Pawel Gburzynski

BMA250 accelerometer

driver

May 5, 2017

Introduction

BMA250 is an accelerometer sensor manufactured by Bosch Sensortec. 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, unless you want to use the raw interface variant of the driver (see below). Should a need arise, I will put here more information about the device's register content, so this document will become self-contained.

The sensor replaces CMA3000 in the new version of CHRONOS (dubbed CHRONOS_WHITE). As CMA3000 is probably discontinued, we should consider using BMA250 for our applications requiring acceleration sensing. The new sensor appears way more flexible and (also more power friendly) than its predecessor.

Summary of the sensor's functionality

The sensor has seven configurable modes of operation:

1. passive mode (readout of acceleration data)
2. motion detection (any motion)
3. tap detection (with single and dual tap options)
4. orientation detection
5. flat position detection
6. low g (aka fall) detection
7. high g (i.e., shock) detection

Additionally and independently of the above modes, it is possible to configure the sensor for low power operation in one of 12 levels (level 12 meaning full power). The modes can be combined; for example, it is possible to detect (any) movement and taps at the same time.

The driver can be compiled in one of two versions. When the constant BMA250_RAW_INTERFACE is defined, the praxis interface for setting up the operational parameters of the accelerometer is simplified (and thus frugal in terms of code size), but extremely powerful; basically, the praxis can directly set all the relevant registers of the device. That interface is recommended for production and for advanced (binary) OSS setups where the configurations of register values can be preset in advance or created by external programs.

Note that, for the ultimate simplicity of praxis interface, the sensor can be set up to operate in one of several *dedicated* modes configured by suitably connecting some of the chip's pins (which can also be done in software with the right connections). In that mode the sensor basically requires no driver: it only sends (signals) interrupts upon detection of the preconfigured events. In our boards, we may want to provide for configuring the sensor in dedicated modes (which may be useful in some applications).

For experimenting (with simple OSS), the second, verbose version of the driver interface is recommended. This interface is described first.

The verbose interface

The verbose interface encompasses 7 praxis-accessible functions for configuring the operation modes + 1 function for configuring the power level. Multiple event types can be enabled simultaneously; however, for tap detection, you can have either single tap mode or double tap mode, but not both at the same time.

As all typical accelerometers, the sensor measures acceleration along three axes (X, Y, Z). The readings are 10-bit signed integers. The resolution is determined by the settable range (in g) which is approximately evenly spread over the discrete range of values represented by a signed 10-bit value. There are four ranges: 2g, 4g, 8g, and 16g. The resolution at 2g is roughly 3.91mg.

The acceleration data is available continuously (via *read_sensor*), although it is updated at some intervals determined by the configurable bandwidth. Technically, one can select between raw data and filtered data (meaning passed through a bandwidth filter). With the verbose interface, he driver only uses filtered data (which coincides with the default setting of the sensor and makes better sense in all conceivable applications). The bandwidth is configurable in 8 discrete steps from 7.81Hz to 1000Hz. The update frequency of (filtered) acceleration data is twice the bandwidth, e.g., at 1000Hz, the data is updated at 0.5ms intervals. You shouldn't worry about this. It only means that if you issue two tightly spaced *read_sensor* operations, you may get the same data twice (if the time separation between the two operations is less than the update interval).

In any of the true low power modes, the sensor sleeps for some time, then wakes up, then goes to sleep again. The low power setting determines the duration of the sleep interval. The operation after wake-up basically looks like normal operation in full power mode. Depending on the mode, the wake-up interval accommodates the necessary number of value samples to properly diagnose the presence or absence of the configured event(s).

The sensor provides for calibration and offsets that can be stored by the user in EEPROM. No factory calibration is preset, but the sensor offers functions for assisting/automating user calibration. The present driver doesn't touch that part. We shall see if calibration is needed in our applications. If so, then it will be probably done by a special praxis using a special (extended) driver.

The sensor also offers temperature readouts (almost for free). According to the datasheet, the temperature is a single 2's complement byte with the center (value zero) at 24 degrees centigrade and the resolution of 0.5 degrees. This implies the range between –40 and +87.5 degrees. My sensor shows 17 (translating into 32.5 degrees) at the room temperature around 22 degrees, which suggests that it does need calibration, at least as far as temperature. For acceleration, a flat-laying watch (at 2g range setting) yields the readings <-3, -25, 266>, which doesn't look bad. With bandwidth-filtered data, there is no need to calibrate the sensor for motion or tap detection.

