# IoT in Five days

# Short introduction to Contiki

How to install (orientative, everything should be pre-installed)

- Know the Z1 mote: identify sensors, connectors, antenna.
- Check the installation: toolchain location, hello world example

My first application: hello world with LEDs

- Use the LEDs and printf to debug, use printf arguments.
- Change timer values, triggers.

Adding sensors: analogue and digitals

- Difference between both, basics.
- How to connect and read: ADC, I2C
- How to debug: enable modules printf macro, logic analyser (quick show, no hands-on for this)

# How to install

## Install VMWare for your platform

On Win and Linux: https://my.vmware.com/web/vmware/free#desktop_end_user_computing/vmware_player/6_0

On OSX you can download VMWare Fusion: http://www.vmware.com/products/fusion

## Download Instant Contiki:

Instant Contiki is an entire Contiki development environment in a single download. It is an Ubuntu Linux virtual machine that runs in VMWare player and has Contiki and all the development tools, compilers, and simulators used in Contiki development installed. http://www.contiki-os.org/start.html

## Start Instant Contiki

In VM, use 32 bit version, Instant_Contiki_Ubuntu_12.04_32-bit.vmdk Start Instant Contiki by running InstantContiki2.7.vmx. Wait for the virtual Ubuntu Linux boot up. Log into Instant Contiki. The password and user name is **user**. Don't upgrade right now.

## Update Contiki

### What is GIT

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. The main difference with previous change control tools like Subversion, is the possibility to work locally as your local copy is a repository, and you can commit to it and get all benefits of source control. Making branches and merging between branches is really easy.

To learn more about GIT there are some great tutorials online:

http://try.github.io http://excess.org/article/2008/07/ogre-git-tutorial/ A nice graphical introduction to git: http://rogerdudler.github.io/git-guide/

Contiki source code is maintained and hosted at Github https://github.com/contiki-os/contiki. GitHub is a GIT repository web-based hosting service, which offers all of the distributed revision control and source code management (SCM) functionality of Git as well as adding its own features. Unlike Git, which is strictly a command-line tool, GitHub provides a web-based graphical interface and desktop as well as mobile integration. It also provides access control and several collaboration features such as wikis, task management, and bug tracking and feature requests for every project.

The advantage of using GIT and hosting the code at github is allowing people to fork the code, develop on its own, and then contribute back and share your progress.

To update our code to the latest available contiki code, open a terminal and write:

```
cd $HOME/contiki
git fetch origin master
git pull origin master
git log to see the latest log
```

ADD GIT FOR EXAMPLES

## Check installation: toolchain location

Open a terminal and write:

```
msp430-gcc --version
```

Check if you have version 4.7. If not, download the latest toolchain version here: http://sourceforge.net/projects/zolertia/files/Toolchain/

## Check installation: examples

Open a terminal and write:

```
cd examples/hello-world
make TARGET=z1 savetarget
```

so it know that when you compile you do so for the z1 mote. You need to do this only once per application.

```
make hello-world
```

it starts compiling (ignore the warnings)

## Check z1 connection to the virtual machine

Connect the mote via USB.

In VM player: Player    Removable Devices    Signal Integrated Zolertia Z1    Connect

In VMWare Fusion: Devices    USB Devices    Silicon Labs Zolertia Z1

Open a terminal and write:

```
make z1-motelist
user@instant-contiki:~/contiki/examples/hello-world$ make z1-motelist
using saved target 'z1'
../../tools/z1/motelist-z1
Reference  Device       Description
---------- ---------------- ---------------------------------------------
Z1RC0336   /dev/ttyUSB0 Silicon Labs Zolertia Z1
```

Save the reference ID for next lab sessions (Z1RC0336). Each mote has a unique reference number. The port name is useful for programming and debugging.

Upload the preloaded Hello World application to the Z1 mote

Open a terminal and write:

```
make hello-world.upload  MOTES=/dev/ttyUSB0
```

if you don’t use the MOTE part, the system will install on the first device it finds. It is OK if you only have one device.

