PANDORA PRODUCTS

PWMLib
PWM Library

Jim Schimpf

Document Number: PAN-2015112301
Revision Number: 0.8
3 April 2016

Pandora Products.
215 Uschak Road
Derry, PA 15627

Pandora Products.

215 Uschak Road

Derry, PA 15627

Phone: 724-539.1276

Email: jim.schimpf@gmail.com

Document Revision History

Version	Author	Description	Date
0.1	js	Initial Version	23-Nov-2015
0.2	js	Add example	24-Nov-2015
0.3	js	Handle differing IOCON cmds needed	26-Nov-2015
0.4	js	Added test and summary	26-Nov-2015
0.5	js	Add info on prescale	1-Dec-2015
0.6	js	Interrupt support	10-Dec-2015
0.7	js	Better API documentation	15-Mar-2015
0.8	js	Better API doc on Start	3-Apr-2016

Contents

List of Figures

1 PWM LPC1114 Description

1.1 Introduction

Pulse **W**idth **M**odulation or PWM is the production of a train of square waves with a changeable duty cycle. A square wave is normally produced with the high part of the wave equal to the low part or 50% duty cycle. When PWM is employed you are able to set the duty cycle anywhere from 0% (i.e. OFF) to 100% (i.e. ON). The picture shows some examples.

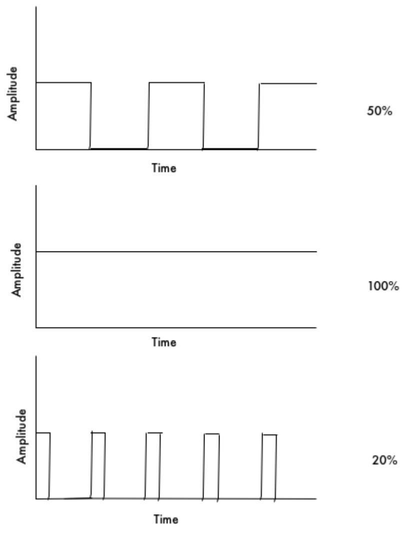

Figure 1: PWM Waveforms

These waveforms can be fed to a capacitor and with different duty cycles can produce a different DC voltage outputs. Or they can be fed to a HEXFET and allow current control of some high power device like a motor or heater.

1.2 LPC1114 PWM Hardware

1.2.1 Pins Available

The LPC1114 has 2 16 bit timers and 2 32 bit timers each of which can run PWM outputs. The LPC1114FN28/102 (28 pin DIP package) has a limited number of these pins brought out. Using Figure 25 in [1] We can build the following table.

Depending on what I/O lines you need to control your system a particular timer's PWM pins may or may not be available. The highlighted pins have other important uses, thus won't be good candidates for PWM use. The design using the processor will determine which of these lines could be used. Also the registers without pin numbers are present but not connected externally on the chip.

Timer	Pin	GPIO Pin	Package #	Other Use
CT16B0	MAT0	PI00_8	1	
	MAT1	PIO0_9	2	
	MAT2	**PIO0_10**	**3**	**SWCLK - Debug**
	MAT3			
CT16B1	MAT0	PIO1_9	18	
	MAT1			
	MAT2			
	MAT3			
CT32B0	**MAT0**	**PIO1_6**	**15**	**Serial Out**
	MAT1	**PIO1_7**	**16**	**Serial In**
	MAT2	**PIO0_1**	**24**	**Proc Boot**
	MAT3	PIO0_11	4	
CT32B1	MAT0	PIO1_1	10	
	MAT1	PIO1_2	11	
	MAT2	**PIO1_3**	**12**	**SWDIO - Debug**
	MAT3	PIO1_4	13	

Table 1: LPC1114FN28/102 PWM Pins

1.2.2 Counter Timer Hardware

The 16 bit counter/timers hardware looks like this:

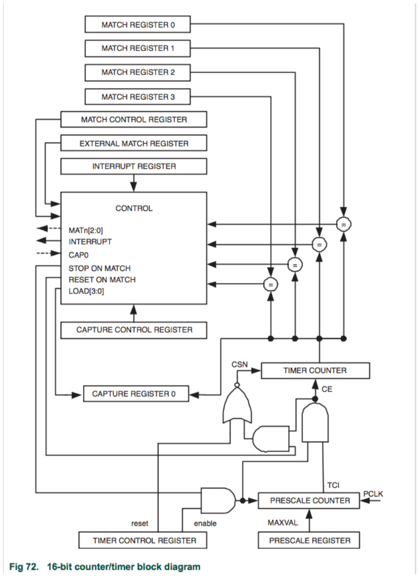

Figure 2: 16 Bit Counter/Timer

While the 32 bit counter timers look like this:

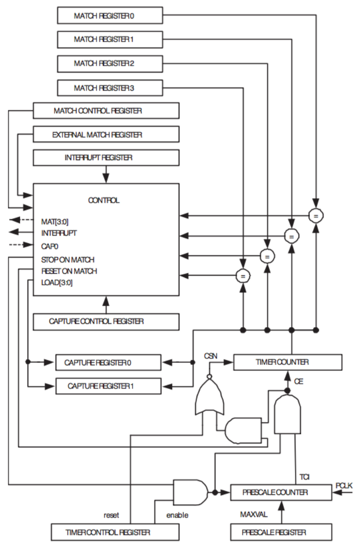

Fig 84. 32-bit counter/timer block diagram

Figure 3: 32 bit Counter Timer

The major difference other than the size of the counter register is that the 32 bit counter timers have 2 capture registers and the 16 bit only have one. For PWM we don't need to consider the capture action.

1.2.3 Operation

The counter/timer is set to count up to a certain value and reset. This is done by putting a value into a match register and setting the counter to reset when it hits this value. Then a second match register is set with a count that when reached causes its output pin to go high. When the reset value is reached that output pin is again reset to 0 for the next cycle.

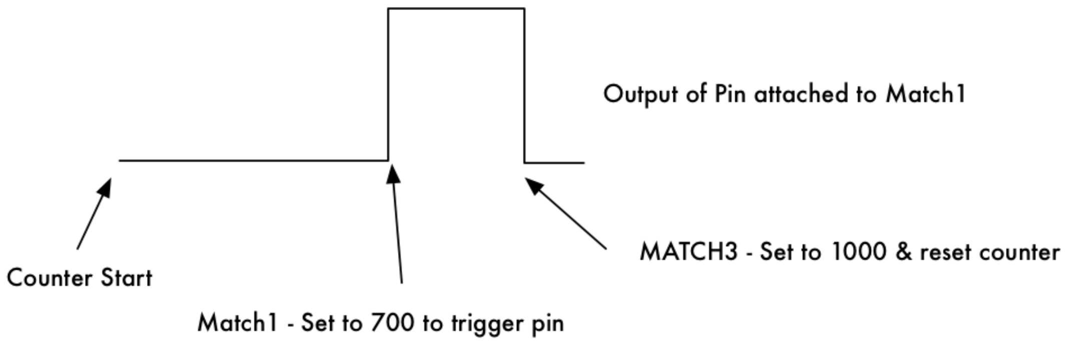
Output of Pin attached to Match1
Counter Start
MATCH3 - Set to 1000 & reset counter
Match1 - Set to 700 to trigger pin

Figure 4: PWM Cycle

This means for PWM operation it requires two Match registers. One to specify the full cycle time and the second hooked to an output pin to supply the PWM waveform. Looking at Table 1 on page

3 when creating a new PWM output, use the register that is NOT connected to a pin for the reset register if possible.

Note that the value put into the output pin match register will be the difference between the full cycle value and your new value. Say the full cycle is 1000 counts. You want a 10% duty cycle, this would be 100 counts. If you put in 100 in the pin match register the waveform would not be correct it would be low (as the counter always starts from reset and the match is low) for 100 cycles then high for 900. The opposite (i.e. 900) is what you want, the API will handle this subtraction for you. This way the waveform stays low till 900 is reached and resets at 1000, giving a duty cycle of 10%.

Finally the frequency of the PWM is somewhat important. Too slow and controlled devices will jitter, too fast and the current controlling devices (FET's etc) that the PWM drives might not respond correctly. The frequency can be controlled to some extent by the counter prescaler register. The board API has a call to get the system clock frequency (usually 48 MHz). That would be the value fed through the prescaler to the counter register. The output frequency of the PWM wave form is

```
f = <System Clock>/(<full cycle count> * <prescale>)

    thus

prescale = <System Clock>/(<full cycle count> * <f>)
```

The PWMLib will calculate the prescale value, the user will need to supply the full cycle count and a desired frequency. There is a slight correction on this as you can see in Figure 2 on page 3 and Figure 3 on page 4 the prescaler works like the match register. The prescale value is put into the prescale register. The prescale counter run by the <System Clock> counts up to the match (prescale value) and one tick is then fed into the main counter.