Turning the sensor on and off

To switch the sensor on, execute this function:

```
Boolean bma250_on (byte range, byte bandwidth,
      byte events);
```

where *range* is one of these values: BMA250_RANGE_2G (3), BMA250_RANGE_4G (5), BMA250_RANGE_8G (8), BMA250_RANGE_16G (12) and *bandwidth* is a value between 0 and 7, inclusively, with 0 selecting 7.81Hz (the lowest bandwidth) and every next value doubling the bandwidth up to 1000Hz.

The last argument is a collection of flags indicating which events should be detected by the sensor. It can be zero, in which case no events will be detected and triggered (*wait_sensor* will be pointless), but the program will still be able to use *read_sensor* to poll for the acceleration data.

The event bits are as follows:

```
#define BMA250_STAT_LOWG      0x01
#define BMA250_STAT_HIGHG     0x02
#define BMA250_STAT_MOVE      0x04
#define BMA250_STAT_TAP_D     0x10
#define BMA250_STAT_TAP       0x20
#define BMA250_STAT_TAP_S     BMA250_STAT_TAP
#define BMA250_STAT_ORIENT    0x40
#define BMA250_STAT_FLAT      0x80
```

and represent these events: LOW-G, HIGH-G, MOTION, DOUBLE TAP, SINGLE TAP, ORIENTATION CHANGE, FLAT. The constants are named as they are, because the same bits can be looked up in the status field returned as part of sensor data (see below) to check which events have been triggered.

For example,

```
bma250_on (BMA250_RANGE_2G, 7,
                  BMA250_STAT_TAP_D | BMA250_STAT_MOVE);
```

starts the sensor in 2g range at 1000Hz detecting motion and sensing double taps. Note that, except for the range and bandwidth, the sensor operates with the default values of all parameters, which may be fine. The default values of parameters can be modified by calling the functions presented in subsequent sections.

Normally the function returns **YES**. Its value can be ignored, unless the sensor has been configured as optional such that its absence is detected at system initialization. In such a case, the function will return **NO** when the sensor is absent (in which case the function does nothing).

To switch the sensor off, execute this function:

```
void bma250_off (byte mode);
```

with the argument 0. Any other value of the argument will merely affect the low power status of the sensor, but only if the sensor was on (otherwise, the operation will be void). Values 1–11 select true low-power options with the sleep intervals (in milliseconds): 1000, 500, 100, 50, 25, 10, 6, 4, 2, 1, 0.5 (lower values of mode correspond to lower power). Value 12 and higher puts the sensor into the (default) full power mode (no sleeping). According to the datasheet, the current drain of the sensor at the respective power setting (in µA) is: 0.7, 0.9, 2.3, 4.0, 7.4, 16.4, 25.2, 34.5, 55.0, 78.8, 100.5, 139.0 (the last value is the current drain in full power mode).

Note: in CHRONOS, the sensor is switched off electrically (meaning switching off the supply voltage), so it drains no current in the *off* state. It is also possible to put the sensor into the so-called suspend mode in which it remains dormant (sustaining the last settings) at the current drain of 0.5uA. The driver doesn't use this mode on CHRONOS, but it can be configured to do so instead of powering the sensor down.

After the sensor has been switched off (*bma250_off (0)*), it must be powered up (with *bma250_on*) to resume operation. Any required event mode(s) (see below) must be setup again, i.e., the sensor always starts from scratch (*bma250_on* always resets all parameters to default values). It is OK to call *bma250_on* after a previous call to *bma250_on* (no need to invoke *bma250_off* in between).

Extracting data from the sensor

For as long as the sensor is on, the program can use *read_sensor* to read the sensor data. The data is returned in this format:

```
typedef struct {
      word  stat;
      sint  x, y, z;
      char  temp;
} bma250_data_t;
```

where the three signed integers are the most recent *x*, *y*, and *z* acceleration values, *stat* is the sensor's status (a collection of bits), and *temp* is the temperature readout. Here is the recommended way to call *read_sensor*:

```
      bma250_data_t val;
      …
      state READ_BMA250:
            read_sensor (READ_BMA250, SENSOR_MOTION,
                  (address)(&val));
            …
```

assuming that SENSOR_MOTION is the sensor number.

Bits in the lower byte of the *status* word mark the occurrence (or rather the pending status) of the respective events (the bits are the same as for the *events* argument of *bma250_on*). Only those events that have been enabled by *bma250_on* can ever show up in *status*.

