 +node d# 504 /ram d# 0 /p ] { ,
  ... } [ d# 503 +node d# 503 /ram d# 0 /p ] { ,
  ... } [ d# 502 +node d# 502 /ram d# 0 /p ] { ,
  ... } [ d# 501 +node d# 501 /ram d# 0 /p ] { ,
  , }
  [ d# 2 ship panel upd ?ram ]
 }block
```

30a ⟨*final-pwm1* 30a⟩≡
```
D# 852 code{
 ( pwm demo ) [ d# 500 node d# 0 org ] { br }

: pol < 00 > @b h# 2000 ( dw ) and if { cr }
 ( ... ) < 03 > ( ... ) down b! @b push ex { cr }
: rtn < 06 > ( ... ) io b! then < 08 > [ h# 16 ] end { br }

: cyc < 09 > ( ie- ) h# 1FFFF and over . + -if { cr }
 ( ... ) < 0c > ( ... ) h# 20000 !b pol ; { cr }
 ( ... ) < 0f > then h# 10000 ( h# 0 ) !b pol ; { br }

: upd < 12 > ( xex- ) drop push drop h# 1100 { cr }
 ( ... ) < 14 > pop pop ( iex- ) rtn ; < 16 > { br }

: chk < 16 > ( xex- ) drop push @b h# 8000 ( rw ) { cr }
 ( ... ) < 18 > and if drop drop right . { cr }
 ( ... ) < 1b > ( ... ) b! @b dup io b! then { cr }
 ( ... ) < 1e > drop pop ( ie- ) cyc ; < 20 >
}block
```

30b ⟨*final-pwm2* 30b⟩≡
```
D# 854 code{
 ( pwm demo ) [ d# 200 node d# 0 org ] { br }

: pol < 00 > @b h# 8000 ( rw ) and if { cr }
 ( ... ) < 03 > ( ... ) right b! @b push ex { cr }
: rtn < 06 > ( ... ) io b! then < 08 > drop { br }

 ( cyc ie- ) h# 1FFFF and over . + -if { cr }
 ( ... ) < 0c > ( ... ) h# 20000 !b pol ; { cr }
 ( ... ) < 0f > then h# 10000 ( h# 0 ) !b pol ; { br }

: upd < 12 > ( xex- ) drop push drop h# 8100 { cr }
 ( ... ) < 14 > pop pop ( iex- ) rtn ; < 16 >
}block
```

31a ⟨*final-pwm3* 31a⟩≡
```
D# 856 code{
 ( pwm demo ) [ d# 100 node d# 0 org ] { br }

: pol < 00 > @b h# 2000 ( dw ) and if { cr }
 ( ... ) < 03 > ( ... ) down b! @b push ex { cr }
: rtn < 06 > ( ... ) io b! then < 08 > drop { br }

 ( cyc ie- ) h# 1FFFF and over . + -if { cr }
 ( ... ) < 0c > ( ... ) h# 20000 !b pol ; { cr }
 ( ... ) < 0f > then h# 10000 ( h# 0 ) !b pol ; { br }

: upd < 12 > ( xex- ) drop push drop h# 10100 { cr }
 ( ... ) < 14 > pop pop ( iex- ) rtn ; < 16 >
}block
```

31b ⟨*final-pot* 31b⟩≡
```
D# 858 code{
 ( potentiometer code ) [ d# 709 node d# 0 org ] { br }

: pol @b h# 8000 ( rw ) and if { cr }
 ( ... ... ) right b! @b push ex io b! drop ; { cr }
 ( ... ) then d# 0 + d# 0 + d# 0 + @b + . drop ; { br }

: check ( v-v ) io b! @b - h# 4000 ( dr ) and if { cr }
 ( ... ... ) down a! over ! drop ; { cr }
 ( ... ) then d# 0 + drop . @b drop ; { br }

: adc < 19 > ( mode ) h# 0 !b { cr }
 ( ... initial ) data a! dup ! @ push { br }

: read ( -v ) pol data a! dup ! @ dup a! { cr }
 ( ... ) - d# 1 . + pop . + h# 1FFFF and { cr }
 ( ... ) a push check read ; < 29 >
}block
```

32 ⟨*final-wire* 32⟩≡
```
D# 860 code{
 ( wire code ) { , }
 [ d# 609 node d# 0 org reclaim ]
: pass down b! up a! begin @b ! end ; { , }
 [ d# 509 node d# 0 org reclaim ]
: pass up b! right a! begin @b ! end ; { , }
 [ d# 508 node d# 0 org reclaim ]
: pass right b! left a! begin @b ! end ; { , }
 [ d# 507 node d# 0 org reclaim ]
: pass left b! right a! begin @b ! end ; { , }
 [ d# 506 node d# 0 org reclaim ]
: pass right b! left a! begin @b ! end ; { , }
 [ d# 505 node d# 0 org reclaim ]
: pass left b! right a! begin @b ! end ; { , }
 [ d# 504 node d# 0 org reclaim ]
: pass right b! left a! begin @b ! end ; { , }
 [ d# 503 node d# 0 org reclaim ]
: pass left b! right a! begin @b ! end ; { , }
 [ d# 502 node d# 0 org reclaim ]
: pass ( right b! left a! begin @b ! end ; ) { , }
 [ d# 501 node d# 0 org reclaim ]
: pass ( left b! right a! begin @b ! end ; )
}block
```

33a ⟨*final-scaler* 33a⟩≡
```
D# 862 code{
 ( scaling code ) [ d# 501 node d# 0 org ] { br }

 ( reduction ) d# 0 { br }

: pol @b h# 8000 ( rw ) and if { cr }
 ( ... ) right b! @b push ex { cr }
 ( ... ) io b! then drop { br }

: check ( -n ) @b h# 800 ( lw ) and if { cr }
 ( ... ... ) left a! push @ { br }

: read ( rn-vr ) over over - push . + ( -rf ) -if { cr }
 ( adjust lo ) drop drop pop dup push d# 3 . + d# 2 { cr }
 ( ... ) then dup - d# 18 . + ( -rfo ) -if { cr }
 ( adjust hi ) drop drop drop pop dup push { cr }
 ( ... ... ) d# 16 . + d# 15 d# 0 { cr }
 ( ... ) then drop pop drop if { cr }
 ( ... ... ) h# 3FFFF . + push d# 1 begin 2* unext { cr }
 ( swap ) then a! push a pop ( -vr ) { br }

 ( ... ... ) right a! over ! pop { cr }
 ( ... ) then drop pol ; < 2e >
}block
```

33b ⟨*final-avg* 33b⟩≡
```
D# 864 code{
 ( averaging buffer code ) [ d# 502 node d# 0 org ]
: pass h# 20 a! d# 31 for { , }
 ( ... ... ) right b! @b !+ { , }
 ( ... ... ) a push h# 20 a! @+ { , }
 ( ... ... ) d# 30 for @+ + 2/ unext { , }
 ( ... ... ) pop a! left b! !b { , }
 ( ... ) next pass ; < 11 >
}block
```