If we use the above calculation and say we get 1. That is the main counter should be incremented at 48 MHz or 1/clock. But the prescale register would be 1. The prescale counter would then count up to 1 and increment the main counter timer register. But wait the prescale counter would go 0...1 then tick the counter/timer register at 24 MHz rather than 48.

Desired Frequency	Calculated Prescale	Output Frequency	Corrected Prescale	Output Frequency
48	1	24	0	48
24	2	16	2	24
16	4	12	3	16
:	:	:	:	:

Table 2: Prescale Table

As you can see in the above table if we subtract 1 from the calculation above then we will get the desired frequency. This correction done automatically in the library.

1.2.4 Interrupts

The counters can also generate an interrupt each time the counter resets. This can be useful if you wish to say change the PWM duty cycle at every pulse. The is simply done in the setup (see PWMLib_Serup()2.3.1) with the inter flag TRUE.

This enables the interrupt so that when that match register resets the interrupt is generated. The interrupt is vectored though the interrupt controller and in the NXP code it will a named routine for each counter:

```
void TIMER16_0_IRQHandler(void)
void TIMER16_1_IRQHandler(void)
void TIMER32_0_IRQHandler(void)
void TIMER32_1_IRQHandler(void)
```

All you have to do in your code is have a routine named this for the counter you are using counter and on interrupt the processor will jump there and execute your code.

The library will turn on this interrupt when you start the PWM and stop it when you stop the PWM.

2 PWMLib API

2.1 Internal Structure

The PWMLib has an internal structure that is built with the first call and used in all subsequent calls:

```
typedef struct {
                LPC_TIMER_T            *timer;     // Timer used
                uint32_t               preScale;       // Prescaler for timer
                uint32_t               cycleLength;    // PWM cycle in clock ticks
                uint32_t               frequency;      // PWM Frequency
                CHIP_IOCON_PIO_T       pin;            // Processor pin
                uint8_t                pin_ioset;      // Pin Function setting
                int8_t                 rmreg;          // Reset on match
                int8_t                 omreg;          // Output on match
               LPC11CXX_IRQn_Type     irq_vec;  // Interrupt vector
                uint32_t               pwm_set;        // Current value
                bool                   running;
} PWM_Data;
```

And a set of return ENUMs

```
typedef enum {
                PWM_OK = 0,
                PWM_BAD_VALUE_TOO_SMALL = -1,
                PWM_BAD_VALUE_TOO_LARGE = -2,
                PWM_BAD_ACTION          = -3,
                PWM_PIN_NOT_AVAILABLE   = -4,
} PWM_RETURN;
```

2.2 PWM Pin Enable

When pins shown in Table 1 on page 3 are used they must be switched from GPIO use to connect with their respective MAT# register. This is done through I/O setup but does not use the same IOCON_FUNC for each pin. Here is the layout for the LPC1114FN28/102. The following table was developed using Chapter 8 of [1] and looking up each pin.

Timer	Pin	GPIO Function	GPIO Pin	Package #	Other Use
CT16B0	MAT0	IOCON_FUNC2	PIO0_8	1	
	MAT1	IOCON_FUNC2	PIO0_9	2	
	MAT2	IOCON_FUNC3	**PIO0_10**	**3**	**SWCLK - Debug**
	MAT3				
CT16B1	MAT0	IOCON_FUNC1	PIO1_9	18	
	MAT1				
	MAT2				
	MAT3				
CT32B0	**MAT0**	IOCON_FUNC2	**PIO1_6**	**15**	**Serial Out**
	MAT1	IOCON_FUNC2	**PIO1_7**	**16**	**Serial In**
	MAT2	IOCON_FUNC2	**PIO0_1**	**24**	**Proc Boot**
	MAT3	IOCON_FUNC3	PIO_11	4	
CT32B1	MAT0	IOCON_FUNC3	PIO1_1	10	
	MAT1	IOCON_FUNC3	PIO1_2	11	
	MAT2	IOCON_FUNC3	**PIO1_3**	**12**	**SWDIO - Debug**
	MAT3	IOCON_FUNC2	PIO1_4	13	