If you get this error:

```
serial.serialutil.SerialException: could not open port /dev/ttyUSB0: [Errno 13]
Permission denied: '/dev/ttyUSB0'
```

you need to add yourself to the dialout group of Linux. You do it this way:

```
sudo usermod -a -G dialout user
```

enter the root password, which is **user**

```
sudo reboot
```

password is **user**.

open terminal and go again to /contiki/examples/hello-world  and enter again

```
make hello-world.upload  MOTES=/dev/ttyUSB0
make z1-reset && make login
```

the first command resets the mote and the second one connects to the serial port and displays the result on the screen

Sceenshot missing

Note that the node ID is displayed.

# My first applications

## Hello world with LEDs

Let’s see the main components of the Hello World example. View the code with:

```
gedit hello-world.c
```

When starting Contiki, you declare processes with a name. In each code you can have more processes. You declare the process like this:

```
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);
```

① hello_world_process is the name of the process and "Hello world process" is the readable name of the process when you print it to the terminal.

② The AUTOSTART_PROCESSES(&hello_world_process); tells Contiki to start that process when it finishes booting.

```
/*---------------------------------------------------------------------------*/
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(hello_world_process, ev, data)
{
  PROCESS_BEGIN();
  printf("Hello, world\n");
  PROCESS_END();
}
```

① You declare the content of the process in the process thread. You have the name of the process and callback functions (event handler and data handler).

② Inside the thread you begin the process,

③ do what you want and

④ finally end the process.

The next step is adding a LED and the user button. Add picture.

Let’s create a new file. Go to: /home/user/contiki/examples/z1 with

```
cd /home/user/contiki/examples/z1
```

Let’s name the new file `test_led.c` with

```
gedit test_led.c.
```

You have to add the `dev/leds.h` which is the library to manage the LED lights. To check the definition go to `/home/user/contiki/core/dev`.

Available LEDs commands:

```
unsigned char leds_get(void);
void leds_set(unsigned char leds);
void leds_on(unsigned char leds);
void leds_off(unsigned char leds);
void leds_toggle(unsigned char leds);
Available LEDs:
LEDS_GREEN
LEDS_RED
LEDS_BLUE
LEDS_ALL
```

Now try to turn ON only the red LED and see what happens

```
#include "contiki.h"
#include "dev/leds.h"
#include <stdio.h>
//----------------------------------------------------------------
PROCESS(led_process, "led process");
AUTOSTART_PROCESSES(&led_process);
//----------------------------------------------------------------
PROCESS_THREAD(led_process, ev, data)
{
  PROCESS_BEGIN();
  leds_on(LEDS_RED);
  PROCESS_END();
}
```

We now need to add the project to the makefile. So edit `Makefile` and make sure you have:

```
CONTIKI_PROJECT = test-phidgets blink test-adxl345 test-tmp102 test-light-ziglet test-
battery test-sht11 test-relay-phidget test-tlc59116
CONTIKI_PROJECT += test_led
```

Now let’s compile and upload the new project with:

```
make clean && make test_led.upload MOTES=/dev/ttyUSB0
```

the make clean is used to erase previously compiled objects. Now the LED is red!

Missing pictures.

**TIP** Exercise: try to switch on the other LEDs.

## Printf

You can use prinf to visualize on the console what is happening in your application. It is really useful to debug your code, as you can print values of variables. Let’s try to print the status of the LED, using the `unsigned char leds_get(void);` function that is available in the documented functions (see above). Get the LED status and print its status on the screen.

```
#include "contiki.h"
#include "dev/leds.h"
#include <stdio.h>
char hello[] = "hello from the mote!";
//-----------------------------------------------------------------
PROCESS(led_process, "led process");
AUTOSTART_PROCESSES(&led_process);
//-----------------------------------------------------------------
PROCESS_THREAD(led_process, ev, data)
{
  PROCESS_BEGIN();
  leds_on(LEDS_RED);
  printf("%s\n", hello);
  printf("The LED %u is %u\n", LEDS_RED, leds_get());
  PROCESS_END();
}
```

If one LED is on, you will get the LED number (LEDs are numbered 1,2 and 4).

