

Pawel Gburzynski

# The SHOCK praxis

# (preliminary document)

**Version 0.1 (CC1350)**



August 5, 2021

© Copyright 2013-2021, Olsonet Communications Corporation.

All Rights Reserved.

# Preamble

The role of this praxis[[1]](#footnote-1) is to demonstrate the operation of the set of sensors onboard the CC1350 SensorTag (aka CC1350STK) and provide a generic tool for collecting sensor data from the device without having to connect it over a wire to a PC. It is intended for experiments where possibly large (relatively speaking) sets of data must be collected and stored for off-line analysis and interpretation.

After we have come up with some decent (tentative) GUI, I am going to upgrade the praxis with mechanisms facilitating high-speed data acquisition from the IMU (accelerometer) possibly involving compression and some kind of acknowledgments. The idea is to be able to collect IMU data with high reliability at the highest possible rate. While data from the remaining sensors may also be of interest, IMU is the primary concern.

This document is intentionally sketchy. Detailed information regarding the PCBs, the sensors, the PicOS platform, the RF interface, is contained in other documents by Texas Instruments and/or Olsonet Communications. Those documents are publicly available and can be easily obtained, e.g., from me.

# The devices

The setup involves two devices: a CC1350STK (see Figure 1) called the Tag in the sequel, and a LAUNCHXL-CC1350 (Figure 2) called the Peg and acting as the wireless interface between the Tag and the PC. The Tag is powered from a battery and can be freely moved around within the Tag-Peg transmission range, which is of order 100 m in open space. The wireless communication channel is neither BT nor WiFi. It is built around the so-called proprietary mode of CC1350 and operates within the 916 MHz ISM band.

