ES6

PWM

Rick van Melis & Simon Lit

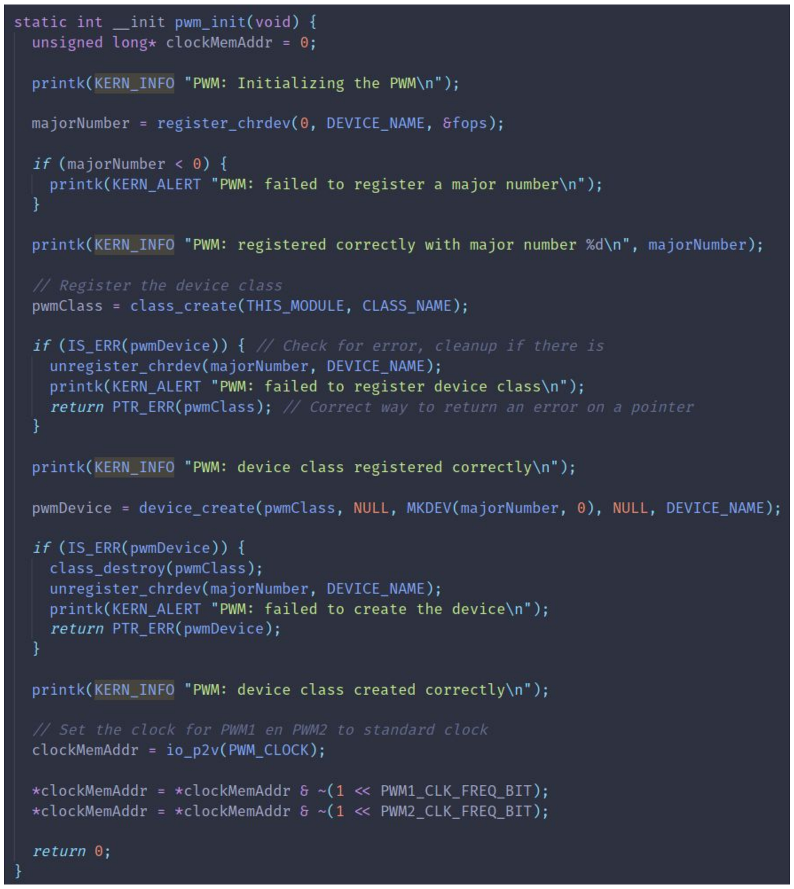
29-03-2019

Pulse width modulation

Our implementation

Creating a devfs node

The first step of creating a devfs node is done programmatically by loading the driver. We register through the init function as listed. In this function we register our drive with a major number and the corresponding file operations. Furthermore we create the device class and the device itself. We check for errors with the function IS_ERR clean up if need be and return the error code with PTR_err supplied by 'linux/err.h'.



```c
static int __init pwm_init(void) {
  unsigned long* clockMemAddr = 0;

  printk(KERN_INFO "PWM: Initializing the PWM\n");

  majorNumber = register_chrdev(0, DEVICE_NAME, &fops);

  if (majorNumber < 0) {
    printk(KERN_ALERT "PWM: failed to register a major number\n");
  }

  printk(KERN_INFO "PWM: registered correctly with major number %d\n", majorNumber);

  // Register the device class
  pwmClass = class_create(THIS_MODULE, CLASS_NAME);

  if (IS_ERR(pwmDevice)) { // Check for error, cleanup if there is
    unregister_chrdev(majorNumber, DEVICE_NAME);
    printk(KERN_ALERT "PWM: failed to register device class\n");
    return PTR_ERR(pwmClass); // Correct way to return an error on a pointer
  }

  printk(KERN_INFO "PWM: device class registered correctly\n");

  pwmDevice = device_create(pwmClass, NULL, MKDEV(majorNumber, 0), NULL, DEVICE_NAME);

  if (IS_ERR(pwmDevice)) {
    class_destroy(pwmClass);
    unregister_chrdev(majorNumber, DEVICE_NAME);
    printk(KERN_ALERT "PWM: failed to create the device\n");
    return PTR_ERR(pwmDevice);
  }

  printk(KERN_INFO "PWM: device class created correctly\n");

  // Set the clock for PWM1 en PWM2 to standard clock
  clockMemAddr = io_p2v(PWM_CLOCK);

  *clockMemAddr = *clockMemAddr & ~(1 << PWM1_CLK_FREQ_BIT);
  *clockMemAddr = *clockMemAddr & ~(1 << PWM2_CLK_FREQ_BIT);

  return 0;
}
```

