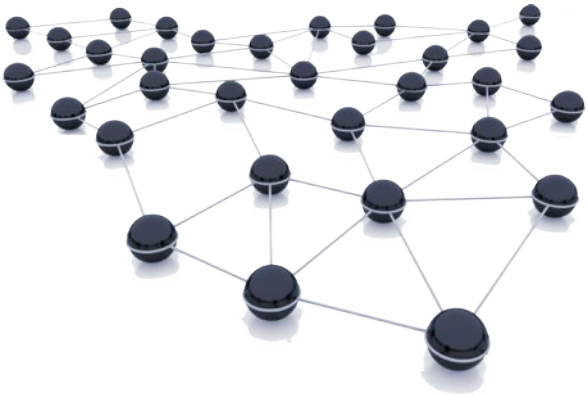
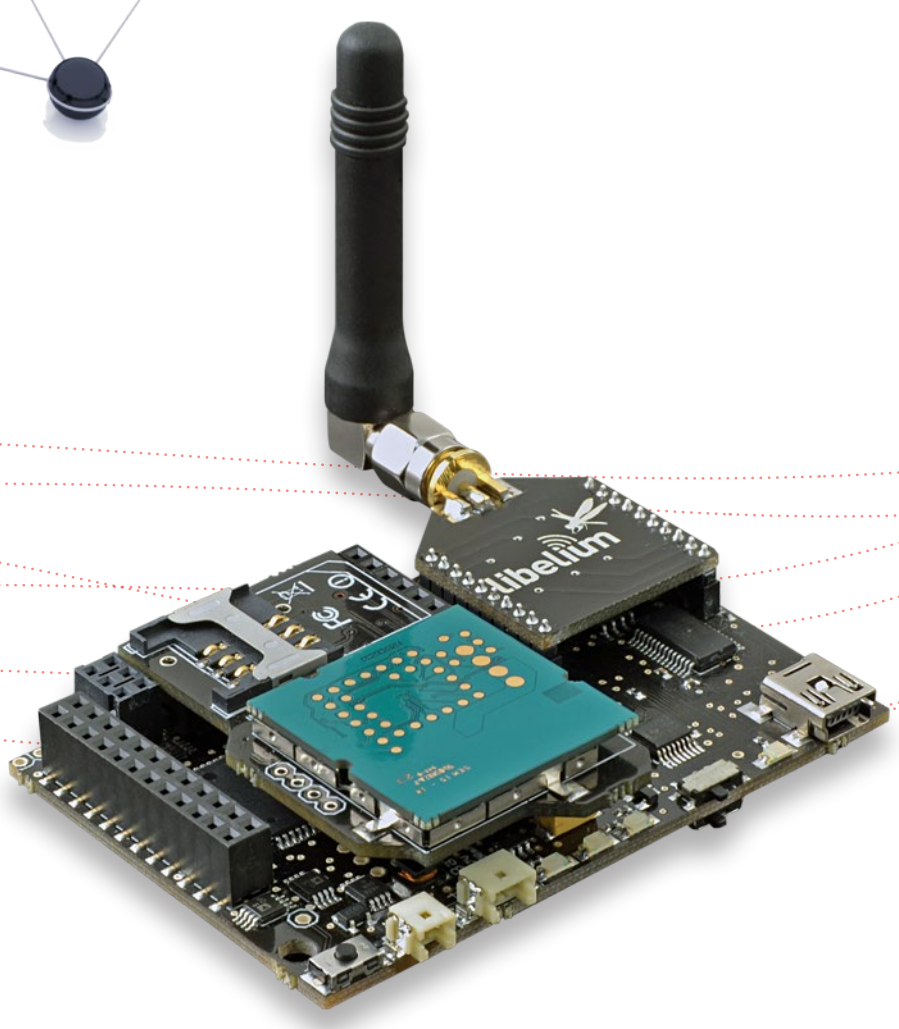
Waspmote Interruptions

Programming Guide





libelium

wasp mote

Document Version: v4.2 - 09/2013

INDEX

1. General Considerations

1.1. Waspmote Libraries

1.1.1. Waspmote Interruptions Files

Winterrupts.c

1.1.2. Structure

The functions used to manage interruptions are stored in 'Winterrupts.c'. This C file has no header file associated, but the functions are declared in 'wiring.h'. Since it is a C file, no constructor is needed and no functions declared in other C++ Waspmote libraries can be used, only the libraries developed in C.

2. Architecture

2.1. Interruption pins

The microcontroller receives all type of interruptions as **hardware interruptions**. For external interruptions, Waspmote uses the following interruption pins:

- **RXD1**
- **TXD1**

***NOTE**: keep in mind that the interruption pins are the same UART1 transmission/reception pins. This means that the same pin is used for both communication and interruption signals.*

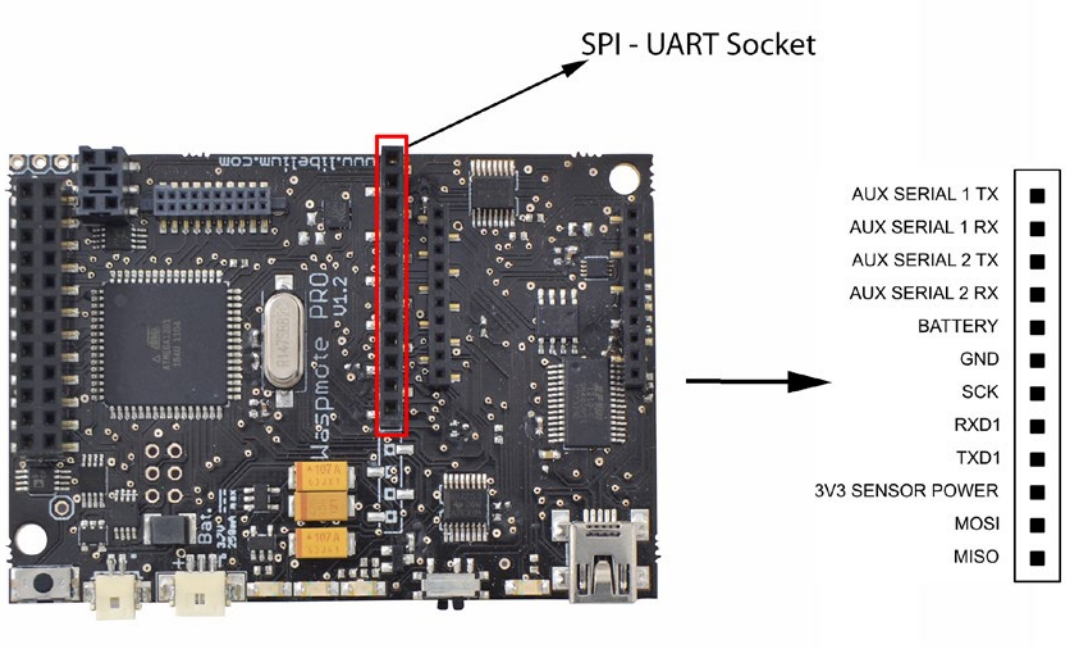


Figure 1: Interruption pins within the SPI-UART socket

2.2. Interruption sources

Waspmote has several interruption sources which can be used:

- Watchdog
- RTC
- Accelerometer
- XBee
- Sensors

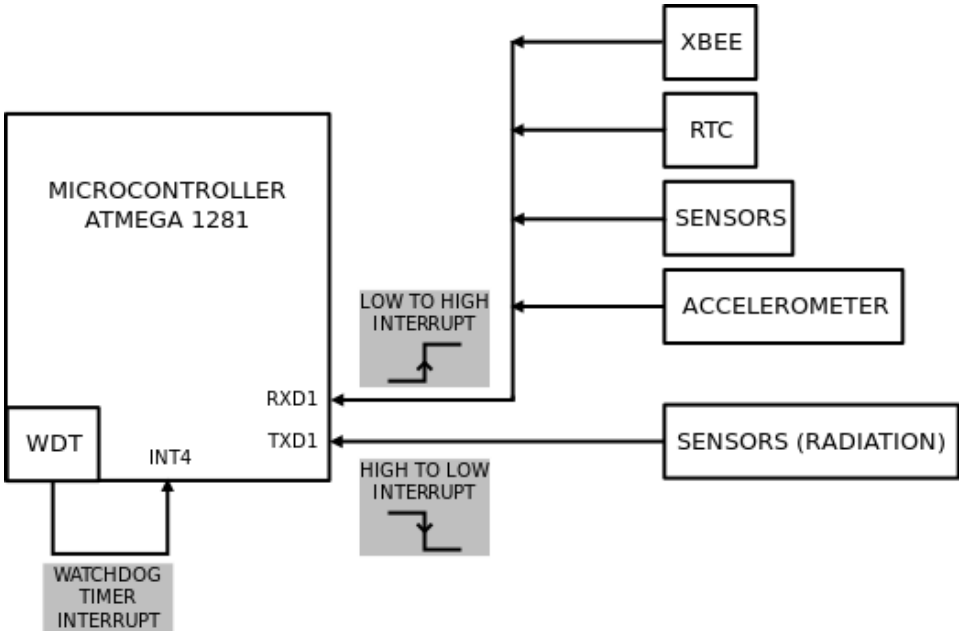


Figure 2: Interruptions diagram

The interruption signals from the RTC, accelerometer, XBee and sensors are connected to **RXD1** pin. The internal Watchdog 'simulates' a hardware interruption using a reserved digital pin (DIGITAL0). This simulation by the Watchdog has been implemented to maintain the same functionality structure in all the interruption sources and can fill in flags in the same way.

The interruption pin of the modules is connected through a diode to **RXD1** or **TXD1** in order to avoid interferences between the different interruption sources. Besides, there is a unique **monitoring pin** for each module. Thus, when the interruption is captured, the monitoring pin of each module is analyzed to see which one generated it.

The definition of the monitoring and interruption pins can be found in the WaspConstants.h file:

ACC_INT: Accelerometer interruption

RTC_INT: RTC interruption

WTD_INT: Watchdog interruption

SENS_INT: Generic Sensor Board interruption

PLV_INT: Pluviometer interruption

HIB_INT: Hibernate interruption

RAD_INT: Radiation sensor interruption

XBEE_INT: XBee module interruption

PIR_3G_INT: PIR sensor interruption (Video Camera Board)

2.3. Interruption Flags

Because of the multiplexing of these interruption signals, a series of flag vectors have been created in WaspVariables.h to find out the module which generated the interruption. These flags are:

- **intConf**
- **intFlag**
- **intCounter**
- **intArray**

2.3.1. intConf

This bitmap vector is used to set which interruptions are enabled to be captured by Waspmote. It is defined as a two-byte variable in WaspVariables.h:

```
uint16_t   intConf;
```

Only interruptions previously enabled in this vector will be captured. Otherwise, if the interruption signal arrives, but `intConf` is not set correctly, the interruption will be lost.

A bit set to '0' indicates the interruption is disabled and even if it arrives it will not be captured.

A bit set to '1' indicates the corresponding interruption is enabled to be captured in the case it arrives.

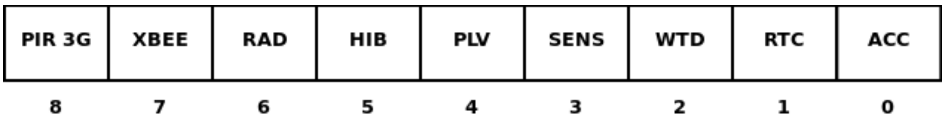


Figure 3: Structure of the 'intConf' flag

For instance, if accelerometer and RTC interruptions have been enabled, then the corresponding bits inside `intConf` are set to '1'. The rest of the bits remain set to '0':

PIR 3G	XBEE	RAD	HIB	PLV	SENS	WTD	RTC	ACC
0	0	0	0	0	0	0	1	1

This means `intConf` is equal to binary B000000011 which is the same as decimal 3.

2.3.2. intFlag

This bitmap vector is used to indicate which module generated the captured interruption. It Is defined as a two-byte variable in WaspVariables.h:

```
uint16_t   intFlag;
```

Each bit corresponds to a specific interruption source.

A bit set to '0' indicates no interruption has been captured for a specific source.

A bit set to '1' indicates the corresponding interruption has been captured.

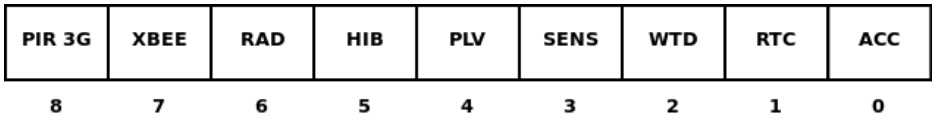


Figure 4: Structure of the 'intFlag' flag

For instance, if the RTC interruption has been captured, then the corresponding bit inside `intFlag` is set to '1'. The rest of non-captured bits remain set to '0':

PIR 3G	XBEE	RAD	HIB	PLV	SENS	WTD	RTC	ACC
0	0	0	0	0	0	0	1	0

This means `intFlag` is equal to binary B000000010 which is the same as decimal 2.

The API defines the following constants to check what interruption has been captured in `intFlag`:

```
#define  ACC_INT       1      // bit 0
#define  RTC_INT       2      // bit 1
#define  WTD_INT       4      // bit 2
#define  SENS_INT      8      // bit 3
#define  PLV_INT       16     // bit 4
#define  HIB_INT       32     // bit 5
#define  RAD_INT       64     // bit 6
#define  XBEE_INT      128    // bit 7
#define  PIR_3G_INT    256    // bit 8
```

2.3.3. intCounter

This variable is used to count the total number of interruptions which have been captured. Each time an interruption is captured this counter is increased independently of the interruption source.

2.3.4. intArray

This array of bytes is used to count the total number of interruptions that have been captured for each module. It is defined in WaspVariables.h:

```
uint8_t   intArray[8];
```

Each time an interruption is captured the position corresponding to the module is increased.

