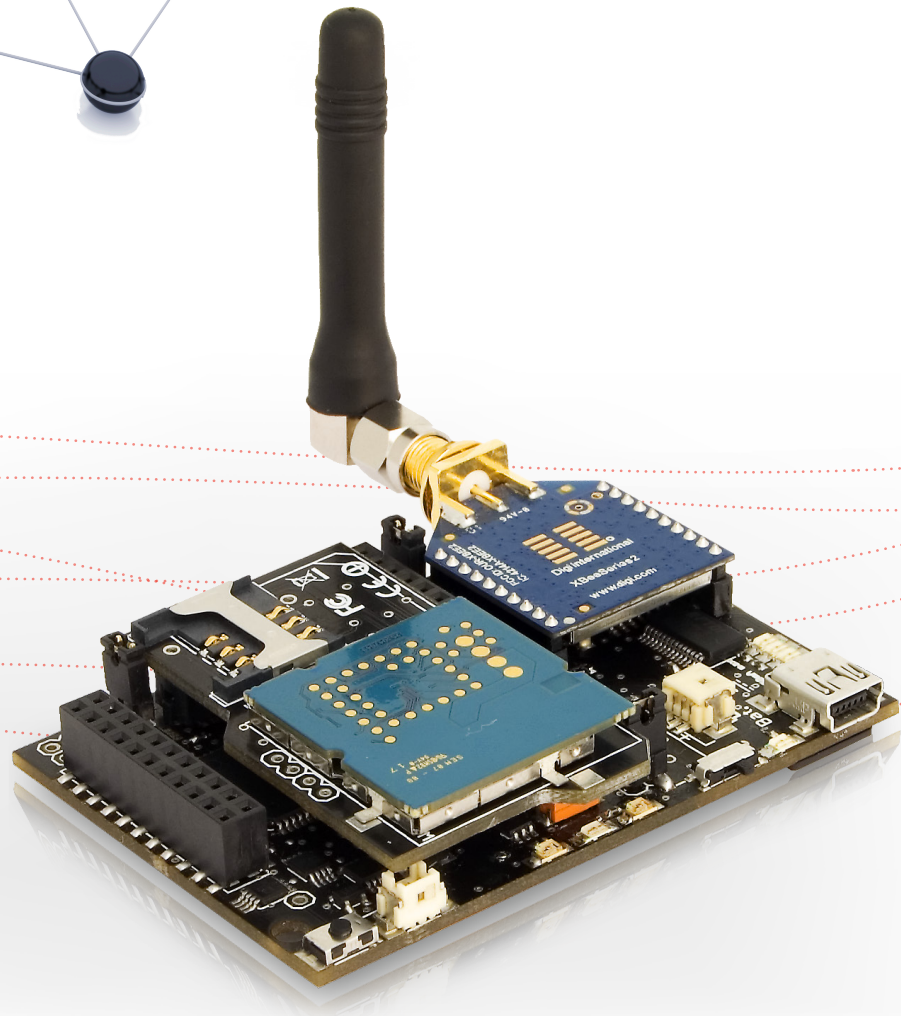
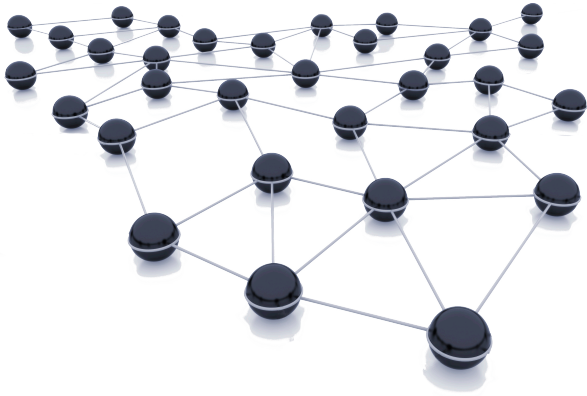