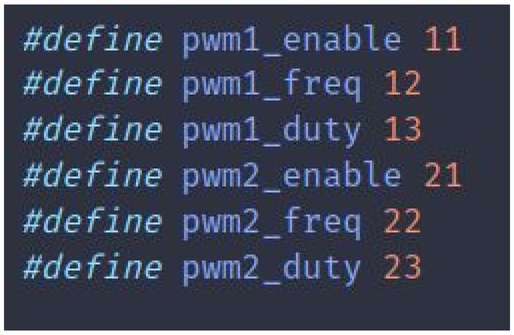
At the end of the init function we set a few bits in the PWM clock control register to select the clock frequency divider. Here we could also set the PWM clock source.

Linking & Communicating with a devfs node

Step 2 requires us to both make and link the device node with mknod pathname mode and dev(Major, Minor). By using mknod multiple times we can create several dev files for the same driver (Major number), each corresponding to a sub-module of our driver (minor).

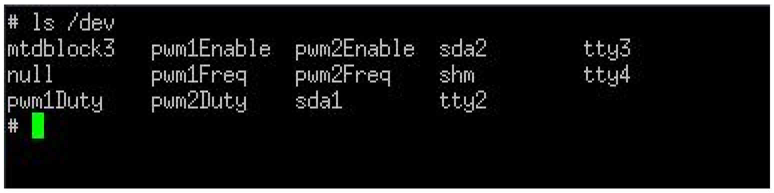
As you can see we have created several files. We have chosen to implement the two pwm with each an enable, freq and duty file. The minor number contains two digits, the first digit

representing the pwm index and and the second digit representing the function.
Three functions are possible: 1 for enable, 2 for frequency and 3 for duty cycle.
This works out to the following options:



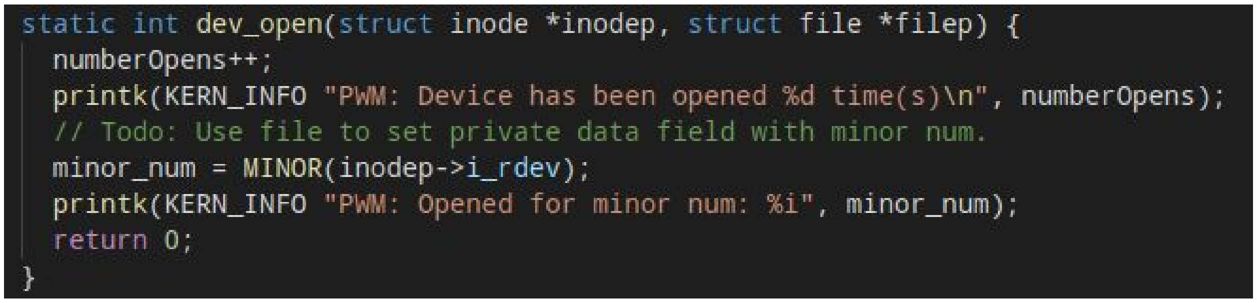
```
#define pwm1_enable 11
#define pwm1_freq 12
#define pwm1_duty 13
#define pwm2_enable 21
#define pwm2_freq 22
#define pwm2_duty 23
```

And after is linked with mknod the dev folder on the device looks like this:



```
# ls /dev
mtdblock3   pwm1Enable  pwm2Enable  sda2        tty3
null        pwm1Freq    pwm2Freq    shm         tty4
pwm1Duty    pwm2Duty    sda1        tty2
#
```