The upper byte of *status* contains some other flags whose meaning is probably less important. Here are the constants providing symbolic names for those bits:

```
#define BMA250_STAT_HX          0x0100
#define BMA250_STAT_HY          0x0200
#define BMA250_STAT_HZ          0x0400
#define BMA250_STAT_HS          0x0800
#define BMA250_STAT_O0          0x1000
#define BMA250_STAT_O1          0x2000
#define BMA250_STAT_ZU          0x4000
#define BMA250_STAT_ISFLAT      0x8000
```

Note that the above constants refer to the entire status word (not just to the upper byte). The first three of them are to indicate which axis has triggered the last high-g event (was first to see the shock). Note that the bits can only be set if the event is enabled. The fourth flag tells the sign of the slope along the triggering axis (0 – positive, 1 – negative). Bits 0x3000 should be taken together as an indication of the current orientation of the device (assuming the orientation event is enabled). Their meaning is: 00 – portrait upright, 01 – portrait upside-down, 10 – landscape left, 11 – landscape right. The next bit (0x4000) tells the orientation of the z-axis (0 – up, 1 – down). Finally, the most significant bit tells whether the device is lying flat (1) or otherwise (0).

The sensor has numerous parameters that can be changed by writing values to some registers. Instead of simply making the "write register" operation available to the praxis (which would make controlling the sensor rather cryptic), I have decided to provide six functions for changing those parameters in a more systematic manner, by relating them to specific event types. I am not sure if that makes much better sense (and is in fact less cryptic).

Motion detection

The motion detection parameters are configured by calling this function:

```
bma250_move (byte nsamples, byte threshold);
```

According to the datasheet, motion is detected by examining the difference between successive acceleration values. They call that measure “slope”. To trigger an event, the slope (i.e., the absolute difference between two consecutive acceleration values, along any axis) must be above *threshold* (the second argument of *bma250_move*) for at least *nsamples* consecutive cases. The default values of *threshold* and *nsamples* are 20 and 1, respectively.

Note that the time spacing between two consecutive acceleration values depends on the bandwidth. When discussing the dedicated modes, the datasheet lists some of their (implicitly assumed) parameters. For motion detection, the range is set to 2g and the bandwidth to 125Hz (setting 4). This, in combination with the default values of *threshold* and *nsamples*, should be viewed as the factory-recommended parameters for motion detection.

The sensor can restrict motion detection to any two or just one axis. The present driver doesn't offer these options.

Tap detection

The parameters pertaining to this mode can be modified by calling this function:

```
void bma250_tap (byte md, byte th, byte del, byte ns);
```

A (single) tap event occurs when there is a momentary high slope followed by a period of (relative) silence. The slope threshold for the event is determined by the *th* argument of *bma250_tap* which can take values between 0 and 63 interpreted in units of 16 x resolution. Thus, for example, with the 2g range (and the resolution of 3.9mg+) the unit of threshold is about 62.5mg.

More specifically, for a single tap event, there must be a slope-above-threshold occurrence which may last for some time, but it must die out relatively quickly, then for some time there must be a period of relative silence. The first period is called *shock*: any high-slope occurrence within that period (following the first high-slope signal) is ignored. The second period is called *quiet*: any slope-above-threshold occurrence within that period will cancel the first event. The duration of each of the two periods is determined by *md* which can be:

`0`	shock = 50ms, quiet = 30ms
`BMA250_TAP_SHOCK`	shock = 75ms, quiet = 30ms
`BMA250_TAP_QUIET`	shock = 50ms, quiet = 20ms
`BMA250_TAPSHOCK | BMA250_TAP_QUIET`	shock = 75ms, quiet = 20ms

The third argument, *del*, is only meaningful for double tap detection. It selects the duration of the window for the second shock event (of the second tap) measured after completion of the previous event, i.e., from the end of the first *quiet* period. The settings are: 0 – 50ms, 1 – 100ms, 2 – 150ms, 3 – 200ms, 4 – 250ms, 5 – 375ms, 6 – 500ms, 7 – 700ms.

The last argument, *ns*, is only interpreted in the low-power mode when the chip wakes up from sleep and has to look at some acceleration samples to check if an interesting event is commencing. The argument determines the number of samples on which the decision will be based. Its value can be 0 – 2 samples, 1 – 4 samples, 2 – 8 samples, and 3 – 16 samples.

The defaults are *md* = 0, *th* = 10, *del* = 4, *ns* = 0.

The bandwidth used by the chip in the dedicated tap sensing mode is 1000Hz.

Orientation change detection

This function affects the parameters of the orientation change event:

```
void bma250_orient (byte bl, byte md, byte tht, byte hyst);
```

The purpose of this event is to tell whether the orientation of the chip has changed among one of these states: *portrait upright*, *portrait upside-down*, *landscape left*, *landscape right*. If the event is enabled, then the device's orientation can be seen in the *status* word of the sensor's data.

