



μTasker Document

μTasker – Hardware Timers

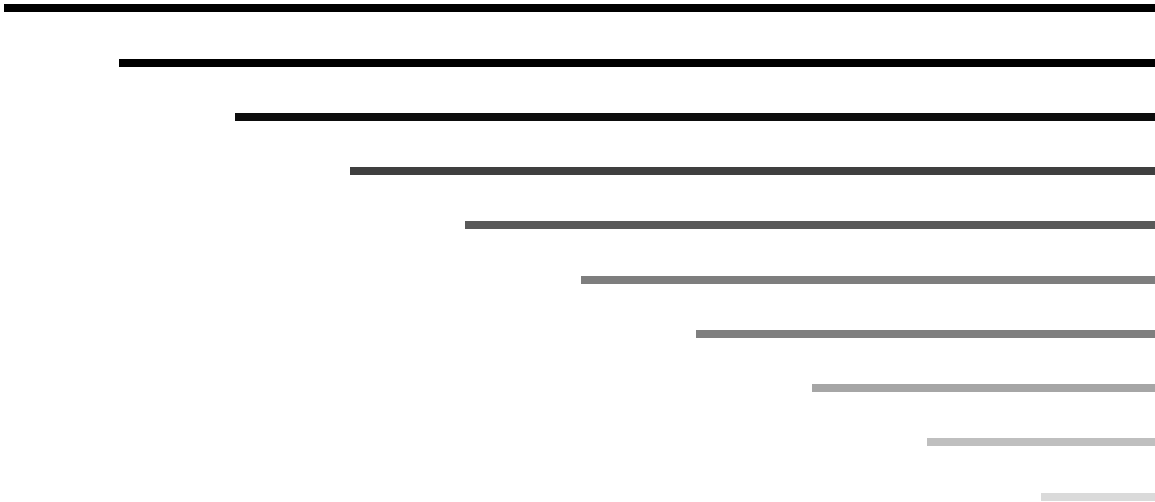


Table of Contents

1. Introduction.....	3
2. Timer Control Interface.....	3
3. Configuring a Single-Shot Time Delay.....	4
4. Configuring a Periodic Interrupt.....	5
5. Configuring a Frequency on a Timer Output Pin.....	5
6. Configuring a Pulse-Width-Modulation Signal on a Timer Output Pin.....	6
7. Configuring a Timer with external Clock Input.....	7
8. Conclusion.....	9
Appendix A – List of Processors and Timer Modules Supported.....	11
Appendix B – Examples of Single-Shot Interrupt Delays.....	11
Appendix C – Examples of Periodic Interrupt.....	11
Appendix D – Examples of Generating Frequencies.....	11
Appendix E – Examples of Generating PWM Signals.....	12

1. Introduction

Processors generally contain a number of timers. These are used, for example, to generate periodic interrupts, delays, frequencies or pulse-width-modulation signals; for counting external events or measuring periods of external signals.

The capabilities and use of such timers can vary greatly depending on the processor type.

This document describes the timer interface in the μTasker project which aids in simple control of such timers in a generic manner. Much of the timers' capabilities can also be simulated in the μTasker simulator, making the verification of new configurations and timer behaviour possible in user projects.

2. Timer Control Interface

The μTasker project uses a common interface for control of various interrupt capable peripherals.

```
fnConfigureInterrupt (*void)
```

The timer control is a particular case of using this interface and its use will be further detailed in the following sections.

In order to use the hardware timer support the specific hardware module(s) in the processor should first be activated. See appendix A for a complete list of timer modules supported in various processor packages.

Most processor types have a general purpose timer module which is activated in `app_hw_XXXX.h` (where `XXXX` is the processor type) by the define `SUPPORT_TIMER` (or similar).

3. Configuring a Single-Shot Time Delay

```
static void fnConfigure_Timer(void)
{
    static TIMER_INTERRUPT_SETUP timer_setup = {0}; // interrupt configuration parameters

    timer_setup.int_type = TIMER_INTERRUPT;
    timer_setup.int_priority = PRIORITY_TIMERS;
    timer_setup.int_handler = timer_int;
    timer_setup.timer_reference = 2; // timer channel 2
    timer_setup.timer_mode = (TIMER_SINGLE_SHOT | TIMER_US_VALUE); // single shot us timer
    timer_setup.timer_value = 100; // 100µ delay
    fnConfigureInterrupt((void *)&timer_setup); // enter interrupt and start timer
}
```

This example (for *Luminary Micro devices*) shows a timer being configured to generate an interrupt after a delay of 100µs. It uses a general purpose timer, whereby channel 2 is used for the delay. When the single-shot timer fires the interrupt call-back `timer_int(void)` is called from within the timer interrupt routine.

```
static void timer_int(void)
{
    TOGGLE_TEST_OUTPUT();
    fnConfigure_Timer();
}
```

This example interrupt routine is toggling an output (for visibility) and restarting a further single-shot hardware timer.