When any file is opened, whether it's through the use of cat/echo or a user space program we receive an inode structure with data on which we can use the MINOR function to retrieve the minor number. To decrease development time we save this number as a global variable which we then use in dev_read. Our research however shows that the golden standard would be to use struct file * filep to set the private data field with the minor number. As the file* is also known to dev_read and dev_write.



```
static int dev_open(struct inode *inodep, struct file *filep) {
    numberOpens++;
    printk(KERN_INFO "PWM: Device has been opened %d time(s)\n", numberOpens);
    // Todo: Use file to set private data field with minor num.
    minor_num = MINOR(inodep->i_rdev);
    printk(KERN_INFO "PWM: Opened for minor num: %i", minor_num);
    return 0;
}
```

Depending upon the minor number we handle reads/writes with the correspondingly named functions of <linux/fs.h>.

Verification & Proof

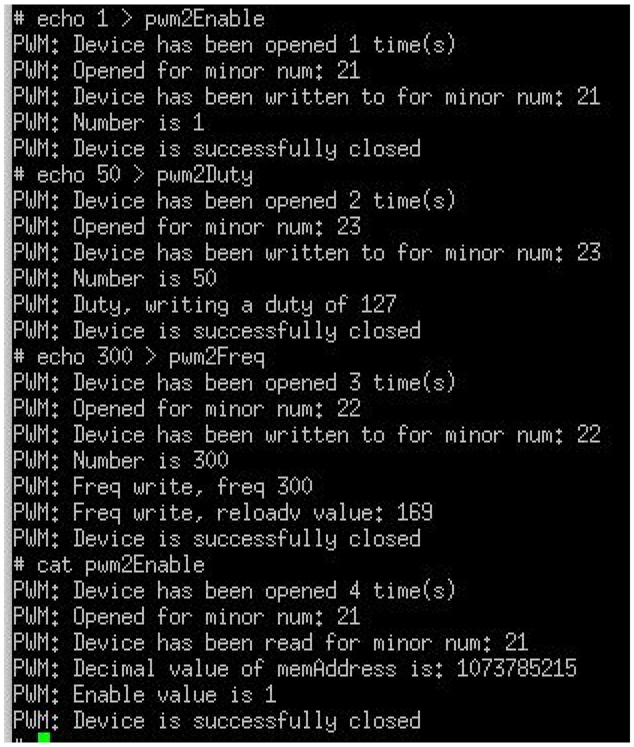
First step would be to see if we can write a value to our register and then read it back from that register. Let's do that!

We write a one into the enable bit (bit 31 of the pwm control register), write a duty cycle of 50% which translates to a value of 127 into bits 7:0. And finally we set the frequency to 300 Hz, resulting in a value of 168 for the 15:8 bits of the register.

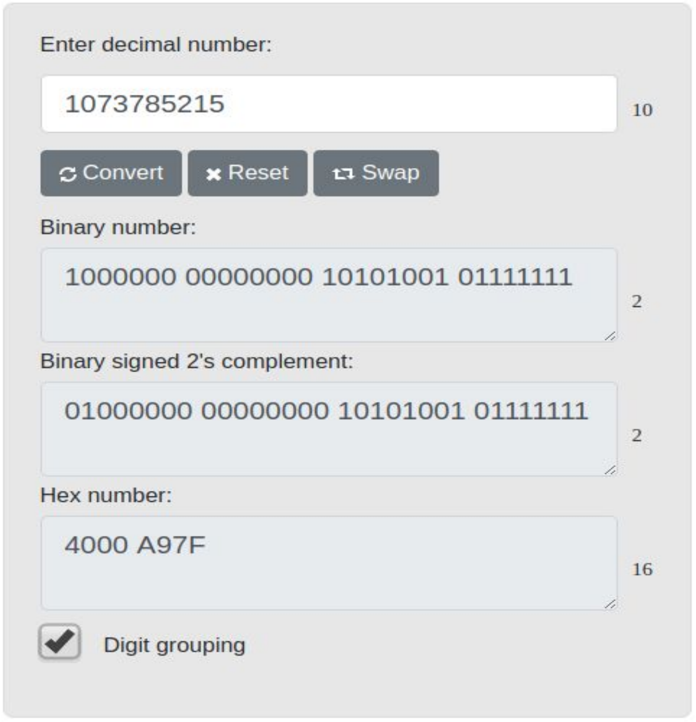
If we read par example the enable of PWM2 we get a value of 1. We also print the decimal value of whatever is in the memory at the PWM 2 Control register.

