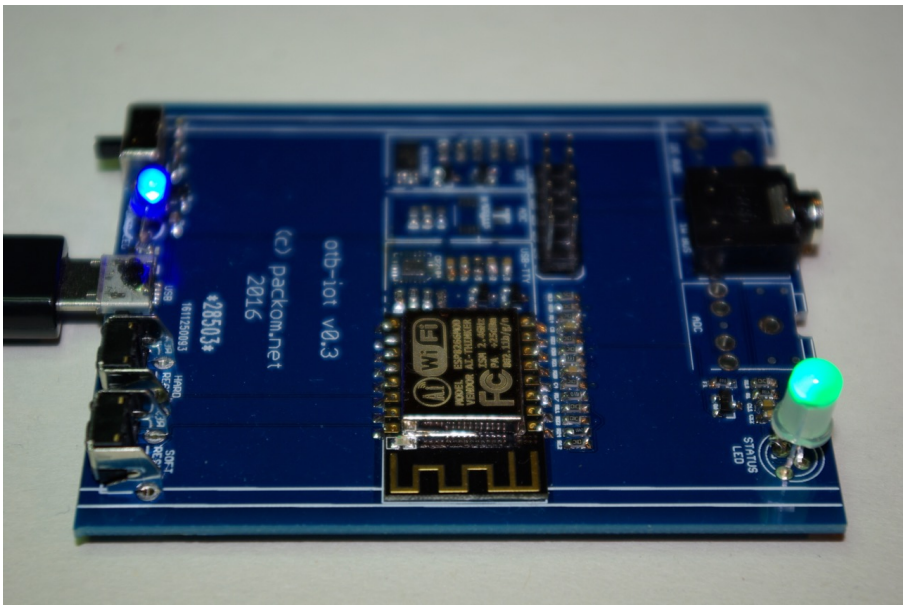
otb-iot Documentation

Release 0.1

Piers Finlayson

Feb 28, 2019

Contents


otb-iot v0.3
(c) packom.net
2016
STATUS
LED

CHAPTER 1

Contents

otb-iot docs are hosted at: https://otb-iot.readthedocs.io/en/latest/

1.1 Features

- Designed to work "Out of The Box" (OTB) - no coding required to use extensive feature set
- Hardened and reliable implementation of an IoT device running on the ESP8266 chip
- Fully open source
- Includes error handling, logging and ability to diagnose issues
- Over the air (OTA) updates for application software
- Robust and customized bootloader to automatically recover from corrupted images
- Default WiFi AP and Captive Portal to allow device to be set up quickly and easily from smart phone and other WiFi capable devices
- Controlled using lightweight MQTT protocol for IoT devices
- Provides multiple external bus options, for connecting peripherals including
  - I2C
  - ADC
  - One wire protocol
  - WS2812B and similar LEDs
- Supports many application types, including:
  - Temperature reporting
  - Power monitoring
  - Heating control

  - Remote controlled devices
  - Automated home
  - Lighting control
- Factory reset including read only factory application image
- Optional internal I2C bus for on-board devices such as additional ADCs, eeproms, etc
- Optional internal eeprom to store hardware identifying information (serial numbers, part numbers, details of on board peripherals)
- RGB status LED communicating rich status information to user
- Hardware reference designs available for
  - control devices
  - compatible peripherals

1.2 Hardware Requirements

1.2.1 Supported Modules

otb-iot has been run on many off the shelf esp8266 based modules with 4MB flash, including:

- ESP-01 (various variants, with flash upgraded to 4MB)
- ESP-12 (various variants with 4MB flash)
- ESP-07S (not the original ESP-07 which had 512KB/1MB of flash)
- WEMOS D1 Mini

1.2.2 CPU

Only the ESP8266 is currently supported.

The ESP8285 is not, as it doesn't provide sufficient flash.

The ESP32 may be supported in future.

1.2.3 Flash

otb-iot has been designed as a hardened IoT imlementation for the ESP8266, and as such includes 2 bootable application images, plus a third factory image, which is only used to replace a bootable image (it is not booted itself). This means otb-iot requires >= 4MB (32Mbit) flash.

It would probably be possible to squeeze the entire implementation onto a 2MB flash chip, but the author has never seen any ESP8266 modules with 2MB chips, so it doesn't seem worthwhile to try.

Most ESP8266 modules, with the notable exception of the ESP-01, now contain 4MB of flash.

If you need to check the flash size of your module, connect it to your serial port and run:

```
$SDK_BASE/xtensa-lx106-elf/bin/esptool.py flash_id # $SDK_BASE is esp-open-sdk␣
↪location
```

You should get an output like the following:

```
Manufacturer: ef

Device: 4016
```

In this example the flash chip is made by manufacturer 0xef (Winbond) and the device size is shown by the least significant bit of the devce ID. Here that's 0x16. This indicates that the flash is 2^0x16 bytes in size - that's 2^22 = 4MByte.

If your chip shows 4014 you have a 1MB chip, and 4013 indicates a 512KByte chip.

Note, some manufacturers may use different device ID schemes to indicate the flash size.

The other way of figuring out the size is to get magnifying glass out and read the part number. W25Q08FVSIG indicates an 8Mbit = 1MByte chip.

Upgrading the flash on a ESP-01 module is straightforward with the aid of a cheap (~$30) hot air station:

- Identify the flash chip - it's the larger of the two chips, with 8 legs.
- Note the location of pin 1 - normally indicated by a dot nearest this pin.
- With the hot air station heat the flash chip while tugging very gently with a pair of tweezers.
- Once removed solder a new chip in place with the hot air station - making sure to locate pin 1 appropriately.

The author typically replaces smaller flash chips with W25Q32FVSIG. At the time of writing, these are available on aliexpress for US$2.45 for 10 including shipping.

1.2.4 GPIO

There's no fundamental requirement to expose or use any GPIO pins to run otb-iot - with the exception of ensuring GPIO 0, 2 and 15 are in the appropriate states for flashing and running.

However, not using any GPIO pins would defeat the point of otb-iot - it's designed to interface with the real world, for which GPIO pins are required. See *GPIO Pins* for a description of what pins are used for what purpose.

1.3 Building

1.3.1 Introduction

otb-iot is intended to be built on linux and has been successfully built on various distributions including:

- Ubuntu
- Raspbian
- Arch Linux

1.3.2 Quick-Start

Plug your ESP8266 device into a USB port on your linux machine (or, if you're using VirtualBox, map the USB device through from your host to your guest).

Then run:

```
dmesg | grep usb
```

You should see output like this (in this example I am using a Wemos D1 mini - the precise text with vary depending on the USB TTL device you are using):

```
[90279.476382] usbcore: registered new interface driver usbserial_generic
[90279.476412] usbserial: USB Serial support registered for generic
[90279.480721] usbcore: registered new interface driver ch341
[90279.480755] usbserial: USB Serial support registered for ch341-uart
[90279.481709] usb 2-1.8: ch341-uart converter now attached to ttyUSB1
```

Note the value provided right at the end - here *ttyUSB1* - you'll need this in a sec.

Install docker if not already installed:

```
curl https://get.docker.com/|sh
```

You may need to log out and log back in again at this point so that your user is part of the docker group.

Run the container containing pre-built otb-iot images. Change <usb-device> to the value you noted earlier - in my example this would be *ttyUSB1*:

```
docker run --rm -ti --device /dev/<usb-device>:/dev/ttyUSB0 piersfinlayson/otbiot
```

Once the container has been pulled and run, Flash the device and connect to it over serial:

```
make flash_initial && make con
```

Use Ctrl-] to terminate the serial connection.

When you want to terminate the container run:

```
exit
```

1.3.3 Do It Yourself

If you'd rather download and build everything yourself read on.

Pre-requisites

Install the esp-open-sdk by following the instructions here: https://github.com/pfalcon/esp-open-sdk

Getting the Code

Get the code from: https://github.com/piersfinlayson/otb-iot

```
git clone https://github.com/piersfinlayson/otb-iot --recursive

cd otb-iot
```

Configuring the Makefile

You may need to modify various values within the Makefile - in particular:

```
SDK_BASE ?= /opt/esp-open-sdk  # Should point at where you installed the esp-open-sdk

SERIAL_PORT ?= /dev/ttyUSB0    # Port your ESP8266 is connected to for programming
```

Alternatively export shell variables to override the values in the Makefile:

```
export SDK_BASE=/opt/esp-open-sdk

export SERIAL_PORT=/dev/ttyUSB0
```

You could add these lines to your .bash_profile file.

Building

Run:

```
make all
```

Installing

Ensure your ESP8266 device is connected to the serial port you configured earlier and run:

```
make flash_initial
```

This command will erase the flash and then write:

- the bootloader
- ESP8266 SDK init data
- the application
- a backup "factory" application which can be used to recover the device using a factory reset

First Steps

You should now be ready to take your first steps with otb-iot. *Continue here*.

1.4 First Steps

Once you've *built the software and flashed your device* you're ready to connect to and configure otb-iot.

1.4.1 Configuring

otb-iot devices that aren't yet configured expose a WiFi Access Point (AP) and captive portal to enable the user to provide the minimal configuration required to get up and running.

Open the WiFi settings on a phone or computer and look for an access point with the named otb-iot.xxxxxx, where xxxxxx is unique to your device. (xxxxxx is the chip ID of your ESP8266.)

Connect to the AP and your device should automatically open up a browser window containing captive portal. If this doesn't happen, open a browser and navigate to http://192.168.4.1/

In the window that appears enter the following information:

- SSID of an available WiFi AP to connect to
- Password for the WiFi services
- IP address of your MQTT server (look *here* for help on setting this up)

When you press Submit the otb-iot device will reset, and attempt to connect to the WiFi AP and MQTT server you have specified. If this succeeds it will send a message to the MQTT server using topic:

```
/otb-iot/xxxxxx
```

Where xxxxxx again is your ESP8266's chipid.

If the otb-iot device fails to connect it will continue exposing its own AP to allow you to correct any configuration errors

1.4.2 Logging

Depending on your persuasion, the first thing you may chose to do after flashing your device is connect to the serial port and review logs. See *serial* for how to do this.

1.4.3 Next Steps

Now your otb-iot device is connected send it a message to check it's working. Send the following MQTT topic and message:

```
/otb-iot/xxxxxx trigger/ping
```

With mosquitto and a local MQTT server you'd run the following command:

In one terminal, monitor messages from the MQTT broker:

```
mosquitt_sub -v -t /otb-iot/#
```

And in another:

```
mosquitto_pub -t /otb-iot/xxxxxx trigger/ping
```

You should receive:

```
/otb-iot/xxxxxx ok:pong
```

in response.

Your now ready to try some more advanced stuff - head over to *mqtt* for more on the supported MQTT commands.

1.5 GPIO Pins

The GPIO pins are used for the various purposes by otb-iot. The purposes can be configured via an onboard *eeprom*, or otb-iot will use defaults.

The defaults assume a WeMoS D1 Mini board:

Pin	Purpose
0	internal I2C
1	*Reserved - UART Transmit*
2	internal I2C
3	*Reserved - UART Receive*
4	available GPIO
5	available GPIO
6	*Reserved - SD_CLK*
7	*Reserved - SD_DATA0*
8	*Reserved - SD_DATA0*
9	*Reserved - SD_DATA0*
10	*Reserved - SD_DATA0*
11	*Reserved - SD_CMD*
12	available GPIO
13	available GPIO
14	available GPIO
15	available GPIO
16	available GPIO

THe otb-iot main board v0.4 uses pins for following purposes - these are configured using an onboard 24LC128 *eeprom*.

Pin	Purpose
0	SDA - internal I2C
1	*Reserved - UART Transmit*
2	SCL - internal I2C
3	*Reserved - UART Receive*
4	available GPIO
5	available GPIO
6	*Reserved - SD_CLK*
7	*Reserved - SD_DATA0*
8	*Reserved - SD_DATA0*
9	*Reserved - SD_DATA0*
10	*Reserved - SD_DATA0*
11	*Reserved - SD_CMD*
12	available GPIO
13	available GPIO
14	Soft reset (reboot) and factory restore
15	Drives status LED (WS2812B)
16	Connected to RST for software driven reset

1.6 Serial

The ESP8266's serial port is used for two main purposes in otb-iot:

- To flash the otb-iot binaries.
- To output logs from the otb-iot device.

otb-iot also supports *Serial over MQTT*.

1.6.1 Pins

The standard ESP8266 serial pins (1-TX, 3-RX) are used by otb-iot.

1.6.2 Logs

A wealth of very useful logs are produced and transmitted over the serial port by otb-iot.

1.6.3 Connecting

There are a variety of different solutions for accessing the serial port of an ESP8266 based device:

1 On board USB-TTY. Some modules, such as the WEMOS D1 Mini come with an onboard USB-TTY chip, which allows the device to be powered and the serial port to be accessed via a single USB connector from a Windows, Mac or Linux device.

2 Using a separate USB-TTY device. This connects to the computer using a USB port and then the TX and RX pins (+ GND) are connected to the ESP8266.

3 Connecting directly to the serial pins ising GPIOs from another device - for example an Arduino, another ESP8266 or a raspberry pi.

The first solution is the simplest, but also most expensive, particular if producing devices in quantity.

Note that there are different chips used by different vendors for USB-TTY conversion, such as:

- CP2102
- CP2104
- CH340G
- FT232RL

It has been the author's experience that some converters can be flaky.

1.6.4 Programming

It is much easier to program the device if a pair of transistors are used to allow the RTS and DTR lines of the serial port to drive GPIO 0 and the RST pin of he ESP8266. This allows automatic switching of the device to be programmed into flash mode and back out - rather than requiring manual fiddling with the ESP8266 pins.

1.6.5 Speed

otb-iot's default serial port speed is 115,200 baud. However, due to a limitation in the ESP8266 chip, upon initial boot, or immediately after flashing, the bootloader will communicate at 74,880 baud. However, after any soft resets the bootloader will communicate at 115,200.

Therefore the first time after flashing the bootloader will start at 74,880, and you must connect to the device (quickly) at this speed to collect first boot logs.

1.7 Logs

1.7.1 Categories

otb-iot categorises the types of logs produced as follows:

Type	Reason
Boot	Messages from the bootloader
Debug	Disabled by default, provides very verbose logging
Info	An interesting event has occurred
Warn	A minor failure has occurred
Error	A major failure has occurred

As indicated, all logs are contained in builds of otb-iot by default, with the exception of debugging logging, which introduces significantly larger binary sizes, so these logs should only be selectively turned on.

1.7.2 Accessing

There are four ways of accessing logging information on otb-iot:

1 Using the *serial* port.

2 Once connected via MQTT, logs of type error will be reported automatically via MQTT.

3 The last N logs are stored in a circular RAM buffer. These can be retrieved individually via MQTT.

4 The last N logs are stored on the flash chip before a reboot, if the device is able to do so. These can then be read off by directly (physically) accessing the flash.

5 Not yet supported - accessing the logs when connecting to the otb-iot AP.

1.7.3 Using

When developing devices based on otb-iot the serial port should be used wherever possible to provide the quickest and easiest access to as much diagnostic information as possible. However, where this isn't possible, the stored logs can be accessed either by MQTT or reading the flash chip directly.

1.7.4 Debug Logs

As noted earlier, these should be used sparingly, as the binary will be very large with debugging turned on, and this will also slow the software down significantly. The author recommends that debugging logs are only used temporarily and when absolutely necessary.

1.8 MQTT

1.9 Flash

1.9.1 Map

The flash map for otb-iot is provided below. Note that this article may get out of date from time to time. The master version of this map can be found in otb_flash.h.

Location	Length	Contents
0x0	0x6000	Bootloader
0x6000	0x2000	Bootloader Config
0x8000	0xF8000	Application slot 0 (upgradeable, default), only 0xf4000 may be used
0x100000	0x1000	Logs (only 0x400 bytes are used)
0x101000	0x1000	Last reboot reason (only 0x200 bytes are used)
0x102000	0xFE000	Unused
0x200000	0x1000	otb-iot application configuration
0x201000	0x7000	Reserved
0x208000	0xF8000	Application slot 1 (upgradeable), only 0xf4000 may be used
0x300000	0x1000	Reserved
0x301000	0x7000	Reserved
0x308000	0xF4000	Factory application image (treated as read only)
0x3FC000	0x4000	Used by ESP8266 SDK

1.9.2 Notes

1 As can be seen, despite there being 0xF8000 bytes available for application images in slot 0 and 1, the factory image only has 0xF4000 available.

2 The last reboot reason is only written by otb-iot if it differs from the value already contained within this location. This avoids circular reboots for the same reason causing serious degredation of the flash chip (by too many erases taking place).

3 Logs are only stored in flash upon a reboot (assuming the device is able to store in flash before the reboot takes place - if power is removed this is not possible).

4 There is no separate SPIFFS filesystem in otb-iot. This is because there is very little information required to be served up as files (from the captive portal) - the filesystem for the HTTP server is linked into the application image. This has the added benefit of upgrading the HTTP files on OTA upgrade.

5 Neither bootloader nor otb-iot configuration are written to the device as part of the flash_initial process. The bootloader and application software test the configuration on flash, and if it is invalid a new version is provided.

6 It is important that the application images and factory image are at the same offset from the beginning of the MB they are stored within, as the bootloader knows this offset, loads the correct 1MB of flash (aligned on a 1MB boundary) and jumps to a location in the application image based on this offset.

1.10 Raspberry Pi Build Server

A raspberry pi can be used as a headless build server for otb-iot. This can be especially useful if you're writing hardware configuration into an I2C eeprom - as the raspberry pi and otb-iot include this programming function.

This section contains the steps to follow to set up the raspberry pi before building otb-iot.

1 Download and install raspbian lite to an sd card following the instructions here.

2 Before installing the sd card in the pi, insert in into a linux machine and identify the disk identifier the machine has allocated it. We'll assume /dev/sde here.

3 On the linux machine, mount the boot parition /dev/sde1 and create a zero length file ssh

```
sudo mount /dev/sde1 /mnt
touch /mnt/ssh
sudo umount /mnt
```

4 Now mount /dev/sde2 and set up the pi's networking. For example, to set up wifi, edit the following files:

- /etc/network/interfaces - change all instances of manual in this file to dhcp
- /etc/wpa_supplicant/wpa_supplicant.conf - add the following at the end:

```
network={
        ssid="your ssid"
        psk="your password"
}
```

5 Unmount /dev/sde2 and run sync:

sudo umount /dev/sde2 sudo sync

6 Now insert the sd card into the pi and boot it up.

7 Once booted, ssh into the IP address of the pi (if it's been allocated via DHCP you may need to query your DHCP server to find its addres):

```
ssh pi@ip_address # password is raspberry
```

8 At this point I like to install my public RSA key into ~/.ssh/authorized_keys to save entering the password again

9 Now run raspi-config to expand the filesystem, and enable I2C (optional)

```
sudo raspi-config
```

10 After rebooting to allow the changes to take effect update your pi and install the necessary packages:

```
sudo apt-get update

sudo apt-get dist-upgrade

sudo apt-get install make unrar-free autoconf automake libtool gcc g++ gperf flex␣
↪bison texinfo gawk ncurses-dev libexpat-dev python-dev python python-serial sed git␣
↪unzip bash help2man wget bzip2 libtool-bin hexedit
```

11 Install esp-open-sdk:

```
cd ~/

git clone --recursive https://github.com/pfalcon/esp-open-sdk.git

cd esp-open-sdk

make
```

(continues on next page)

(continued from previous page)

```
cd ..
```

You're now ready to *get started* with otb-iot.

1.11 Serial over MQTT

1.11.1 Introduction

otb-iot supports controlling serial devices remotely, controlled over the MQTT channel.

There are three key primitives that are supported:

- set - to set up serial port configuration (which pins to use, speed of communication)
- get - to read back serial port configuration (similar set of fields as set)
- trigger - to send data via serial, and to receive received packets

1.11.2 Setting it Up

An example of a series of MQTT commands to set up serial communication are as follows:

```
topic: /otb-iot/chipid message: set/config/serial/tx/15
topic: /otb-iot/chipid message: set/config/serial/rx/13
topic: /otb-iot/chipid message: set/config/serial/speed/2400
topic: /otb-iot/chipid message: set/config/serial/enable
```

1.11.3 Sending and Receiving Data

otb-iot maintains a circular buffer of received serial data. As it's possible some noise on the line has caused data to be random received, it's worth clearing this buffer before sending data:

Sent:

```
topic: /otb-iot/chipid message: trigger/serial/buffer/dump
```

Received:

```
topic: /otb-iot/chipid/status message: ok:xx
```

xx indicates the buffer is empty. A maximum of 25 bytes will be returned in each status message - multiple messages will be sent if the buffer contains more than 25 bytes.

To send serial data:

```
topic: /otb-iot/chipid message: trigger/serial/send/105b015c16
```

Here 0x10 0x5b 0x01 0x5c 0x16 will be sent.

Data is returned as hex bytes:

```
topic: /otb-iot/chipid message: trigger/serial/buffer/dump
topic: /otb-iot/chipid/status   message: ok:01/02/03/04/xx
```

Here 0x01, 0x02, 0x03, 0x04 were received in that order.

The buffer is cleared as bytes are read. If the buffer is read again immediately:

```
topic: /otb-iot/chipid message: trigger/serial/buffer/dump
topic: /otb-iot/chipid/status   message: ok:xx
```

1.11.4 More Details

For more details on the commands supported see otb_cmd.h.

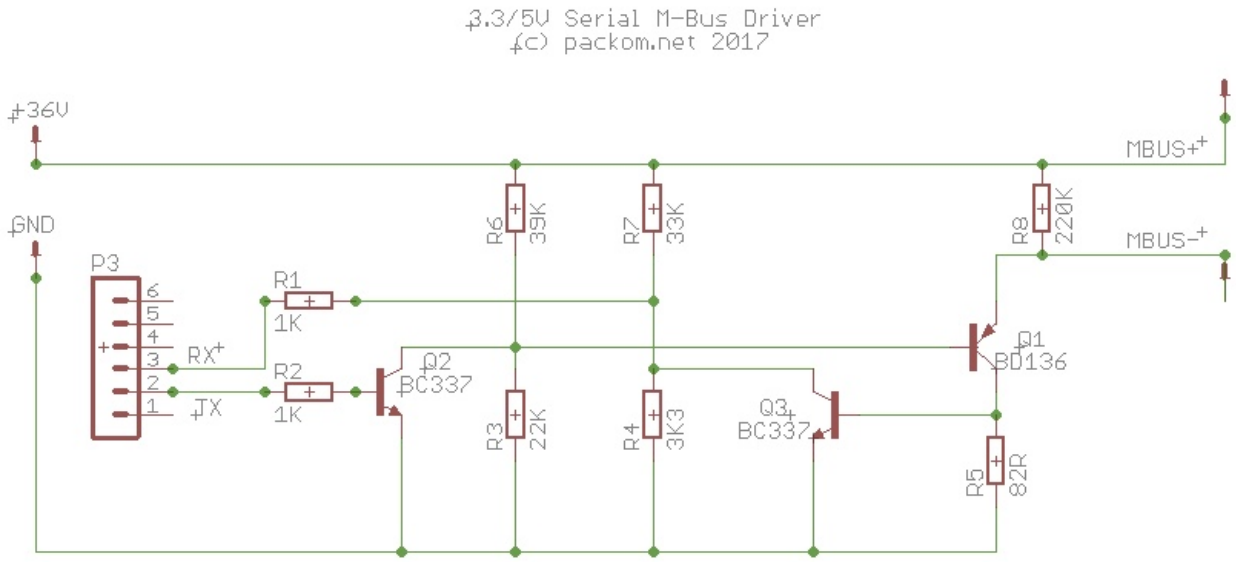
1.12 M-Bus Support

otb-iot supports communication with devices that support the M-Bus protocol. This is used by a variety of heat and other meters. The standard (wired) M-Bus protocol is based on serial communications, but the two lines run at +36V and +24V respectively. This allows the meter to be powered by the bus.

To communicate with M-Bus devices using otb-iot a simple circuit must be built to allow the 3.3 or 5V support by the otb-iot device (depending on what unit you're using) to be converted to the +36/24V used by M-Bus.

Once this circuit is place between the otb-iot module and the M-Bus capable meter, otb-iot's *Serial over MQTT* function can be used to communicate with the M-Bus device.

1.12.1 Circuit



This circuit is based on: https://github.com/rscada/libmbus/blob/master/hardware/MBus_USB.pdf

1.12.2 Communicating with Device

A sequence of MQTT commands such as the following should return M-Bus data from the device:

```
topic: /otb-iot/chipid          message: set/config/serial/disable
topic: /otb-iot/chipid          message: set/config/serial/tx/15
topic: /otb-iot/chipid          message: set/config/serial/rx/13
topic: /otb-iot/chipid          message: set/config/serial/speed/2400
topic: /otb-iot/chipid          message: set/config/serial/enable
topic: /otb-iot/chipid          message: trigger/serial/buffer/dump
topic: /otb-iot/chipid          message: trigger/serial/send/105b015c16
topic: /otb-iot/chipid          message: trigger/serial/buffer/dump
```

In the data string sent, change 01 to the meter's M-Bus address.

Data such as the following should be returned - although the exact contents will depend on your meter. (This output is from an Itron CF Echo II.)

```
topic: /otb-iot/5ccf7f0b583b/status message: ok:68/4d/4d/68/08/01/72/60/31/27/14/77/
↪04/09/04/97/10/00/00/0c/78/60/31/27/14
topic: /otb-iot/5ccf7f0b583b/status message: ok:04/07/cd/03/00/00/0c/15/80/63/01/00/
↪3b/2d/99/99/99/0b/3b/00/00/00/0a/5a/53
topic: /otb-iot/5ccf7f0b583b/status message: ok:01/0a/5e/21/02/3b/61/99/99/99/04/6d/
↪24/13/2b/22/02/27/70/03/09/fd/0e/22/09
topic: /otb-iot/5ccf7f0b583b/status message: ok:fd/0f/47/0f/04/00/ff/16/xx
```

1.12.3 Decoding

rscada's M-Bus code can be used to decode this output. A forked copy is available here: https://github.com/piersfinlayson/libmbus

In addition to libmbus and various utilities from rSCADA this fork also contains a utility to decode hex bytes returned my otb-iot.

To use:

```
git clone https://github.com/piersfinlayson/libmbus
cd libmbus
./build.sh
bin/mbus_process_hex_dump <hex string>
```

(Alternatively, libmbus is a submodule within otb-iot - just run make mbus_tools. The binary will be in extras/mbus_tools/bin.)

For example, given the output above, the following command would be run:

```
bin/mbus_process_hex_dump␣
↪684d4d68080172555555557704090455100000c78555555550407cd0300000c15806301003b2d9999990b3b0000000a5a2
```

This would return:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<MBusData>
    <SlaveInformation>
        <Id>55555555</Id>
        <Manufacturer>ACW</Manufacturer>
        <Version>9</Version>
        <ProductName>Itron CF Echo 2</ProductName>
        <Medium>Heat: Outlet</Medium>
        <AccessNumber>85</AccessNumber>
        <Status>10</Status>
```

(continues on next page)

(continued from previous page)

```
        <Signature>0000</Signature>
    </SlaveInformation>
    <DataRecord id="0">
        <Function>Instantaneous value</Function>
        <StorageNumber>0</StorageNumber>
        <Unit>Fabrication number</Unit>
        <Value>55555555</Value>
    </DataRecord>
    <DataRecord id="1">
        <Function>Instantaneous value</Function>
        <StorageNumber>0</StorageNumber>
        <Unit>Energy (10 kWh)</Unit>
        <Value>973</Value>
    </DataRecord>
    <DataRecord id="2">
        <Function>Instantaneous value</Function>
        <StorageNumber>0</StorageNumber>
        <Unit>Volume (1e-1  m^3)</Unit>
        <Value>16380</Value>
    </DataRecord>
    <DataRecord id="3">
        <Function>Value during error state</Function>
        <StorageNumber>0</StorageNumber>
        <Unit>Power (100 W)</Unit>
        <Value>999999</Value>
    </DataRecord>
    <DataRecord id="4">
        <Function>Instantaneous value</Function>
        <StorageNumber>0</StorageNumber>
        <Unit>Volume flow (m m^3/h)</Unit>
        <Value>0</Value>
    </DataRecord>
    <DataRecord id="5">
        <Function>Instantaneous value</Function>
        <StorageNumber>0</StorageNumber>
        <Unit>Flow temperature (1e-1 deg C)</Unit>
        <Value>220</Value>
    </DataRecord>
    <DataRecord id="6">
        <Function>Instantaneous value</Function>
        <StorageNumber>0</StorageNumber>
        <Unit>Return temperature (1e-1 deg C)</Unit>
        <Value>300</Value>
    </DataRecord>
    <DataRecord id="7">
        <Function>Value during error state</Function>
        <StorageNumber>0</StorageNumber>
        <Unit>Temperature Difference (1e-2  deg C)</Unit>
        <Value>999999</Value>
    </DataRecord>
    <DataRecord id="8">
        <Function>Instantaneous value</Function>
        <StorageNumber>0</StorageNumber>
        <Unit>Time Point (time &amp;amp; date)</Unit>
        <Value>2017-01-30T21:17:00</Value>
    </DataRecord>
    <DataRecord id="9">
```

(continues on next page)

(continued from previous page)

```
        <Function>Instantaneous value</Function>
        <StorageNumber>0</StorageNumber>
        <Unit>Operating time (days)</Unit>
        <Value>868</Value>
    </DataRecord>
    <DataRecord id="10">
        <Function>Instantaneous value</Function>
        <StorageNumber>0</StorageNumber>
        <Unit>Firmware version</Unit>
        <Value>22</Value>
    </DataRecord>
    <DataRecord id="11">
        <Function>Instantaneous value</Function>
        <StorageNumber>0</StorageNumber>
        <Unit>Software version</Unit>
        <Value>47</Value>
    </DataRecord>
    <DataRecord id="12">
        <Function>Manufacturer specific</Function>
        <Value>04 00</Value>
    </DataRecord>
</MBusData>
```

(Note SlaveInformation->Id and Fabrication number have been redacted.)

1.12.4 Automated Reading

otb-iot contains a script to automatically query an otb-iot device connected to an M-Bus capable meter and upload retrieved information to influxdb.

1.13 Eeprom

1.13.1 Introduction

otb-iot supports an onboard I2C eeprom continuing hardware and basic configuration information. This eeprom is expected to reside at address 0x57, and to be an 24LC128 (128kbit, 16kbyte) eeprom.

Data is stored on the eeprom using the structures defined in otb_eeprom.h.

1.13.2 Information

The first structure is otb_eeprom_info and lives at address 0x0. At the end of this structure there are a series of otb_eeprom_info_comp structures. These point to the other structures available on the eeprom.

On an otb-iot main board, the following other stuctures are present:

- otb_eeprom_main_board
  - contains global information about the board such as chipid, mac addresses, number of modules
- otb_eeprom_main_board_sdk_init_data
  - stores sdk_init_data which is reflashed to the board in the event of a factory reset

- otb_eeprom_main_board_gpio_pins
  - purposes of the board's *gpio_pins*
- otb_eeprom_main_board_module (may be multiple of)
  - information about each of the module exposed by this main board (number of headers, pins, what each pin is, etc)

1.13.3 Generating Eeprom Data

The hwinfo application is provided to burn the eeprom with the appropriate strucures over I2C. It needs to be run on a device which has native I2C support and GPIO pins - such as the raspberry pi.

The following is an example invocation of hwinfo:

```
hwinfo -v -e 128 -i 123456 -1 abcdef -2 bcdef1 -f 4096 -d 0 -t 2 -c 1 -s 1 -m 1 -z␣
↪987123
```

Breaking this down:

- -v
  - runs in verbose mode
- -e 128
  - size in kbit of eeprom (128kbit, 16kbyte)
- -i 123456
  - chipid of esp8266
- -1 abcdef
  - prefix of station mac address (first 3 bytes)
- -2 bcdef1
  - prefix of AP mac address (first 3 bytes)
- -f 4096
  - Size of esp8266 flash in kbyte (4Mbyte)
- -d 0
  - Type of external ADC installed on the main board (0 = none)
- -t 2
  - Configuration of esp8266 ADC configuration (2 = 220K/100K resistive divider)
- -c 1
  - otbiot hardware code (1 = otbiot main board)
- -s 1
  - otbiot hardware subcode (1 = main board v0.4)
- -m 1
  - esp module type (1 = ESP12)
- -z 987123

  - Board serial number

This generates a binary file, hwinfo.out, with the data to be burnt to the eeprom

1.13.4 Setting up Pi to Burn Eeprom

- Install raspbian
- Set up I2C on the pi
  - sudo rasp-config
  - select option 5 (interacing options)
  - select option P5 (I2C)
  - select yes
  - exit raspi-config
  - sudo reboot
  - i2cdetect -y 1 (may need to use 0 on an early raspberry pi)
    - check this doesn't fail
- Check out otb-iot on your pi and build hwinfo:
  - git clone –recursive https://github.com/piersfinlayson/otbiot
  - cd otb-iot
  - make hwinfo
  - make i2c-tools
- Wire your pi up to the I2C bus the eeprom is install on - these instructions assume using a pi zero and that the I2C bus is an otb-iot v0.4 module 1
  - Wire pi pin 3 (SDA) to module 1, header 1, pin 9
  - Wire pi pin 5 (SCL) to module 1, header 1, pin 10
  - Wire pi pin 6 (GND) to module 1, header 1, pin 7
  - Wire header 1 pin 11 (GND) to header 2 pin 1 (to disable write protect on the eeprom)
- Power on the eeprom/otb-ot device
- Get the chipid and mac address information from the otbiot/esp device to provide hwinfo
  - make flash_stage (this flashes an application to the esp8266 which retrieves this information)
  - connect to the esp8266 device over serial and note down the chipid and mac address prefixes
- Run hwinfo to get hwinfo.out

1.13.5 Burning Eeprom

- Burn the eeprom with this command:
  - bin/eeprog /dev/i2c-1 0x57 -16 -f -w 0x0 < hwinfo.out
- Disconnect the write protect pin (to renable write protect)
- Check the data flashed corectly by reading it back:

- bin/eeprog /dev/i2c-1 0x57 -16 -xf -r 0x0:0x3fff > eeprom.out
- hexedit eeprom.out
  - Check the data is as expected (and not all zeros or all fs)

CHAPTER 2

Contribute

- Issues: https://github.com/piersfinlayson/otb-iot/issues
- Code: https://github.com/piersfinlayson/otb-iot

CHAPTER 3

Support

Raise bugs as github issues or report to piers@piersandkatie.com.

CHAPTER 4

License

The project is licensed under the GPLv3 license.

https://www.gnu.org/licenses/gpl-3.0.en.html

CHAPTER 5

Search

- search