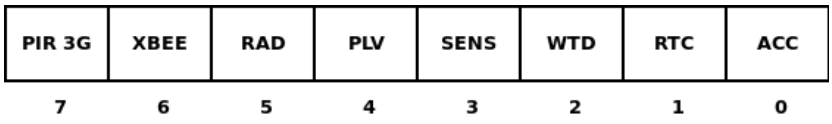


Figure 5: Diagram of the 'intArray' flag

The API defines the following constants to read the number of interruptions captured for each module:

```
#define  ACC_POS       0
#define  RTC_POS       1
#define  WTD_POS       2
#define  SENS_POS      3
#define  PLV_POS       4
#define  RAD_POS       5
#define  XBEE_POS      6
#define  PIR_3G_POS    7
```

So, in Waspmote codes you only need to access this way:

Example of use:

```
{
  USB.println( intArray[ACC_POS], DEC );

  USB.println( intArray[RTC_POS], DEC );
  // etc...
}
```

For instance, if the RTC interruption has been **captured** for 3 times, then the corresponding byte inside `intArray` stores this information:

PIR 3G	XBEE	RAD	HIB	PLV	SENS	WTD	RTC	ACC
0	0	0	0	0	0	0	3	0

3. How to use interruptions

The most common use of interruptions in Waspmote is to mix them with low power consumption states. The repeating procedure shall be:

- **Step 1:** Enable the interruption you want to listen to
- **Step 2:** Set low-power consumption states and wait an interruption to wake Waspmote up from this state
- **Step 3:** Interruption event happens
- **Step 4:** Disable interruptions
- **Step 5:** Check the interruption flags in order to know the interruption source which generated the alarm
- **Step 6:** Do normal operation (measuring and sending information)
- **Step 7:** Clear interruption flags
- **Step 8:** Clear the interruption pin to increase the code robustness

3.1. Enable interruptions

Keep in mind that there are two basic steps involved when enabling interruptions:

- Enable the microcontroller pin to generate interruptions (RXD1 and/or TXD1 pin)
- Set up the interruption source to generate the interruption signal

Enable the interruption pin

There is a general function `enableInterrupts` which modifies `intConf` to add a new active interruption related to a specific module. Besides, the interruption pin (RXD1 or TXD1) is enabled to generate interruptions attaching the corresponding subroutine to this event.

***NOTE**: When enabling an interruption for a module, other modules are capable of interrupting the microcontroller through the same interruption pin.*

Example of use

```
{
        // Enables the specified interruption
        enableInterrupts(ACC_INT);
}
```

Set up the interruption source

Each module has its own way to set up interruptions. Normally, the API functions that enable the interruption sources also enable the corresponding microcontroller interruption pin (RXD1 or TXD1) so usually it is not necessary to call `enableInterrupts` function.

The best way to understand each setting is to read more about them in the specific chapter dedicated for each module in this document.

3.2. Set low-power consumption states

When entering a low-power consumption state, **some peripherals are switched off by default:**

- Both microcontroller's UARTs are closed
- SOCKET1 power supply is switched off
- RTC is powered down to its backup battery pin in order to maintain the interruption and date info
- I2C bus is unset
- SD card is switched off

Besides, the low-power consumption functions in the API permit to **select what power supplies of Waspmote are switched off.** On the contrary, it is possible to leave some of these supplies powered on if the user is interested on it.

The input arguments for the low-power consumption state are:

`SENS_OFF:` Switch off the Sensor Board power supply (5V and 3V3 supply lines)
`SOCKET0_OFF:` Switch off the SOCKET0 power supply
`ALL_OFF:` Switch off all power supplies. SOCKET0 and Sensor Board are switched off
`ALL_ON:` Do not switch off any power supply. The power supplies are not switched on, they are kept as they are.

There are several modes to set low-power consumption states mixed with external interruptions:

- **sleep** mode
- **sleep** mode with Watchdog interruption
- **deep sleep** mode with RTC interruption
- **sleepAgr** mode (when using Agriculture Board)

3.2.1. Sleep

In this state, Waspmote only expects external interruptions, so the user must keep in mind to set up them to be able to wake up the microcontroller.

The sleep options are specified as input:

Examples of use

```
{
   // case 1: switch off all power supplies
   PWR.sleep( ALL_OFF );

   // case 2: switch off SOCKET0's power supply
   PWR.sleep( SOCKET0_OFF );

   // case 3: switch off Sensor Board's power supply
   PWR.sleep( SENS_OFF );

   // case 4: do not switch off any power supply. Keep them as they actually are
   PWR.sleep( ALL_ON );
}
```

3.2.2. Sleep with Watchdog interruption

In this mode, Waspmote expects external interruptions. Besides, the Watchdog interruption is always enabled to ensure the microcontroller wakes up after the specified time if any other interruption arrives. Possible interval values from 16ms to 8s:

```
WTD_16MS
WTD_32MS
WTD_64MS
WTD_128MS
WTD_250MS
WTD_500MS
WTD_1S
WTD_2S
WTD_4S
WTD_8S
```

The sleep options are also specified as input:

Example of use

```
{
   // Set sleep mode:
   // 1. Enabling Watchdog interruption to be generated after 32ms
   // 2. All modules are switched off
   PWR.sleep(WTD_32MS, ALL_OFF);
}
```

3.2.3. Deep Sleep

In this mode, Waspmote expects external interruptions. Besides, the **RTC interruption is always enabled** to ensure the microcontroller wakes up if no interruption arrives. The RTC alarm is set as specified in the **RTC Programming Guide**.

The sleep options are also specified as input:

Example of use

```
{
   // Set deep sleep mode:
   // 1. Enabling RTC interruption to be generated after 10 seconds from now
   // 2. All modules are switched off
   PWR.deepSleep("00:00:00:10", RTC_OFFSET, RTC_ALM1_MODE2, ALL_OFF);


   // Set deep sleep mode:
   // 1. Enabling RTC interruption to be generated at 17:00 on day 15th
   // 2. All modules are switched off
   PWR.deepSleep("15:17:00:00", RTC_ABSOLUTE, RTC_ALM1_MODE2, ALL_OFF);
}
```

3.2.4. Agriculture Board Sleep mode

The Agriculture board uses its self low-power consumption mode because there are some digital pins which need to be disabled.

```
{
   // Set Agriculture sleep mode:
   // 1. Enabling RTC interruption to be generated after 30 seconds from now
   // 2. SOCKET0 is switched off
   SensorAgrv20.sleepAgr("00:00:00:30", RTC_OFFSET, RTC_ALM1_MODE3, SOCKET0_OFF );


   // Set Agriculture sleep mode:
   // 1. Enabling both RTC and pluviometer interruption
   // 2. SOCKET0 and SOCKET1 are switched off
      SensorAgrv20.sleepAgr("00:00:00:30",  RTC_OFFSET,  RTC_ALM1_MODE3,  SOCKET0_OFF,  SENS_AGR_
PLUVIOMETER);
}
```

3.2.5. Interruptions with no low-consumption state

It is possible to set up interruptions with no sleep mode in Waspmote by simply avoiding this step. During the execution, when the interruption arrives, the microcontroller enters the corresponding subroutine and marks the interruption flags. So, the user's program should poll `intFlag` so as to check if there are pending interruptions to be treated.

It is important to remark that when listening to interruptions, it is not possible to communicate via UART1 because there would be interferences in the interruption line. This means it is not possible to use SOCKET1, nor GPS nor auxiliar sockets.

3.3. Interruption events execute subroutines

A **subroutine** is a function called when an interruption occurs in order to treat the interruption event. Waspmote API has been designed not to do any long-term operation inside the subroutines, but to mark the corresponding flag associated to the interruption event.

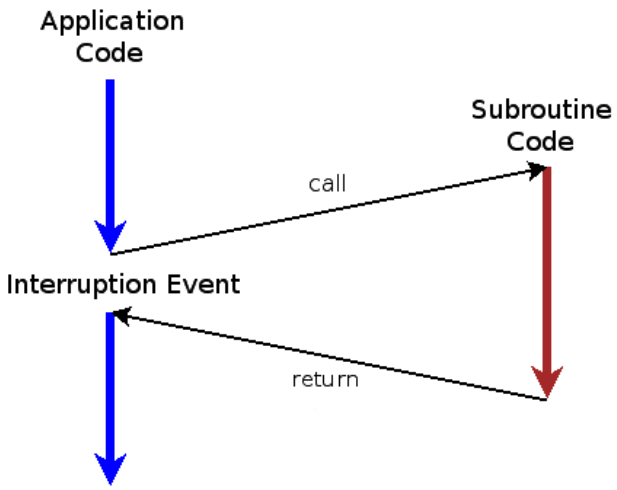


Figure 6: Subroutine is executed when an interruption is received

Two subroutines defined in Winterruptions.c are associated to the interruption pins when the interruptions are enabled:

- **onHAIwakeUP** (On High Active Interruption). This subroutine is run by the modules which generate rising-edge interruptions (usually in RXD1 pin).
- **onLAIwakeUP** (On Low Active Interruption). This subroutine is run by the modules which generate falling-edge interruptions (usually in TXD1 pin).