Note that the user interrupt handler doesn't need to reset any hardware flags since the driver interrupt handler is responsible for this work. The user must however be aware that the code is running in a sub-routine to the timer interrupt handler and so should generally be kept as short as possible. It is typical for such routines to send an event to a task so that extra work can be triggered (eg. `fnInterruptMessage(OWN_TASK, TIMEDELAY_1);`).

The timer module will generally be set to low power mode (power down or similar) after a single-shot timer has fired, in order to optimise power requirements when the timer is no longer in use.

See Appendix B for further examples of generating a single-short interrupt delay for various processor types and using various timer modules in the processors.

4. Configuring a Periodic Interrupt

To be added.

5. Configuring a Frequency on a Timer Output Pin

To be added.

6. Configuring a Pulse-Width-Modulation Signal on a Timer Output Pin

It is often possible to generate PWM signals from general purpose timers. Some processors have, in addition, dedicated PWM modules optimised for this task.

The following example shows two PWM signals (CCP0 and CCP1) being generated from a single general purpose timer channel on a Luminary Micro device.

```
static void fnConfigure_Timer(void)
{
    static TIMER_INTERRUPT_SETUP timer_setup = {0}; // interrupt configuration parameters
    timer_setup.int_type = TIMER_INTERRUPT;
    timer_setup.int_priority = PRIORITY_TIMERS;
    timer_setup.int_handler = 0; // no interrupt
    timer_setup.timer_reference = 0; // timer channel 0
    timer_setup.timer_mode = (TIMER_PWM_B); // generate PWM signal on timer output port
    timer_setup.timer_value = TIMER_US_DELAY(TIMER_FREQUENCY_VALUE(1000)); // generate 1000Hz
    timer_setup.pwm_value = 20; // 20% PWM (high/low)
    fnConfigureInterrupt((void *)&timer_setup); // enable PWM signal
    timer_setup.timer_mode = (TIMER_PWM_A | TIMER_DONT_DISTURB);
    // now set output A but don't disturb (reset) output B
    timer_setup.timer_value = TIMER_US_DELAY(TIMER_FREQUENCY_VALUE(1500)); // generate 1500Hz
    timer_setup.pwm_value = 35; // 35% PWM (high/low)
    fnConfigureInterrupt((void *)&timer_setup); // enable PWM signal
}
```

There is no interrupt involved with a PWM channel and the PWM output runs continuously until stopped.

The initialisation includes also the configuration of the port output for PWM use.

By recalling the initialisation but with different frequencies or PWM percentage values (0..100% in steps of 1%) changes to the present setting can be achieved. Whether the `TIMER_DONT_DISTURB` flag is used depends on whether a timer reset (takes place when called without the flag) is desired or not.

The following shows first one channel being stopped (the other will continue to operate) and then the second channel being disable. In this case, when both channels have been disabled, the timer channel will be set back to its power-down state to ensure lowest power consumption when not used.

```
static void fnStop_PWM(void)
{
    static TIMER_INTERRUPT_SETUP timer_setup = {0}; // interrupt configuration parameters
    timer_setup.int_type = TIMER_INTERRUPT;
    timer_setup.timer_reference = 0; // timer channel 0
    timer_setup.timer_mode = (TIMER_STOP_PWM_B | TIMER_DONT_DISTURB);
    // stop B but don't disturb A
    fnConfigureInterrupt((void *)&timer_setup); // disable PWM signal

    timer_setup.timer_mode = (TIMER_STOP_PWM_A); // stop A and power down timer module
    fnConfigureInterrupt((void *)&timer_setup); // disable PWM signal
}
```

Note that once a PWM channel has been disabled its PWM port output state may not be defined (resulting in a continuous '0' or '1'). It may therefore be necessary to convert the port back to GPIO use by adding, for example, `_FLOAT_PORT(B, PORTB_BIT0);` (assuming the PWM output is on port B-0 and the input floating state is suitable). *The state after a channel power down may also be different to the case of simply disabling a module.*

See Appendix E for further examples of generating PWM signals for various processor types and using various timer modules in the processors.

7. Configuring a Timer with external Clock Input

In some cases it is necessary to count external pulses; for example in order to measure an external frequency, measure a pulse width, duty cycle or phase between two inputs. This mode of operation is normally referred to as capture mode.

