Fontys University of Applied Sciences

Software & Technology - 6th Semester

Assignment: **PWM**

Elviro Pereira Junior, **2719584**
Rafal Grasman, **289290**

16/04/2018

Assignment:

The LPC3250 has three PWM's: one sophisticated motor-PWM and two normal PWMs.
We want to use the latter two.
In the LPC3250 User Manual, they are called PWM1 and PWM2. After some searching, you can find out the output pins of the IC. Then, you can find the corresponding connector-pin on the OEM-board, and finally: the pins on the large connectors of the development board. Please note that these documents are not error-free! (names might change between the document…).
On the LPC3250, there are only a few hardware registers to control the PWM's. No multiplexers need to be set.

• write a device driver to control both PWM's. A PWM is controlled via three device nodes, for example with names like pwmX_enable, pwmX_freq (or: pwmX_period) and pwmX_duty. Those files are accessible via echo and cat. Typical content of pwmX_enable is 0..1 (off or on), typical content of pwmX_freq is a frequency in Hz, typical content of pwmX_period is the period in milliseconds, typical content of pwmX_duty is 0..100 (in percents)

All those files are connected to one device driver (so in the init_module(), there is one call of register_chrdrv()), and they all share the same file operations.

So the read- or write-operation itself has to find out which file is affected. Examine parameter inode to get this information. When you create the files with mknod, make sure that they have the same major number but a different minor number. See also paragraph 3.2 of http://www.makelinux.net/ldd3/ .

Operations device_read() and device_write() do not have an inode parameter, but device_open() does. A common, legal trick is that device_open() saves the minor number in file>private_data, and that device_read() uses its file->private_data as the minor number. The same applies for device_write().

Please implement both device_read() and device_write(). Write lets a user set a new value, read allows a user to read back the current value.

Proof your implementation by using a scope: either demonstrate your work during class or attach pictures of various PWM settings (both PWM1 and PWM2 together). Give your personal interpretation of these pictures (e.g. with text balloons).

Goal: Communicate with devices through devfs

Research Results

Device Driver Type

First we looked into the different device drivers such as block and char device driver. The main difference between this two is that the char device performs actions immediately (unbuffered) and is used for sending/receiving single bytes of data and a block device is usually to send blocks (multiple bytes) of data which are buffered.[1]

To solve this assignment we decided that we would write a char device driver. The reason we chose a char driver instead of a block driver is because we'll be sending 1 byte at a time and not a whole block of data, and need the action to be performed immediately.

PWM on LPC3250 board

The LPC3250 board has two output PWM's and one motor control PWM[2]. For this assignment the two output PWM's will be controlled.

Features[3]

- Two 8-bit PWM's
- Clocked by the main peripheral clock or the 32 kHz RTC clock.
- Programmable 4-bit prescaler.
- Duty cycle programmable in 255 steps.
- Output frequency up to 50 kHz when using a 13 MHz peripheral clock.

Registers / Addresses

Multiple registers control all the aspects of the PWM's like frequency, duty cycle, on/off.

The address for the PWM's control register is[4]:

0x4005C000 (PWM1)

0x4005C004 (PWM2)

0x4005 C000	PWM1 and PWM2	7	FAB

Table 610. Pulse Width Modulator register map

Address offset	Name	Description	Reset value	Type
0x4005 C000	PWM1_CTRL	PWM1 Control register.	0x0	R/W
0x4005 C004	PWM2_CTRL	PWM2 Control register.	0x0	R/W

31.3.1 PWM1 Control Register (PWM1_CTRL, RW - 0x4005 C000)

Table 611. PWM1 Control Register (PWM1_CTRL, RW - 0x4005 C000)