These subroutines check both `intConf` flag and monitoring pin related to each interruption source in order to know what module triggered the interruption. When the two conditions match:

- `intFlag` is marked on the correct position to show the module that provoked the interruption.
- `intCounter` is incremented each time an interruption is detected.
- `intArray` is incremented in the correct position to know how many times a module has activated the interruption.

Initially no further action is taken, in an attempt to add the minimum time delay possible in the execution trace of the program's main code. Once these flags are updated, the microcontroller returns to the last executed instruction before capturing the interruption.

It is in the **main code** where the interruption must be treated by the user when the interruption flag is marked in `intFlag` vector. The aim of this design is to avoid pausing actions that Waspmote may be doing at the moment the interruption triggers, an example would be sending a packet through the XBee radio.

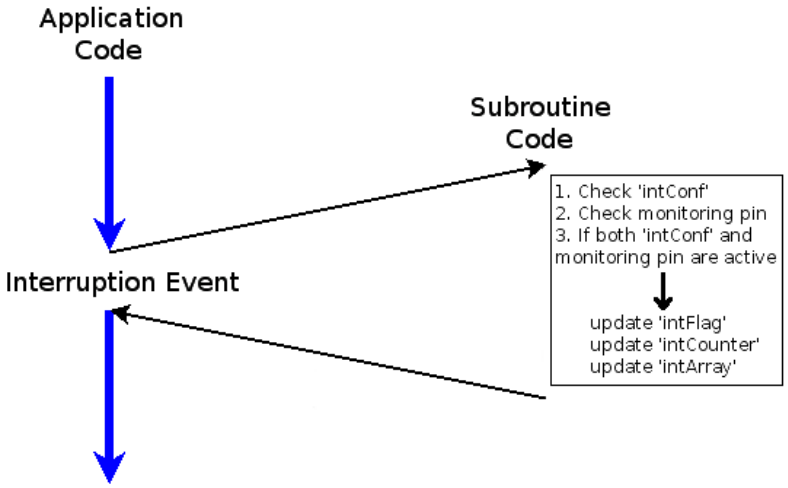


Figure 7: aSubroutine operation when an interruption arrives

3.4. Disable interruptions

Similarly to the interruption enabling functions, keep in mind that there are two basic steps involved in disabling interruptions:

- Disable the microcontroller pin not to generate interruptions (RXD1 and/or TXD1 pin)
- Disable the interruption source not to generate any interruption signal in the interruption/UART pins

Disable the interruption pin

There is a general function `disableInterrupts` which modifies `intConf` to remove enabled interruptions related to specific modules. Besides, the interruption pin (RXD1 or TXD1) is disabled to avoid generating interruptions in that pin.

***NOTE**: When disabling an interruption for a module, also interruptions are disabled for the rest of the modules which share the same interruption pin but intConf keeps configured for those other modules.*

Example of use

```
{
   // Disables the specified interruption
   disableInterrupts(ACC_INT);
}
```

Disable the interruption source

Each module has its own way to disable interruptions. The best way to understand it is to read more about them in the specific chapter dedicated for each module in this document.

3.5. Check the interruption flags

As explained before, there are several flags which are updated inside the interruption subroutines which permit to know if an interruption has happened and which of the modules has triggered it.

After an interruption occurs the user must analyze the contents of `intFlag` in order to know the interruption source. The API defines some constants related to each interruption source inside `intFlag`:

```
#define   ACC_INT    1      // bit 0
#define   RTC_INT    2      // bit 1
#define   WTD_INT    4      // bit 2
#define   SENS_INT   8      // bit 3
#define   PLV_INT    16     // bit 4
#define   HIB_INT    32     // bit 5
#define   RAD_INT    64     // bit 6
#define   XBEE_INT   128    // bit 7
#define   PIR_3G_INT  256   // bit 8
```

Thus, the following comparison allows the user to distinguish between different events:

```
{
      /*** Check the interruption source ***/
      if( intFlag & ACC_INT )
      {
        // accelerometer interruption treatment
      }
      if( intFlag & RTC_INT )
      {
        // RTC interruption treatment
      }
            if( intFlag & WTD_INT )
```

```
        {
          // Watchdog interruption treatment
        }


        // etc...
}
```

3.6. Clear the interruption flags

After checking the interruption source of the event, it is mandatory to clear the 'intFlag' variable. If not, the captured interruption is not cleared and the next checking will determine that the same interruption was provoked again.

The following function clears the global flag 'intFlag'. It sets 'intFlag' to zero.

Example of use

```
        {
           clearIntFlag(); // Clears 'intFlag'
        }
```

3.7. Clear the interruption pin

At the end of the process, it is recommended to use the following function in order to clean the interruption pins (which are shared with the UART1). This will help to increase the robustness of your codes because it permits to avoid interferences in this line between different interruption sources and communication via UART1.

```
Example of use
        {
           PWR.clearInterruptionPin();
        }
```

3.8. Basic example

This is a basic example based on accelerometer interruptions. It is possible to see all the steps previously explained in order to use the interruptions in Waspmote. In this example, a sleep mode is used switching off all modules.

```
void loop()
{
  // Starts accelerometer
  ACC.ON();

  // Enable interruption: ACC Free Fall interruption
  ACC.setFF();

  // Set low-power consumption state
  USB.println(F("Enter low-power consumption state"));
  PWR.sleep(ALL_OFF);

  // Disable interruption
  ACC.ON();
  ACC.unsetFF();

  USB.ON();
  USB.println(F("Waspmote wakes up"));

  // Check the interruption source
  if( intFlag & ACC_INT )
  {
    function_ACC();
  }

  // Clear 'intFlag'
  clearIntFlag();

  // Clear interruption pin
  PWR.clearInterruptionPin();
}
```

Figure 8: Basic example of interruptions in Waspmote

4. Watchdog

The Watchdog allows the microcontroller to be woken up from a low consumption state by generating an interruption.

This is a software-generated interruption. The interruption generation is different to the other Waspmote modules. To unify the operation with the rest of the modules, a hardware interruption is simulated when receiving the software interruption.

Enable interruption

The `PWR.setWatchdog(mode,timer)` function enables the microcontroller interruption in the INT4 pin. Besides, it sets the Watchdog interruption indicating the mode and timer to set.

The possible modes are:

`WTD_ON:` enable the Watchdog
`WTD_OFF:` disable the Watchdog

The possible timers before generating the interruption are:

`WTD_16MS:`	16ms
`WTD_32MS:`	32ms
`WTD_64MS:`	64ms
`WTD_128MS:`	128ms
`WTD_250MS:`	256ms
`WTD_500MS:`	500ms
`WTD_1S:`	1s
`WTD_2S:`	2s
`WTD_4S:`	4s
`WTD_8S:`	8s

Example of use:

```
{
   // enable Watchdog interruption to be generated after 8 seconds
   PWR.setWatchdog(WTD_ON,WTD_8S);
}
```

Interruption Event

When the Watchdog interruption occurs, an internal subroutine is run to generate the hardware interruption simulation. The reserved DIGITAL0 (INT4 pin) is used to monitor this simulated interruption which is a low active interruption. For this reason, when this interruption occurs, the subroutine onLAIwakeUP is executed, marking the corresponding flags.

Disable interruption

The `PWR.setWatchdog(mode,timer)` function disables the INT4 interruption pin and stops the Watchdog timer.

Example of use:

```
{
   // disable Watchdog interruption
   PWR.setWatchdog(WTD_OFF,WTD_8S);
}
```

Check Watchdog interruption

In order to check the interruption capture, it is necessary to poll `intFlag` as seen below:

```
if( intFlag & WTD_INT )
{
  // Watchdog interrupt treatment
}
```

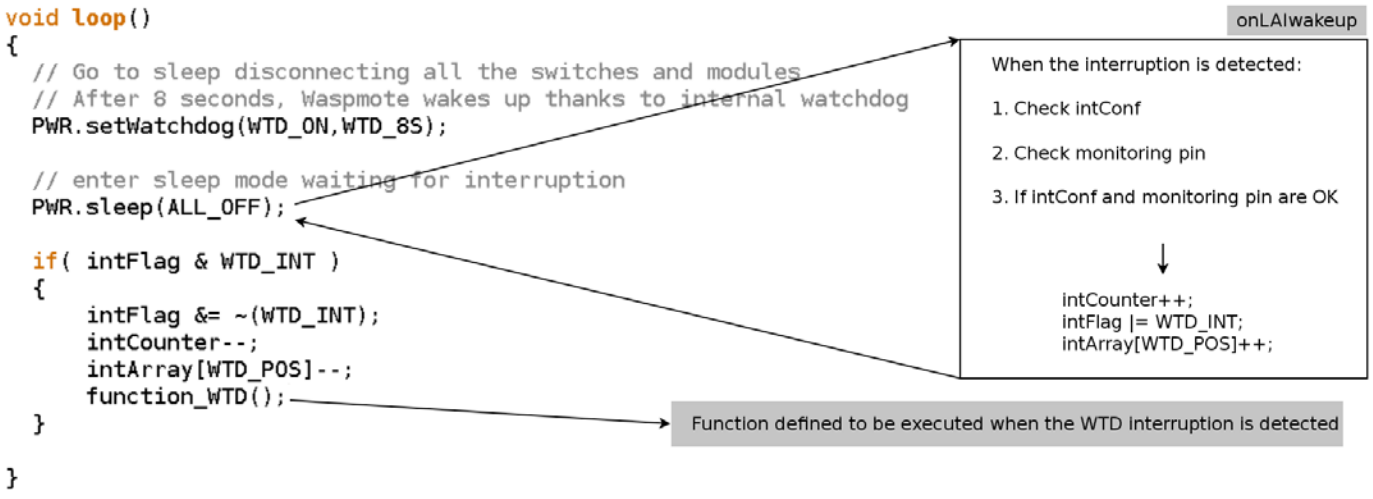


Figure 9: How to use the Watchdog interruption

The generated interruptions can also be used by the Watchdog as timed alarms without the need for the microcontroller to be in energy saving mode. The Watchdog alarms can be used for time less or equal to 8s. For longer times the RTC must be used.

• Watchdog interruption example:

http://www.libelium.com/development/waspmote/examples/int-01-watchdog-timer-interrupt

Related API libraries: **WaspPWR.h, WaspPWR.cpp**

All information about their programming and operation can be found in the document: **Energy and Power Programming Guide**.

All the documentation is located in the **Development section** in the Libelium website.

5. RTC

The RTC can be used to set a common time base in all Waspmotes within a network. Also, it can generate interruptions based on a specified temporal period.

The RTC provides two alarms to enable interruptions and wake up the microcontroller from a low consumption mode :

Alarm 1 is accurate to one second

Alarm 2 is accurate to one minute

The use of the RTC has been associated with Waspmote's **Deep Sleep** and **Hibernate** modes, allowing it to put the microcontroller to sleep, activating alarm 1 in the RTC to wake it. Thus, Waspmote can be put into the lowest consumption mode and woken up using the RTC. For more information about these energy saving modes, consult the **Energy and Power Programming Guide**.

Enable interruption

The `RTC.setAlarm1(time, offset, mode)` function sets Alarm1 to the specified time, offset and mode. This function also enables the RXD1 interruption pin.

The `RTC.setAlarm2(time, offset, mode)` function sets Alarm2 to the specified time, offset and mode. This function also enables the RXD1 interruption pin.

Refer to the **RTC Programming Guide** in order to know more about the setting of the RTC alarms.

Interruption event

The RTC generates a rising-edge interruption through the **RXD1 pin**. It also has a unique monitoring pin. When the interruption occurs in the RXD1 pin, the subroutine 'onHAIwakeUP' is run marking the corresponding flags.

Disable interruption

The `RTC.detachInt()` function disables the RXD1 interruption pin and also disables both RTC alarm1 and alarm2 so as not to provoke any interruption. After calling this function, no module will be able to interrupt the microcontroller via RXD1 because it has been disabled.

Example of use:

```
{
   // disable RTC interruption
   RTC.detachInt();
}
```

Check RTC interruption

In order to check the interruption capture, it is necessary to poll `intFlag` as seen below:

```
if( intFlag & RTC_INT )
{
  // RTC interrupt treatment
}
```

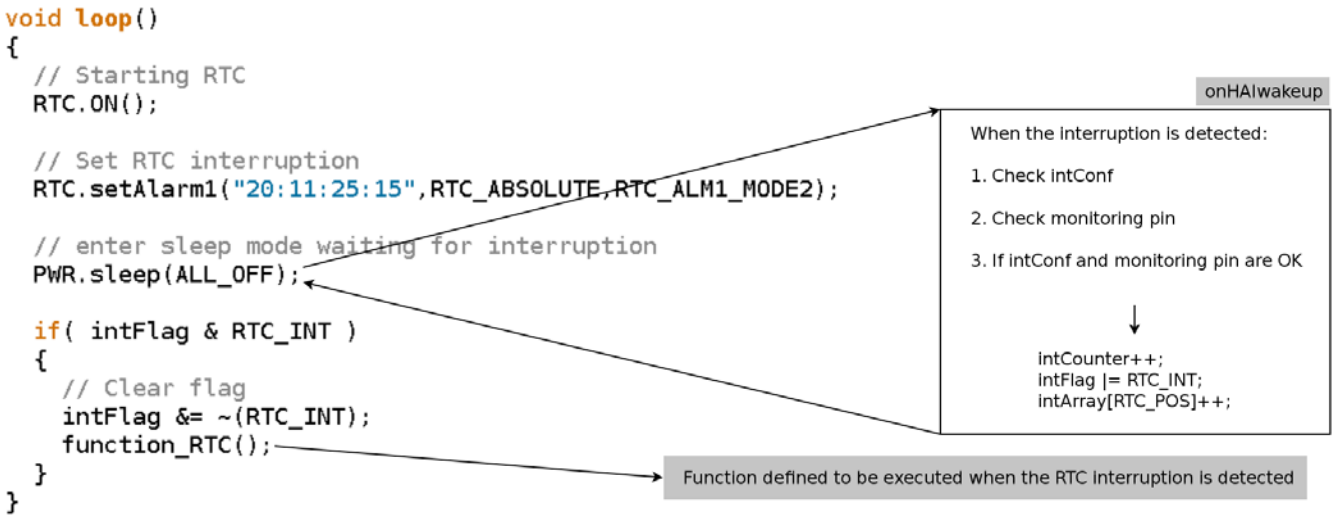


Figure 10: RTC alarm operation

It is also possible to use the interruptions generated by the RTC as timed alarms without the need for the microcontroller to be in energy saving modes. The use of RTC alarms is recommended for times longer than 8s, since for shorter timers the microcontroller's internal Watchdog can be used.