The following example shows how an input can be configured on the SAM7X for use as clock and subsequently how the timer counter value is read.

The SAM7X has 3 individual 16 bit timers; there are several pins that can be used by the timer as outputs or inputs.

TCLK0, TCLK1, TCLK2 – these are inputs (called external inputs – these can be used as clock inputs)

TIOA0, TIOA1, TIOA2 – these can be inputs or outputs (called internal I/O signals – these can be used as clock inputs)

TIOB0, TIOB1, TIOB2 – these can be inputs or outputs (called internal I/O signals – these can be used as trigger inputs but not clock inputs)

The timer counter can be incremented on either the rising or falling edge of the signal.

When an external clock source is selected it can be XC0, XC1 or XC2, where these are sourced by the following possible combinations:

XC0 can be TCLK0, TIOA1 or TIOA2

XC1 can be TCLK1, TIOA0 or TIOA2

XC2 can be TCLK2, TIOA0 or TIOA1

The maximum frequency of an external signal is $2/5^{\text{th}}$ the master clock.

The timers are flexible so this example is just one of various configurations – it simply configures the input as clock to the timer counter so that the counter is incremented at the frequency of the external signal. The counter runs from its initial value of 0x0000 up to a maximum value of 0xffff. After the value 0xffff is reached it overruns to 0x0000. By reading the timer counter value at two instances in time the external frequency (assuming a stable pulse rate) can be measured. If the timer overflows, an interrupt on the overflow allows the timer width to be increased by incrementing a further variable (creating a 32 bit timer value).

```

static void fnConfigure_Timer(void)
{
    static TIMER_INTERRUPT_SETUP timer_setup = {0}; // interrupt configuration parameters
    timer_setup.int_type = TIMER_INTERRUPT;
    timer_setup.int_priority = PRIORITY_TIMERS;
    timer_setup.int_handler = fnOverflow;           // interrupt handler on overflow
    timer_setup.timer_reference = 0;               // timer channel 0
    timer_setup.timer_mode = (TIMER_SOURCE_TCLK0 | TIMER_SOURCE_RISING_EDGE); // timer clock input and edge
    fnConfigureInterrupt((void *)&timer_setup);  // enable PWM signal
}

```

This example shows timer 0 being set to be clocked from the TCLK0 input, incremented on its rising edge. An interrupt handler is specified for timer counter overflows (a value of 0 for the interrupt handler lets the timer overflow without generating an interrupt).

The possible timer sources are (only one may be defined)

```

TIMER_SOURCE_TCLK0
TIMER_SOURCE_TCLK1
TIMER_SOURCE_TCLK2
TIMER_SOURCE_TIOA0
TIMER_SOURCE_TIOA1
TIMER_SOURCE_TIOA2

```

It is possible to define the same source for more than one timer. The user must however be aware that not all source combinations are possible – for example if TCLK0 and TCLK1 are used by two timers, the third timer cannot use TIOA2 (this is because the external signals XC0 and XC1 have been allocated and TIOA2 is not available via XC2. If this were attempted no clock would be connected. Furthermore, since the allocation of TIOA inputs can be from two possible XC sources the allocation priority is defined as:

- 1) TIOA0 will be taken from XC1, if available. If not it will be taken from XC2
- 2) TIOA1 will be taken from XC2, if available. If not it will be taken from XC0
- 3) TIOA2 will be taken from XC0, if available. If not it will be taken from XC1

Calling `fnConfigureInterrupt((void *)&timer_setup)` with `timer_setup.timer_mode = TIMER_DISABLE;` will disable the timer (power down) and also disconnect its source. This pin will however be left configured as timer input so will need to be reconfigured if required for a different function afterwards.

Assuming that the overflow interrupt is incrementing a variable called `usCounterOverflow` a 32 bit timer value can be read by performing

```

ulCounter = (_COUNTER_VALUE(0) + (usCounterOverflow << 16));

```

This shows the macro `_COUNTER_VALUE()` which is used for direct timer counter register access

8. Conclusion

This document is in progress – not officially released

Modifications:

- V0.01 29.8.2009: First preliminary version with only Luminary Micro specific use
- V0.02 14.1.2010: Add SAM7X PWM details in appendix
- V0.03 10.3.2010: Add external counter mode

Appendix A – List of Processors and Timer Modules Supported

Appendix B – Examples of Single-Shot Interrupt Delays

Appendix C – Examples of Periodic Interrupt

Appendix D – Examples of Generating Frequencies

Appendix E – Examples of Generating PWM Signals

AT91SAM7X – PWM

The SAM7X has a PWM controller with 4 channels. The outputs are called PWM0..PWM3 and each is available on two possible output pins.

In addition to the PWM controller, the general purpose timers can also be used to generate PWM signals. Only the PWM controller is discussed here.

To enable the PWM controller support in the µTasker project the define `SUPPORT_PWM_CONTROLLER` must be set.

The following is an example of using the PWM controller together with the SAM7X. It shows 4 PWM signals being generated, using all 4 PWM channels.

```

TIMER_INTERRUPT_SETUP timer_setup = {0};           // interrupt configuration parameters
timer_setup.int_type = PWM_CONFIGURATION;
timer_setup.timer_reference = 2;                   // PWM channel 2
timer_setup.int_type = PWM_CONFIGURATION;
timer_setup.timer_mode = (TIMER_PWM_ALT);          // configure PWM
signal on alternative PWM2 output
timer_setup.timer_value = TIMER_US_DELAY(TIMER_FREQUENCY_VALUE(1000)); // generate 1000Hz
on timer output
timer_setup.pwm_value = _PWM_PERCENT(20, TIMER_US_DELAY(TIMER_FREQUENCY_VALUE(1000)));
// 20% PWM (high/low)
fnConfigureInterrupt((void *)&timer_setup);      // enter
configuration for PWM test
timer_setup.timer_reference = 3;
timer_setup.timer_mode = (TIMER_PWM);             // generate PWM
signal on PWM3 output and synchronise all PWM outputs
timer_setup.timer_value = TIMER_US_DELAY(TIMER_FREQUENCY_VALUE(1500));
timer_setup.pwm_value = _PWM_TENTH_PERCENT(706,
TIMER_US_DELAY(TIMER_FREQUENCY_VALUE(1500))); // 70.6% PWM (high/low) on different channel
fnConfigureInterrupt((void *)&timer_setup);      // enter
configuration for PWM test
timer_setup.timer_reference = 0;
timer_setup.timer_mode = (TIMER_PWM_ALT);         // generate PWM
signal on PWM0
timer_setup.timer_value = TIMER_US_DELAY(TIMER_FREQUENCY_VALUE(2000));
timer_setup.pwm_value = _PWM_TENTH_PERCENT(995,
TIMER_US_DELAY(TIMER_FREQUENCY_VALUE(2000))); // 99.5% PWM (high/low) on different channel
fnConfigureInterrupt((void *)&timer_setup);      // enter
configuration for PWM test
timer_setup.timer_reference = 1;
timer_setup.timer_mode = (TIMER_PWM | TIMER_PWM_START_0 | TIMER_PWM_START_1 |
TIMER_PWM_START_2 | TIMER_PWM_START_3); // generate PWM signal on PWM1 output and synchronise
all PWM outputs
timer_setup.timer_value = TIMER_US_DELAY(TIMER_FREQUENCY_VALUE(3000));
timer_setup.pwm_value = _PWM_TENTH_PERCENT(25,
TIMER_US_DELAY(TIMER_FREQUENCY_VALUE(3000))); // 2.5% PWM (high/low) on different channel
fnConfigureInterrupt((void *)&timer_setup);      // enter interrupt
for timer test

```

Note the following points:

- 1) Since the PWM controller is being used, and not a general purpose timer, the `int_type` is set to `PWM_CONFIGURATION`. The `TIMER_INTERRUPT_SETUP` is otherwise used as for timer control.
- 2) `timer_reference` is used to specify the PWM controller channel (0..3). In the example all 4 channels are configured.

- 3) The primary output pin is used when the mode is set to `TIMER_PWM` and the alternative output is used when `TIMER_PWM_ALT` is set. The primary outputs are PB19, PB20, PB21, PB22 of the PWM controller channels 0, 1, 2 and 3. The secondary outputs are PB27, PB28, PB29 and PB30 for the PWM controller channels 0, 1, 2 and 3.
- 4) Although the 4 channels are configured independently, their counters are not actually started until the final channel is configured. All 4 are started using `TIMER_PWM_START_0 | TIMER_PWM_START_1 | TIMER_PWM_START_2 | TIMER_PWM_START_3`, which has the effect of synchronising all 4 channels.
- 5) To stop channels from operating the mode flags `TIMER_PWM_STOP_0`, `TIMER_PWM_STOP_1`, `TIMER_PWM_STOP_2` and/or `TIMER_PWM_STOP_3` can be used. Should no further channels be running after this command the PWM controller will be powered down. In the powered down state the settings are however retained in the PWM module in the SAM7X.