**TIP** Exercise: what happens when you turn on more than one LED? What number do you get?

## Button

We now want to detect if the user button (see picture) has been pushed.

Create a new file in `/home/user/contiki/examples/z1` called `test_button.c` The button in Contiki is considered as a sensor. We are going to use the `dev/button-sensor.h` library. It is a good process to give the process a meaningful name so it reflects what the process is about. Here is the code to print the button status:

```
#include "contiki.h"
#include "dev/leds.h"
#include "dev/button-sensor.h"
#include <stdio.h>
//-----------------------------------------------------------------
PROCESS(button_process, "button process");
AUTOSTART_PROCESSES(&button_process);
//-----------------------------------------------------------------
PROCESS_THREAD(button_process, ev, data)
{
  PROCESS_BEGIN();
  SENSORS_ACTIVATE(button_sensor);
  while(1) {
  PROCESS_WAIT_EVENT_UNTIL((ev==sensors_event) && (data == &button_sensor));
  printf("I pushed the button! \n");
  }
  PROCESS_END();
}
```

Let’s modify the `Makefile` to add the new file.

```
CONTIKI_PROJECT += test_button
```

You can leave the previously created test_led in the makefile. This process has an infinite loop (given by the wait()) to wait for the button the be pressed. The two conditions have to be met (event from a sensor and that event is the button being pressed) As soon as you press the button, you get the string printed.

**TIP** Exercise: switch on the LED when the button is pressed. Switch off the LED when the button is pressed again.

## Timers

Create a new file in `/home/user/contiki/examples/z1` called `test_timer.c`. You don’t need any new library as the timer is part of Contiki’s core. We create a timer structure and we set the timer to expire after a given number of seconds. Then when the timer is expired we execute the code and restart the timer. This is the basic type of timer. Contiki has three types of timers.

```
#include "contiki.h"
#include "dev/leds.h"
#include "dev/button-sensor.h"
#include <stdio.h>
#define SECONDS 2
//-----------------------------------------------------------------
PROCESS(hello_timer_process, "hello world with timer example");
AUTOSTART_PROCESSES(&hello_timer_process);
//-----------------------------------------------------------------
PROCESS_THREAD(hello_timer_process, ev, data)
{
  PROCESS_BEGIN();
  static struct etimer et;
  while(1) {
  etimer_set(&et, CLOCK_SECOND*SECONDS);
  PROCESS_WAIT_EVENT();
  if(etimer_expired(&et)) {
    printf("Hello world!\n");
    etimer_reset(&et);
  }
  }
  PROCESS_END();
}
```

① CLOCK_SECOND is a variable that relates to the microcontroller ticks. As Contiki runs on different platforms, the value of CLOCK_SECOND is different in different devices. This is related to the frequency of the processor. In Z1 it is 128.

② PROCESS_WAIT_EVENT(); waits for any event to happen.

**TIP** Excercise: can you print the value of CLOCK_SECOND to count how many ticks you have in one second? Try to blink the LED for a certain number of seconds. A new application that starts only when the button is pressed and when the button is pressed again it stops.

## Sensors

The Z1 has two built in digital sensors: temperature and 3 axis acceleration. The main difference between analog and digital sensors are the power consumption (lower in digital) and the protocol they use. Analog sensors require being connected to ADC (analog to digital converters) which translate the analog (continuous) reading to a digital value (in millivolts). The quality and resolution of the measure depends on both the ADC (resolution is 12 bits in the Z1) and on the sampling frequency. As a rule of thumb, you need to have double sampling frequency as the phenomena you are measuring. As an example, if you want to sample human sound (8 kHz) you need to sample at twice that frequency (16 kHz minimum).