• RTC interruption example:

http://www.libelium.com/development/waspmote/examples/int-02-rtc-alarm-interrupt

Related API libraries: **WaspRTC.h, WaspRTC.cpp**

All information about their programming and operation can be found in the document: **RTC Programming Guide.**

All the documentation is located in the **Development section** in the Libelium website.

6. Accelerometer

The Waspmote accelerometer is used to get the acceleration in 3-axes. Besides, it can generate different types of interruptions. But only one type of interruption at a time.

Enable interruption

The Waspmote accelerometer can generate several types of interruption: free fall, inertial wake up, 6D movement and 6D position. Different functions have been created in order to enable each kind of interruption (only once at a time).

The `ACC.setFF()` function enables the accelerometer free-fall interruption and enables the RXD1 interruption pin.

The `ACC.setIWU()` function enables the accelerometer inertial wake up interruption and enables the RXD1 interruption pin.

The `ACC.set6DMovement()` function enables the accelerometer 6-direction movement interruption and enables the RXD1 interruption pin.

The `ACC.set6DPosition()` function enables the accelerometer 6-direction position interruption and enables the RXD1 interruption pin.

Interruption event

The accelerometer generates a rising-edge interruption through the RXD1 pin. It also has a unique monitoring pin. When the interruption occurs in the RXD1 pin, the subroutine 'onHAIwakeUP' is run marking the corresponding flags.

The alarms are generated when defined thresholds are reached. These thresholds determine what is the needed acceleration for triggering the interruption. These thresholds are defined in 'WaspACC.h'.

Disable interruption

Depending on the type of interruption previously enabled, it is necessary to call a specific function in order to disable the interruption.

The `ACC.unsetFF()` function disables both accelerometer interruption and RXD1 interruption pin.

The `ACC.unsetIWU()` function disables both accelerometer interruption and RXD1 interruption pin.

The `ACC.unset6DMovement()` function disables both accelerometer interruption and RXD1 interruption pin.

The `ACC.unset6DPosition()` function disables both accelerometer interruption and RXD1 interruption pin.

After calling any of these functions, no module will be able to interrupt the microcontroller via RXD1 because it has been disabled.

Check ACC interruption

In order to check the interruption capture, it is necessary to poll `intFlag` as seen below:

```
if( intFlag & ACC_INT )
{
  // Accelerometer interruption treatment
}
```

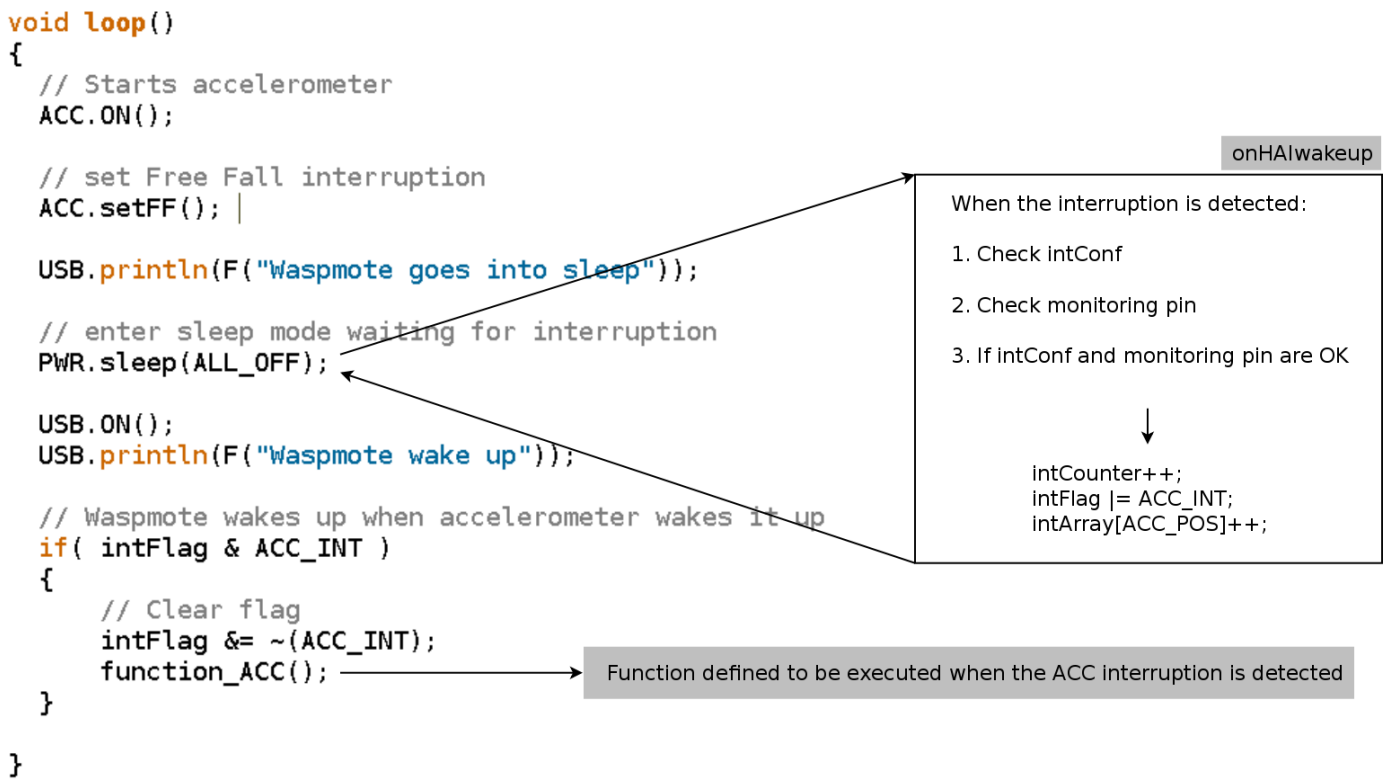


Figure 11: Accelerometer interruption operation

It is also possible to use the interruptions generated by the ACC without the need for the microcontroller to be in energy saving modes.

• Accelerometer interruption example:

http://www.libelium.com/development/waspmote/examples/int-03-accelerometer-interrupt

Related API libraries: **WaspACC.h, WaspACC.cpp**

All information about their programming and operation can be found in the document: **Accelerometer Programming Guide.**

All the documentation is located in the **Development section** in the Libelium website.

7. Sensors

The sensors are connected to Waspmote through the microcontroller's analog and digital inputs. Some of Libelium Sensor Boards can be used to generate interruptions:

- Agriculture
- Events
- Smart Cities
- Video Camera
- Radiation

Enable interruption

The **Agriculture Board** interrupts through the pluviometer sensor interruption. The `SensorAgrv20.attachPluvioInt()` function enables both pluviometer interruption (`PLV_INT` is enabled in `intConf`) and RXD1 interruption pin.

The **Events Board** interrupts through one of the different sensor which can be plugged to several sockets available in this board. The `SensorEventv20.attachInt()` function enables both Sensor Board interruption (`SENS_INT` is enabled in `intConf`) and RXD1 interruption pin. Besides, it is necessary to set a threshold for each socket to set up the interruption generation as needed. The function used to set thresholds is `SensorCities.setThreshold(sensor,threshold)`.

