

OSK5912 General Purpose Timers

Matthew Percival

25th August 2005

Contents

| | | |
|----------|---|-----------|
| 1 | The Usual Business | 2 |
| 2 | Introduction | 3 |
| 3 | General Usage | 4 |
| 3.1 | Simple sysfs Operations | 4 |
| 3.1.1 | Additional Operations | 5 |
| 3.2 | Pulse Width Modulation sysfs Operations | 6 |
| 3.3 | Event Capture sysfs Operations | 7 |
| 3.4 | Complex sysfs Operations | 7 |
| 4 | Example Usage | 10 |
| 4.1 | Interrupt on Timer Match | 10 |
| 4.2 | Event Capture from Pulse Width Modulation | 10 |
| 4.3 | Automatic Wake-Up Configuration | 11 |
| 4.4 | Pulse Width Modulation | 11 |
| 4.5 | Frequency Sweep | 11 |
| 4.6 | Event Capture | 11 |
| 4.7 | Frequency Sweep Chained to Event Capture | 12 |

1 The Usual Business

Copyright ©2005 Matthew Percival

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license can be found at <http://www.gnu.org/licenses/fdl.html>.

If you see anything wrong or missing in this document, please let me know so that I can update it. Additional areas where people trip-up are also welcome, if they are included with a solution and a means by which to verify this.

2 Introduction

This document explains the workings of the General Purpose (or Dual Mode) Timers for the OMAP5912 Starter Kit, specifically how they are used with the operations available in sysfs — mostly found in arch/arm/plat-omap/dmtimers.c. The OSK comes with eight GP Timers, three of which have programmable pulse width modulation exportable to multiplexed pins, while another two also have input capture available from other pins. These timers can also continue to count while the system is in deep sleep, and as such can be used to wake the system up on an event. In addition, because the GP Timers are on the shared bus, both the ARM and DSP can use them, at the discretion of the ARM.

Beyond the obvious use of counting a period of time, such as counting how long the system has been in deep sleep for, they can be used for a variety of other tasks. Obviously, as mentioned, they can wake the system after a specified period of time, but this is not the full extent of their uses.

Timers 1–3 are available on external pins to output a programmable pulse. This can either be a continuous frequency, run either over a long period of time or for a short burst, or a frequency sweep, smoothly transitioning from high to low frequencies, or vice versa. Because these events are available on pins, they can be used by other devices, perhaps driving a clock on a bus, or even interacting with a completely different device.

Timers 4–5 are also available on pins, however, they receive input, and can be configured to capture their current count upon receiving an edge from this input frequency. At its most basic usage, this could be used to time how long it takes for a particular external event to occur. Alternatively, it could take a series of samples, from which the frequency of the input wave could be calculated. It should be noted that Texas Instruments seem have withdrawn support for this feature, as it is no longer mentioned in the newest revision of the timer manual (spru891). It was also noted in the silicon errata (DM_TIMER_1 in sprz209d) that TI had no intention to correct this error. The device will fail to trigger on around 2% of edges, and seems to interrupt at unusual times for the first couple of edges.

3 General Usage

The most simple way to use a GP Timer is to start it, and let it tick until overflow: with default settings, once it reaches overflow (0xFFFFFFFF) it will reset (to 0x0) and stop. To make things more interesting, the match and load registers can be set, compare and auto reload can be enabled, and interrupts configured. With just a few operations, things can become even more interesting: by selecting a pulse mode and trigger, and configuring a multiplexed pin, the timer can output a programmable pulse width modulation.

Normally this can mostly be done by directly changing the registers, however, by taking advantage of the sysfs some operations can be done more simply. By mounting the sysfs, and changing to the `/sys/devices/system/dmtimer/` directory, you will find eight subdirectories, each relating to a different GP Timer. Some of these timers have special operations which are only available to them, but there are several operations available to all timers.