The datasheet includes an elaborate explanation of the algorithm for determining the device's orientation and triggering the orientation change event. The feature may turn out to be interesting and useful (so it is enabled in the driver). Roughly, the parameters have this meaning:

bl selects one of the four blocking modes preventing spurious orientation changes; its value can be one of:

BMA250_ORIENT_BLKOFF
BMA250_ORIENT_BLKTHT
BMA250_ORIENT_BLKSLP
BMA250_ORIENT_BLKANY

md selects one of the modes of the algorithm for switching between the different orientations; its value can be one of:

BMA250_ORIENT_MODSYM
BMA250_ORIENT_MODHAS
BMA250_ORIENT_MODLAS
BMA250_ORIENT_MODSYB

tht defines the blocking angle and is only meaningful when *bl* selects angle blocking, i.e., equals BMA250_ORIENT_BLKTHT; its range is 0 – 63

hyst is the hysteresis in units of 62.5mg, regardless of the range selection; its range is 0 – 7

The default settings are:

bl	=	BMA250_ORIENT_BLKTH
md	=	BMA250_ORIENT_MODSYM
tht	=	8
hyst	=	1

In the dedicated orientation detection mode, the bandwidth is 62.5Hz.

Flat position detection

This function sets the parameters of the flat position detection algorithm:

```
void bma250_flat (byte tht, byte hold);
```

The position status is a single bit, i.e., the device is either flat or not. That bit can be read from the *status* word returned with the senor's data.

An event is triggered whenever the value of the “flat” bit changes. The first argument of the above function describes the threshold angle. Its value can be between 0 and 63 (translating into 0 to 48 degrees). The second argument provides the holding time. The device should remain in the new state for at least that long for the change to be signaled. The settings are: 0 – 0, 1 – 512ms, 2 – 1024ms, 3 – 2048ms. The default values are 8 and 1, respectively.

There is no dedicated flat position detection mode.

Low-g (fall) detection

This function describes the parameters of low-g detection:

```
void bma250_lowg (byte md, byte th, byte dur, byte hyst);
```

The first argument can be either BMA250_LOWG_MODSGL, selecting the *single* mode, or BMA250_LOWG_MODSUM, selecting the *sum* mode. In the single mode, the acceleration along each axis is compared to the threshold separately (acceleration on all three axes must be below the threshold), while in the sum mode, the sum of absolute values of acceleration for all three axes is compared against the threshold. The threshold is specified in the second argument, as a full byte, in units of 7.81mg (regardless of the range selection).

The event is triggered, if the threshold criterion holds for at least the amount of time determined by *dur*. The actual time is (*dur* + 1) x 2ms. The event is reset if the threshold condition ceases to hold when the threshold is incremented by the hysteresis. The latter is determined by *hyst* which can be between 0 and 3. The value is multiplied by 32 x the resolution (depending on the range) to yield the offset added to the threshold.

The default values are *md* = BMA250_LOWG_MODSGL, *th* = 48 (375mg), *dur* = 9 (20ms), *hyst* = 1 (125mg).

There is no dedicated low-g detection mode.

High-g (shock) detection

This function describes the parameters of high-g detection:

```
void bma250_highg (byte th, byte dur, byte hyst);
```

The idea is similar to low-g detection except that there is just one (single) mode. The event is generated if the acceleration along any axis exceeds the threshold. The values of threshold, duration, and hysteresis are calculated as for low-g.

There is no dedicated high-g detection mode.

The simplified interface

This option is selected at compilation if the constant BMA250_RAW_INTERFACE is defined. The semantics of operations *read_sensor* and *wait_sensor* (including the format of sensor data) is the same as for the verbose interface. What changes is the set of operations for configuring the sensor. Only two of those operations are left and their semantics are modified. This function configures the sensor:

```
Boolean bma250_on (bma250_regs_t *regs);
```

where the argument points to this structure:

```
typedef struct {
      lword rmask;
      byte regs [20];
} bma250_regs_t;
```

which describes the set of registers to be loaded into the device.

Normally the function returns **YES**. Its value can be ignored, unless the sensor has been configured as optional such that its absence is detected at system initialization. In such a case, the function will return **NO** when the sensor is absent (in which case the function does nothing).

The byte array stores the values to be put into these registers (in exactly this order): 0x0F, 0x10, 0x11, 0x13, 0x16, 0x17, 0x1E, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x28, 0x2A, 0x2B, 0x2C, 0x2D, 0x2E, 0x2F (20 registers altogether), with the mask indicating which of the values are valid (one on bit number *i* from the right stands for the *i*-th location in *regs*). The registers not mentioned in *rmask* are skipped, i.e., not set. Refer to the sensor's datasheet for the meaning of the registers. The byte array need not extend farther (the structure can be shorter) than to the highest register number indicated in the mask.