Bits	Description	Reset value
31	PWM1_EN This bit gates the PWM_CLK signal and enables the external output pin to the PWM_PIN_STATE logical level. 0 = PWM disabled. (Default) 1 = PWM enabled.	0
30	PWM1_PIN_LEVEL If the PWM1_EN bit is set to 0, the PWM_OUT1 pin is set to the PWM1_PIN_LEVEL logical state. (Default = 0)	0
29:16	Not used Write is don't care, Read returns random value.	0
15:8	PWM1_RELOADV Reload value for the PWM output frequency. Fout = (PWM_CLK / PWM_RELOADV) / 256. A value of 0 is treated as 256. (Default = 0)	0x0
7:0	PWM1_DUTY Adjusts the output duty cycle. [Low]/[High] = [PWM_DUTY] / [256-PWM_DUTY], where 0 < PWM_DUTY <= 255. (Default = 0)	0x0

Table 612. PWM2 Control Register (PWM2_CTRL, RW - 0x4005 C004)

Bits	Description	Reset value
31	PWM2_EN This bit gates the PWM_CLK signal and enables the external output pin to the PWM_PIN_STATE logical level. 0 = PWM disabled. (Default) 1 = PWM enabled.	0
30	PWM2_PIN_LEVEL If the PWM1_EN bit is set to 0, the PWM_OUT1 pin is set to the PWM1_PIN_LEVEL logical state. (Default = 0)	0
29	PWM2_INT 0 = Normal PWM_OUT2 functionality 1 = PWM_OUT2 outputs the internal interrupt status. A low level means that neither nIRQ or nFIQ is active. PWM_OUT2 = nIRQ 'NAND' nFIQ.	0
28:16	Not used Write is don't care, Read returns random value.	0
15:8	PWM2_RELOADV Reload value for the PWM output frequency. Fout = (PWM_CLK / PWM_RELOADV) / 256 (A value of 0 is treated as 256) Default = 0	0x0
7:0	PWM2_DUTY Adjusts the output duty cycle. [Low]/[High] = [PWM_DUTY] / [256-PWM_DUTY], where 0 < PWM_DUTY <= 255. (Default = 0)	0x0

The bits of interest overlap in both PWM registers. EN bits to enable/disable (1/0) the PWM, DUTY for setting the duty cycle (0 - 0%, 255 - 100%) and RELOADV for scaling the output frequency.

There is also the PWM Clock Control register (PWMCLK_CTRL - 0x4000 40B8) which allows you to select the clocks for the PWMs[5]:

Bit	Function	Reset value
31:12	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	-
11:8	PWM2_FREQ. Controls the clock divider for PWM2. 0000 = PWM2_CLK = off 0001 = PWM2_CLK = CLKin : : 1111 = PWM2_CLK = CLKin / 15	0
7:4	PWM1_FREQ. Controls the clock divider for PWM1. The encoding is the same as for PWM2_CLK above.	0
3	PWM2 clock source selection: 0 = 32 kHz RTC_CLK 1 = PERIPH_CLK	0
2	PWM2 block enable 0 = Disable clock to PWM2 block. 1 = Enable clock to PWM2 block.	0
1	PWM1 clock source selection: 0 = 32 kHz RTC_CLK 1 = PERIPH_CLK	0
0	PWM1 block enable 0 = Disable clock to PWM1 block. 1 = Enable clock to PWM1 block.	0

With these registers, the frequency, duty cycle and on/off can be controlled for PWM1 and PWM2.
For how the registers are controlled from C refer to the Peek & Poke documentation.

Frequency

The formula for the output frequency is (as can be seen in the screenshot from the documentation table):

Fout = (CLK / RELOADV) / 256

Then there is the FREQ register which controls the CLK, when 0 the CLK is doing nothing, every other value for FREQ divides the CLK by the value, so a FREQ of 2 would make the CLK / 2.

There are two clock sources which allow for different frequencies:

- RTC: a 32768 Hz clock [6]
- Peripheral clock: a clock running at 13 MHz which is ultimately based on OSC_CLK (at default settings) [6]

So the final frequency formula is:

Fout = ((SELECTED_CLOCK / FREQ) / RELOADV) / 256

Which means two options (and eliminating the divisions and [/ 256] at the same time):

Can be rewritten to Fout = 128 / (FREQ * RELOADV)
Or:
Fout = 50781.25 / (FREQ * RELOADV)

FREQ is at minimum 1 and RELOADV is default 256, minimum 1 (else the PWM is off).