3.1 Simple sysfs Operations

These operations are available to all GP Timers, and are operations which either cannot be performed in user-space, such as setting an interrupt service routine, or are more easily done through an interface such as this.

dm_use: This file is perhaps the most important in each of the `dmtimer[0-7]/` directories, as it acquires a lock on the timer for you to be able to use it. On a read, it calls `omap_dm_timer_use_read()`, which returns a 1 if you have access to the timer, and 0 if you do not. Writing to this file calls `omap_dm_timer_use_toggle()`, which in turn calls `omap_dm_timer_request_sel()`, which attempts to get you a lock on the timer. Writing 'claim' allows you to get the timer, while 'release' frees it again. NOTE: You cannot perform any write actions to any other files in the `dmtimer[0-7]/` directories until you have a lock on the specific timer.

dm_clock: The GP Timers can select between three clocks for their functional domain: these are the ARMXOR peripheral clock, the 32KHz input clock and the an external clock pin. Reading this file calls `omap_dm_timer_clock_read()`, which returns the `GPTIMERx_CLK_SEL` bit of the `MOD_CONF_CTRL_1` register: 0 for the ARMXOR clock, 1 for the 32KHz clock, and 2 for the external clock. Writing calls `omap_dm_timer_clock_set()`, which in turn calls `omap_dm_timer_set_source()`. Writing 'ARMXOR', '32_KHZ' or 'EXTCLK' selects their respective clocks. It should

also be noted that `EXTCLK_HZ` will need to be modified with the frequency of the external clock (in `include/asm-arm/arm/arch-omap/dmtimer.h`) before it can be used with some of the functions mentioned later.

dm_interrupt: There are three interrupts available for the GP Timers: on match, overflow, or capture. Reading this file calls `omap_dm_timer_int_read()` which returns the last interrupt to occur: 1 is a match interrupt, 2 an overflow interrupt, and 4 a capture interrupt. Writing to this file calls `omap_dm_timer_int_set()`, which in turn calls `omap_dm_timer_set_int_enable()`. Writing 'match' or 'overflow' enables their respective interrupts. 'capture' will also enable the capture interrupt in `dmtimer3/` and `dmtimer4/`. This method only uses the simple `omap_dm_timer_interrupt()` handler, however, which merely clears the interrupt.

dm_wake: The GP Timer can also be set to wake the system from Deep Sleep using any of the three interrupt conditions. This file operates in almost the same manner as the `dm_interrupt` file, only it calls `omap_dm_timer_wake_read()` on read, and `omap_dm_timer_wake_set()`, which calls `omap_dm_timer_set_wake_enable()`, on write. It also uses the `omap_dm_timer_interrupt()` to clear the interrupt.

3.1.1 Additional Operations

These were originally debugging operations which have been kept available, in case anyone may wish to use them. You simply need to define 'DM_TIMER_FULL_SYSFS' to unlock these files. Most of them are operations which can be done just as easily in other ways, and in general have very limited functionality.

dm_active: This is tied to the actual running of the timer. Reading the file calls `omap_dm_timer_active_read()`, which returns the value of the ST bit of the `GPTMR_TCLR` register: 1 means the timer is running, 0 means it is not. Writing to this file calls `omap_dm_timer_active_toggle()`: writing 'start' or 'stop' will carry out their respective operations.

dm_autoreload: The timer can be set to reload on overflow and continue counting. Reading this file calls `omap_dm_timer_autoreload_read()`, which returns the AR bit of the `GPTRM_TCLR` register: 1 for on, 0 for off. Writing to this file calls `omap_dm_timer_autoreload_enable()`, which in turn calls `omap_dm_timer_enable_autoreload()` when 'enable' is written.

dm_compare: The timers can be set to observe when the counter reaches a specified point — this is mostly useful for interrupts. Reading this file calls `omap_dm_timer_compare_read()`, which returns the CE bit of the GPTMR_TCLR register: 1 for on, 0 for off. Writing to this file calls `omap_dm_timer_compare_enable()`, which in turn calls `omap_dm_timer_enable_compare()` when ‘enable’ is written.

dm_counter: Reading this file calls `omap_dm_timer_counter_read()`, which in turn calls `omap_dm_timer_read_counter()`, returning the current value of the timer’s counter. Writing to this file calls `omap_dm_timer_counter_set()`, which in turn calls `omap_dm_timer_set_counter()`, allowing you to set a new counter value.