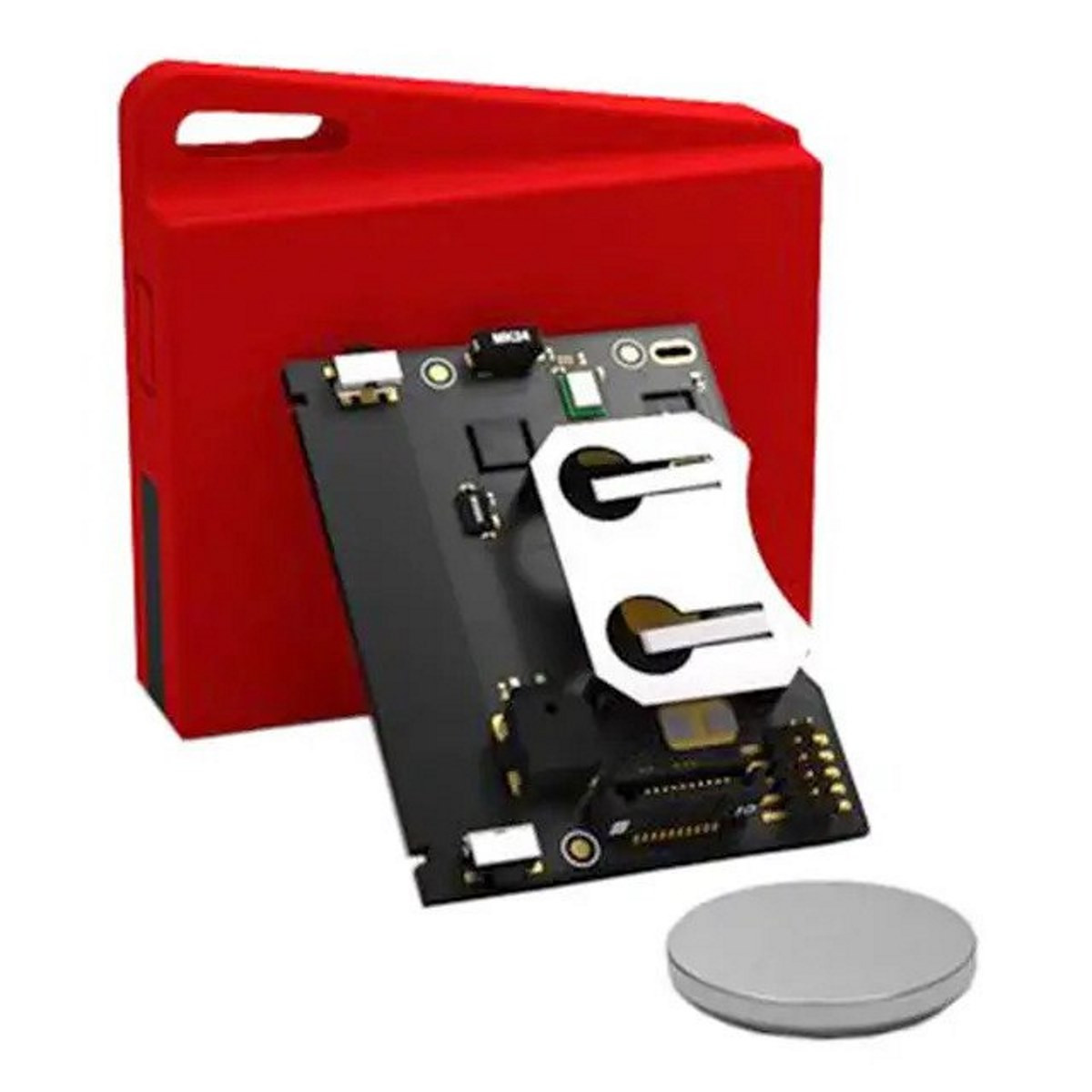


Figure 1: CC1350STK

The Tag is typically stored in a dormant state in which it uses practically zero battery power. There is no OFF switch, as such, on the device; however, the deepest sleep mode of the microcontroller effectively amounts to an OFF state because the battery drain is then negligible. To wake up the device from the dormant state, Switch 1 (see Figure 3) should be pressed once. If the device was in fact dormant, the device will reset, and the LED will blink twice.

The device will start in its initial power-up state. Thus, the return from the dormant state looks like switching the device on. Note that powering the device up (say by inserting the battery or connecting it via the USB debug interface) will cause it to go immediately dormant: pushing Switch 1 is necessary to bring it to life after that. This way Switch 1 does its best to emulate the power switch (as one of its multiple functions).

Pressing Switch 1 while the device is not dormant will cycle it through the three states of the RF module: off, WOR, and full-on. Following the push, the LED will blink quickly: 2 times when the new assumed state is off, 4 times for WOR, and 6 times for full-on. When the device starts (after a wakeup from dormant), the initial state of the RF module is full-on, so the Tag can immediately receive commands from the Peg over the radio channel.



Figure 2: LAUNCHXL-CC1350

The WOR (Wake-On-Radio) state is a low-power state where the Tag is still able to respond to RF signals from the Peg. It consists in brief periods of receiver attention separated by longish periods of idleness. The Peg must send a special signal (a packet preceded by a very long preamble) to trigger a successful reception by the Tag.

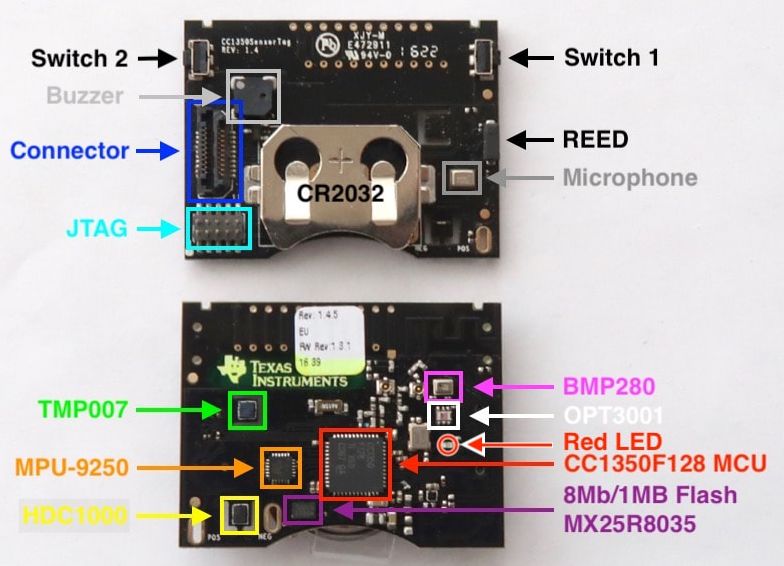


Figure 3: Tag layout

If, while the Tag is alive, Switch 1 is pressed continuously for more than 5 seconds, the LED will start to blink very quickly. It will blink 64 times and then the device will enter the dormant state. You should release the switch when the LED starts blinking. Otherwise, if the switch is still pressed when the LED is done blinking, the device will reset.