Wasp mote Accelerometer Programming Guide



INDEX

1. General Considerations	4
1.1. Wasmote Libraries	4
1.1.0. Wasmote ACC Files	4
1.1.1. Constructor	4
1.1.2. Configuration Registers and Constants	4
1.1.3. Flags.....	4
2. Initialization	4
2.1. Initializing the accelerometer.....	5
2.2. Rebooting the accelerometer	5
2.3. Closing the accelerometer	5
2.4. Getting status.....	5
2.5. Getting correct working.....	5
3. Getting acceleration on axis	6
3.1. Registers.....	6
3.2. Getting acceleration on axis X	6
3.3. Getting acceleration on axis Y.....	6
3.4. Getting acceleration on axis Z.....	7
4. Free Fall interruption	7
4.1. Registers.....	7
4.2. Setting the Free Fall interruption	7
4.3. Unsetting the Free Fall interruption.....	8
5. Direction Change interruption	7
5.1. Registers.....	8
5.2. Setting the Direction Change interruption	9
5.3. Unsetting the Direction Change interruption.....	9
6. Accelerometer Modes and Events	9
6.1. Accelerometer Modes	9
6.1.1. Setting Accelerometer Modes	9
6.1.2. Getting Accelerometer Modes.....	10
6.2. Accelerometer Events	10
6.2.1. Setting Accelerometer Events.....	10
6.2.2. Getting Accelerometer Events	10
7. Advanced Functions	11
7.1. Writing a Register.....	11
7.2. Reading a Register.....	11
7.3. Setting 'CTRL_REG1'	11
7.4. Getting 'CTRL_REG1'	11

7.5. Setting 'CTRL_REG2'	12
7.6. Getting 'CTRL_REG2'	12
7.7. Setting 'CTRL_REG3'	12
7.8. Getting 'CTRL_REG3'	12
7.9. Getting ADC Mode	13
7.10. Setting ADC Mode	13
7.11. Setting sampling rate	13
8. Code examples and extended information	14

1. General Considerations

1.1. Waspote Libraries

1.1.0. Waspote ACC Files

WaspACC.h ; WaspACC.cpp

1.1.1. Constructor

To start using Waspote ACC library, an object from class 'WaspACC' must be created. This object, called 'ACC', is created inside the Waspote ACC library and it is public to all libraries. It is used through the guide to show how the Waspote ACC library works.

When creating this constructor, no variables are initialized by default.

1.1.2. Configuration Registers and Constants

There are many registers and constants defined in Waspote ACC library. These registers and constants are used to define some configuration parameters used in Free-Fall or Wake-Up events and other tasks in the library. Through this guide, all these values will be explained.

The thresholds used to generate interruptions are values selected according to the tests performed. However, each developer should analyze its scenario and decide if the thresholds have to be modified.

1.1.3. Flags

There are various flags used to handle interruptions and errors while using the Waspote accelerometer.

1.1.3.1. Status flag

It is used to set reading or writing errors or hardware interruptions. Possible values are:

- ACC_ERROR_READING: error reading register.
- ACC_ERROR_WRITING: error writing register.
- ACC_COM_ERROR: error communicating with accelerometer.

1.1.3.2. interruption Flag

It is used to check which interruption got active, its structure is:

```
(MSB) 7 6 5 4 3 2 1 0 (LSB)
      XX XX XX XX XX TH DD FF
TH – threshold ; DD - direction change ; FF - free fall
```

1.1.3.3. Global Interruption Flag

It is used to check the port in which the interruption has got activated. It is used in this library, and it will be used in other libraries which generate interrupts too.

2. Initialization

Before getting information from the accelerometer, it has to be initialized, setting its registers with the appropriate values.

2.1. Initializing the accelerometer

It initializes communication with I2C bus and configures the accelerometer, open the communication and set up the registers. It makes no tests and checks nothing from the communication, therefore returns nothing and modifies no flags.

Example of use

```
{
  ACC.ON(); // Initializes the accelerometer
}
```

2.2. Rebooting the accelerometer

It reboots the accelerometer, taking for granted the accelerometer is on and forcing the sensor to reboot by writing to the control register #2 into the right pin.

It makes no tests and checks nothing from the communication, therefore returns nothing and modifies no flags.

Example of use

```
Example of use
{
  ACC.reboot(); // Executes the reboot process
}
```

2.3. Closing the accelerometer

It sets the accelerometer module to the Hibernate mode taking for granted the accelerometer is on and powering down the sensor by writing to the control register #1 into the right pin.