dm_load: Reading this file calls `omap_dm_timer_load_read()`, which then calls `omap_dm_timer_read_load()`, returning the timer’s reload value. Writing to this file calls `omap_dm_timer_load_set()`, which in turn calls `omap_dm_timer_set_load()`, allowing you to set a new reload value.

dm_match: Reading this file calls `omap_dm_timer_match_read()`, which in turn calls `omap_dm_timer_read_match()`, returning the timer’s counter match value. Writing to this file calls `omap_dm_timer_match_set()`, which in turn calls `omap_dm_timer_set_match()`, allowing you to set a new match value.

3.2 Pulse Width Modulation sysfs Operations

GP Timers 1, 2 and 3 can be programmed for pulse width modulation, which is exportable to three different multiplexed pins. As such, `dm_timer[0–2]/` have a few extra files available for this purpose.

dm_pwm_mode: This is tied to the pulse/toggle option: in other words, does an event trigger a short pulse, or does it invert the polarity of the line. Reading this file calls `omap_dm_timer_pwm_mode_read()`, which returns the PT bit of the GPTMR_TCLR register: 0 for pulse, 1 for toggle. Writing to this file calls `omap_dm_timer_pwm_mode_toggle()`, which in turn calls `omap_dm_timer_enable_pulse()` when ‘pulse’ is written, and `omap_dm_timer_enable_toggle()` when ‘toggle’ is written.

dm_pwm_mux: Even if you configure the timer for PWM, it is not much use unless you export this to an external pin. Reading this file calls `omap_dm_timer_pwm_mux_read()`, which returns 1 if the timer is active on a pin; 0 if it is not. Writing to this file calls `omap_dm_timer_pwm_mux_enable()`, which in turn calls `omap_dm_timer_mux_pwm()` when ‘enable’ is written.

dm_pwm_output: There are two events which can act as triggers for PWM: timer overflow and, optionally, counter match. Reading this file calls `omap_dm_timer_pwm_output_read()`, which returns the TRG field of the GPTMR_TCLR register: 0 for no trigger (ie PWM off), 1 for overflow only, and 2 for both overflow and match. Writing to this file calls `omap_dm_timer_pwm_output_toggle()`, which in turn calls `omap_dm_timer_trigger_overflow()` when 'overflow' is written, and `omap_dm_timer_trigger_ofandmatch()` when 'match' is written.

3.3 Event Capture sysfs Operations

GP Timers 4 and 5 are also tied to multiplexed pins, but in this case for input to trigger an event capture. As such, `dm_timer[3-4]/` have a couple of extra files to take advantage of this feature. I will reiterate that the silicon errata notes that event capture does not work perfectly, so please use this feature with a degree of caution.

dm_event_edge: Event capture can be triggered on rising, falling or both edges of the input frequency. Reading this file calls `omap_dm_timer_event_edge_read()`, which returns the TCM field of the GPTMR_TCLR register: 0 for no capture (ie EVENT off), 1 for rising edge, 2 for falling edge, and 3 for both edges. Writing to their file calls `omap_dm_timer_event_edge_toggle()`, which in turn calls `omap_dm_timer_capture_rising()` when 'rising' is written, `omap_dm_timer_capture_falling()` when 'falling' is written, and `omap_dm_timer_capture_both()` when 'both' is written.

dm_event_mux: As with PWM, EVENT is tied to a multiplexed pin, and needs to be configured before it can work. Reading this file calls `omap_dm_timer_event_mux_read()`, which returns 1 if the timer is active on a pin; 0 if it is not. Writing to this file calls `omap_dm_timer_event_mux_enable()`, which in turn calls `omap_dm_timer_mux_event()` when 'enable' is written.

3.4 Complex sysfs Operations

A few more interesting operations have also been made available on the sysfs, which make use of some of the better features made available by the GP Timers. Most of these are only available to specific timers, however, so I have noted which timers in these cases.

dm_config_capture: Only available in `dmtimer[3-4]/`, this file will perform a series of event captures, which can later be averaged (in user-space) and used to identify an unknown frequency. Reading this file calls `omap_dm_timer_config_capture_read()`, which returns two values: the first is the sum of the counts taken between captures; the second is the number of these counts recorded. Writing to this file calls `omap_dm_timer_config_capture_set()`: ‘period X’ will set it to return the sum of the difference between X captures, while writing ‘begin’ will call `omap_dm_timer_configure_event_capture()`, starting the capture sequence. This uses the `omap_dm_timer_capture_interrupt()` handler, which attempts to handle the silicon errata for event capture (DM_TIMER_1) as best possible.

dm_config_chain: Available on all timers, this file allows you to initiate an event capture a specified number of microseconds after a frequency output has completed. Three different timers must be configured for a chain to work, and if `dm_config_frequency` has been used, it *must* have a maximum number of pulses to output selected. Reading this file calls `omap_dm_timer_config_chain_read()`, which returns 0 if the timer is not selected for chaining, or three values if it is: the first is the timer selected for frequency output, the second is the delay between chained items, and the third is the timer selected for event capture. Writing to this file calls `omap_dm_timer_config_chain_set()`: writing ‘select’ locks the current timer as the one being used for the delay step, and writing ‘delay X’ sets this delay to X microseconds. Writing ‘sweep Y’ or ‘pulse Y’ selects timer Y for frequency output — ‘sweep’ for frequency sweep, and ‘pulse’ for fixed-rate pulses. Likewise, writing ‘event Z’ selects timer Z for event capture. Finally, writing ‘begin’ initiates the frequency output, which, followed by the delay, completed with the event capture. For this mode, the clock selected for the delay step *must* be configured to run on the ARMXOR peripheral clock, as the 32KHz clock lacks the resolution for microseconds.

dm_config_frequency: Available in `dmtimer[0-2]/`, this file allows you to output a frequency of a specified number of hertz: the maximum frequency is limited only by the frequency of the clock selected. Reading this file calls `omap_dm_timer_config_frequency_read()`, which returns the frequency the timer is currently outputting (in Hz). Writing to this file calls `omap_dm_timer_config_frequency_set()`: writing ‘freq X’ sets the timer to output a wave at X Hz, and optionally writing ‘pulses Y’ will set the timer to stop after outputting Y pulses. Writing ‘begin’ calls `omap_dm_timer_configure_frequency()`, which starts the pulse width modulation. This may use the `omap_dm_timer_frequency_interrupt()` handler, which handles switching

the timer off after a specified number of pulses. It should be noted that the number of pulses generated may exceed the number specified at higher frequencies (18KHz or more).

dm_config_frequency_sweep: Also only available in dmtimer[0-2]/, this file allows a programmable frequency sweep to be initiated from a specified starting point to a specified ending point, with a hard limit far exceeding its practical limit. Reading this file calls omap_dm_timer_config_frequency_sweep_read(), which returns '1' if a frequency sweep has been completed; 0 if one has not started or is in progress. Writing to this file calls omap_dm_timer_config_frequency_sweep_set(): 'start X' will set the starting frequency to X Hz, 'end Y' will set the ending frequency to Y Hz, and 'period Z' will set the scan to advance by 1/Z of the difference between the start and end frequencies on each pulse. Writing 'begin' will call omap_dm_timer_configure_frequency_sweep(), starting the frequency sweep. This uses the omap_dm_timer_sweep_interrupt() handler, which handles the changes in frequency. In the final version, period will specify the length of time to conduct the frequency sweep over, rather than the divisor.

dm_config_wake: Available on all timers, this file allows you to set the timer to wake the system after a specified number of seconds: the delay can be set to a little longer than a day and a half. Reading this file calls omap_dm_timer_config_wake_read(), which returns the number of seconds it is currently set to wake the system after. Writing to this file calls omap_dm_timer_config_wake_set(), which in turn calls omap_dm_timer_configure_wakeup(). By writing a delay, in seconds, this will configure the timer to do exactly that, before starting the counter.

4 Example Usage

I shall provide a few simple examples now of how to use the dmtimer/ files. Some of these would require the additional files to be activated to use as-is, but those lines could be replaced with direct register writes.

4.1 Interrupt on Timer Match

Not very interesting, but this is perhaps the easiest way to show how the GP Timers work, and how the sysfs interacts with them. Calling 'cat <file>' after entering each of these would show you what is happening. Polling dm_interrupt after starting the timer would also show you when the interrupt has occurred.

```
# cd /sys/devices/system/dmtimer/dmtimer0/  
# echo -n "claim" > dm_use  
# echo -n "0x333333" > dm_match  
# echo -n "enable" > dm_compare  
# echo -n "1" > dm_interrupt  
# echo -n "start" > dm_active
```

4.2 Event Capture from Pulse Width Modulation

This would require you to wire the PWM0 pin to the EVENT3 pin, and it does not do anything with the values it captures, but is a good example of how these two operations both work.

```
# cd /sys/devices/system/dmtimer/dmtimer0/  
# echo -n "claim" > dm_use  
# echo -n "0xFFFF6666" > dm_counter  
# echo -n "0xFFFF6666" > dm_load  
# echo -n "enable" > dm_autoreload  
# echo -n "enable" > dm_pwm_mux  
# echo -n "toggle" > dm_pwm_mode  
# echo -n "overflow" > dm_pwm_output  
# echo -n "start" > dm_active  
# cd ../dmtimer3/  
# echo -n "claim" > dm_use  
# echo -n "enable" > dm_event_mux  
# echo -n "rising" > dm_event_edge  
# echo -n "start" > dm_active
```

4.3 Automatic Wake-Up Configuration

This would configure the system to wake after a minute, then put it into deep sleep — perhaps the most useful of these examples, as it could be a very real situation in many systems.

```
# cd /sys/devices/system/dmtimer/dmtimer0/  
# echo -n "claim" > dm_use  
# echo -n "60" > dm_config_wake  
# echo -n "standby" > /sys/power/state
```

4.4 Pulse Width Modulation

This would see the timer output a 32KHz wave on the PWM0 pin.

```
# cd /sys/devices/system/dmtimer/dmtimer0/  
# echo -n "claim" > dm_use  
# echo -n "freq 32000" > dm_config_frequency  
# echo -n "begin" > dm_config_frequency
```

4.5 Frequency Sweep

This would initiate a frequency sweep of 32Hz to 32KHz, with a 0.5% advancement — thus seeing it spend less time in the lower frequencies than it does the higher ones.

```
# cd /sys/devices/system/dmtimer/dmtimer0/  
# echo -n "claim" > dm_use  
# echo -n "start 32" > dm_config_frequency_sweep  
# echo -n "end 32000" > dm_config_frequency_sweep  
# echo -n "period 200" > dm_config_frequency_sweep  
# echo -n "begin" > dm_config_frequency_sweep
```

4.6 Event Capture

This would require an external frequency source to scan, which could be another timer as in the earlier example.

```
# cd /sys/devices/system/dmtimer/dmtimer3/  
# echo -n "claim" > dm_use  
# echo -n "period 100" > dm_config_capture  
# echo -n "begin" > dm_config_capture  
# cat dm_config_capture
```

4.7 Frequency Sweep Chained to Event Capture

This also requires an external frequency source to scan. It will run a frequency sweep from 32Hz to 32KHz, wait 100 μ s, then initiate an event capture.

```
# cd /sys/devices/system/dmtimer/dmtimer0/
# echo -n "claim" > dm_use
# echo -n "start 32" > dm_config_frequency_sweep
# echo -n "end 32000" > dm_config_frequency_sweep
# echo -n "period 200" > dm_config_frequency_sweep
# cd ../dmtimer3/
# echo -n "claim" > dm_use
# echo -n "period 100" > dm_config_capture
# cd ../dmtimer7/
# echo -n "claim" > dm_use
# echo -n "select" > dm_config_chain
# echo -n "sweep 1" > dm_config_chain
# echo -n "delay 100" > dm_config_chain
# echo -n "event 4" > dm_config_chain
# echo -n "begin" > dm_config_chain
# sleep 3
# cat ../dmtimer3/dm_config_capture
```