Before writing the registers prescribed by the argument, the function initializes the sensor, i.e., powers it up, resets, and performs some standard settings expected by the driver (e.g., configures the interrupts). Thus, not all formally settable registers can appear in the argument to *bma250_on*. For example, registers 0x14 (reset), 0x19 (interrupt configuration), 0x21 (clear interrupts) are used internally by the driver in some standard fashion and are not considered configurable.

For example, the following set of operations of the verbose interface variant:

```
bma250_on (BMA250_RANGE_2G, 6, BMA250_STAT_MOVE);
bma250_move (3, 72);
```

can be replaced by this sequence:

```
bma250_regs_t regset = { 0x00003013,
                          { 0x03, 0x0E, 0x00, 0x00, 0x07,
                            0x00, 0x00, 0x00, 0x00, 0x00,
                            0x00, 0x00, 0x03, 0x48} };
…
bma250_on (&regset);
```

The registers that must be set to configure the sensor are: 0x0F (to 0x03) [range = 2G], 0x10 (to 0x0E) [bandwidth], 0x11 (to 0x00) [full power – this in fact corresponds to the default setting of the register after reset, and can thus be skipped], 0x16 (to 0x07) [enabling motion <aka slope> interrupts], 0x27 (to 0x03) [the number of samples], and 0x28 (to 0x48 = 72) [the threshold]. Although at first sight, judging by the above example, the word “verbose” may appear misplaced, note that any configuration of the sensor, including combinations of multiple event types, can be accomplished with a single call to *bma250_on*.

To switch the sensor off, use this function:

```
bma250_off ();
```

Note that, in contrast to the verbose variant, it takes no arguments. Its sole purpose is to switch the sensor off completely. For low-power operation, you can set register 0x11 with *bma250_on*.

The test praxis for CHRONOS

There is a test praxis for CHRONOS, which you can use to experiment with the sensor (as well as CMA3000). It is called SENSTEST and has been derived from CMATEST, i.e., the previous test praxis for CMA3000. It does involve the same AP (access point) setup as RFTEST to pass commands to (and receive feedback from) the CHRONOS. The praxis recognizes whether the sensor is BMA250 or CMA3000. Some commands differ depending on the case. For the BMA250 sensor, the praxis uses the verbose interface.

Commands

a mode threshold time **[CMA3000]**

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.

a range bandwidth events **[BMA250]**

Starts the report FSM. The three numbers stand for the range (0 – 2g, 1 – 4g, 2 – 8g, 3 – 16g), bandwidth selection (0 – 7), and the events mask (as for *bma250_on*). The last argument is specified in hexadecimal.

For both sensors, 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. Note that this is capital ***P***.

p

Reports the values of the pressure/temperature sensor, which can be SCP1000 or BP085, depending on the board.

pc **[BP085 only]**

Reports the 11 calibration values of BP085. Note that the sensor readings returned by the driver of BP085 are not calibrated.

These commands are only applicable to BMA250 and translate into calls to the mode functions (mm – *bma250_move*, mt – *bma250_tap*, and so on):

```
mm nsamples threshold
mt md th del ns
mo bl md tht hys
mf tht hold
ml md th dur hyst
mh th dur hyst
```

The value of *md* for *mt* is 0 – 3 with bit 0 mapped into BMA250_TAP_SHOCK, and bit 1 into BMA250_TAP_QUIET. The value of *bl* for *mo* is 0 – 3, translating into BMA250_ORIENT_BLKOFF, BMA250_ORIENT_BLKTHT, BMA250_ORIENT_BLKSLP, and BMA250_ORIENT_BLKANY. The value of *md* for *mo* is also 0 – 3 translating into the mode constants BMA250_ORIENT_MODSYM, BMA250_ORIENT_MODHAS, BMA250_ORIENT_MODLAS, BMA250_ORIENT_MODSYB. Any nonzero value of md for ml translates into BMA250_LOWG_MODSUM, while 0 is interpreted as BMA250_LOWG_MODSGL. Note that when calling the actual functions (rather than invoking commands of the test praxis) you should be using symbolic constants.

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*).

<u>This applies to CMA3000 only</u>

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. 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.

This applies to BMA250

For BMA250, there is no difference between the interpretation of the first call to *read_sensor* following an event and any other call, except that the first call clears the events that have been pending before the call (and is thus required after receiving an event).

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.

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

```
u: [s] t <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; *s* is the status word shown in hexadecimal, *t* is the raw temperature reading of the sensor, and *x*, *y*, *z*, are the three acceleration values shown as signed integers.