### Analog Sensors

There is one analog sensor in the Z1, and it provides the battery level expressed in milliVolts. There is an example in the Contiki example folder called `test-battery.c`. The example includes the battery level driver (`battery-sensor.h`). It activates the sensor and prints as fast as possible (with no delay) the battery level. When working with the ADC you need to convert the ADC integers in milliVolts. This is done with the following formula:

```
float mv = (bateria * 2.500 * 2) / 4096;
```

We are powering the Z1 with 5 Volts (this is why we multiply by 5). We divide the raw value by 4096 as it is a 1 shifted to 12 positions on the left (as the precision of the ADC is 12 bits). The internal power of the Z1 is 3V. There is an internal voltage divider that converts from 5V to 3.3V. If you connect the Z1 to the USB port, you will always get the highest value (around 3V). Two value are printed by the code in two columns: the raw value from the ADC and the converted value in milliVolts.

## External analog sensor:

We can connect an external analog sensor. As an example, let's connect the precision light sensor. It is important to know the voltage required by each sensor. If the sensor can be powered at 3V, it should be connected to the phidgets connector in the top row. If the sensor is powered at 5V it can be safely connected to the phidgets bottom row. Only if the mote is powered by USB, then you can use the 5V sensor. Insert phidgets Insert picture. If you use the phidgets cable, there is a single way to connect the node. Insert datasheet of the light sensors.

You need to convert the values coming from a 5V sensors as there is an internal voltage divider.

There is an example called `test-phidgets.c`. This will read values from an analog sensor and print them to the terminal.  Connect the light sensor to the 3 V phidget connector. As this is an official example, there is no need to add it to the Makefile (it is already there!). Let's compile the example code:

```
make clean && make test-phidgets.upload MOTES=/dev/ttyUSB0
```

and connect to the node:

```
make z1-reset && make login
```

This is the result:

```
Starting 'Test Button & Phidgets'
Please press the User Button
Phidget 5V 1:123
Phidget 5V 2:301
Phidget 3V 1:1710
Phidget 3V 2:2202
```

The light sensor is connected to Phidget 3V 2, so the raw value is 2202. Try to illuminate the sensor with a flashlight (from your mobile phone, for example) and then to keep it in your hand so that no light can reach it. http://www.phidgets.com/products.php?product_id=1127_0

Sensor Properties Sensor Type Light Sensor Output Type Non-Ratiometric Response Time Max 20 ms Measurement Error Max ± 5 % Peak Sensitivity Wavelength 580 nm Light Level Min 1 lx Light Level Max (3.3V) 660 lx Light Level Max (5V) 1 klx

As you can see, the light sensor can be connected to both the 5 V and 3.3 V phidget connector. The max measurable value changes depending where you connect it. The formula to translate SensorValue into luminosity is: Luminosity(lux)=SensorValue

**TIP** Exercise: make the sensor take sensor readings as fast as possible. Print on the screen the ADC raw values and the millivolts (as this sensor is linear, the voltage corresponds to the luxes). What are the max and min values you can get? What is the average light value of the room? Create an application that turns the red LED on when it is dark. When it is light, turn the green LED on. In between, switch off all the LEDs. Add a timer and measure the light every 10 seconds.

## Internal digital sensor

The Z1 has an internal digital sensor: the 3 axis accelerometer. There is an example called `test-adxl345.c`. You don’t need to add it to the Makefile. Once uploaded, this is the result:

```
~~[37] DoubleTap detected! (0xE3) -- DoubleTap Tap
x: -1 y: 12 z: 223
~~[38] Tap detected! (0xC3) -- Tap
x: -2 y: 8 z: 220
x: 2 y: 4 z: 221
x: 3 y: 5 z: 221
x: 4 y: 5 z: 222
```