Switch 2 is not used at present. Switch 1 is the only means to control the device manually (without connecting to it from the Peg over the RF link). Note that the RF mode can be changed remotely by a command from the Peg, and the device can also be put into the dormant state this way. The device responds to such a request in a manner slightly different from its response to a push on Switch 1. The actual action is postponed for 1 second, so the device can still exchange packets with the Peg and (in particular) acknowledge the command. Then, if the receiver is being switched off, the LED will blink 16 times. If the Peg requests the Tag to go dormant, the LED will blink 64 times, exactly as for a long push on Switch 1.

The Tag uses most power when sending data over the RF channel. With the transmitter continuously turned on, the battery will last for about 10 hours. With the receiver fully on (but no transmissions), the time extends to 13-15 hours. In the WOR mode, the battery will last for a few months (probably around 5).[[2]](#footnote-2)

With the RF module turned off, the device operates in a low-power state (even if it is not formally dormant) using very little battery power. However, there is a significant conceptual difference between that state and the dormant state. In the dormant state, the device is completely inactive: it is only tuned to a push of Switch 1, to reset when the switch is pressed. In the idle state, the device can respond to events (like sensor events), so it can be set to react (and send packets over RF) when something happens. In such a case, the RF module is only turned on briefly for the transmission, and the receiver never goes on, so the Tag cannot receive commands from the Peg. Battery drain during such operation depends on the report frequency, the configuration of sensors that are turned on, their parameters, etc., but can be very low.

# Tag sensors and operation

The Tag is equipped with 5 sensors:

* MPU9250: accelerometer, gyro, compass, temperature combo
* HDC100: humidity and temperature combo
* SPH0641LM4H-1: one bit microphone
* OPT3001: ambient light sensor
* BMP280: air pressure and temperature combo

Technical details are provided in separate documents.[[3]](#footnote-3) Here I only include rudimentary description needed to understand the functionality of the application at hand. From now on, the sensors will be referred to by their selectors, i.e., names used in the application, which are, respectively: IMU, HUMIDITY, MICROPHONE, LIGHT, PRESSURE.

Three sensors, IMU, HUMIDITY, and PRESSURE, combine multiple functions (they consists of multiple, selectable, semi-independent components). IMU is the most prominent (most complex) representative of this group. Note that there are three different temperature components.[[4]](#footnote-4) Not a big surprise: many sensors provide temperature components on the side, so this is expected when you have a bunch of them.

The sensors can be independently configured. Also, for the three sensors consisting of multiple components, the components can be selected independently. Then, the sensors can be independently turned on and off. The operations of configuring a sensor and turning it on and off are separated.

The present application is a bit messy because the program currently running in the Tag caters to numerous tests and implements miscellaneous hooks that I needed to try out various things. No problem. It can all be cleaned up if we know what we want. If any *actual* functionality is expected from the thing, it is easier to remove and trim than to create and refine.

## Sampling and streaming

There are two ways to collect sensor data which I call *sampling* and *streaming*. Sampling is available for all sensors and for combinations of their selections at once meaning that groups of data coming from different sensors can be sampled simultaneously as compound samples). Streaming is presently only available for the accelerometer component of the IMU. It allows for high-reliability, timed (synchronous) data collection from the accelerometer and is intended for research in canine behavior. Streaming looks like a clumsy extension of sampling that went a bit sideways. But here it is.

The sensors can be configured and turned on independently. I mean that usually a sensor is configured before being turned on, and it cannot be configured while being turned on, so it must be turned off, reconfigured, and turned on again. There is a streaming shortcut where the IMU (the only sensor that can be streamed) is automatically configured and turned on, and then the sampling starts.

For sampling, you go through these steps:

1. configuring the sensors,
2. turning on those sensors that you want to deliver data,
3. starting sample collection,

Then, all the sensors that have been turned on will be sampled according to their configuration.