The maximum divider is 15 (FREQ) * 256 (RELOADV) = 3840.
The minimum divider is 1.

This means that the minimum frequency is 128/3840 = 0.033 Hz and maximum frequency is 50781.25 Hz (50.78125 kHz).

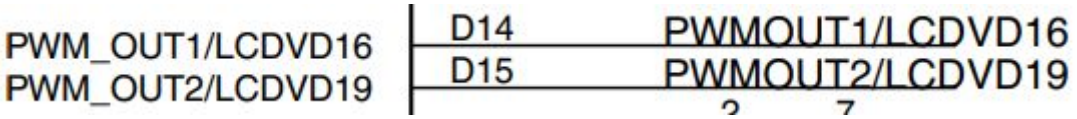
Because 50781.25/3840 = 13.22 Hz, and falls below the maximum 128 Hz that can be made with the help of the RTC, this means that the whole range between 0.033 Hz and 50.78125 kHz is available (no gap). The error is the smallest at respectively 0.033 Hz when using RTC_CLK and 13.22 Hz when using PERIPH_CLK, using higher frequencies (nearing the maximums) means larger errors.

To keep it simple the FREQ and RELOADV values can be determined by trying which combination of values yields the least distance to the wanted value [wanted value = FREQ * RELOADV as in wanted value = 50781.25 / wanted frequency, or wanted value = 128 / wanted frequency]. A function shall be written to determine the best approximation.

Physical Ports

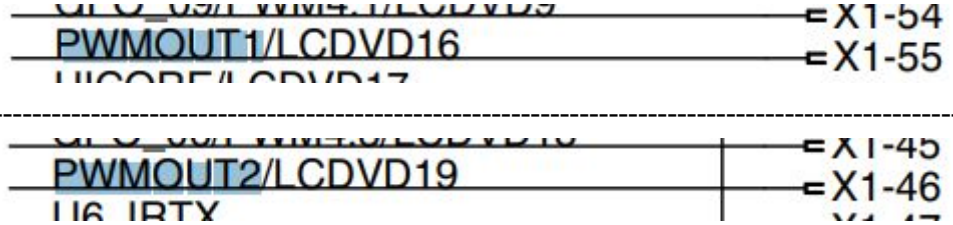
Next to be able to test if the ports go on/off, have the right frequency and cycle using for example a logic analyser or an oscilloscope, the physical pins need to be determined.

On the OEM board schematic we started by looking for PWM_OUT1 (and PWM_OUT2) as named in chapter 31.2 of the DataSheet:



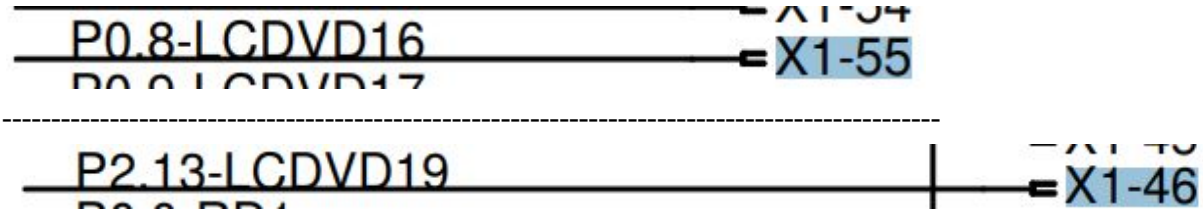
(base_schematic_v1.1 schematic)

These go to PWMOUT1 (LCDVD16) and PWMOUT2 (LCDVD19):



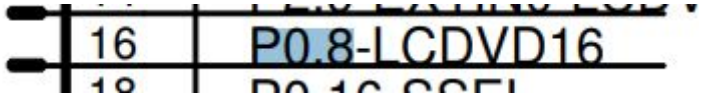
(base_schematic_v1.1 schematic)