Table 3: LPC1114FN28/102 GPIO & IO Function

From this table we can develop a structure specific for the LPC1114FN28/102 mapping the MAT# register to an I/O pin and the IOCON_FUNC# needed to set it for PWM output. (If you want the library to run a different LPC1114 package you will have to develop this table for your copy of the library). First we create a structure that holds the data for a single timer.

```
// Timer Charaistics
typedef struct {
LPC_TIMER_T *timer;
// Match GPIO setting
uint8_t mat0;
uint8_t mat1;
uint8_t mat2;
uint8_t mat3;
// GPIO pin
CHIP_IOCON_PIO_T pin_mat0;
CHIP_IOCON_PIO_T pin_mat1;
CHIP_IOCON_PIO_T pin_mat2;
CHIP_IOCON_PIO_T pin_mat3;
                LPC11CXX_IRQn_Type irq;
} TIMER_SPEC;
```

Then in the code we build an IOTable[] that holds the data for all 4 timers. (See in PWMLib.c). When the PWMLib_Setup 2.3.1 is called it has the pointer to the particular timer passed as a parameter. This allows look up by comparing it to the first value in the structure. The code then uses the match register passed in for the PWM output match to pick the IOCON_FUNC in the matX items above and the correct pin label IOCON_PIOX_Y in the pin_matX items for that particular timer.

For interrups the LPC11CXX_IRQn_Type was added to hold the vector value for the particular counter's interrupt. When interrupts are enabled then this is used to set it up for the particular counter

2.3 API

2.3.1 PWMLib_Setup Set up timer for PWM

```
PWM_RETURN PWMLib_Setup( LPC_TIMER_T *pTMR,int freq,int size,
      CHIP_IOCON_PIO_T reset_match,CHIP_IOCON_PIO_T out_match,
      PWM_Data *data,bool inter)
```

INPUT	NAME	USE
	pTMR	Pointer to timer used
	freq	Desired PWM frequency
	size	Full cycle count
	reset_match	Reset counter (i.e. 0-3 for MAT0-3)
	out_match	PWM pin (i.e 0-3 for MAT0-3)
	data	Filled out PWM_Data structure
	inter	TRUE if you want an interrupt at each counter reset
OUTPUT	PWM_RETURN	Status of setup

The reset_match and out_match values specify which match registers (MAT0-MAT3) of the chosen counter are to be used for the end of waveform match (reset_match, MATCH3 in the picture) the output PWM pin (out_match, MATCH1 in the picture). See Figure 4 on page 4.

This call fills out the data in the PWM_Data structure2.1. It first does the table lookup described to get the timer pin data. Note if the values found are 0 then that pin is not connected to the outside and an error is returned. It then calculates the prescale as described in 1.2.3 and returns an error is there is a problem.

NOTE: The duty cycle of the PWM is initially set to 0 by this call. You can call PWMLib_DutyCycle() at any before or after **PWMLib_Start()** time to change the duty cycle.

2.3.2 PWMLib_Start Start PWM

```
void PWMLib_Start(PWM_Data *data)
```

INPUT	**NAME**	**USE**
	data	PWM_Data from PWMLib_Setup
OUTPUT	none	

This call starts the PWM output. The code is rather straightforward except for the PWMC register. This has a bit set for PWM output match register used. bit 0 is set for MAT0, bit 1 for MAT1 etc. So a bit shift of 1 using the MAT# is used to set it.

2.3.3 PWMLib_DutyCycle Set the PWM duty cycle

```
PWM_RETURN PWMLib_DutyCycle(PWM_Data *data,int dutyCycle)
```

INPUT	**NAME**	**USE**
	data	PWM_Data from PWMLib_Setup
	dutyCycle	# Counts
OUTPUT	PWM_RETURN	Status of setup

This allows the setting of the PWM cycle time. The value input is # counts and the PWM duty cycle is equals 100% * #counts/<full cycle count>. Thus if the full cycle count was 2000 then 1000 would be 50% duty cycle. This call can be made before or after PWMLib_Start is called. If run before start then the PWM will begin with that value rather than the default 0.