The accelerometer can give data in x,y and z axis and has three types of interrupts: when you do a single tap, when you do a double tap and when you let the sensor free-fall (pay attention not to damage the mote!). Try tapping once and twice. The code has two threads, one for the interruptions and the other for the LEDs. When Contiki starts, it triggers both the processes. The led_process thread triggers a timer that waits before turning off the LEDs. This is mostly done to filter the rapid signal coming from the accelerometer. The other process is the acceleration one. It assigns the callback for the led_off event. Interrupts can happen at any given time, are non periodic and totally asynchronous. They can be triggered by external sources (sensors, interrupt pins, etc) and should be cleared as soon as possible. When an interrupts happens, the interrupt handler (which is a process that checks the interrupt registers to find out which is the interrupt source) manages it and forwards it to the subscribed callback. In this example, I first start the accelerometer and then map the interrupts from the accelerometer to a specific callback function. Interrupt source 1 is mapped to the free fall callback handler and the tap interrupts are mapped to the interrupt source 2.

```
  /* Start and setup the accelerometer with default values, eg no interrupts enabled. */
  accm_init();
  /* Register the callback functions for each interrupt */
  ACCM_REGISTER_INT1_CB(accm_ff_cb);
  ACCM_REGISTER_INT2_CB(accm_tap_cb);
  /* Set what strikes the corresponding interrupts. Several interrupts per pin is
    possible. For the eight possible interrupts, see adxl345.h and adxl345 datasheet. */
```

We then need to enable the interrupts like this:

```
accm_set_irq(ADXL345_INT_FREEFALL, ADXL345_INT_TAP + ADXL345_INT_DOUBLETAP);
```

What happens in the while cycle is that we read the values from each axis every second. If there are no interrupts, this will be the only thing shown in the terminal.

**TIP** Exercise: put the mote in different positions and check the values of the accelerometer. Try to understand what is x, y and z. Measure the max acceleration by shaking the mote. Turn on and off the LED according to the acceleration on one axis. Image with axis

## External digital sensor

The ZIG001-mini is a digital temperature sensor. The advantage of using digital sensors is that you don’t have to do calibration of your own as they come factory-calibrated. They usually have a low power current consumption compared to their analog peers. They allow a more extended set of commands (turn on, turn off, configure interrupts). If you have a digital light sensor, you can set a threshold value when the sensor sends an interrupt, without the need for continuous polling. The example is available as `test-sht11.c`. The light digital sensor is also given as example in the same folder.

```
Temperature:   27 degrees Celsius
Rel. humidity: 67%
Temperature:   27 degrees Celsius
Rel. humidity: 66%
Temperature:   27 degrees Celsius
Rel. humidity: 65%
```

**TIP** Exercise: convert the temperature to Fahrenheit. Try to get the temperature and humidity as high as possible (without damaging the mote!). Try to print only “possible” values (if you disconnect the sensor, you should not print anything, or print an error message!).

## Sending Data to Ubidots:

What is Ubidots Get API key and variable ID

We can send data from the sensors to Ubidots to visualize and generate events. We need two software components to send data. We will modify an existing Z1 example from today. **We will send data from the temperature sensor to the serial port of the PC. The PC will then parse the data and send it to Ubidots using their API with a python script.** For the data to be parsed correctly it must follow a certain format (like tab separated, comma separated or others).

In this case we will use a Contiki application that handles the serial formatting to be sent to the python script. Apps are created to provide extra features that can be used directly by other applications. Apps are placed in the apps folder of Contiki. The Makefile of an APP has the following naming convention:

```
Makefile.serial-shell
```

And inside you must specify which are the source codes that will be used, in this case:

```
serial-ubidots_src = serial-ubidots.c
```

This is what the serial-ubidots serial will look like:

```
#include <string.h>
#include "serial-ubidots.h"
void
send_to_ubidots(const char *api, const char *id, uint16_t val)
{
  unsigned char buf[6];
  snprintf(buf, 6, "%d", val);
  printf("\n\r%s\t", api);
  printf("%s\t", id);
  printf("%s\n\r", buf);
}
```

You need to declare a header file `serial-ubidots.h` as well:

```
#define VAR_LEN 24
#define UBIDOTS_MSG_LEN (VAR_LEN + 2)
struct ubidots_msg_t {
  char var_key[VAR_LEN];
  uint8_t value[2];
};
void send_to_ubidots(const char *api, const char *id, uint16_t val);
```

We have also created a data structure which will simplify sending this data over a wireless link, we will talk about this a bit later.