Which go to eXternal connectors X1-55 and X1-46:

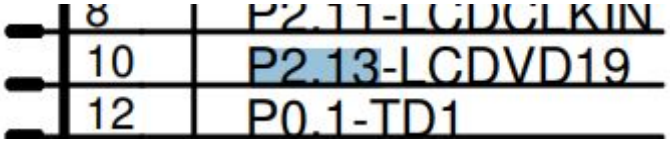


(LPC32x0_OEM_Board_v1.3 schematic)

And P0.8, P2.13 connectors go to:



(pin 16 on Expansion connector J2)



(pin 10 on Expansion connector J3)

(LPC32x0_OEM_Board_v1.3 schematic)

So, J2.Pin16 is PWM OUT 1, and J3.Pin10 is PWM OUT 2.

DevFS Usage/Implementation

For this assignment we have created a kernel driver module (PWM) which is controlled by 3 device nodes (PWM_frequency, PWM_duty, PWM_enable). Device nodes are responsible, for communicating with the hardware. The module registers all the device nodes, which means all devices are dependent/belong to the module (they share the same major number).

The module can now be used to interact with those device nodes and to interact with the physical pins. The device filesystem allows <echo “value” > /dev/DesiredDevice > to write and <cat /dev/DesiredDevice> to read, and the kernel will call our module, pass the minor number of the DesiredDevice, and then the module decides which function to be executed depending on the minor number.

NOTE: It is required to manually create device nodes in the dev file system which corresponds to the driver major and minor number.

Mknod /dev/<device> <major> <minor>

For the driver we write the following values have been assigned:

mknod /dev/PWM1_enable c 253 0
mknod /dev/PWM1_frequency c 253 1
mknod /dev/PWM1_duty c 253 2
mknod /dev/PWM2_enable c 253 3
mknod /dev/PWM2_frequency c 253 4
mknod /dev/PWM2_duty c 253 5

NOTE: the same notation works for the PWM1 & PWM2, the minor numbers go from 5 - 0 (0-2 for PWM1 and 3 - 5 for PWM2).

Design Decisions

For both PWM's each, three device files will be made:

/dev/PWM<number>_enable (either 1 or 0)
/dev/PWM<number>_frequency (in millihertz)
/dev/PWM<number>_duty (in percent without %)

Writing 1 to 'enable' will enable the PWM, writing '0' will disable it. Reading the file will return the current status.
The duty cycle is a value between [0, 100] which will determine the duty cycle of the PWM in % (10 = 10% and so on). Reading this file will return the current duty cycle.
Writing the frequency in millihertz to the frequency file will make the PWM go at (Approximately) the selected frequency. Reading from this file will return the actual (approximated from input) frequency at which the PWM is running.

The following files will be created:

- /dev/PWM1_enable
- /dev/PWM1_frequency
- /dev/PWM1_duty
- /dev/PWM2_enable
- /dev/PWM2_frequency
- /dev/PWM2_duty

Millihertz has been chosen for the frequency to allow for greater precision, because kernel modules don't support floating point operations natively which means we need to work with integers. Making the integers bigger allows for less errors when dividing and doing further operations.

Testing

To J2.Pin16 and J3.Pin10 a logic analyzer will be connected. Both PWM's will be measured independently and different duty cycles, frequencies and on/off stated will be tested, for example:

- PWM1 ON, 50% duty, 50Hz, PWM2 OFF
- PWM1 OFF, PWM2 ON, 75% duty, 25Hz
- PWM1 ON, 33% duty, 10Hz, PWM2 ON, 25% duty, 20Hz
- PWM1 OFF, PWM2 OFF

By doing:

- A
    - echo "1" > /dev/PWM1_enable
    - echo "50" > /dev/PWM1_duty
    - echo "50000" > /dev/PWM1_frequency
    - echo "0" > /dev/PWM2_enable
- B
    - echo "0" > /dev/PWM1_enable
    - echo "1" > /dev/PWM2_enable
    - echo "75" > /dev/PWM2_duty
    - echo "25000" > /dev/PWM2_frequency
- C
    - echo "1" > /dev/PWM1_enable
    - echo "33" > /dev/PWM1_duty
    - echo "10000" > /dev/PWM1_frequency
    - echo "1" > /dev/PWM2_enable
    - echo "25" > /dev/PWM2_duty
    - echo "20000" > /dev/PWM2_frequency
- D
    - echo "0" > /dev/PWM1_enable
    - echo "0" > /dev/PWM2_enable

The results of using the accompanying Logic Analyzer software are visible in "Screenshots" section, and will be compared to the expected values, by using read operations (for all elements):

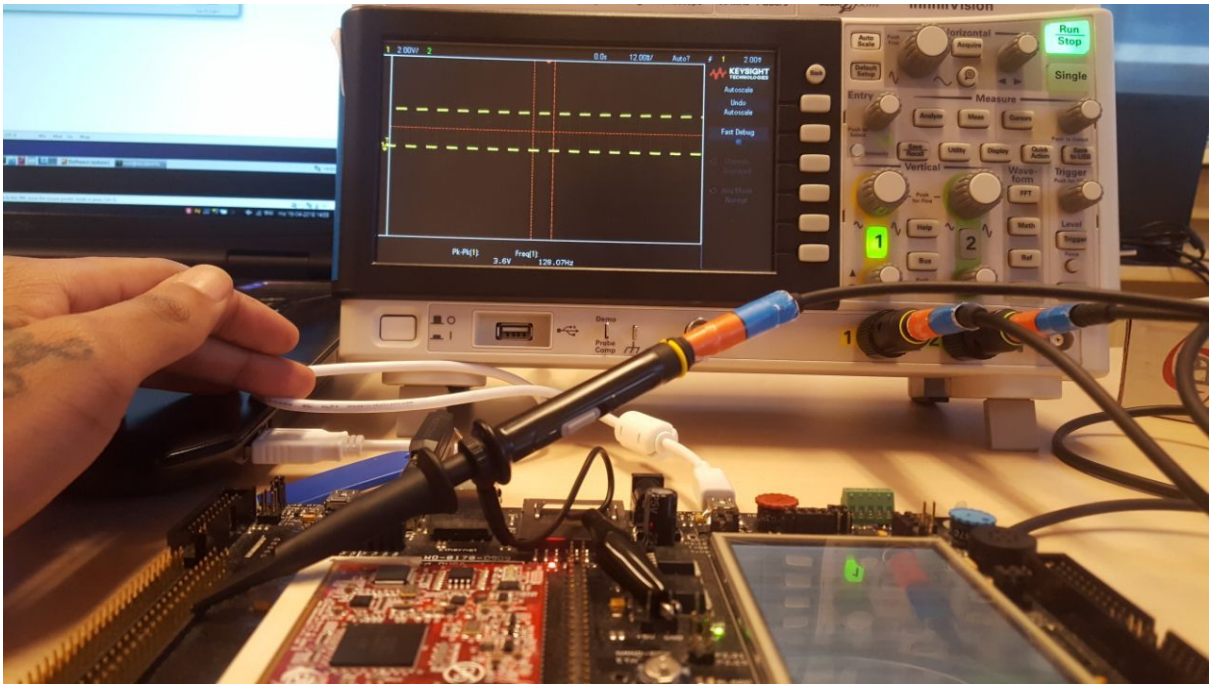
- cat /dev/PWM1_enable
- cat /dev/PWM1_frequency
- cat /dev/PWM1_duty
- cat /dev/PWM2_enable
- cat /dev/PWM2_frequency
- cat /dev/PWM2_duty

To confirm the program works as expected, the module from assignment 1 (Peek & Poke) is used to r/w the registers too.

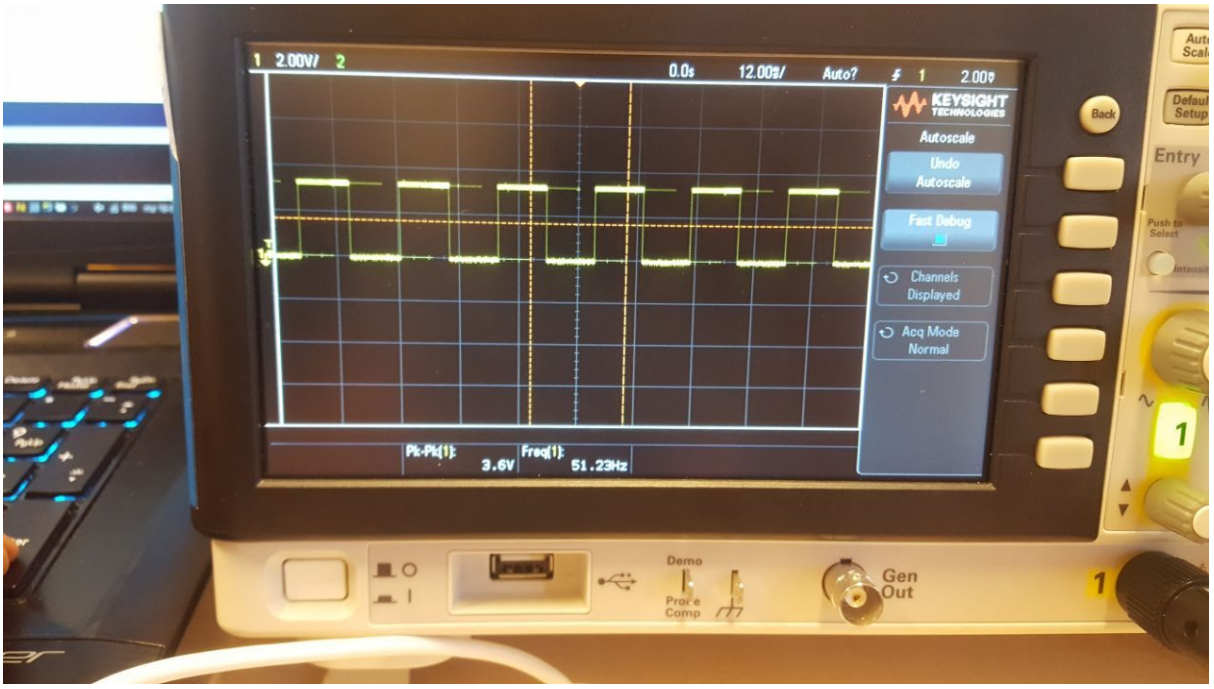
ScreenShots / Results

(debug clutter has been removed from code and filtered out below)

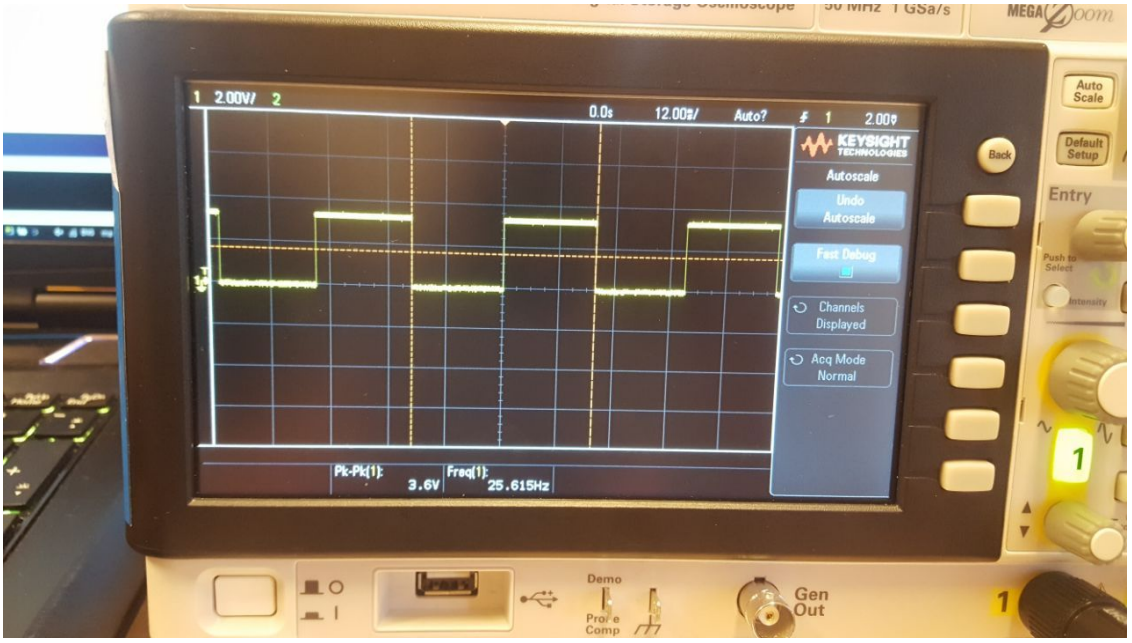
---
insmod mpwm.ko
mpwm: module license 'Custom' taints kernel.
Disabling lock debugging due to kernel taint
mknod /dev/PWM1_enable c 253 0
mknod /dev/PWM1_frequency c 253 1
mknod /dev/PWM1_duty c 253 2
mknod /dev/PWM2_enable c 253 3
mknod /dev/PWM2_frequency c 253 4
mknod /dev/PWM2_duty c 253 5
mknod /dev/PWM1_enable c 253 0
mknod /dev/PWM1_frequency c 253 1
mknod /dev/PWM1_duty c 253 2
mknod /dev/PWM2_enable c 253 3
mknod /dev/PWM2_frequency c 253 4
mknod /dev/PWM2_duty c 253 5



Test A:
echo "1" > /dev/PWM1_enable
echo "50" > /dev/PWM1_duty
echo "50000" > /dev/PWM1_frequency
echo "0" > /dev/PWM2_enable
cat /dev/PWM1_enable
cat /dev/PWM1_frequency
49981
cat /dev/PWM1_duty
49
cat /dev/PWM2_enable
0



Test B:
echo "0" > /dev/PWM1_enable
echo "1" > /dev/PWM2_enable
echo "75" > /dev/PWM2_duty
echo "25000" > /dev/PWM2_frequency
cat /dev/PWM1_enable
0
cat /dev/PWM2_enable
1
cat /dev/PWM2_frequency
49981
cat /dev/PWM2_duty
74



Test C:

```
# echo "1" > /dev/PWM1_enable
# echo "33" > /dev/PWM1_duty
# echo "10000" > /dev/PWM1_frequency
# echo "1" > /dev/PWM2_enable
# echo "25" > /dev/PWM2_duty
# echo "20000" > /dev/PWM2_frequency
# cat /dev/PWM1_enable
1
# cat /dev/PWM1_frequency
13224
# cat /dev/PWM1_duty
32
# cat /dev/PWM2_enable
1
# cat /dev/PWM2_frequency
13328
# cat /dev/PWM2_duty
24
```

Test D:

```
# echo "0" > /dev/PWM1_enable
# echo "0" > /dev/PWM2_enable
# cat /dev/PWM1_enable
0
# cat /dev/PWM2_enable
0
```

Reference

[1] Reference to mknod manual page
[2] DataSheet (Chapter 1: LPC32x0 Introductory information) [page 6]
[3] DataSheet (Chapter 1: LPC32x0 Introductory information) [page 26]
[4] DataSheet (Table 4. Peripheral devices on the LPC32x0) [page 34]
[5] DataSheet (Table 49. PWM Clock Control register) [page 82]
[6] DataSheet (Table 10. Clocks and clock usage) [page 39]