If we decode to the individual parts of enable, frequency and duty. We get the following hex codes. For duty we get hex 7F and for frequency we get A9 which translates to 127 and 169 respectively.

As you can see in the binary number there is a 1 at position 31, for the enable bit of the register.



```
# echo 1 > pwm2Enable
PWM: Device has been opened 1 time(s)
PWM: Opened for minor num: 21
PWM: Device has been written to for minor num: 21
PWM: Number is 1
PWM: Device is successfully closed
# echo 50 > pwm2Duty
PWM: Device has been opened 2 time(s)
PWM: Opened for minor num: 23
PWM: Device has been written to for minor num: 23
PWM: Number is 50
PWM: Duty, writing a duty of 127
PWM: Device is successfully closed
# echo 300 > pwm2Freq
PWM: Device has been opened 3 time(s)
PWM: Opened for minor num: 22
PWM: Device has been written to for minor num: 22
PWM: Number is 300
PWM: Freq write, freq 300
PWM: Freq write, reloadv value: 169
PWM: Device is successfully closed
# cat pwm2Enable
PWM: Device has been opened 4 time(s)
PWM: Opened for minor num: 21
PWM: Device has been read for minor num: 21
PWM: Decimal value of memAddress is: 1073785215
PWM: Enable value is 1
PWM: Device is successfully closed
```



Enter decimal number:
1073785215 (10)

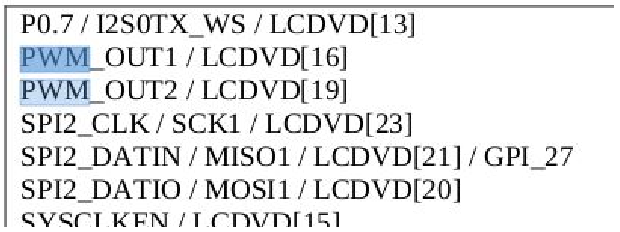
Binary number:
1000000 00000000 10101001 01111111 (2)

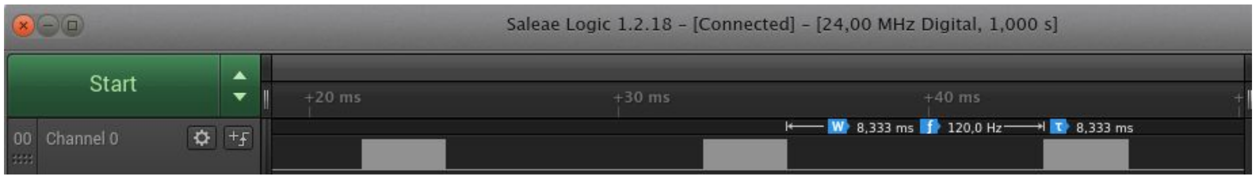
Binary signed 2's complement:
01000000 00000000 10101001 01111111 (2)

Hex number:
4000 A97F (16)

[✓] Digit grouping

If we look at our logic analyzer which is connected to the pins of PWM1 and PWM2. Those are labeled LCDVD 16 and 19 which corresponds to pin 16 of expansion connector J2 and pin 10 of expansion connector J3 respectably. We get the following result.

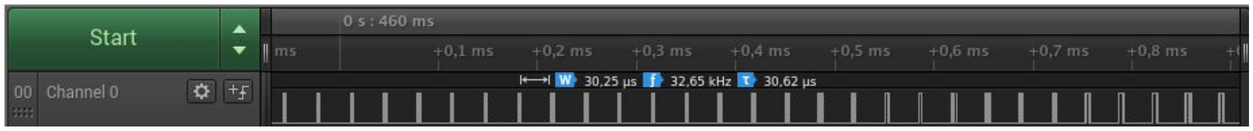

P0.7 / I2S0TX_WS / LCDVD[13]
PWM_OUT1 / LCDVD[16]
PWM_OUT2 / LCDVD[19]
SPI2_CLK / SCK1 / LCDVD[23]
SPI2_DATIN / MISO1 / LCDVD[21] / GPI_27
SPI2_DATIO / MOSI1 / LCDVD[20]
SYSCLKEN / LCDVD[15]