It closes I2C bus as well.

It makes no tests and checks nothing from the communication, therefore returns nothing and modifies no flags.

Example of use

```
{
  ACC.close(); // Closes I2C bus and sets accelerometer to hibernate mode
}
```

2.4. Getting status

It gets the accelerometer status.

It returns a byte containing the status of the accelerometer reading from the proper register. It won't activate any flags by itself, but activates ACC_COMM_ERROR in case there was an error writing to the register.

Example of use

```
{
  uint8_t status;
  status=ACC.getStatus(); // Gets the accelerometer status
}
```

2.5. Getting correct working

It gets if the accelerometer is present and is working properly.

It sends the accelerometer dummy register on 0x0F that should always answer 0x3A, otherwise an error occurred.

This function can be used for determining if the accelerometer is on the board and checking if it is still working properly.

Since it calls 'readRegister', it won't activate any flags by itself, but the other functions will activate ACC_COMM_ERROR in case there was an error communicating to the register.

Example of use

```
{
  uint8_t status;
  status=ACC.check(); // Gets if accelerometer is present and is working properly
}
```

Available Information

On correct working, 'status' should be equal to 0x3A.

3. Getting acceleration on axis

3.1. Registers

There are several registers used to get acceleration on three axis. These registers are defined in the Wasp mote ACC library as constants, that can be modified by the user if required. However, these definitions should not be modified unless registers addresses change in future accelerometer revisions.

NOTE: Accelerometer can use two scales: $\pm 2g$ and $\pm 6g$. The following examples have been tested using the $\pm 2g$ scale.

NOTE2: Our tests have proved 960 is the value returned by the next functions with Wasp mote on rest.

3.2. Getting acceleration on axis X

It checks accelerometer's acceleration on X axis.

It returns the combined contents of data registers 'outXhigh' and 'outXlow' as an integer according to ADC mode configuration (16 bit mode or 12 bit mode).

Since it calls readRegister, it won't activate any flags by itself, but the other functions will activate ACC_COMM_ERROR in case there was an error communicating to the register.

Example of use

```
{
  int16_t accX;
  accX=ACC.getX(); // Gets acceleration on X axis.
}
```

accX can return a value in the range [-2048,2048]

3.3. Getting acceleration on axis Y

It checks accelerometer's acceleration on Y axis.

It returns the combined contents of data registers 'outYhigh' and 'outYlow' as an integer according to ADC mode configuration (16 bit mode or 12 bit mode).

Since it calls readRegister, it won't activate any flags by itself, but the other functions will activate ACC_COMM_ERROR in case there was an error when communicating to the register.

Example of use

```
{
  int16_t accY;
  accY=ACC.getY(); // Gets acceleration on Y axis.
}
```

accY can return a value in the range [-2048,2048]

3.4. Getting acceleration on axis Z

It checks accelerometer's acceleration on Z axis.

It returns the combined contents of data registers 'outZhigh' and 'outZlow' as an integer according to ADC mode configuration (16 bit mode or 12 bit mode).

Since it calls readRegister, it won't activate any flags by itself, but the other functions will activate ACC_COMM_ERROR in case there was an error when communicating to the register.

Example of use

```
{
  int16_t accZ;
  accZ=ACC.getZ(); // Gets acceleration on Z axis.
}
```

accZ can return a value in the range [-2048,2048]

4. Free Fall interruption

4.1. Registers

There are several registers used to set free fall interruptions. These registers are:

- FF_WU_CFG: free-fall and inertial wake-up configuration register.
- FF_WU_SRC: shows the axis on which interruption was generated.
- FF_WU_ACK: dummy register. Data read is not significant.
- FF_WU_THS_L: free-fall and inertial wake-up acceleration threshold less significant bits.
- FF_WU_THS_H: free-fall and inertial wake-up acceleration threshold most significant bits.
- FF_WU_DURATION: minimum duration of the free-fall/wake-up event to be recognized.