Now that we have created this APP, we should add it to our example code (that sends temperature to Ubidots), the proper way is to edit the Makefile we have already know at `examples/z1` and add serial-ubidots to the APPS argument:

```
APPS = serial-shell serial-ubidots
```

And now let's edit the `test-tmp102.c` example to include the serial-ubidots application, first add the serial-ubidots header as follows:

```
#include "serial-ubidots.h"
```

Then we should create 2 new constants with the API key and Variable ID, obtained at Ubidots site as follows:

```
static const char api_key[] = "fd6c3eb63433221e0a6840633edb21f9ec398d6a";
static const char var_key[] = "545a202b76254223b5ffa65f";
```

It is a general good practice to declare constants values with as "const", this will save some valuable bytes for the RAM memory :) Change the polling interval to avoid flooding Ubidots and kicking us out :)

```
#define TMP102_READ_INTERVAL (CLOCK_SECOND * 15)
```

Then we are ready to send our data to Ubidots, first change the call to the tmp102 sensor to have the value with 2 digits precision, and send it over to Ubidots, replace as follows:

```
PRINTFDEBUG("Reading Temp...\n");
raw = tmp102_read_temp_x100();
send_to_ubidots(api_key, var_key, raw);
```

Upload the code to the Z1:

```
make MOTES=/dev/ttyUSB0 test-tmp102.upload && make MOTES=/dev/ttyUSB0 z1-reset && make
MOTES=/dev/ttyUSB0 login
```

This is what you will see on the screen:

```
fd6c3eb63433221e0a6840633edb21f9ec398d6a
545a202b76254223b5ffa65f    2718
```

Notice that you must divide by 100 to get the 27.18ºC degree value, this can be done easily on Ubidots.

Ubidots Python API Client

The Ubidots Python API Client makes calls to the Ubidots Api. The module is available on PyPI as "ubidots". To follow this quickstart you'll need to have python 2.7 in your machine (be it a computer or an python-capable device), which you can download at http://www.python.org/download/.

You can install pip in Linux and Mac using this command:

```
$ sudo easy_install pip
```

Installing the Python library Ubidots for python is available in PyPI and you can install it from the command line:

```
$ sudo pip install ubidots==1.6.1
```

The python script on the PC is called `UbidotsPython.py` and is located in the `tools/z1` directory. The script parses the serial data and sends it to Ubidots. to execute it, run:

```
user@instant-contiki:~/contiki/tools/z1$ python UbidotsPython.py -p /dev/ttyUSB0
```

In which -p is the argument to tell the Python script to connect to our mote at the given port. If you connect to Ubidots, you will immediately see the values coming in. To enable printing debug information from the Python script to the console, enable the following value at the `UbidotsPython.py` file:

```
# Enable to print extended information
DEBUG_APP = 1
```

The data is sent to Ubidots as long as the pyhton script is running. You can have it working in the background by adding `&` at the end of the script call. While the python script is running, you cannot program the node! As the temperature sensor is located next to the USB connector, it tends to heat up. A realistic value is few degrees lower than the measured on. To get more reliable temperature measurements while connected to the USB, use an external temperature sensor! Don't forget that Ubidots will not accept more than one measurement every 15 seconds. If you send data more frequently, you will lose the connection.

To divide the incoming data by 100, you should name it as derived variable as follows: create a temperature variable with the raw data and then the derived variable by dividing the temperature variable by 100.

In Ubidots your data will show up as follows: