open source watch Documentation

Release 1.2.0

jj

Apr 30, 2021

CONTENTS


Zephyr™ Project

CHAPTER

ONE

COPYRIGHT

The book is subject to copyright.

You cannot use the book, or parts of the book into your own publications, without the permission of the author.

I have put in most of the code a copyright with my mail address. This does not mean anything. It is just to let you know I modified existing code. The code has been tampered with ...

1.1 author:

Jan Jansen najnesnaj@yahoo.com

1.2 LICENSE:

All the software is subject to the Apache 2.0 license (same as the Zephyr RTOS) A permissive license whose main conditions require preservation of copyright and license notices. Contributors provide an express grant of patent rights. Licensed works, modifications, and larger works may be distributed under different terms and without source code.

You could use nRF Connect SDK (often referred as NCS),it is Nordic Semiconductor SDK based on zephyr. It has additional features that could be useful for pinetime, namely bluetooth modules and bluetooth services. Important to mention that even though, NCS forks zephyr it is kept close to it and it is regularly updated to latest zephyr (typically every 2-3 weeks).

NCS is using BSD-5-Clause-Nordic license (https://github.com/NordicPlayground/fw-nrfconnect-nrf/blob/master/LICENSE) which is why I did not use their samples.

(see also : bluetooth serial communication)

CHAPTER

TWO

ZEPHYR SMARTWATCH FRAMEWORK

Suppose you want to build a cycle computer quickly. You need : - bluetooth - a touchscreen - a button - time (clock) - a method of setting the clock - wireless firmware updating - timer functions - power saving - battery management - several graphical screens, with buttons graphics

You could start from scratch, or use a framework and only add the stuff you need for your cycling computer : - cadence sensor - display speed - record speed

Suppose you have an idea for a portable device, before running to the shop to buy a dev-board, develop hardware .... You could use the framework as a starting point, since it supports a virtual posix-board.

```
this document started off in setting up zephyr RTOS on the PineTime smartwatch.

https://wiki.pine64.org/index.php/PineTime

It evolved to accomodate other nordic nrf52832 based watches (Desay D6....) and a␣
↪Virtual watch (native_posix_64)

Zephyr allows for hardware abstraction. The framework is not limited to Nordic nrf52.
In theory : compiling with a different board definition file should be enough. (minor␣
↪changes might be necessary - see the button sample)


The virtual watch lets you create and debug software without a watch nor a debugprobe.
↪ (=cheap)
There exist a lot of cheap (chinese) watches based on Nordic microcontrollers.
The hardware will probably differ, but with minor adjustments it should be possible␣
↪to adapt the framework.
```

```
the approach in this manual is to get quick results :
    - minimal effort install
    - try out the samples
    - inspire you to modify and enhance


The masterpiece is the firmware toolkit.
Since it is a big chunk of code, you might be lost.

The samples in the /app directory, contain parts of the toolkit.
Like building blocks, which form the final firmware.
```

suggestion :

- follow the Zephyr installation instructions
- try some examples

- if you like it copy the /app directory for some more fun

CHAPTER

THREE

THE IDEA BEHIND THE FRAMEWORK

I adapted the pinetime hypnos firmware to run on the Desay D6 smartwatch.

This watch has a small (128x32) oled display and is rather minimalistic.

It does not have a touchscreen, but a single touch button.

One cannot run the software of a feature-rich smartwatch on a simple one. But the reverse can be done.

3.1 Building blocks

A smartwatch has drivers for motion sensors, bluetooth, HR sensor, a battery, a screen, uses a clock, sets the clock, has a button. . .

If all put together, you'll end up with the firmware and a big chunk of code.

This project explores features separately, in the "samples" directory.

Reading out a button, how is it done? How do you make a distinction between click, double click, long press? You can find this in the samples directory. These resembles building blocks or modules.

The oswatch-xxxx directories assemble some of the building blocks.

The oswatch-cts shows how the time is set with bluetooth, but also uses the button, the screen, the clock.

Oswatch-full would include everything.

For your own project you can choose the oswatch-template which suits you most.

3.2 Screen

The OLED screen can only display a few lines. This is used as a basis for screen layout.

Multiple screens can be displayed, but each screen has little info.

3.3 Touchscreen

Although the LVGL graphics library offers touchscreen support and the Pinetime watch has a touchscreen, the emphasis was put on the side button. Most watches have a button.

Selecting a screen and a parameter on the screen is done with the button instead of the touchscreen.

3.4 Heart rate sensor

Most heart rate sensors have a light sensor and an LED. Instead of trying to get a heart rate out of it. Emphasis was put on reading out the light sensor and controlling the LED.

This way my cheap oled watch can be used to read out LED signals, and thus be used as a communication device.

The LED cannot be controlled independently on the pinetime. Suppose you want to use the lightsensor without the led, it is not possible.

3.5 Bluetooth

Setting time and timers or other parameters is a nuisance on a small device. It is far more easier to set/read parameters on a linux box or android device and transfer them to the smartwatch.

The easiest would be to use bluetooth UART, this is not a part of the open source zephyr (apache 2 license) , but belongs to Nordic. (which supply their own version of zephyr)

The idea I will explore is to use 2 bluetoothservices, one to request and one to supply the value of a parameter.

Bluetooth is used to set the time (CTS) current time service.

3.6 Power saving

Power saving makes sense when the device is battery operated, which is probably not always the case. Hence, power saving is an option.

3.7 Updating over the air

FOTA : firmware updating over the air, is a useful option. The way firmware updates work, is to have two firmware images on the device. In the case of the cheap OLED device, there is no extra memory to store firmware images. Hence, FOTA is an option.

3.8 Motion sensor

The motion is used as a stepcounter, but could be used for other purposes as well. Knock sensor, position sensor, movement detection … emphasis is put on reading out the sensor rather than processing the data.

3.9 Drivers

Creating drivers takes a lot of time. Chances are that a specific driver does not exist for zephyr, or that the standard one does not work fully.

To overcome this problem a zephyr driver that resembles the needed driver is adapted and placed out of tree. In some cases I have not even bothered to change the name. The max30101 lightsensor is not present in the smartwatches I worked with, but it exists as a driver that can easily be adapted. In zephyr this means that the board definition files need to refer to this driver as well. Board definition has to be placed out of tree as well.

CHAPTER

FOUR

INSTALL ZEPHYR

4.1 update on 31-12-2020

Pinetime has become part of the standard zephyr distribution!

These days you can install zephyr and execute a pinetime sample!

```
west build -p -b pinetime_devkit0 samples/boards/pine64_pinetime
```

4.2 How to install zephyr

https://docs.zephyrproject.org/latest/getting_started/index.html

the documentation describes an installation process under Ubuntu/macOS/Windows

4.3 How to install the open source watch frame kit

The kit should work alongside the zephyr installation. Just get a copy of the "app" directory.

```
<work>  /app
     ├──/zephyr
     ├──.....
```

the app-directory contains the drivers and source code and(!) modified board definitions.

4.4 Linux rules

The development has been done on a virtual linux machine. For testing bluetooth functionality, I have used linux as well. In this manual you'll find a procedure to communicate to the device from a linux box. Updating the firmware wireless? Possible from a linux box!

Suppose you want to measure how many steps employees do a day? You could readout everything using a linux box, and handle the data using all the available tools of the box.

TIP : sometimes you run into trouble compiling: removing the build directory can help in that case

CHAPTER

FIVE

OUT OF TREE

A technique used in zephyr/samples/application-development, is “out of tree” development.

When you tinker with watches, you will soon find out that not all the drivers exist.

You can adapt existing zephyr drivers, but placing them within the zephyr repository could cause issues (upgrading zephyr).

The samples provided contain the board definition and the drivers within their directory outside the zephyr directory.

Have a look at the samples, on how it is done.

```
All the development, board definition files, drivers are contained in the "app"␣
↪directory.
Copy this directory alongside the zephyr rtos repository, and it should work.
No extra configuration needed.
You can use the latest Zephyr distribution.
```

5.1 Top Tip:

Adapting an existing driver is easier than writing one from scratch. In the Kconfig you just set another name : instead of SSD1306, you use SSD1306NEW. You might need to adapt the board definition file <watch>.dts as well. Zephyr stitches together parts based on labels. Make sure they match.

CHAPTER

SIX

DISPLAY

6.1 Display Types

There are three types of display included.

- a st7789 color display used in the pinetime
- a monochrome SSD1306 OLED display used in the desay D6
- a on-screen display SDL simulated on linux

6.1.1 Several methods of using the display

Included in the zephyr distribution are : - the Little Graphics Library (LVGL) - Character frame buffer (cfb)

For the pinetime the LVGL library might be a good option, since it has a big screen, colors and (!) is touch sensitive.

For the DS-D6 which has a monochrome OLED small display, the character frame buffer could be the best option.

6.1.2 The human eye

You can print something to the serial port, use the Segger RTT, but having a sample which displays something on the screen is the equivalent of blinking a LED.

Instead of the blinky blinking the LED, most samples include the screen.

CHAPTER

SEVEN

STARTING WITH SOME BASIC APPLICATIONS

The best way to get a feel of zephyr for the smartwatch, is to start building applications.

The watch framework is under /app.

The framework contains a clock, bluetooth, a procedure to upgrade over the air, cts ...

To reduce the complexity, samples are provided. Each sample contains a single feature of the framework.

7.1 Push the button

- A button for the posix simulation watch, is a lvgl push button.
- A button for the pinetime is a real button, which needs 2 (!) gpio ports
- A button for the Desay D6 is a real button, which needs 1 gpio port.

To complicate matters, a button on the pinetime can be both an lvgl button and a real gpio-enabled button.

Using a framework, which would suit these watches, needs to distinguish between the boards (use a condition with parameter : CONFIG_BOARD)

7.2 posix

7.2.1 Building and Running

The "native_posix_64" board is your own linux-box. This means that you can execute the code on your system. You do not need a smartwatch.

```
west build -p -b native_posix_64 samples/button/sdlbutton
```

running : ./build/zephyr/zephyr.exe

7.3 pinetime

7.3.1 Building and Running

```
west build -p -b pinetime_devkit1 samples/button/pinebutton
```

running : west flash

Reading out the button on the watch

```
The pinetime does have a button on the side.
The desay D6 has a touchbutton in front.
The virtual POSIX watch has a touchbutton
```

Note:: *The pinetime watch has a button out port (15) and button in port (13). You have to set the out-port high. Took me a while to figure this out...*

CHAPTER

EIGHT

LITTLEVGL BASIC SAMPLE

8.1 Overview

This sample application displays “Hello World” in the center of the screen and a counter at the bottom which increments every second.

LittlevGL is a free and open-source graphics library providing everything you need to create embedded GUI with easy-to-use graphical elements, beautiful visual effects and low memory footprint.

8.2 Simulation

In order to avoid uploading to check what the display looks like, there is a simple way to simulate this. I’ve tested this on Ubuntu 18.04 64bit. You’ll need the SDL2 library.

```
west build -p -b native_posix_64 samples/display/lvgl
```

after the building, you can find build/zephyr/zephyr.exe (and execute this to see display-layout)

the sample is provided as samples/display/lvgl-posix

8.3 Pinetime

The program has been modified to light up the background leds.

TIP: matching label : DISPLAY

```
Matching labels are necessary!
pinetime.conf:CONFIG_LVGL_DISPLAY_DEV_NAME="DISPLAY"
pinetime.overlay:               label = "DISPLAY"; (spi definition)
```

8.4 Building and Running

```
west build -p -b pinetime samples/lvgl
```

8.4.1 modifying the font size :

```
west build -t menuconfig
```

goto:

- additional libraries
- lvgl gui library

(look for fonts, and adapt according to your need)

8.4.2 apply changes of the changed config:

```
west build
```

(instead of west build -p (pristine) which wipes out your customisation)

8.5 References

https://docs.littlevgl.com/en/html/index.html

LittlevGL Web Page: https://littlevgl.com/

CHAPTER

NINE

REAL TIME CLOCK

The compile-time is used to set the initial time. (CMAKE file)

The elapsed time since the startup can be calculated.

Setting the exact time is done via bluetooth current time service.

9.1 Overview

9.2 References

CHAPTER

TEN

CURRENT TIME SERVICE

https://www.bluetooth.com/specifications/gatt/services/

https://www.bluetooth.com/specifications/gatt/characteristics/

- 0x1805 current time service
- 0x2A2B current time characteristic

10.1 Requirements:

You need :

- **a CTS server (use of bluez on linux explained)**
  - start the CTS service (python script: gatt-cts-server.py provided)
  - connect to the CTS client
- a CTS client (the oswatch)

10.2 BLE Peripheral CTS sample for zephyr

This example demonstrates the basic usage of the current time service. It is based on the https://github.com/Dejvino/pinetime-hermes-firmware.

It starts advertising it's UUID, and you can connect to it. Once connected, it will read the time from your CTS server (bluez on linux running the gatt-cts-server script in my case)

first build the image

```
$  west build -p -b pinetime_devkit1 oswatch-cts
```

10.3 Using bluez on linux to connect

The pinetime zephyr sample behaves as a peripheral:

```
- make sure your linuxbox has bluez running (type bluetoothctl if you're not sure)
- first of all start the cts service : python gatt-cts-server.py
- connect to the pinetime with bluetoothctl
```

Using bluetoothctl:

```
#bluetoothctl
[bluetooth]#scan on


[NEW] Device 60:7C:9E:92:50:C1 Zephyr Peripheral Sample Long
once you see your device
[blueooth]#connect 60:7C:9E:92:50:C1 (the device mac address as displayed)
```

10.4 Howto use Bluez on linux to set up a time service

Within the bluez source distribution there is an example GATT (Generic Attribute Profile)server. It advertises some standard service such as heart rate, battery ... Koen zandberg adapted this script, so it advertises the current time : https://github.com/bosmoment/gatt-cts/blob/master/gatt-cts-server.py

You might have to install extra packages:

```
apt-get install python-dbus
apt-get install python-gi
apt-get install python-gobject
```

10.5 Howto use Android to set up a time service

As soon as a device is bonded, Pinetime will look for a CTS server (Current Time Service) on the connected device. Here is how to do it with an Android smartphone running NRFConnect:

Build and program the firmware on the Pinetime Install NRFConnect (https://www.nordicsemi.com/Software-and-Tools/Development-Tools/nRF-Connect-for-desktop)

Start NRFConnect and create a CTS server : Tap the hamburger button on the top left and select "Configure GATT server" Tap "Add service" on the bottom Select server configuration "Current Time Service" and tap OK Go back to the main screen and scan for BLE devices. A device called "PineTime" should appear Tap the button "Connect" next to the PineTime device. It should connect to the PineTime and switch to a new tab. On this tab, on the top right, there is a 3 dots button. Tap on it and select Bond. The bonding process begins, and if it is successful, the PineTime should update its time and display it on the screen.

CHAPTER

ELEVEN

DRIVERS

11.1 configuring I2C

11.1.1 board level definitions

```
under boards/arm/pinetime are the board definitions
- pinetime.dts
- pinetime_defconfig


The sensors in the pinetime use the I2C bus.

&i2c1 {
        compatible = "nordic,nrf-twi";
        status = "okay";
        sda-pin = <6>;
        scl-pin = <7>;

      };
```

11.1.2 definition on project level

In the directory of a sample, you will find a prj.conf file. Here you can set values specific for you project/sample.

```
In the "prj.conf" file we define the sensor (eg adxl372)

CONFIG_STDOUT_CONSOLE=y
CONFIG_LOG=y
CONFIG_I2C=y
CONFIG_SENSOR=y
CONFIG_ADXL372=y
CONFIG_ADXL372_I2C=y
CONFIG_SENSOR_LOG_LEVEL_WRN=y
```

note: this gets somehow merged (overlayed) with the board definition pinetime_defconfig

11.2 SSD1306 modification

11.2.1 board level definitions

The desay D6 OLED display gets it power trough a GPIO pin.

In the board definition file one can set it high.

Some minor adjustment in board.c was needed.

```
under boards/arm/ds_d6 are the board definitions
- ds_d6.dts
- ds_d6_defconfig



vdd_pwr: vdd-pwr-ctrl {
        compatible = "regulator-fixed";
        label = "vdd-pwr-ctrl";
        regulator-name = "vdd-pwr-ctrl";
        enable-gpios = <&gpio0 26 GPIO_ACTIVE_HIGH>;
        regulator-boot-on;
};
```

The driver has been placed “out of tree”. It is a modified ssd1306 driver. The ssd1306 behaved somewhat differently. It was renamed to ssd1306new to avoid confusion. In the board definition file the label was adapted.

```
ssd1306new@0 {
        compatible = "solomon,ssd1306fb";
        spi-max-frequency = <80000000>;
        label = "SSD1306NEW";
        reg = <0>;
        segment-offset = <0>;
        page-offset = <0>;
        display-offset = <0>;
        multiplex-ratio = <31>;
        height = <32>;
        width = <128>;
        segment-remap;
        prechargep = <0xF1>;
        reset-gpios = <&gpio0 4 0>;
        data_cmd-gpios = <&gpio0 28 0>;
};
```

11.2.2 definition config

```
In the "ds_d6_defconfig" file we define the driver


 CONFIG_LVGL_DISPLAY_DEV_NAME="SSD1306NEW"
 CONFIG_SSD1306NEW=y
```

11.3 sensors on the I2C bus

0x18: Accelerometer: BMA423-DS000 https://github.com/BoschSensortec/BMA423-Sensor-API

0x44: Heart Rate Sensor: HRS3300_Heart

0x15: Touch Controller: Hynitron CST816S Touch Controller

11.4 Bosch BMA421

this driver does not exist, so it has been created. Still work in progress ....

```
west build -p -b pinetime samples/gui/lvaccel
```

11.4.1 Overview

BMA421 is not a part number available to the general public, and therefore all the supporting documentation and design resources are neither discussed in public forums, nor disclosed on GitHub.

CHIP_ID=0X11 (so the Bosch BMA423 drivers need to be adapted)

The Bosch documentation on the bma423 seems to apply to the bma421.

11.4.2 Requirements

for this sensor does not exist any driver, so here's what I did to create one under zephyr

adapt CMakeLists.txt

~/zephyrproject-2/zephyr/drivers/sensor add_subdirectory_ifdef(CONFIG_BMA280 bma280) add_subdirectory_ifdef(CONFIG_BMA421 bma421)

adapt Kconfig

~/zephyrproject-2/zephyr/drivers/sensor

add yaml file

~/zephyrproject-2/zephyr/dts/bindings/sensor cp bosch,bma280-i2c.yaml bosch,bma421-i2c.yaml

edit KConfig

source "drivers/sensor/bma280/Kconfig" source "drivers/sensor/bma421/Kconfig"

source "drivers/sensor/bmc150_magn/Kconfig"

source "drivers/sensor/bme280/Kconfig"

create driver

see under drivers/sensor/bma421

complement the pinetime.dts file with the following (under samples/sensor/bma280)

```
&i2c1 {
        bma421@18 {
                    compatible = "bosch,bma421";
                    reg = <0x18>;
                    label = "BMA421";
                    int1-gpios = <&gpio0 8 0>;
                   };
       };
```

Create a file: */dts/bindings/sensor/bosch,bma421-i2c.yaml*. Which contains:

```
compatible: "bosch,bma421"
include: i2c-device.yaml
properties:
       int1-gpios:
       type: phandle-array
       required: false
```

11.4.3 Building and Running

11.4.4 Todo

- the driver is interrupt driven as well – need to test software
- the sensor has algorithm for steps – read out register
- temperature some attempt has been made, but . . . (OK, temp can be read)

11.4.5 References

Bosch has documented the BMA423 very well. I kind of hope it will apply to the bma421.

A mechanism to adapt the 0x5E register is provided. (burst read/write)

All kind of parameters can be set to trigger an interrupt. (e.g. number of steps taken : think of the 10000 steps threshold)

11.5 HYNITRON CST816S

Zephyr has evolved and now there is something that serve as a touchscreen device. (KSCAN_TOUCH)

the board definition file has been adapted slightly, using the focaltech ft5336 as a touch_controller. A minor change in this driver is enough to get data from the hynitron cst816S.

The big advantage : almost standard zephyr install! Unfortunately I cannot get the KSCAN_TOUCH working in an out-of-tree setup.

I have created the renamed ft9999 driver.

```
west build -p -b pinetime_devkit0 samples/display/lvgl
```

this driver does not exist, so it has been created. Still work in progress ....

11.5.1 Overview

(this is the old way of doing things ....)

the Hynitron cst816s is a touchscreen. Zephyr doesn't handle touchscreens yet. In order to investigate, the touchscreen driver has been created as a sensor. In fact it senses your finger ;)

11.5.2 Requirements

for this sensor does not exist any driver, so here's what I did to create one under zephyr

adapt CMakeLists.txt adapt Kconfig add yaml file

create driver

The driver reads only one position. Multitouch is possible, but the screen is small....

see under drivers/sensor/cst816s

have a look at the pinetime.dts (under board/arm/pinetime) file:

```
&i2c1 {
        cst816s@15 {
                    compatible = "hynitron,cst816s";
                    reg = <0x15>;
                    label = "CST816S";
                    };
       };
```

11.5.3 Building and Running

There are two samples :

- samples/gui/lvtouch (graphical)
- samples/sensor/cst816s (no graphics)

11.5.4 Todo

The graphical sample doesn't handle interrupts.

11.5.5 References

There is little available for this touchscreen.

11.6 HX HRS3300

this driver does not exist, so it has been created. Still work in progress ....

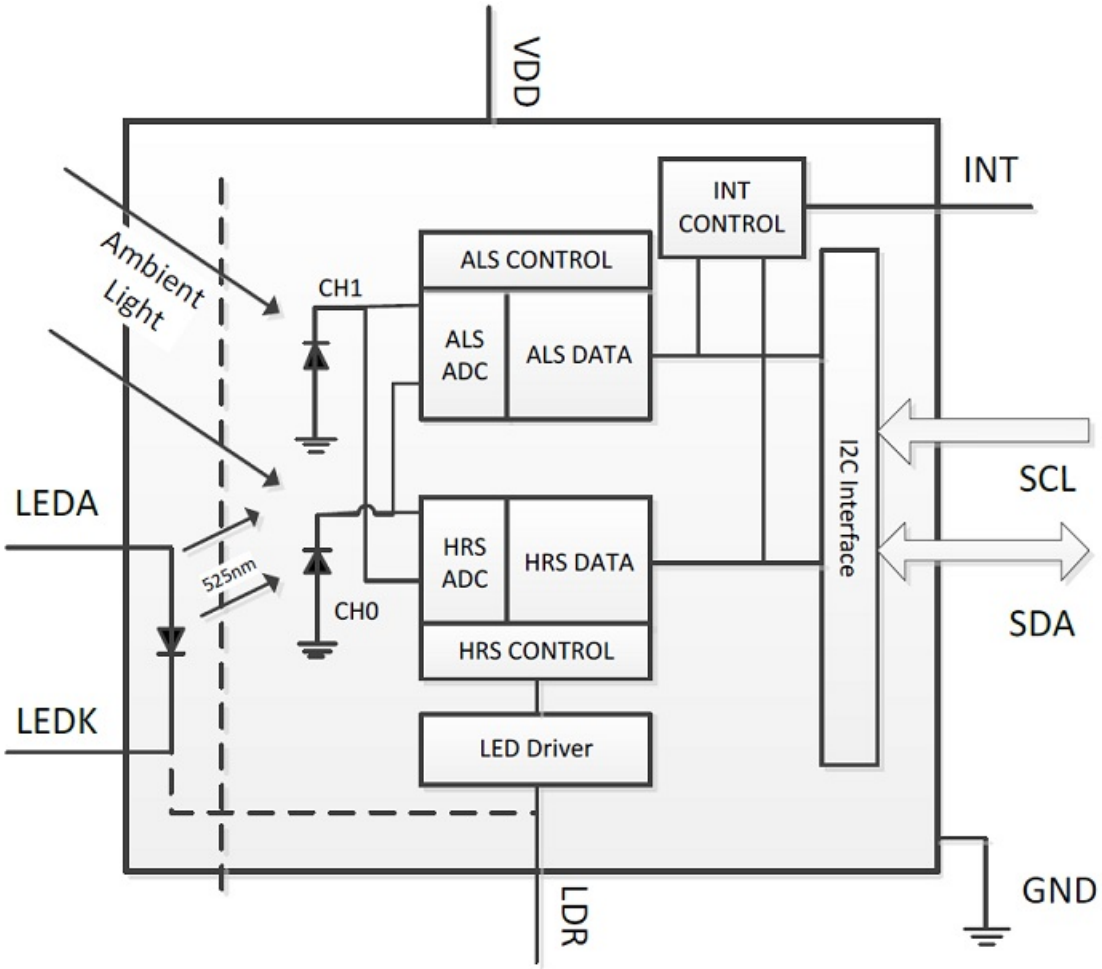
there is a sample in this repository which can be copied to the zephyr samples directory

```
west build -p -b pinetime samples/sensor/hrs3300
```

11.6.1 Overview

The HX HRS3300 sensor is a heart rate sensor, it produces 2 values: ALS and HRS. Ambient LIGHT SENSOR and HEART RATE SENSOR. Which have to be processed by an algorithm. I have no knowledge of a good open source algorithm yet.

I have used the settings of an arduino port of this library.



11.6.2 Requirements

for this sensor does not exist any driver, so here's what I did to create one under zephyr

adapt CMakeLists.txt

~/zephyrproject-2/zephyr/drivers/sensor add_subdirectory_ifdef(CONFIG_HRS3300 hrs3300)

adapt Kconfig

~/zephyrproject-2/zephyr/drivers/sensor

add yaml file

~/zephyrproject-2/zephyr/dts/bindings/sensor add hx,hrs3300.yaml

edit KConfig

source “drivers/sensor/hrs3300/Kconfig”

create driver

see under drivers/sensor/hrs3300

complement the pinetime.dts file with the following (under samples/sensor/bma280)

```
&i2c1 {
        hrs3300@44 {
                   compatible = "hx,hrs3300";
                   reg = <0x44>;
                   label = "HRS3300";
                   };
       };
```

Create a file: */dts/bindings/sensor/hx,hrs3300.yaml*. Which contains:

```
compatible: "hx,hrs3300"
properties:
```

11.6.3 Building and Running

11.6.4 Todo

- algorithm for heartrate
- power saving
- switching off/on mechanism

11.6.5 References

HRS3300 Heart Rate Sensor.pdf https://github.com/atc1441/HRS3300-Arduino-Library

11.7 Serial Nor Flash

```
west build -p -b pinetime samples/drivers/spi_flash -DCONF=prj.conf
```

11.7.1 Overview

This sample application should unlock the serial nor flash memory. This can be very useful to store e.g. background for the watch.

compilation problematic ....

/root/zephyrproject/zephyr/samples/drivers/spi_flash/src/main.c:17:22: error: 'DT_INST_0_JEDEC_SPI_NOR_LABEL' undeclared (first use in this function); did you mean 'DT_INST_0_NORDIC_NRF_RTC_LABEL'?

Turns out this is some problem with the board definition file.

I found it to be very useful to consult the generated dts file. Here you can check if everything is present.

Guess the dts-file has to be well intended.(structured)

**TIP: consult the generated dts board file **

consulting the generated board definition file

```
vi /root/zephyrproject/zephyr/build/zephyr/include/generated/generated_dts_board.conf
```

11.7.2 Requirements

complement the pinetime.dts file with the following (under spi) #define JEDEC_ID_MACRONIX_MX25L64 0xC22017

```
&spi0 {
  compatible = "nordic,nrf-spi";
  status = "okay";
  sck-pin = <2>;
  mosi-pin = <3>;
  miso-pin = <4>;
  cs-gpios = <&gpio0 27 0>,<&gpio0 5 0>;
  st7789v@0 {
          compatible = "sitronix,st7789v";
          label = "DISPLAY";
          spi-max-frequency = <8000000>;
          reg = <0>;
          cmd-data-gpios = <&gpio0 18 0>;
          reset-gpios = <&gpio0 26 0>;
          width = <240>;
          height = <240>;
          x-offset = <0>;
          y-offset = <0>;
          vcom = <0x19>;
          gctrl = <0x35>;
          vrhs = <0x12>;
          vdvs = <0x20>;
          mdac = <0x00>;
          gamma = <0x01>;
          colmod = <0x05>;
          lcm = <0x2c>;
          porch-param = [0c 0c 00 33 33];
          cmd2en-param = [5a 69 02 01];
          pwctrl1-param = [a4 a1];
```

(continues on next page)

(continued from previous page)

```
        pvgam-param = [D0 04 0D 11 13 2B 3F 54 4C 18 0D 0B 1F 23];
        nvgam-param = [D0 04 0C 11 13 2C 3F 44 51 2F 1F 1F 20 23];
        ram-param = [00 F0];
        rgb-param = [CD 08 14];

};

mx25r64: mx25r6435f@1 {
        compatible = "jedec,spi-nor";
        reg = <1>;
        spi-max-frequency = <1000000>;
        label = "MX25R64";
        jedec-id = [0b 40 16];
        size = <67108864>;
        has-be32k;
};
```

11.7.3 Building and Running

```
west build -p -b pinetime samples/drivers/spi_flash
```

11.7.4 Todo

- detect ID memory : it is not the macronix one as suggestion on the pinetime website

I found the following : jedec-id = [0b 40 16]; (OK: can execute sample program)

- create working board definition (OK: see above)

11.7.5 References

http://files.pine64.org/doc/datasheet/pinetime/MX25L6433F,%203V,%2064Mb,%20v1.6.pdf

11.8 Battery

the samples just gets an analog reading from the battery

```
west build -p -b pinetime samples/sensor/battery
```

11.8.1 Overview

The battery level is measured on port 31, trough an ADC conversion.

voltage = (value * 6)/1024 percentage remaining ((voltage - 3.55)*100)*3.9;

A module should be able to report battery status in millivolts and charge level in percentage. Additionally, it should notify when external power is connected and when battery is being charged. Module will use adc (saadc peripheral) to measure battery voltage and gpio driver to monitor charge indication pin (pin 0.12) and power presence pin (0.19). Battery voltage can be in range from 3.0V - 4.2V (?). Unfortunately, internal reference (0.6V) can only be used for

voltages up to 3.6V (due to minimal gain of 1/6). VDD/4 reference can be used with 1/6 gain to measure voltages up to 4.95V. Test is needed to check how accurate is VDD as reference. Discharge curve (https://forum.pine64.org/showthread.php?tid=8147) will be used to calculate charge level in percent. Things to consider: saadc periodical calibration (spec suggests calibration if temperature changes by 10'C) inaccuracy of results: oversampling? never report higher level than before (if charge not connected), etc.