Setting these registers to the desired values, interruptions will be generated on different axis and with different thresholds. After many tests, some thresholds have been established but it is recommended to test them on each scenario and determine the correct values for each application.

Values for these registers are defined as constants in the Waspote ACC library, and may be modified by the user.

By default, these values are :

- FF_WU_CFG_val : LIR | ZHIE | YHIE | XHIE (enable X,Y and Z)
- FF_WU_THS_L_val : 0x00 (8 lower bits)
- FF_WU_THS_H_val : 0x3C (8 higher bits) . 0X3C00 is a little below +1g
- FF_WU_DURATION_val : 0x05 (duration value)

Changing these values will make a change on the free-fall interruption. If the threshold is set higher, the interruption will be executed when the acceleration reaches that threshold. By default, a +1g is set, but it is recommended to test it on the real scenario.

4.2. Setting the Free Fall interruption

It sets the Free Fall interruption.

First of all, it clears previous interruptions and writes into the below explained registers the selected values. After writing the registers, the hardware interruption is attached to the pin.

It returns 'flag' to check if there was any error while writing to registers or attaching the interruption.

Example of use

```
{
  int16_t flag;
  flag=ACC.setFF(); // Sets Free Fall interruption
}
```

4.3. Unsetting the Free Fall interruption

It clears and unsets the Free Fall interruption.

First of all, it clears FF_WU_CFG register and the hardware interruption is detached to the pin.

It returns 'flag' to check if there was any error while writing to registers or attaching the interruption.

Example of use

```
{
  int16_t flag;
  flag=ACC.unsetFF(); // Unsets Free Fall interruption
}
```

NOTE: To detect more than one interruptions, it is recommended to set and unset interruptions every time an interruption is generated to prevent accelerometer catch only the first one and discard the rest of the interruptions.

5. Direction Change interruption

5.1. Registers

There are several registers used to set direction change interruptions. These registers are:

- DD_CFG: direction detector configuration register.
- DD_SRC: direction detector source register. It shows the axis on which interruption was generated.
- DD_ACK: dummy register. Read data is not significant.
- DD_THSI_L: direction detection internal threshold less significant bits.
- DD_THSI_H: direction detection internal threshold most significant bits.
- DD_THSE_L: direction detection external threshold less significant bits.
- DD_THSE_H: direction detection external threshold most significant bits.

Setting these registers to the desired values, interruptions will be generated on different axis and with different thresholds. After many tests, some thresholds have been established but it is recommended to test them on each scenario and determine the correct values for each application.

Values for these registers are defined as constants in Waspnote ACC library, and may be modified by the user.

By default, these values are :

- DD_CFG_val : IEND | LIR | YHIE | YLIE | XHIE | XLIE (enable X and Y)
- DD_THSI_L_val : 0x00 (8 lower bits)
- DD_THSI_H_val : 0x32 (8 higher bits) . Internal threshold.
- DD_THSE_L_val : 0x00 (8 lower bits)
- DD_THSE_H_val : 0x3C (8 higher bits) . External threshold.

Changing these values will make a change on the direction detection interruption. If the threshold is set higher, the interruption will be executed when acceleration reaches that threshold. This interruption has two thresholds, one internal and another external. The interruption is generated when both thresholds are reached.

5.2. Setting the Direction Change interruption

It sets the Direction Change interruption.

First of all, it clears previous interruptions and writes into the below explained registers the selected values. After writing the registers, the hardware interruption is attached to the pin.

It can be called using the pre-defined value in Waspmote ACC library, or setting a direction value as input.

It returns 'flag' to check if there was any error while writing to registers or attaching the interruption.

Example of use

```
{
  int16_t flag;
  flag=ACC.setDD(); // Sets Direction Change interruption with pre-defined direction
  uint8_t direction=YHIE | YLIE;
  flag=ACC.setDD(direction); // Sets Direction Change interruption
}
```

5.3. Unsetting the Direction Change interruption

It clears and unsets the Direction Change interruption.

First of all, it clears DD_CFG register and the hardware interruption is detached to the pin.

It returns 'flag' to check if there was any error while writing to registers or attaching the interruption.

