Pawel Gburzynski

CMA3000 accelerometer

driver

April 18, 2012

Introduction

CMA3000 is an accelerometer sensor manufactured by VTI Technologies. Documentation available from the manufacturer (data sheet + specification) covers all the technical intricacies. That documentation is not required to understand how to use the driver and what you can expect from the sensor.

Summary of the sensor's functionality

The sensor has three configurable modes of operation:

1. motion detection
2. free fall detection
3. perpetual measurement

The first two modes are inherently event-oriented, meaning that the sensor uses the acceleration data internally to decide when to trigger an interrupt signaling the respective event. The last mode is intended for high-speed readouts of raw acceleration data. The maximum accuracy/resolution of the acceleration data is only available in measurement mode.

The sensor measures acceleration along three axes (X, Y, Z). The readings are 8-bit signed integers. The resolution is determined by the range (in g) which is approximately evenly spread over the discrete range of values represented by a signed byte. There are two ranges, roughly:

8g (resolution = 67mg), note that this is (-8g, +8g)
2g (resolution = 17mg)

Motion detection mode

In the motion detection mode, a motion event is triggered when the delta of acceleration measurements, sampled at 10Hz, exceeds a configurable threshold for a configurable amount of time. Only 8g range (67mg resolution) is available in this mode.

The acceleration data is passed through a band pass filter whose purpose is to ignore high and low frequency changes (like some external vibrations). The document says that the –3dB cutoffs of the filter are at 1.3Hz and 3.8Hz. The exact meaning of the threshold and time parameters is not immediately clear. The threshold is probably applied against the absolute delta after the signal has passed through the filter. The time should be probably interpreted as the amount of time (in $1/10^{th}$ of a second) for which the filtered signal should exceed the threshold to trigger the event (this is what the document basically says).

From my very rough experiments, the most sensitive setting of time is 3 (this is also the default). If you wonder why not 0 or 1, then my explanation would be that the time must fall in the range where the filter passes a significant portion of the signal, and the center of the band is at 2.55Hz. This would also mean that using values larger than 5 is pointless, except that they work (!) although the sensitivity is significantly lower for time > 3. So there is something there I don't quite understand.

The best (and useless) sensitivity is at threshold = 0, time = 3. Useless, because with this setting the sensor generates spurious events when completely stationary. You can manipulate either parameter to reduce the sensitivity. For example, the spurious events cease when time becomes 0 (on the low end) or 8 (on the high). The maximum settable

value of time is 15. For threshold it goes formally up to 255, but I couldn't get any triggers with 20, so perhaps 15 is a good practical limit for both (the driver cuts it at 31).

The most reliable kind of extra information returned along with a motion event is the axis, i.e., X, Y, or Z on which the threshold has been reached/exceeded. At one point in the document they say that the acceleration registers read in the motion detection mode show the filtered acceleration data, i.e., the current deltas on the three axes (according to my understanding). However, some other part (under the "known issues" header) says that *"Interrupt (INT-pin) based acceleration data reading can be used only in measurement mode"*. The driver does return this data also in the motion detection mode (as described below) and you can decide for yourself how much sense it makes.

Motion detection mode is the most frugal mode in terms of energy usage (the current drawn by the sensor is below 11uA).

Free fall detection mode

A free fall event is triggered when all three acceleration values (sampled at 400Hz) remain below a configurable threshold (are close to zero) for a configurable amount of time. Formally, the device offers two options for the sampling frequency: 100Hz and 400Hz, as well as the two range/resolution options (8g versus 2g). I have noticed no significant power advantages of the 100Hz option – the sensor draws about 70uA in both cases), so the driver only implements the 400Hz option at 2g range (high resolution).

The threshold and time parameters are similar to the two parameters of the motion detection mode (although their interpretation may be simpler, because there is no mention of any filters in the documentation). I haven't tested the fall detection mode at all, because I couldn't trigger any free-fall event within the confines of my simplistic lab.

About the only useful information returned by the event is the event itself. The document says that you can read the acceleration data after the event (and it is the actual acceleration data); then again, the same qualifier (known issues) applies. The driver does give you access to the acceleration data as read at the time of the event, and you can do with it whatever you please.

Measurement mode

In this mode, raw acceleration data is available sampled at 10Hz, 100Hz, 400Hz, within the range of 2g or 8g (the former not available at 10Hz). The driver only implements 400Hz and 2g range. No events are triggered.

The driver doesn't implement the measurement mode directly, but reading the sensor without accepting an event amounts to switching it to measurement mode. This happens in a transparent way, as explained below.

Driver interface

The driver uses the new (as of April 2012) sensor API function **`wait_sensor`** to await events, in addition to reading sensor values with (the old function) **`read_sensor`**.

Switching the sensor on and off

Before it can deliver events and values, the sensor must be explicitly switched on with this function:

```
void cma3000_on (byte mode, byte threshold, byte time);
```


where **mode** is 0 for motion detection and 1 for free-fall detection, and the remaining two values are the two parameters of the respective mode, as described above. Note that **time** is forcibly bounded to 15 (any larger value will translate into 15) and **threshold** is forcibly bounded to 31.

This function:

```
void cma3000_off ();
```

powers the sensor down.

The operation of switching the sensor on is relatively fast (normally taking about 300us); however, I have noticed that writing to the sensor's registers is not 100% reliable, so the operations are retried until the read-back value matches the one that has been written. This happens on both specimens of CHRONOS that I have. On initialization, I sometimes have to write the reset/mode registers up to 5 or 6 times before the sensor catches on. In any case, the maximum delay incurred by the function is of order millisecond, so there's little penalty for switching the sensor on and off.

Calling **cma3000_on** always resets the sensor, so you don't have to insert **cma3000_off** between two calls to **cma3000_on**. In other words, if the sensor happens to be on, **cma3000_on** will act as if the sensor has been powered off first.

Receiving events

Note that **wait_sensor** is a void-type function invoked like this:

```
wait_sensor (CMA3000_SENSOR, TARGET_STATE);
```

Normally, the function immediately blocks (forces release), so it should be the last statement in the current state.[1] The issuing FSM is suspended and will be resumed in **TARGET_STATE** when the event (motion, free fall) occurs. Needless to say, the FSM can issue other wait requests before calling **wait_sensor**.

Note that an event causing **wait_sensor** to unblock does not go away until you execute **read_sensor** to retrieve the event's data. This means that a subsequent **wait_sensor**, executed before **read_sensor**, will fire immediately. An alternative way to erase the pending event is to reset the sensor by executing **cma3000_off** followed by **cma3000_on**.

If the state argument is **WNONE**, the function performs a special action consisting in reverting the sensor to the event mode (as described below) and immediately returns. Note that this is a subset of the function's standard action.

Reading values

The generalized “value” returned by the sensor (**read_sensor**) is a sequence of 4 single-byte signed integer numbers, i.e., a 4-element **char** array, e.g.,

```
char acc_data [4];
...
read_sensor (RETRY_STATE, CMA3000_SENSOR, (address)acc_data);
```

[1] It has to do so because of race conditions with interrupts.

The values returned in the array depend a bit on the context. If there is a pending event (motion, free fall), then the first (pilot) item in the array will be nonzero. The remaining three items will be the readings of the acceleration registers (X, Y, Z) obtained in the original mode immediately after receiving the event interrupt.

In the motion detection mode, the pilot value will be 1, 2, or 3, depending on whether the trigger occurred on the X, Y, or Z axis. In the free-fall detection mode, the value can only be 4 (indicating a free-fall event).

Once the sensor has been switched on (**cma3000_on**), it begins monitoring the respective events. After every event occurrence (signaled by the sensor via an interrupt), the driver saves the four values (with the pilot value guaranteed to be nonzero) to be returned to the praxis via the nearest invocation of **read_sensor**. If another event is received before **read_sensor** gets called, the new event overwrites the old one.

The idea is that following the internal reception of an event, the driver remembers that fact until the nearest **read_sensor** (or until the sensor is reset) and will return the event data to the first invocation of **read_sensor**. More accurately, it will return the data pertaining to the last occurrence of the event preceding the **read_sensor** call. Then the event data will be erased. If there are no intervening events, subsequent calls to **read_sensor** will be returning raw acceleration data updated at 400Hz and ranged to 2g. The first (pilot) value will then be zero, so the program can easily tell which is the case.

In this context, the role of **wait_event** is to immediately notify the praxis that an event is pending (so the praxis doesn't have to poll). The function does it in a manner preventing event loss. This means that if an event is pending already, or becomes pending while the function is putting the FSM to sleep, the FSM will be immediately rescheduled in the indicated state.

If you invoke **read_sensor** with no event data outstanding (e.g., there was no event to begin with it, or it has been cleared by a previous call to **read_sensor** and no new event has materialized in the meantime), the driver will switch the sensor to measurement mode. The sensor will remain in that mode until the nearest call to **wait_sensor** (which will resume the previous event mode, i.e., motion or free-fall detection) or until the sensor is reset (re-powered). The first call to **read_sensor** in a possible series of measurement calls does the extra job of switching the sensor to the measurement mode, so it will take more time than any of the subsequent measurement calls. The extra delay is about 120msec and does involve the scheduler (the state specified as the first argument to **read_sensor** is actually used). Subsequent measurement calls will return immediately in no more than 200usec each.

The sensor is always reverted to the respective event mode by a call to **wait_sensor.** This operation is much faster than in the opposite direction (because the waiting time for the transition is determined by the current sampling frequency of the sensor). Note that if the praxis relies solely on polling (it never uses **wait_sensor** to intercept events, but wants to perceive them via polling), it still has to make sure to revert the sensor to event mode following a reception of measurement data (the pilot value equal zero), as otherwise, no further events will be signaled in the pilot value (the sensor will be stuck in measurement mode). This is accomplished by a special call to **wait_sensor** with the state argument equal **WNONE**. Gratuitous calls cost practically nothing, so there is no need to check beforehand if the action is in fact needed, i.e., the previous invocation of **wait_sensor** has returned measurement (as opposed to event) data.

Note that the state argument of **read_sensor** is relevant: the operation may incur a non-trivial delay when it forces the sensor into measurement mode. As a special case, when the state argument is **WNONE**, **read_sensor** makes sure never to force the sensor into measurement mode. Instead, if there is no event data pending, the function will set the pilot value to zero and return immediately (leaving the remaining three values intact). This is the recommended way to invoke **read_sensor** when the praxis only wants to learn about events (and it doesn't care about the measurement data at all).

Also note that you can ignore the event modes altogether and use the sensor exclusively in measurement mode (via polling). If you never issue a **wait_sensor** request, then all invocations of **read_sensor** (possibly with exception of the first one) will be "measurement" calls (the pilot value will always be zero). Needless to say, the state argument of those invocations cannot be **WNONE**.

Also note that event-related acceleration values obtained in motion-detection mode (whatever sense they make) refer to the range of 8g (the granularity is 67mg), while the readings obtained in measurement mode (when the pilot value is zero) always refer to the range of 2g (at 17mg granularity).

Finally, note that multiple FSMs may be waiting for the sensor at the same time (with pending **wait_sensor** requests). While this is formally OK (all those FSMs will be awakened by the same nearest sensor event), they may get into a mess when they subsequently begin executing **read_sensor** and **wait_sensor** in multiple interleaved copies. I cannot think of a scenario when this kind of operation would make sense.

Test praxis for CHRONOS

There is a test praxis for CHRONOS, especially tailored for CMA3000, which you can use to experiment with the sensor. It is called CMATEST and has been derived from RFTEST. It does involve the same AP (access point) setup as RFTEST to pass commands to (and receive feedback from) the CHRONOS.

Commands

a mode threshold time

Starts the report FSM. The three numbers directly translate into the arguments of **cma3000_on** invoked in the first step of the resulting action. A missing number defaults to zero. The FSM's behavior (described below) is controlled by three parameters settable with this command:

s tmout nrds rintv

where *tmout* is a timeout in seconds (60 max), *nrds* is the number of required readouts after an event, and *rintv* is the interval between two consecutive readouts in PicOS milliseconds. The default setting of the three parameters (before the first use of the command) is 0, 1, 0. A missing number in the command line defaults to zero as well. Note that *tmout* and *nrds* cannot be both zero.

q

Stops the report FSM and powers the sensor down. Note: you have to execute *q* between two *a*'s.

h n

Switches off the radio for *n* seconds (60 max) or until the nearest sensor event or report, whichever comes first. Used to measure the current drawn by the node. If the argument is absent, it defaults to zero meaning “no timeout”, i.e., just wait for a sensor event (or a periodic readout).

d m

Switches the LCD off (*m* = 0) or on (*m* != 0). Intended for current measurements.

p m

Switches between the powerdown (*m* = 0) or powerup (*m* != 0) modes of the CPU. The initial mode is powerdown.

The following commands are directly inherited from RFTEST:

r year month day dow hour minute second

Sets or reads the real-time clock. The *year* should be written modulo 100 (only the LS byte of the specified integer number is used); *dow* is the day of the week, e.g., 0 meaning Sunday, 1 for Monday, and so on.

If the number of arguments is less than 7 (e.g., zero), the command ignores them and instead displays the current reading of the real-time clock.

b time

Activates the buzzer for the specified number of milliseconds.

fr a

Reads FIM word number *a* (specified as unsigned decimal).

fw a b

Writes word value *b* at FIM location *a* (both unsigned decimals).

fe a

Erases FIM block containing location *a*.

The praxis also responds to push buttons reporting them to the access point.

The report FSM

Here is the action carried out by the report FSM. Its main loop commences with:

1. a **`delay`** request for *tmout* seconds, if *tmout* is nonzero
2. a **`wait_sensor`** request to the accelerometer, if *nrds* is nonzero

In other words, if *tmout* is nonzero, the FSM will be periodically polling the sensor at *tmout* intervals (in seconds). If *nrds* is nonzero, the FSM will be (additionally) expecting sensor events. At least one of the two parameters must be nonzero.

Whenever an event occurs (*nrds* is nonzero) the FSM will read the sensor that many times before returning to the main loop (to issue another call to **wait_sensor**). The first readout is made right away. Note that it will return the event data (the pilot value in the returned 4-tuple will be nonzero). If *nrds* is equal 1, there will be no more readings and the sensor will never switch to the measurement mode. Otherwise, the FSM doesn't return to the main loop, but *rintv* milliseconds after the first readout, calls **read_sensor** again, quite likely (if there was no event in the meantime) producing this time the raw acceleration data in measurement mode. As we said earlier, there is a 120 msec time penalty for that call. Any subsequent calls to **read_sensor** (the case *nrds* > 1), issued at *rintv* millisecond intervals, will return quickly (without dropping the measurement mode). At the end (after all *nrdv* readouts have been made), the FSM will return to the main loop where the call to **wait_sensor** will put the sensor back into event mode.

If *tmout* is nonzero and there has been no event for that many seconds since the last call to **wait_sensor**, the FSM will force a readout and then get back to the main loop. If *nrds* is zero (meaning that the FSM operates in polling-only mode), the sensor is reverted to the event mode via a special call to **wait_sensor**.

The readout format, as presented to the access point, is:

```
u: [p] <x,y,z>
```

where *u* is “E”, if the readout has been caused by an event, or “P”, if it is a forced readout after a timeout; *p* is the “pilot” value, i.e., the first of the four numbers returned by **read_sensor**, shown in hexadecimal, and *x*, *y*, *z*, are the three acceleration values shown as signed integers. Note that *p* equal zero indicates a readout performed in measurement mode.