Development of a BLE 4.2 transceiver and a partial bluetooth stack based on the Semtech SX1281 transceiver module

Heinrich Diesinger,
Univ Lille, Univ Polytech Hauts France, CNRS, Centrale Lille, Junia, Inst Elect Microelect & Nanotechnol (IEMN) UMR 8520, F-59652 Villeneuve d'Ascq, France

Abstract
Bluetooth low energy (BLE) is a communication protocol for low power applications that require longlasting operation in battery powered applications at low weight and size (typical: smartwatches, sensors, drones, cargo tracking). With baud rates betweeen 125 kbps and 2 Mbps it is located between large bandwidth (WiFi) short range, and low bandwidth long range (LoRa, GFSK), FLRC being in the middle. In the sensitivity performance, FLRC and LoRa are closer to the Shannon bound than the 3 others due to the use of spread spectrum techniques.
Another interesting aspect of GFSK based BLE protocol is the number of device classes or profiles and the option by definition of the standard to display data beyond the device name or SSID already in the advertisement mode (previous BT versions: discovery) in the preview, prior to the handshake and pairing. Therefore, BLE devices are able to display for example a reduced set of sensor data to the public in the preview, or transmit public messages prior to the pairing process involving cryptography.

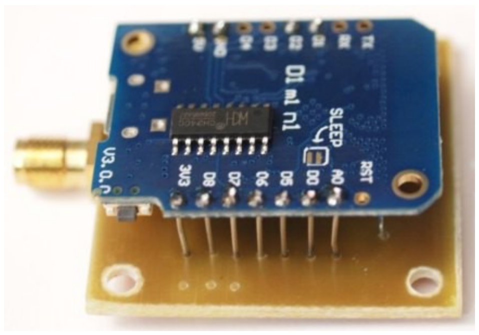
Requirements

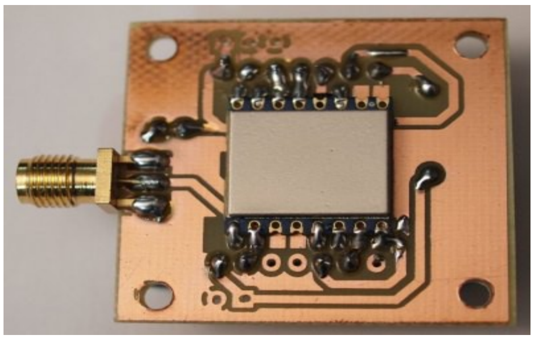
For research purposes (sensitivity performance, protocol immunity, Doppler or carrier frequency offset stability), we require a BLE transceiver that can communicate at the standard 1 Mbps and reduced baud rate of 125 kbps. BLE 4.2 radios are now widely available but often lacking the 125 kbps rate. The Semtech SX128x chip seems to offer it at low cost and ease of use e.g. compared to the Nordic Semiconductor modules that come with their own IDE or an Arduino core that lacks advanced functionality. Moreover, the SX128x module is a most versatile 2.4 GHZ transceiver that features, in addition to basic GFSK and GFSK based BLE, also the modes FLRC, 2.4 GHz LoRa and in the 1280 version also a ranging engine. While GFSK, FLRC and LoRa stacks are implemented in hardware, BLE 4.2 compatibility is implemented only up to the physical layer.
We preferred this module despite the absence of the stack since for the intended characterization, a partial stack implementation below handshake would readily enable the advertisement mode. A first approach of testing it with available libraries (Stuart Robinson, Sandeep Mistry, Radiohead McCaulny, RadioLib) was disappointing because either the communication could not be established at all in BLE (authors focusing on LoRa ?) and/or some parameters (sync word aka access address) were not accessible through the library. In the end, the decision was to abandon available libraries and to write our own library based on the commands given by the manufacturer datasheet [https://semtech.my.salesforce.com/sfc/p/#E0000000JelG/a/2R000000HoCb/X1yTNr5aeVl mviwNhvHyX9a2wSTSla.JWmnEtvAAlRk], using only the standard SPI library to transfer machine code and register values over the port.

Hardware

The aim is a transceiver that can be controlled over serial USB from a computer, tablet or smartphone and combines a radio module and a microcontroller, also enabling to run autonomously and store own code and some data. Whereas a choice of boards (Lillygo, TTgo etc.) are available that combine an ESP32 with a SX1278 LoRa radio, the SX128x can up to now (mai 2022) not be purchased on a similar board (susceptible of changing rapidly...). In principle it would also be possible to control a SX128x over serial port solely by a usb to serial bridge since its novelty is the ability to pass commands either via SPI or RS232, but this would require software on the host computer in order to be used ergonomously and preclude running code on the board. Our choice is to build our own experimental platform while a commercially available solution is still unavailable. Since the ESP32 is oversized for basic control of the SX128x, we opt for a ESP8266 D1 mini module. A requirement for us is a 50 Ohm output for connecting external antennas and measurment gear, rather than a PCB inverted F antenna often seen on bluetooth and WiFi modules. A SX1281 module in a can package by niceRF has a solder pad antenna output (neither IFA nor IPEX connectors). We run a coplanar waveguide from the solder pad to a SMA edge connector.

The board front and back

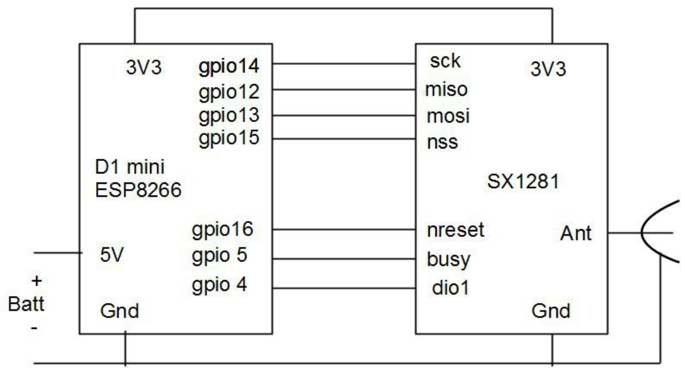




PCBoard design:

The PCBoard is a single layer double sided component design that holds the D1 mini and SX128x on opposite sides. Compared to the SX1278 433 MHz LoRa radio used in a previous project, there are two more connections, DIO1 shall be connected in case interrupts are mapped to an output and the Busy pin is also additional. These two pins are connected to the nearby SDA and SCL pins of the Wemos respectively, meaning the I2C port becomes unusable and we sacrify the option of connecting sensors to it. The antenna is a normal solder pad, not an IPEX connector. We route it with a cm of (approximative...) coplanar waveguide towards an external sma edge connector. The SX128x module is located between the solder pad rows of the D1 mini DIL package. PCBoard design has the minor drawback that lanes pass underneath some unused solder pads of the SX128x module that must be isolated, e.g. by putting some mica or insulating tape underneath the module. At large scale production, solder stop mask could do the job. An external voltage regulator revealed to be unnecessary because the radio draws less power and can be hooked to the LDO regulator even of the budget clones of the D1 mini.

Circuit Diagram



The code

is programmed in Arduino IDE with the ESP8266 core. The code is commented and self explaining. The name of the functions are close to the ones used in the datasheet [link]. Features are the ability to enter PDUs (protocol data unit) by entering a space separated list of hex values or by ascii, and to form packets by calculating a header which the hardware doesnt perform for BLE since the module only provides BLE compatibility at the hardware level.. The same way, byte arrays for configuration registers can be entered. Configuration data and the output packet can be stored on EEPROM to be loaded at the next power up. If EEPROM is absent, the hardcoded defaults will be used instead. The serial interface uses the settings 9k6 8N1. At startup a help screen is displayed. It can be re-displayed by typing "help". Upon packet reception, it is displayed along with the length, PDUtype from the header, the present frequency, RSSI, and the checksum.

The help screen:

```
SX128x utilities with own functions as suggested by datasheet
and direct SPI access to Opcodes/registers/buffers
By heinrich.diesinger@cnrs.fr, F4HYZ
Usage: type commands without prefix
System commands:
batlevel: shows the supply voltage
help: displays this info again
eepromstore: stores the settings to EEPROM (must be called manually)
eepromdelete: clears EEPROM
eepromretrieve: reads settings from EEPROM
reboot: restarts the firmware
Radio related settings:
packettype <0x04>: sets mode (GFSK, LoRa, range, FLRC, BLE)
freq <2410.0>: sets the LoRa frequency in MHz [2400..2500]
txparams <0x1f 0x80>: sets LoRa power in dBm + 18 [0..31] and ramptime (manual)
power <13>: sets RF power in dBm
syncarray <0x00 0x12 0x34 0x56 0x78 0x00 0x00 0x00 ...>: acc addr or sync words
modparams <0x45 0x01 0x20>: baud/BW, mod_indx, gauss_filt
packparams <0x80 0x10 0x18 0x00 0x00 0x00 0x00>: con_st, crc, test, whiten
irqmask <0x00 0x00 0x00 0x00>: irq masks interrupt, dio1, dio2, dio3
buffbase <0x20 0x80>: tx/rx buffer base addresses
autotx <0x00 0x5c>: auto tx delay required by BLE
setupble: manually applies the BLE radio settings
required at startup and to apply changed settings !!!
setrffreq: manually applies the freq only
setrftxparams: manually applies power, ramptime only
Header (and pre-header) settings for computing outgoing packets:
outpdutype <7>: [0..8]
outtx <0>: [0, 1]
outrx <0>: [0, 1]
outllid <0>: [0..3]
outnesn <0>: [0, 1]
outsn <0>: [0, 1]
outmd <0>: [0, 1]
Payload, PDU and SPI raw data input and transmitting:
blemessage  < >: [any text] sets blemessage char array and
    converts it to outpayload byte array (6..37 adv, 0..31 dat)
blepayload: [0x12 0x34 0x56 ..] creates outpayload byte array from user entered 2 digit hex list
makebleadv: from outpayload, calculates header for advertising packet and composes outpdutotal
makebledat: from outpayload, calculates header for data packet and composes outpdutotal
sendoutpdutotal: transmits outpdutotal over the air
spiupload:  [0x12 0x34 0x56 ..] creates uploadbytearray from user entered 2 digit hex list
ATTENTION this gets overwritten by loop activity if intvl is set
spitransfer: transfers uploadbytearray to radio transceiver, retrieves downloadbytearray
     and displays it as hex, ascii and decomposed (upper/lower 3 bit) status bytes
Radio control commands:
clearirqstatus: clears IRQ, needed before settx etc.
setstandby: sets the radio to standby RC 13 MHz
setrx: sets the radio to single receive mode
settx: sets the radio to single transmit mode
setcw: sets radio to transmit a carrier
getstatus: gets radio status
Loop related:
intvl <5000>: repeat intvl for rx tx in ms; inactive if 0
scanflag <0>: if set, freq steps through advertisement channels
txflag <1>: if set, settx() is issued each intvl
```

A bit of Syntax for the basic usage :

Let us assume that we want to generate the BLE advertisement package as described by the sample 2 of the document https://docs.silabs.com/bluetooth/latest/general/adv-and-scanning/bluetooth-adv-data-basics#example It means we have a list of hex values from which we want to generate an advertisement packet. We first define the payload by typing

```
blepayload 0x02 0x01 0x06 0x11 0x07 0x07 0xb9 0xf9 0xd7 0x50 0xa4 0x20 0x89 0x77
0x40 0xcb 0xfd 0x2c 0xc1 0x80 0x48 0x09 0x08 0x42 0x47 0x4d 0x31 0x31 0x31 0x20
0x53
```

Next, to generate an advertisement package from that payload, we type

```
makebleadv
```

this adds a header depending on the different settings (outpdutype....outmd) but let's leave the defaults for now. To transmit it over the air, type

```
sendoutpdutotal
```

Later on, if we want to transmit the packet periodically, we define an interval <> 0 by setting it e.g. to 5 seconds:

```
intvl 5000
```

and telling our board to do a transmission every time the interval is due, otherwise it would only receive and display the result every interval.

```
txflag 1
```

the loop does reception by default and interrogates the receive buffer also every interval.

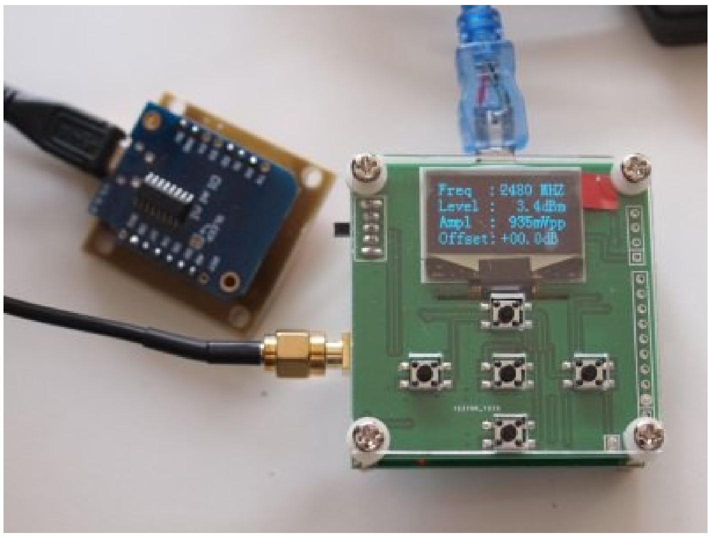
Packet received :

Upon packet reception, it is displayed along with the length, PDUtype from the header, the present frequency, RSSI, and the checksum :

```
Packet received:
RSSI = -60.00 dBm
freq = 2410.00 MHz
Length extracted from getrxbufferstatus : 136 characters
Length extracted from header : 34 characters
PDUtype extracted from header : 7
Reading PDU content :
Hex: 0x2 0x1 0x6 0x11 0x7 0x7 0xB9 0xF9 0xD7 0x50 0xA4 0x20 0x89 0x77 0x40 0xCB 0xFD 0x2C 0xC1 0x80 0x48
0x9 0x8 0x42 0x47 0x4D 0x31 0x31 0x31 0x20 0x53 0xFC 0x69 0xBE
ASCII: # # # # # # ¹ ù × P ¤   ‰ w @ Ë ý , Á € H         # B G M 1 1 1   S ü i ¾
Reading checksum CRC :
Hex: 0xC9 0x38 0x88
ASCII: É 8 ˆ
```

Measurement of the output power :

For this test, the power was set to 5 dBm and the continuous transmission of a carrier was enabled by the command setcw. A connected power meter yields 3.4 dBm, the difference 1.6 dB is attributed approximately to the losses of the GSG output lane between radio and SMA connector, the connectors themselves and cable losses.



Range test over 600 meters

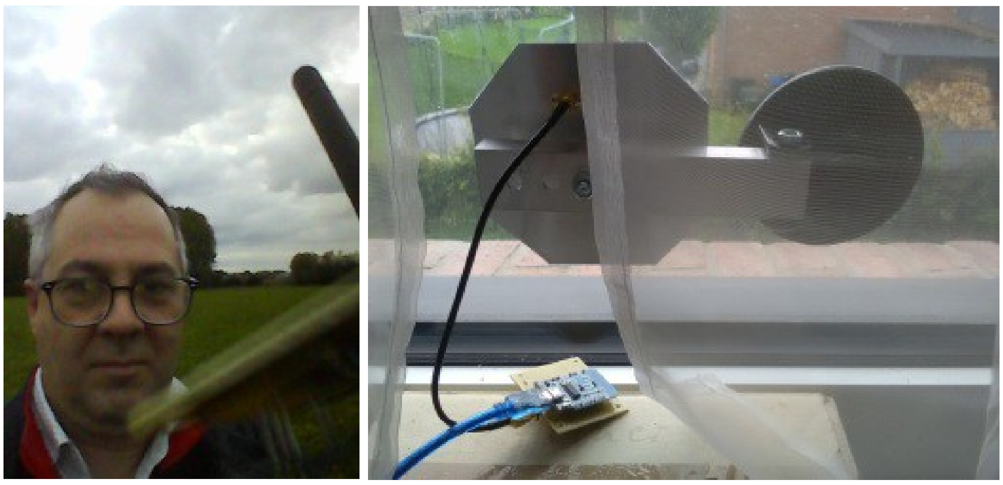


Photo of equipment for field test: window patch antenna and 2 dBi rubber antenna on mobile unit

A first field test was performed with the equipment of above photo, 2 dBi nominal rubber antenna on mobile unit and window mount patch antenna, version with 22 mm washers. The protocol was BLE 1 Mbps, f= 2410 Khz, p= 5 dBm.

The displayed RSSI of the last received packet at a distance of 600 m was -96 dBm.

Comparison with the link budget:

P_r = P_t + G_t + G_r - PL - L_pol - L_conn =

5 dBm + 7.4 dB -1.2 dB - 95.6 dB - 3 dB – 2 dB = -89.4 dBm

Received power P_r

P_t is the transmiiter power = 5 dBm nominal, the powermeter displays 3.4 dBm so 1.6 is taken in account in the connection loss L_conn. Another 0.4 dB are added for the connectors on the receiver side, yielding 2 dB declared as connection loss L_conn.

G_t is the gain of the transmitting antenna 10.4 dB according to our antenna study and we subtract 3 dB because the antenna is behind a double glass window, and we obtain 7.4 dB

G_r is the gain of the receive antenna which is -1.2 dB according to our antenna study.

For the antenna study, see:
https://github.com/technologiesubtile/2.4-GHz-ISM-antennas

Path loss: PL = 20 log (4 pi d f / c) = -95.6 dB

Polarization loss: L_pol: -3 dB between circular and linear, -20 dB between V and H

L_conn see above P_t

The 6.6 dB disagreement between RSSI and link budget can tentatively be attributed as follows:

- -96 dBm is better than on the datasheet p. 25 that states a sensitivity of -94 dBm for high sensitivity mode at FSK at 1 Mbps. The RSSI reading is not accurate.

- the polarization loss of 3 dB is between ideally circular and linear and can be more

- the propagation is through clear line of sight. However, it is close to the ground and there are obstacles along the way, both leading to multipath fading. In air and space communication, this effect will not occur.

Extrapolation and perspective:

The 600 meters range test was performed with artificially throttled performance. We can increase output power to 13 dBm, replace the antennas by 14 dBi quad patch on both sides, eliminating also the polarization loss, and we can communicate from ground to air/space to eliminate the multipath fading. Making use of all effects will increase the range to several (approaching 10) kilometers. To increase the range further, the signal can be amplified with WiFi amplifiers or even more performant antennas can be used, for example a simple patch antenna with a parabolic mirror as for the QO-100 uplink. Such configurations would likely (almost certainly) exceed the maximum EIRP authorized for ISM use, but might be authorized in different context e.g. operating in satellite amateur band with the dish aimed at a satellite to avoid interference with terrestrial services.

More syntax for the expert mode

In general, all settings appearing in the help screen under the category “Radio related settings“ must be followed by the command “setupble“ in order to be applied to the SX1281 and not only to the variables of the ESP8266. For example, to change the packet type to GFSK, we would use the commands

```
packettype 0x00
setupble
```

However for changing only the frequency, it is also possible to type

```
freq 2402.0
setrffreq
```

The setrffreq, similarly as setupble, commits the modified frequency only to the transceiver, but it avoids committing all other radio settings. Similarly, for the power, it can be set e.g. to 10 dBm by

```
power 10
setrftxparams
```

or by

```
txparams 0x1c 0x80
setrftxparams
```

Warning: if the frequency is changed only by “freq 2402.0“ and the committing by “setrffreq“ or “setupble“ is omitted, then it is not changed in the receiver. However, for incoming packets, the resulting message would nevertheless wrongly indicate that the packet was received at 2402.0 MHz, although the transceiver would still be receiveing at the previous frequency.

Writing raw lists of Hex values to the transceiver via the SPI port: in the basic usage, the command blepayload writes payload into a variable, the command makebleadv appends an advertisement header in front and stores everything in the variable outpdutotal, and finally the command sendoutpdutotal appends another 2 Hex bytes in front before sending it to the transceiver, to specify that it is a write command and what is the memory offset. In some cases, the expert user (aware of what he is doing) might want to send raw Hex or machine code to the transceiver instead. This is achieved by

```
spiupload 0x12 0x34 0x56
spitransfer
```

Before such operation, make sure the interval for periodic Tx/rx execution is set to 0
intvl 0

because otherwise the ESP8266 is periodically communicating with the transceiver and this would overwrite the spiupload buffer.

Details about the serial port

The communication settings are quite standard, 9.6 kbps, 8N1. The USB UART bridge of the Wemos D1 mini board is typically a CH341 in the case of the chinese clones but they are also available with a FTDI232 upon request. The expected termination is a newline \n character. However, since some terminal software, in particular Android apps, sometimes have difficulties in issuing a \n, there is also a 2 second timeout. As long as the Serial Monitor of the Arduino IDE is used, this is meaningless because the command is first typed into a line and then sent at once when pressing "enter", together with the terminator according to the checkmark at the lower board of the window. However, when using terminal software that sends the characters one by one as they are typed, they must be typed quickly enough to prevent automatic termination by the timeout. An alternative is to paste them. For example in PuTTy, paste is done by Shift-Ins, and a newline can optionally be sent by Ctrl-J.

Disclaimer

No warranty of correctness or suitability for a particular purpose is given. Hard- and software may not be used for applications that require failsafe operation including but not limited to medical and life sustaining systems and passenger transport. It is the sole responsibility of the user to comply with local regulations regarding spectrum allocation and authorized RF power.

Acknowledgement

The transceiver has been developped in collaboration with Prof. Guillaume Ferré and Marwane Rezzouki from ENSEIRB-MATMECA, Laboratoire IMS, UMR 5218 CNRS, F-33400 Talence in the context of a larger project comparing the physical layer performance of different protocols. Fruitful discussions are gratefully acknowledged.