The command to start sampling accepts the sampling rate as the single argument. The amount of data arriving in a single sample depends on the set of sensors that are turned on and the selection of their components. For example, each of the three main components of IMU, i.e., accelerometer, gyro, and compass, sends triplets of 16-bit values interpreted as vectors in 3-space, while the temperature component sends a single 16-bit value. Thus, if all four components of IMU are selected, the data sent by the sensor amounts to bytes. If multiple sensors are on, e.g., the IMU (configured with its all components active) and the LIGHT sensor, then the data sent in a single sample will amount to bytes (the additional four bytes provided by the LIGHT sensor). The data arriving in a sample are tagged with their sensor identifiers, so they can be identified without the configuration knowledge.

The sampling rate is provided as the number of samples per minute. It is not particularly precise (the strobing clock is not extremely accurate), although the Tag tries to adjust it to maintain a consistent long-term rate (so the long-term rate will tend to converge to the specified rate). The maximum rate that can be legitimately specified is 15360 samples per minute, i.e., 256 samples per second, with the effective reachable rate about 160 samples per second. The minimum rate is 1 sample per minute.

## IMU modes

The whole thing has been inspired by the IMU sensor (which is the target of our efforts). The remaining sensors can be read (because they are available) but we see no immediate use for them.

The IMU can operate in two (or three, depending how you count) modes. In the passive mode, the sensor only returns the readings of its selected components when explicitly polled for the next sample, according to the schedule determined by the sampling command in effect. This is also how all the remaining sensors *always* behave, if they are on during sample collection.

Another mode of the IMU is the *motion detection* mode. When the sensor is configured this way, the accelerometer becomes its only active component, and the sensor triggers events on a configurable acceleration threshold. Configuring the IMU in this mode automatically selects the accelerometer component and disables the remaining components. The data sent by the sensor consists of the acceleration vector ( bytes) augmented by two more bytes ( bytes total). The extra two bytes amount to a 16-bit unsigned integer value returning the number of motion events triggered from the last readout.

An additional option available in the motion detection mode is the *report option*. When the option is off, the motion data is only collected via scheduled sampling, as for any other (passive) sensor that can be sampled. If the report option is on (note that the option is only effective in the motion detection mode), a motion event will automatically trigger a spontaneous sample report sent to the Peg. That sample arrives out of band and contains the IMU data only (8 bytes, as explained above), even if other sensors are also active (their values being collected and included in regular samples).

The third mode of the IMU sensor is used in streaming (you may call it the *synchronous* data extraction mode). In this mode, the sensor triggers events at prescribed regular intervals when new data becomes ready for acquisition. The mode provides the Tag with precise strobes driving the streamed samples, so the collection rate can be accurate. The available rates start at samples per second and go down to samples per seconds. They are internally determined by a *divider* of the base Hz rate whereby a value of between and is used to indicate the number of ticks to be skipped. For example, when , the rate is . Similar to the motion detection mode, the only active component of the IMU sensor in the streaming mode is the accelerometer. The values arriving from the sensor are the three coordinates of the acceleration vector reduced to 10 bits each and sent in packages of 12 readings. The packages (aka blocks) are sent using a special communication protocol that (as long as possible) tries to compensate for occasional packet losses in the radio channel with retransmissions. How far we can go with effective streaming rates is to be determined (we have to experiment with the different RF rates and conditions, as well as the parameters of the streaming protocol), but something like (reliable) samples per seconds appears to be easily achievable without trying too hard.

## HUMIDITY

The sensor comes with two components: the *actual* humidity sensor and a temperature sensor. Either component returns two bytes of data, i.e., a 16-bit value. Once turned on, the sensor is internally sampled by the Tag at the interval specified as one of the configuration parameters (even if no scheduled sampling is in progress). The same approach is followed for PRESSURE and LIGHT. The official (scheduled) sample returns the sensor’s last internally sampled value. The role of this internal (background) pre-sampling is to:

1. Prevent the delays in physical sampling (readouts) of the sensor, which may be significant for slow sensors (whose values are not meant to change very fast), from impacting the scheduled sampling. Generally, it is assumed that scheduled sampling is delay-free.
2. Decouple the readouts of slow sensors from scheduled sampling, such that those readouts can be made formally available at high rates (at which one may want to sample other sensors) without overtaxing the slow sensors.

## PRESSURE

Like HUMIDITY, the sensor consists of two components: air pressure and temperature. Each component returns a 4-byte (32-bit) value.

## LIGHT

The sensor has a single component and returns two 16-byte values: light intensity and status.

## MICROPHONE

The sensor is a standard one-bit microphone which I have rather crudely transformed into a counter accumulating the imbalance between zeros and ones in consecutive 8-tuples of bits (bytes) acquired from the microphone. This imbalance can be interpreted as a measure of acoustic noise in the neighborhood. The value returned by the sensor consists of two unsigned 32-bit numbers (8 bytes total). The first value returns the accumulated imbalance counted from the previous readout (sample), the other returns the total number of bits received from the sensor since the previous readout was made. The imbalance should be interpreted relative to the total bit count.

The present driver for the MICROPHONE sensor should be treated as a stub opening the sensor for potentially interesting applications, like barking recognition, to be implemented later.

# Setting things up

After you insert a battery into the Tag, the LED will blink twice, and the device will hibernate (to prevent draining the battery when idle). When you push Switch 1, the Tag will wake up in the WOR mode.

start in the normal up state with the radio turned on. By pushing Switch 1 you can make sure that the device is in the desired state. If you do not want to use the device immediately, you should put into the dormant state.



Figure 4. CC1350STK with attached DevPack board

For GUI development and local test, the Tag can be powered from a USB cable through the DevPack interface (see Figure 4). This is a small board about the same size as the Tag PCB that can be connected (piggybacked) *onto* the Tag device to provide access to the Tag for programming and debugging. When powered through DevPack, the Tag behaves in the same way as when powered from the battery.

The Peg must be connected to a PC to be operable. The USB interface of the Peg appears as two serial ports on the PC. Under Windows, these are COM ports; under Linux (say Ubuntu or Fedora), the devices are named ttyUSBACMx where x is a digit. One of those ports is for programming and debugging, the other maps to the UART of the Peg’s microcontroller. This is the port that the program talking to the Peg is going to use to communicate with the device.

Both the Tag and the Peg have been programmed in PicOS. The present command-line interface to the Peg amounts to a Tcl script and consists of two parts. The first (main) part is the generic OSS module whose intention is to provide a framework for interfacing OSS (Operational Support System) programs to PicOS master nodes. The second part extends the generic module by defining the specific set of commands and messages for the application. For the project at hand, the two parts come together in two files: ossrun.tcl (this is the generic part) and ossi.tcl (the specific part) that must be present the same directory.[[5]](#footnote-5) The whole thing is run by executing:

./ossrun.tcl

The “main” script will find the specific part and “evaluate” it in the proper context. The script can run on Windows (e.g., under ActiveState Tcl/Tk) as well as on Linux (e.g., Ubuntu) using the standard Tcl/Tk packages.

# Talking to the devices

When invoked properly, the script will produce a window looking as shown in Figure 5 (this is how it looks on Ubuntu). The script must connect to the Peg before you can start conversing with the setup. When you press “Connect” (at the bottom), the program will try to identify the serial device into which the Peg’s UART has been mapped.



Figure 5. The OSS window

The script accomplishes that by polling all serial USB devices for a known response code. When it finds the right device, the window’s title will change to let you know that the script is now connected to the Peg.

You converse with the Peg (and thus with the Tag) by entering commands into the field in the left bottom area of the window. The command syntax is described below. It can also be inferred from the comments and declarations at the beginning of ossi.tcl.

# Command syntax

A command starts with a keyword which can be followed by *selectors* and *parameters*. A selector is simply a (single) keyword that selects something, e.g., a sensor. A parameter is a pair: –name value. The name of a parameter is preceded by – (the minus sign). The value of a parameter can be a character string or a number.

Here is a sample command:

configure imu –accuracy high –components at

The piece of string following the command keyword is a *selector*: it selects the sensor (IMU) to which the subsequent parameters apply. The value of the –accuracy parameter is a keyword representing a *magnitude* setting. The value of –components is the string “at” selecting two components of the sensor: the accelerometer and the thermometer.

A keyword can be abbreviated as long as it cannot be confused with another keyword that might legally appear in its place. The above command can be shortened to:

c i –a hi –c at

The reason why “high” cannot be reduced to “h” is that “huge” is also an option for the magnitude setting. Note that “at” is not a keyword but a string of letters indicating the sensor components.

Practically all parameters are optional. When an optional parameter is not specified, it means either “default” or “unaffected”, i.e., retaining the last set value.

# Command interpretation

Most commands issued to the Peg (in fact *all* commands except “ap”, see below) are transformed into commands to the Tag and expedited by the Peg over the RF channel. The packets exchanged between the Peg and the Tag identify a specific Tag node addressing the Tag via its so-called Node Id which is simply a 16-bit integer device identifier in the PicOS network. The Node Id defaults to 1 (this is how the devices have been configured), but it can be changed with the “ap” command (you may have multiple Tags with different Node Ids).

Normally, when the Tag’s receiver is fully on, the commands are sent as standard (short) packets. This should be the normal mode of operation, i.e., for any serious exchange of data between the Peg and the Tag, you should make sure that the Tag’s receiver is fully on. Accounting for the limited reliability of the RF channel, the Peg can be set to repeat every command a prescribed number of times (see below). This costs little and there is no risk that the same command will be interpreted multiple times by the Tag, because the commands are numbered. When you expect the Tag to be in the WOR state, you can set the Peg to precede a command by a long waking packet, but (typically) your first objective in such case will be to bring the Tag’s receiver to full-on (by executing a radio command, see below). You should not operate with the Peg in a sustained WOR-wakeup setting because the packets sent to the Tag are then extremely long and slow.

The Tag is expected to respond to every command, even one that requests it to become dormant. If the command, as seen by the Peg, succeeds (meaning the Tag has responded indicating success and the Peg has received the response), you should see the text “<OK>” in the OSS windows. The lack of (any) response usually means that the Tag can’t be reached or is irresponsive (because it is off, dormant, its receiver is off, or the receiver is WOR and the command was sent as a standard packet [expecting the receiver to be full on]). Other responses are possible (they will always show up as pieces of text encapsulated in < … >) to indicate errors or problems.

The traffic between the Tag and the Peg involves commands (sent by the Peg to the Tag), responses (ACK codes sent by the Tag to the Peg in reply to commands), and messages sent by the Tag to the Peg, e.g., containing sampled data from the sensors. All messages arriving from the Tag are displayed in the OSS window in a formatted (and legible) fashion. Note that the information appearing in the window can be directed to a file (so the samples arriving from the Peg can be preserved). This is what the “Save” button is for. When the “All” box is additionally checked, the saved data also includes the commands typed in the box at the bottom of the OSS window. The “Hdr” button is irrelevant for now.[[6]](#footnote-6)

There are two special (and sometimes useful) features of the command input interpreter. The exclamation sign (!) entered as a complete command acts as a shortcut and has the effect of re-entering the previous (last-typed) command. If the string entered as a command starts with a colon (:), the remaining portion of the string will be executed as a Tcl script within the context of the interpreter (this may be useful for debugging).

# Command: ap

This command (“ap” stands for “access point”) is the only command addressed exclusively to the Peg. Its syntax is:

ap –node *nn* –wake *nw* –retries *nr* –preamble *pl*

All parameters are optional, and their values are nonnegative numbers. Their meaning is as follows:

|  |  |
| --- | --- |
| node | Specifies the Node Id of the Tag (as explained above). The default Node Id (if the parameter is never specified) is 1. The setup may involve multiple Tags, and the parameter can be used to switch among the multiple Tags serviced by the same Peg. |
| wake | The number of waking packets (furnished with long preambles) to precede a command sent to the Tag to wake it up from WOR. Zero means no wake up (and should be used if the Tag’s receiver is fully on). When this parameter is nonzero (usually never more than 1), all commands sent to the Tag are preceded by waking packets (usually a single waking packet). |
| retries | The number of retries for a command send to the Tag. Zero means a single attempt. |
| preamble | The length in PicOS milliseconds[[7]](#footnote-7) of the preamble preceding a wakeup) packet. The default length is 1024 (which translates into exactly one second). This is approximately the length (duration) of the wakeup packet because the preamble is practically all there is. This length should equal (or slightly exceed) the WOR cycle length at the Tag (settable with the “radio” command, as explained below). |

If the command is issued without arguments, it polls the Tag for the current setting of the above parameters. They are sent by the Tag in a special packet and presented in the OSS window in a formatted way.

# Command: configure

The command configures a sensor (or several sensors) at the Tag. The list of arguments consists of a sensor selector (one of: imu, humidity, microphone, light, pressure) followed by the list of parameters specific to the sensor. Any sensor selector can be abbreviated to a single letter. A single “configure” command can apply to multiple sensors. The list of parameters of a sensor can be followed by another sensor selector (which can never be confused with a parameter) followed in turn by the list of parameters for the sensor. For example, this command:

conf imu -motion yes -report yes -thresh low humid -heater yes -com h

configures IMU and HUMIDITY in a single go.

A sensor parameter can be of one of three types:

|  |  |
| --- | --- |
| Boolean | Such a parameter is a piece of text looking like “yes” or “no” (“y” or “n” will do), or a number with 0 standing for “no” and any nonzero value interpreted as “yes”. |
| magnitude | A parameter like this describes a setting that can range from some minimum to some maximum, in a few discrete steps, which we uniformly categorize into eight discrete values represented by these keywords: “tiny”, “low”, “small”, “medium”, “big”, “high”, “huge”, “extreme”. The numerical interpretation of these values depends on the actual range of the respective physical parameter of the sensor. The eight keywords provide a general, canonical view for all settings in this class. |
| component | A parameter of this type is used to select sensor components and is applicable to the three sensors equipped with multiple components: IMU, HUMIDITY, and PRESSURE. The parameter consists of a sequence of letters indicating which components should be selected. The letters can be: a = accelerometer, c = compass, g = gyro, h = humidity, p = pressure t = temperature. For example, “agt” selects the accelerometer, gyro, and temperature components of the IMU sensor. |

Below we list the sets of parameters per sensor. The sampling option, applicable to HUMIDITY, LIGHT, and PRESSURE, refers to the background sampling frequency of the three (slow) sensors (as described earlier). The background sampling frequency is between once per 8 seconds (tiny) to 16 times per second (extreme).

IMU

|  |  |  |
| --- | --- | --- |
| **name** | **type** | **description** |
| motion | b | Selects or deselects motion detection. If set, motion events will be counted or reported. |
| threshold | m | Sets the threshold for motion detections. |
| rate | m | Selects the sampling rate. |
| accuracy | m | Resolution of sensor readings. |
| bandwidth | m | Bandwidth for the pass filter. |
| components | c | Selects the active components of the sensor: a, g, c, t. |
| report | b | Selects the report option for motion events. If selected, motion events will be triggering immediate report packets sent to the Peg (as described earlier). The option is ignored if the motion detection mode has not been set. |

Defaults: no, big, big, huge, medium, a, no

HUMIDITY

|  |  |  |
| --- | --- | --- |
| **name** | **type** | **description** |
| heater | b | Turns on or off the internal heater to eliminate condensation. |
| accuracy | m | Sets the resolution of the result. |
| components | c | Selects the active components: h, t. |
| sampling | m | The background sampling rate for the sensor. |

Defaults: no, extreme, h, big

MICROPHONE

|  |  |  |
| --- | --- | --- |
| **name** | **type** | **description** |
| rate | m | The frequency of bit sampling from 100 kHz (tiny) to 2.475 MHz (extreme). |

Defaults: big (corresponding to 1.5 MHz)

LIGHT

|  |  |  |
| --- | --- | --- |
| **name** | **type** | **description** |
| continuous | b | Selects the continuous mode for internal operation of the sensor, as opposed to the default single-shot mode. |
| accuracy | m | Selects the accuracy of the sensor. There are basically two values <= medium and > medium translating into two internal sampling times. |
| sampling | m | The background sampling rate. |

Defaults: no, medium, big

PRESSURE

|  |  |  |
| --- | --- | --- |
| **name** | **type** | **description** |
| forced | b | Selects the forced mode for internal operation of the sensor where the sensor only responds to direct readouts. In normal mode, the sensor constantly evaluates the pressure, its response is faster and more accurate. Parameters rate and bandwidth (below) only apply if forced is not set. |
| rate | m | Selects the internal sampling rate for the sensor (the reciprocal of the standby interval). |
| accuracy | m | Selects the accuracy of the result determined by the internal oversampling rate. |
| bandwidth | m | Selects the filtering rate used to smooth out the results based on previous values. |
| sampling | m | The background sampling rate. |

Defaults: yes, extreme, big, extreme, low, big

# Command: radio

This command is used to change the status of the RF module at the Tag. The syntax is:

radio *mode* or

radio wor *offdelay worinterval*

In the first case, mode can be on, off, or hibernate (with admissible abbreviations). In the second case, two optional arguments can define (or redefine) two parameters of the WOR mode:

1. The duration of the RF inactivity period after which the Tag will resume the WOR mode following a packet reception (offdelay). When the device receives a packet in the WOR mode, it will temporarily switch to the full reception mode in anticipation of more traffic, even if it formally remains in the WOR mode. If the traffic does not materialize within the specified interval (and the device is not formally switched to full reception), the Tag will return to WOR. The interval is specified in PicOS milliseconds with the default of 5120 (5 seconds).
2. The length of one WOR cycle, i.e., the period of checking for an activity in the RF channel specified in PicOS milliseconds. The default is 1024, i.e., one second. This parameter should match the preamble length used by the Peg in its wakeup packets.

An unspecified argument retains its last setting.

# Command: on

The command turns the indicated sensors on. The syntax is:

on *sen sen … sen*

where the arguments are sensor selectors. Initially all sensors are off. Note that turning the radio off (or changing its mode) does not affect the sensor status.

# Command: off

The command turns the indicated sensors off. The syntax is:

off *sen sen … sen*

where the arguments are sensor selectors.

# Command: status

The command polls the Tag for its status. It takes no argument, so the syntax is just:

status

The message arriving from the Tag in response to the “status” command returns this information (which is shown in the OSS window with legible headers):

1. The up time of the Tag in seconds.
2. The current (wall-clock) time as reported by the Tag. The Tag’s clock is set to the proper (wall) time with the sample command (see below). If that has happened, the reported time should match the proper time.
3. The battery voltage (it should be around 3V).
4. Memory usage (for the heap): current free and minimum free seen so far (probably not very relevant).
5. Active sensors, i.e., the ones that are currently on.
6. Whether the Tag is sampling (collecting), i.e., the Tag is running the last sampling command received from the Peg. Two numbers indicate the rate (in samples per minute) and the number of samples remaining to collect. If the second number is zero, it means that no samples are being collected at the time.

This information is followed by the report on the sensor configuration. For every sensor, regardless of its on/off status, the sensor’s current configuration settings are shown.

# Command: sample

The command instructs the Tag to start sampling the sensors that are currently on. The syntax is:

sample –frequency *freq* –count *cnt*

The first parameter specifies the sampling frequency in samples per minute, the second indicates how many samples should be taken. If –frequency is not specified, it defaults to 60, i.e., 1 sample per second. If –count is not specified, it defaults to 1. Thus, when the command is issued without arguments, it will ask the Tag for a single sample reporting the readouts of all sensors that are on now.

# Command: stop

The command stops sampling in progress. It takes no arguments, so the syntax is just:

stop

# Sample reports

Having received a sample command, the Tag will respond with the requested number of samples taken at the prescribed intervals. A sample arrives at the Peg as an RF packet whose size depends on the active sensors and their selected components. The data is presented in a formatted fashion, by sensor, in a way that should be immediately legible. A spontaneous report arriving from the IMU sensor operating in the motion detection mode is shown in the same manner, except that the data refers to a single sensor only.

1. For simplicity, we may assume that this term means “application” in PicOS parlance, which is particularly true in this case. [↑](#footnote-ref-1)
2. Better measurements are needed to estimate this. [↑](#footnote-ref-2)
3. As mentioned earlier, there are two levels of such documents: 1) PicOS documentation (device drivers) [written by me], 2) sensor datasheets provided by the manufacturer. [↑](#footnote-ref-3)
4. There is a fourth, i.e., the temperature sensor implanted into the microcontroller itself. [↑](#footnote-ref-4)
5. This is all described in separate documents. We have an SDK (development platform) for PicOS where the ossrun.tcl part belongs to the platform and the “project” provides its specific ossi.tcl part. The whole thing can be run independently of the platform by putting the two parts together. [↑](#footnote-ref-5)
6. It is used for development to generate a C header for the Peg program. [↑](#footnote-ref-6)
7. One PicOS millisecond is equal to 1/1024 of a second. [↑](#footnote-ref-7)