The **Smart Cities Board** interrupts through one of the different sensor which can be plugged to several sockets available in this board. The `SensorCities.attachInt()` function enables both Sensor Board interruption (`SENS_INT` is enabled in `intConf`) and RXD1 interruption pin. Besides, it is necessary to set a threshold for each sensor to set up the interruption generation as needed. The function used to set thresholds is `SensorCities.setThreshold(sensor, threshold)`.

The **Video Camera Board** interrupts through the PIR sensor. The `_3G.enablePIRInterrupt()` function enables both PIR interruption (`PIR_3G_INT` is enabled in `intConf`) and RXD1 interruption pin.

Interruption event

All Sensor Boards generate a rising-edge interruption through the **RXD1 pin**. All interruption sources have a unique monitoring pin which permits to know which sensor provoked the alarm. When the interruption occurs in RXD1 pin, the subroutine 'onHAIwakeUP' is run marking the corresponding flags.

Disable interruption

Each Sensor Board uses a specific function so as to disable the interruption:

The **Agriculture Board** uses the `SensorAgrv20.detachPluvioInt()` function to disable both pluviometer interruption (`PLV_INT` is disabled in `intConf`) and RXD1 interruption pin. It is mandatory to power down the Agriculture Board when UART1 is being used in order to avoid interferences in this line.

The **Events Board** uses the `SensorEventv20.detachInt()` function to disable both Sensor Board interruption (`SENS_INT` is disabled in `intConf`) and RXD1 interruption pin.

The **Smart Cities Board** uses the `SensorCities.detachInt()` function to disable both Sensor Board interruption (`SENS_INT` is disabled in `intConf`) and RXD1 interruption pin.

The **Video Camera Board** uses the `_3G.disablePIRInterrupt(state)` function to disable both PIR interruption (`PIR_3G_INT` is disabled in `intConf`) and RXD1 interruption pin.

Check Sensors interruptions

Depending on the board:

The **Agriculture Board** needs the following checking:

```
if( intFlag & PLV_INT )
{
  // Pluviometer interruption treatment
}
```

The **Events Board** needs to load the interruption register after the alarm is generated in order to know which of the sensors in the board caused the interruption:

```
// Load the interruption register
SensorEventv20.loadInt();

// Check Sensor Board interruption
if( intFlag & SENS_INT )
{
   // check SENS_SOCKET1 interruption
   if( SensorEventv20.intFlag & SENS_SOCKET1 )
   {}

   // check SENS_SOCKET2 interruption
   if( SensorEventv20.intFlag & SENS_SOCKET2 )
   {}

   ...

   // check SENS_SOCKET6 interruption
   if( SensorEventv20.intFlag & SENS_SOCKET6 )
   {}
}
```

The **Smart Cities Board** needs to load the interruption register after the alarm is generated in order to know which of the sensors in the board caused the interruption. In this case it is necessary to check directly the `SensorCities.intFlag` because the general `intFlag` is not useful:

```
// Load the interruption register
SensorCities.loadInt();

// check SENS_CITIES_HUMIDITY interruption
if( SensorCities.intFlag & SENS_CITIES_HUMIDITY )
{}

// check SENS_CITIES_AUDIO interruption
if( SensorCities.intFlag & SENS_CITIES_AUDIO )
{}

...
```

```
// check SENS_CITIES_TEMPERATURE interruption
if( SensorCities.intFlag & SENS_CITIES_TEMPERATURE )
{}
```

The **Video Camera Board** needs the following checking:

```
if( intFlag & PIR_3G_INT )
{
  // PIR interruption treatment
}
```

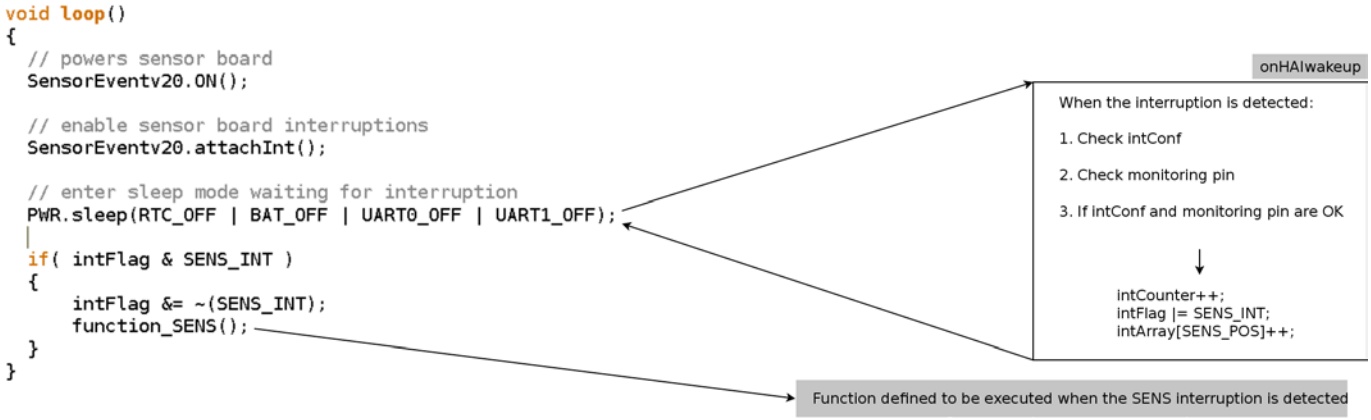


Figure 12: Sensor board alarm operation

• Sensor Event interruption example:

http://www.libelium.com/development/waspmote/examples/int-05-sensor-event-interrupt

• Agriculture interruption example:

http://www.libelium.com/development/waspmote/examples/int-05b-agriculture-interrupt

• Smart Cities interruption example:

http://www.libelium.com/development/waspmote/examples/int-05c-smart-cities-interrupt

• Video Camera interruption example:

http://www.libelium.com/development/waspmote/examples/int-05d-pir-3g-interrupt

Sensor board interruptions are managed quite differently from the rest of the interruptions, so it is recommended to check the sensor boards technical guides.

All the documentation is located in the **Development section** in the Libelium website.

8. Good practices

8.1. Beware of interferences

Waspmote's UART1 and interruption pins are the same microcontroller pins. Thus, it is mandatory to separate the usage of these pins to perform as UART or interruption pins. It is not recommended to use them as both UART and interruption pins at the same time because interference problems will appear, i.e. loss of interruption events, continuous interruptions due to UART communication, etc.