2.3.4 PWMLib_Stop Stop PWM

```
void PWMLib_Stop(PWM_Data *data)
```

INPUT	**NAME**	**USE**
	data	PWM_Data from PWMLib_Setup
OUTPUT	none	

This call stops the PWM output.

2.4 Simple example

For the example the specification is to produce a 2KHz PWM waveform with a 30% duty cycle on pin 10 of the LPC1114FN28/102. From the chart Table 1 on page 3 we can see pin 10 is attached to CT32B1 Match 0 and the output pin is PIO1_1. The only other item to be determined is how much resolution do we want for the PWM (i.e total count). The larger the total count then the more precisely we can specify the duty cycle. (I.e. if the total count was 10 then you could only specify PWM with to 10%). The other limitation is if you specify too large a total count then the prescale value won't be in range. (See 1.2.3).

For this version we will specify a total count of 4000. Thus a 30% duty cycle would be 30% of 4000 or 1200.

```
PWM_RETURN rtnval;
PWM_Data pwm;
/* Initialize GPIO */
Chip_GPIO_Init(LPC_GPIO);
//                      Timer       Count Freq Reset Match Output Match
rtnval = PWMLib_Setup(LPC_TIMER32_1,4000, 2000,     3,          0,       &pwm);
if( rtnval == PWM_OK )
{
    // Set 30% duty cycle
    rtnval = PWMLib_DutyCycle(%pwm,1200);
    PWMLib_Start(&pwm);
}
else
{
    // Handle Timer setup failure
}
```

The result is this waveform.

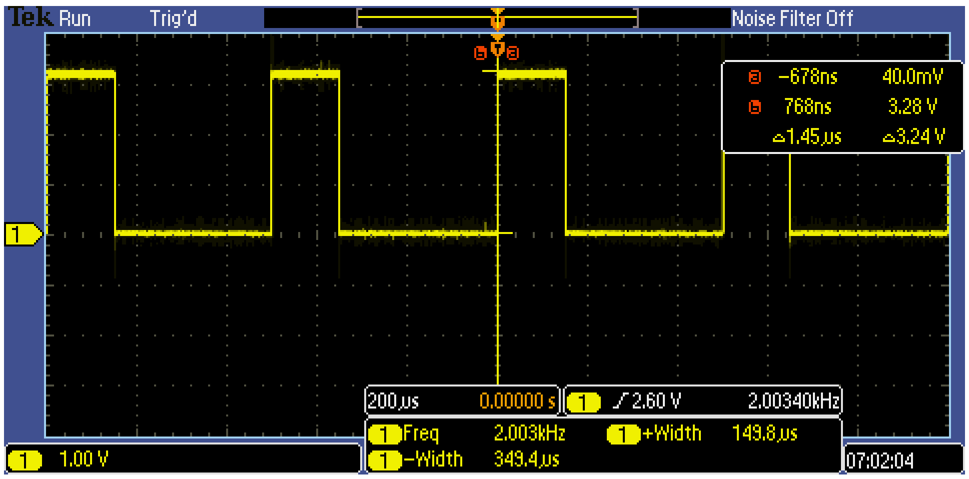

Figure 5: PWM Output

The duty cycle is +Width / (-Width + +Width) = 149.8. usec / (149.8. usec + 349.4 usec) = 0.30008. The frequency is 2.003 kHz which quite close to the desired.

2.5 Interrupt Example

The PWM library also allows you to generate an interrupt on each reset of the reset match register. The code below was written to see if it was possible to change the duty cycle of the PWM on a pulse by pulse basis. At the reset match interrupt, you can write a new value of the duty cycle (with

PWMLib_DutyCycle()) and the next pulse would use that value. You have to do this quickly before the match could occur.

For this test the frequency is 400,000 Hz and the two wave forms we want to produce are:

- 0.5 usec low 2.0 usec high wave
- 1.2 usec low 1.3 usec high wave

This means our interrupt routine will have to change the duty cycle in less than 0.5 usec, the shortest time. This code will test if this is possible with a 48 MHz M0.

2.5.1 Code

```
#include "PWMLib.h"
const unsigned int OscRateIn;
int main(void)
{
    PWM_Data P;
    PWM_Data *pwm = &P;
    /* Initialize GPIO  PIO0_7 as scope marker */
    Chip_GPIO_Init(LPC_GPIO);
    Chip_GPIO_SetPinDIROutput(LPC_GPIO, 0, 7);
    Chip_GPIO_SetPinState(LPC_GPIO, 0, 7, 0);
    // 400,000 Hz frequency
    // MAT3 as reset match
    // MAT0/PIO0_8 PWM bit
    // Enable int on MAT3 reset
    PWMLib_Setup(LPC_TIMER16_0,400000,120,3,0,pwm,true);
    PWMLib_DutyCycle(pwm,24);       // 0.5 us low 2.0 us high
    PWMLib_Start(pwm);
    // Enter an infinite loop
    while(1)
    {
        __WFI();
    }
    return 0 ;
}
void TIMER16_0_IRQHandler(void)
{
    Chip_TIMER_ClearMatch(LPC_TIMER16_0, 3);
    Chip_GPIO_SetPinState(LPC_GPIO, 0, 7, 1);
    Chip_GPIO_SetPinState(LPC_GPIO, 0, 7, 0);
}
```

The code sets up PIO0_7 as a scope marker so we can see when the interrupt takes place. The PWMLib_Setup uses LPC_TIMER16_0 and MAT3 as the reset match and MAT0 (which is attached to PIO0_8 see Table 3 on page 7).

To catch the interrupt, the routine TIMER16_0_IRQHandler() was added. Note the call Chip_TIMER_ClearMatch() MUST be present to clear the interrupt. You need to put in the timer you are using and the MAT register that generated the interrupt. Any other code in the interrupt routine is user dependent. Here we pulsed PIO0_7 so the scope can show us when the routine occured.

2.5.2 Results

The scope was attached to PIO0_8 (PWM out) and PIO0_7 (marker) and we got the following:

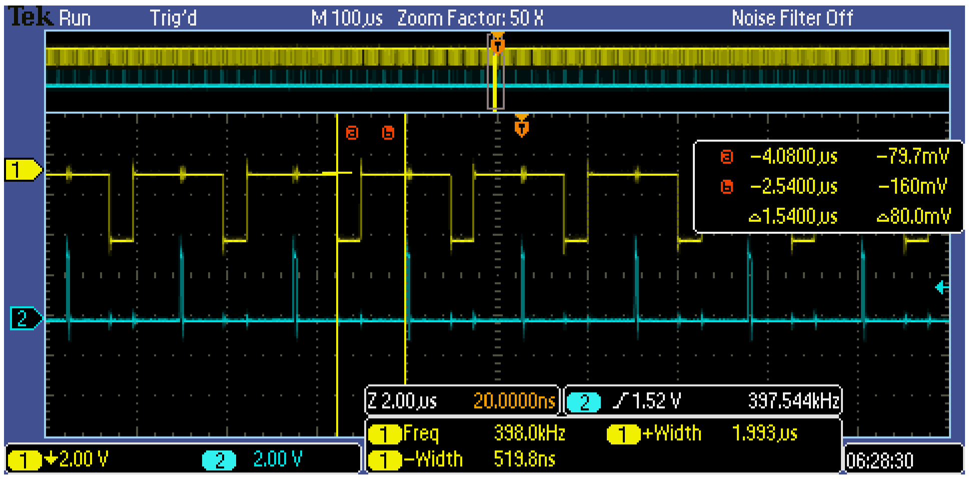

Figure 6: Timer Interrupt

First the good news. Looking at the values at the bottom of the picture the PWM waveform is within 150 ns of the required values so that it is within specification. Looking at the yellow cursors and right hand data block we can see that the interrupt routine occurs 1.5 us after the reset.

The 48 MHz M0 is just not fast enough to change the duty cycle on the fly as the interrupt would have to be less that 0.5 us from the reset match for the cycle by cycle modification to work.

2.6 Summary

The library has been tested on an LPC1114FN28/102 for CT16B0 except for the SWCLK pin. For CT16B1 PIO0_9 (the only pin available). For CT32B0 for all the pins (Serial port was turned off) Note PIO0_1 worked but it was still tied to the 15K pullup for boot. And CT32B1 for all except the SWDIO pin.

References

[1] NXP. *UM10398 LPC111x/LPC11Cxx User manual*. NXP BV, rev. 12.3 edition, June 2014.