Saleae Logic 1.2.18 – [Connected] – [24,00 MHz Digital, 1,000 s]

W 8,333 ms f 120,0 Hz T 8,333 ms

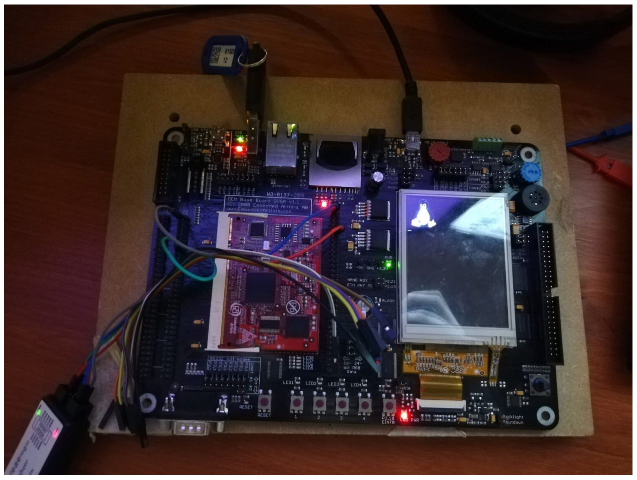
There is a signal so that is good news, however it says 120Hz which does not correspond to the 300 Hz we set for PWM2. No matter what settings we try it does not really move, it is always 120Hz.

If we zoom further in on the blocks of high signal we get the following:


W 30,25 µs f 32,65 kHz T 30,62 µs

Which as far we can tell seems like the signal is set high with a frequency that matches the internal clock frequency of 32KHz.

The logic analyzer we are using is a Saleae logic and the setup is as pictured.

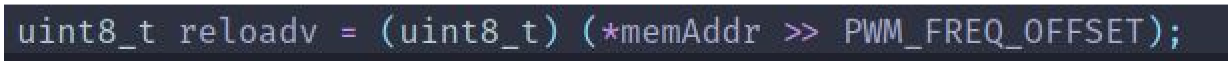


Our calculations for the frequency could be wrong but the frequency should change. We tried setting the CLOCK source values in the PWM Clock register. And as far as we can tell the PWM 1 seems to not send out a signal. Even though is uses the same code as PWM2.

Issues we encountered

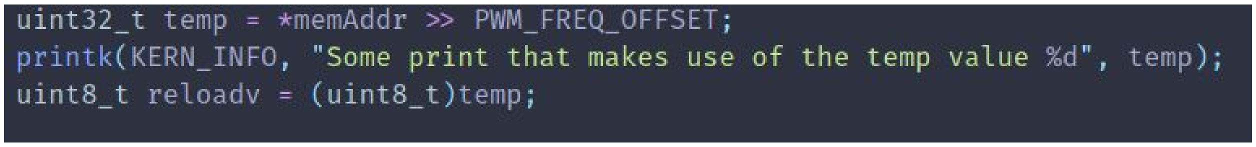
When reading the PWM frequency register we had an interesting problem with reading the correct bytes of the registers. In essence we implemented multiple ways to retrieve 8 bits from a 32 bit register, none seemed to work and we always read 0. When we tried some of these ways locally on our laptop, these all did work and we read the number we wrote.

An example that works local but not on the board is as follows:



```
uint8_t reloadv = (uint8_t) (*memAddr >> PWM_FREQ_OFFSET);
```

On the board though the following does work:



```
uint32_t temp = *memAddr >> PWM_FREQ_OFFSET;
printk(KERN_INFO, "Some print that makes use of the temp value %d", temp);
uint8_t reloadv = (uint8_t)temp;
```

Although if there is no print (or action that makes use of the temp variable) it would not work either.

We figure that what we are seeing here is the compiler doing some optimization, if temp would not be used there is no need to make the temp variable and then that would equal to the same line as code the first example.

Why it works on local and not on arm we don't know.