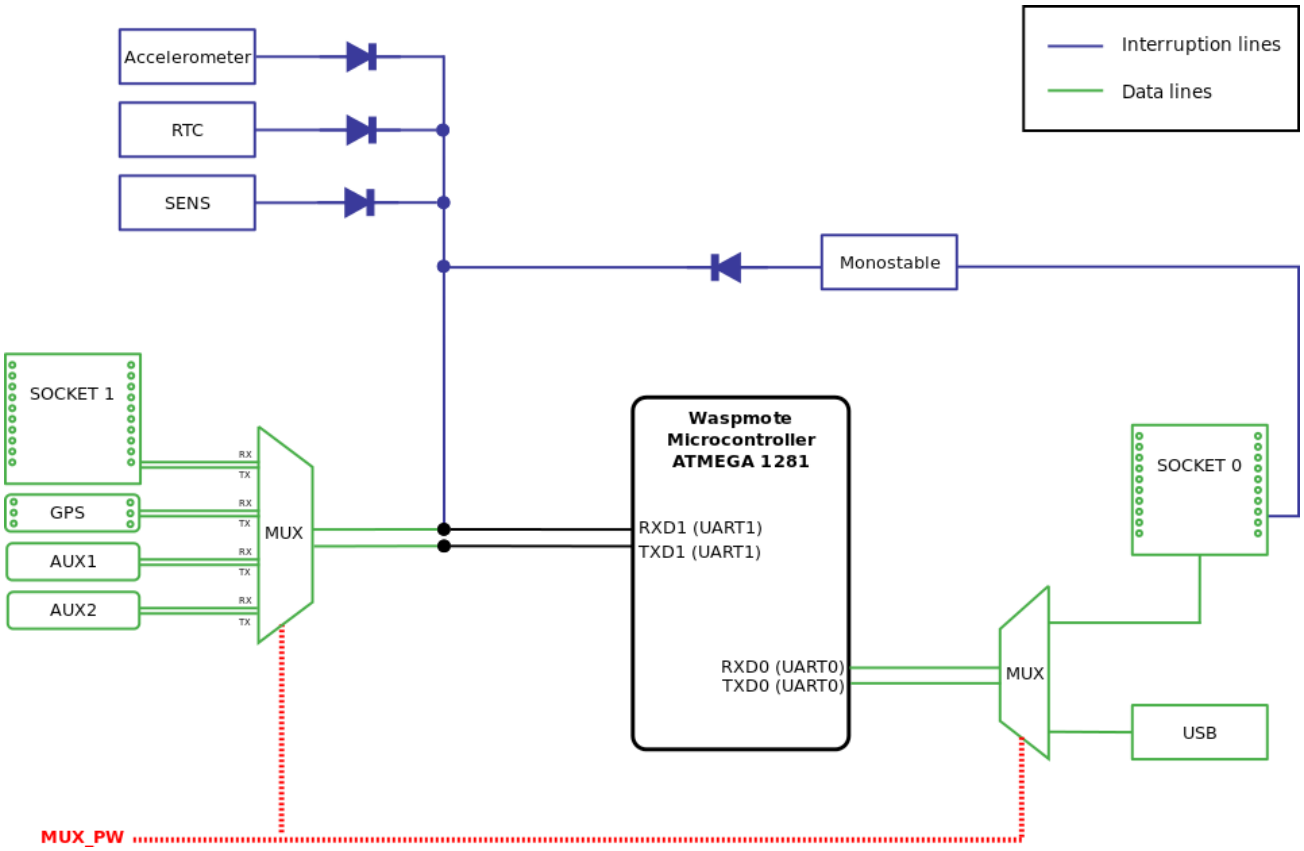


Figure 13: Interruption-UART1 interference lines

It is important to switch off the modules connected to UART1 through the multiplexer when setting up interruptions in order to avoid interferences. Anyway, it is possible to switch off both UARTs multiplexers by calling the following function:

```
{
   // switch off both UARTs multiplexers
   Utils.muxOFF();
}
```

8.2. Code Robustness

Sometimes, it is possible to lose some interruption events due to any reason. This is dangerous because it could happen that in the case the interruption event is lost, the interruption source keeps the interruption signal masking the UART and interruption lines. This does not permit to receive any other interruption.

The `PWR.clearInterruptionPin()` function permits to check each monitoring pin in order to know if there are active interruption signals. Besides, the function disables the interruption source solving the problem.

9. Code examples and extended information

In the Waspmote Development section you can find complete examples:

http://www.libelium.com/development/waspmote/examples

10. API changelog

Function / File	Changelog	Version
intFlag	Variable changes from uint32_t to uint16_t	V003→ v004
intConf	Variable changes from uint32_t to uint16_t	V003→ v004
intArray	Array changes size from 17 to 9	V003→ v004
UART1_OFF	Deprecated. UART1 is always switched off when entering a sleep mode	V003→ v004
UART0_OFF	Deprecated. UART0 is always switched off when entering a sleep mode	V003→ v004
RTC_OFF	Deprecated. RTC is always switched off when entering a sleep mode	V003→ v004
BAT_OFF	Deprecated. Battery level monitor circuit is always switched off when entering a sleep mode	V003→ v004
SOCKET0_OFF	New input argument for sleep modes to indicate to switch off the SOCKET0 power supply	V003→ v004
ALL_ON	New input argument for sleep modes to indicate no switch off is produced when entering to a sleep mode	V003→ v004
HAI_INT_PIN_MON	Deleted	V003→ v004
LAI_INT_PIN_MON	Deleted	V003→ v004
HAI_INT	Deleted definition and interruption management	V003→ v004
LAI_INT	Deleted definition and interruption management	V003→ v004
UART1_INT	Deleted definition and interruption management	V003→ v004
BAT_INT	Deleted	V003→ v004
TIM0_INT	Deleted	V003→ v004
TIM1_INT	Deleted	V003→ v004
TIM2_INT	Deleted	V003→ v004
PIN_INT	Deleted	V003→ v004
HAI_POS	Deleted	V003→ v004
LAI_POS	Deleted	V003→ v004
UART1_POS	Deleted	V003→ v004
BAT_POS	Deleted	V003→ v004
TIM0_POS	Deleted	V003→ v004
TIM1_POS	Deleted	V003→ v004
TIM2_POS	Deleted	V003→ v004
PIN_POS	Deleted	V003 → v004
WaspPWR::switchesOFF	Function changed. Now RTC, I2C and both UARTs are always switched off.	V003→ v004
WaspPWR::sleep	UARTs multiplexers are always swicthed off	V003→ v004
WaspPWR::deepSleep	UARTs multiplexers are always swicthed off. RTC alarm is disabled when exiting the function to ensure no RTC alarm arrives if deepSleep mode has been interrupted by any other interruption source	V003→ v004
WaspPWR::deepSleep	New function. Checks all interruption monitoring pins in RXD1 interruption so as to clean pending interruption sources if interruption was not captured.	V003→ v004
WaspPWR::printIntFlag	New Function. Prints the contents of intFlag indicating which interruption source has been marked in this flag	V003→ v004
WaspPWR::sleep	Function changed to ensure RXD1 pin is switched off before entering sleep mode	V002→ v003
WaspPWR::deepSleep	Function changed to ensure RXD1 pin is switched off before entering sleep mode	V002→ v003
onHAIwakeUP	Changed function. Added additional comparisons for XBee module and 3G interruptions. Deleted the check for anemometer interruptions.	V0.31 → v001

onLAIwakeUP	Changed function. Added new comparison for UART1 interruption. Deleted pluviometer comparison.	V0.31 → v001
enableInterrupts	Changed interruption attachment for UART1 interruption. Deleted anemometer attachment. Added XBee module and 3G interruption attachment.	V0.31 → v001
disableInterrupts	Deleted anemometer interruption detachment. Added XBee module and 3G module detachment.	V0.31 → v001
intCounter	Changed 'intCounter' to static variable	V0.31 → v001
intArray	Increased intArray to a 14-element array	V0.31 → v001

11. Documentation changelog

From v4.1 to v4.2

- Added a chapter about how to use interruptions in Waspmote
- Modified definitions of Interruption flags
- Deleted the interruption and documentation about GPRS interruption
- Deleted the interruption and documentation about 3G/GPRS interruption
- Deleted the interruption and documentation about Low Battery interruption
- Deleted chapter about other functions in the API
- Added new "Good Practises" chapter

From v4.0 to v4.1

- Added chapter "3G/GPRS".