Example of use

```
{
  int16_t flag;
  flag=ACC.unsetDD(); // Unsets Direction Change interruption
}
```

6. Accelerometer Modes and Events

6.1. Accelerometer Modes

There are three work modes for the accelerometer: on, hibernate or custom. To change between these modes, there are 2 functions that have been developed.

6.1.1. Setting Accelerometer Mode

It sets accelerometer's work mode. It configures the accelerometer to a new work mode where the possibilities are:

- ACC_HIBERNATE: go into the lowest power mode possible. This mode has no interruption calls and therefore won't disturb the processor if anything happened.
- ACC_ON: read/send at a constant pace upon request.
- ACC_CUSTOM: user defined work mode, should be programmed by the user.

Since it calls 'writeRegister', it won't activate any flags by itself, but the other functions will activate ACC_COMM_ERROR in case there was an error communicating to the register.

Example of use

```
{
  uint8_t workMode=ACC_HIBERNATE;
  ACC.setMode(workMode); // Sets ACC to Hibernate Mode
}
```

Related Variables

accMode → stores the work mode

6.1.2. Getting the Accelerometer Mode

It checks accelerometer's work mode.

It returns the value for the accelerometer's work mode, the possibilities are:

- ACC_HIBERNATE: go into the lowest power mode possible. This mode has no interruption calls and therefore won't disturb the processor if anything happened.
- ACC_ON: read/send at a constant pace upon request.
- ACC_CUSTOM: user defined work mode, should be programmed by the user.

It doesn't call any other functions and therefore it won't activate any flags

Example of use

```
{
  uint8_t workMode;
  workMode=ACC.getMode(); // Gets ACC Work Mode
}
```

6.2. Accelerometer Events

There are several events that can be set on the accelerometer, such as free fall, direction or threshold. To change among these event modes, there are 2 functions that have been developed.

6.2.1. Setting Accelerometer Events

It sets the event type on the accelerometer. It will set up the type of event. They are non exclusive and could be:

- ACC_NONE
- ACC_FREE_FALL
- ACC_DIRECTION
- ACC_THRESHOLD

Example of use

```
{
  uint8_t eventType=ACC_FREE_FALL;
  ACC.setAccEvent(eventType); // Sets Event to Free Fall
}
```

Related Variables

AccEventMode → stores the accelerometer event

6.2.2. Getting Accelerometer Events

It returns the event type on the accelerometer. It will be zero if there are no values set and bigger otherwise.

It doesn't call any other functions and therefore it won't activate any flags.

Example of use

```
{
  uint8_t eventType;
  eventType=ACC.getAccEvent(); // Gets Event
}
```

Related Variables

AccEventMode → stores the accelerometer event

7. Advanced Functions

The Accelerometer module working mode is to set and read registers, thus when it is wanted to change a data a register is set and when data wants to be read a register is read. There are some functions for writing and reading registers. As well, there are some registers that are used to configure the accelerometer module. These registers are called 'CTRL_REG1', 'CTRL_REG2' and 'CTRL_REG3' and are defined in the Waspote ACC library as constants.

Some functions have been created to set and read these registers. However, these functions are not necessary for a basic accelerometer use.

7.1. Writing a Register

It writes a byte into a register in the accelerometer.
It returns 0 or -1 if error.

Example of use

```
{
  uint8_t reg1=B01000111;// value: powering up and enabling axis X,Y and Z
  /* ctrlReg1 is equal to 0x20, Register address
  ACC.writeRegister(ctrlReg1,reg1); // Sets accelerometer's control register 1
}
```

7.2. Reading a Register

It reads a register from the accelerometer.
It returns its value or -1 if error.

Example of use

```
{
  /* statusReg is equal to 0x27, Status Register Address
  uint16_t statusRegister=0;
  statusRegister=ACC.readRegister(statusReg); // Gets accelerometer's control register 1
}
```

7.3. Setting 'CTRL_REG1'

It sets accelerometer's control register 1.
It returns '1' if error.

Since it calls 'writeRegister', it won't activate any flags by itself, but the other functions will activate ACC_COMM_ERROR in case there was an error when communicating to the register.

Example of use

```
{
  /* Setting Device ON and Enabling Axis X,Y and Z */
  uint8_t reg1=B01000111;
  ACC.setCTRL1(reg1); // Sets accelerometer's control register 1
}
```

7.4. Getting 'CTRL_REG1'

It checks accelerometer's control register 1.
It returns the contents of control register 1.

Since it calls 'readRegister', it won't activate any flags by itself, but the other functions will activate ACC_COMM_ERROR in case there was an error when communicating to the register.

Example of use

```
{
  uint8_t reg1;
  reg1=ACC.getCTRL1(); // Gets accelerometer's control register 1
}
```

7.5. Setting 'CTRL_REG2'

It sets accelerometer's control register 2.

It returns '1' if error.

Since it calls 'writeRegister', it won't activate any flags by itself, but the other functions will activate ACC_COMM_ERROR in case there was an error when communicating to the register.

Example of use

```
{
  /* Setting Interruptions enabled */
  uint8_t reg2=B00001000;
  ACC.setCTRL2(reg2); // Sets accelerometer's control register 2
}
```

7.6. Getting 'CTRL_REG2'

It checks accelerometer's control register 2.

It returns the contents of control register 2.

Since it calls 'readRegister', it won't activate any flags by itself, but the other functions will activate ACC_COMM_ERROR in case there was an error when communicating to the register.

Example of use

```
{
  uint8_t reg2;
  reg2=ACC.getCTRL2(); // Gets accelerometer's control register 2
}
```

7.7. Setting 'CTRL_REG3'

It sets accelerometer's control register 3.

It returns '1' if error.

Since it calls 'writeRegister', it won't activate any flags by itself, but the other functions will activate ACC_COMM_ERROR in case there was an error when communicating to the register.

Example of use

```
{
  /* Setting Clock from internal oscillator */
  uint8_t reg3=B00001000;
  ACC.setCTRL3(reg3); // Sets accelerometer's control register 3
}
```

7.8. Getting 'CTRL_REG3'

It checks accelerometer's control register 3.

It returns the contents of control register 3.

Since it calls 'readRegister', it won't activate any flags by itself, but the other functions will activate ACC_COMM_ERROR in case there was an error when communicating to the register.

Example of use

```
{
  uint8_t reg3;
  reg3=ACC.getCTRL3(); // Gets accelerometer's control register 3
}
```

7.9. Getting ADC Mode

It gets the accelerometer's ADC mode. It reads inside the CTRL2 if the ADC is operating in 12 or 16bits.

Returns '0' for 12 bits right aligned (MSB represents the sign) and '1' for 16 bits left aligned.

Since it calls readRegister, it won't activate any flags by itself, but the other functions will activate ACC_COMM_ERROR in case there was an error when communicating to the register.

Example of use

```
{
  uint8_t adcMode;
  adcMode=ACC.getADCmode(); // Gets the ADC mode: 12 or 16 bits
}
```

7.10. Setting ADC Mode

It sets the accelerometer's ADC mode. It writes inside the CTRL2 whether the ADC should operate in 12 or 16bits.

Values returned are '0' or '-1' if error.

Since it calls writeRegister, it won't activate any flags by itself, but the other functions will activate ACC_COMM_ERROR in case there was an error when communicating to the register.

Example of use

```
{
  uint8_t adcMode=1;
  ACC.setADCmode(adcMode); // Sets ADC Mode to 16 bit left justified
}
```

NOTE: 12 bit right justified mode is used throughout the guide and it is the recommended mode to use. It is the compulsory mode establish the Wake-Up and Free-Fall thresholds 16 bit left justified. If 12 bit mode is used, thresholds wouldn't be properly changed.

7.11. Setting sampling rate

It sets the sampling rate. The different sampling rates are 40 Hz, 160Hz, 640Hz and 2560Hz.

It returns '0' on success, '-1' if error.

Example of use

```
{
  ACC.setSamplingRate(ACC_RATE_640); // Sets a 640Hz sampling rate
}
```

8. Code examples and extended information

For more information about the WaspMote hardware platform go to the Support section:

<http://www.libelium.com/support/waspmote>

Extended information about the API libraries and complete code examples can be found at:

<http://www.libelium.com/development/waspmote>