11.8.2 Todo

check pin when charging

11.8.3 References

https://forum.pine64.org/showthread.php?tid=8147

11.9 Watchdog

```
west build -p -b pinetime samples/drivers/watchdog
```

11.9.1 Overview

Once the pinetime is closed and on your wrist, you still want access.

see : *Firmware Over The Air (FOTA)*

Suppose you upload a application which contains a bug, the watch freezes, and ... you will have to open it up, connect the SWD ...

Here comes the watchdog:

- you launch the watchdog
- you launch the application
- the application feeds the watchdog
- if it cannot feed the watchdog, reset to fota, and you can upload another better version ...

11.9.2 Todo

testing

11.9.3 References

CHAPTER

TWELVE

BLUETOOTH (BLE)

Bluetooth is a very nice feature, it lets you exchange data wireless and (!) update firmware wireless.

The PineTime uses a Nordic nrf52832 chip, which has BLE functionality build into it.

To test, you can compile a standard application : Eddy Stone.

12.1 A word on bluetooth-serial communication

Smartwatch manufacturers usually supply you with a smartwatch app. This app can communicate trough bluetooth, and get/send data to the watch.

Unfortunately this is not a standard Zephyr feature!

You could use nRF Connect SDK (often referred as NCS),it is Nordic Semiconductor SDK based on zephyr. It has additional features that could be useful for pinetime, namely bluetooth modules and bluetooth services. Important to mention that even though, NCS forks zephyr it is kept close to it and it is regularly updated to latest zephyr (typically every 2-3 weeks).

NCS has no impact on build framework and overall user experience. The only downside I can think of is that NCS is based on older version of zephyr (~2 weeks behind).

The main reason why i'm bringing it up is bluetooth shell. Zephyr has very nice shell module with multiple transports (UART, RTT). NCS extends it with bluetooth transport (using Nordic Uart Service) and host tool for using it. There is an application for linux from which you can use shell over bluetooth, get logs, etc.. It is really cool and can be very useful when playing with pinetime where you can get logs or control/tune things with only wireless connection to your PC. Here is a demonstration of bluetooth console used in one of nordic reference kits: https://www.youtube.com/watch?v=3KzTfr6S4pg&t= . It's based on nRF5 SDK (not zephyr) but bluetooth shell (and PC tool) was taken from there.

The Nordic UART Service (NUS) shell transport sample demonstrates how to use the receive shell commands from a remote device.

NCS is using BSD-5-Clause-Nordic license (https://github.com/NordicPlayground/fw-nrfconnect-nrf/blob/master/LICENSE)

Zephyr RTOS and the samples I use/create use the Apache License 2.0. A permissive license whose main conditions require preservation of copyright and license notices. Contributors provide an express grant of patent rights. Licensed works, modifications, and larger works may be distributed under different terms and without source code.

12.2 Eddy Stone

see: bluetooth-eddystone-sample

Note: compile the provided example, so a build directory gets created

```
$ west build -p -b pinetime_devkit0 samples/bluetooth/eddystone
```

```
this builds an image, which can be found under the build directory
```

12.3 Using the created bluetooth sample:

I use linux with a bluetoothadapter 4.0. You need to install bluez.

```
#bluetoothctl
[bluetooth]#scan on
```

And your Eddy Stone should be visible.

It is the peripheral which advertises, and the central that reads the data.

A sample which advertises a heartrate :

```
west build -p -b pinetime_devkit0 samples/bluetooth/peripheral_hr
```

you could use your smartphone or bluez on linux to read out the heartrate.

Or if you have another watch, the central will connect to the peripheral and read out the heartrate.

```
west build -p -b pinetime_devkit0 samples/bluetooth/central_hr
```

12.3.1 the no-bluetooth, no-watch approach : nrf52_bsim

Suppose you have no watch, no development board, nor any bluetooth dongles? You can still test your bluetooth enabled application. https://docs.zephyrproject.org/latest/boards/posix/nrf52_bsim/doc/index.html

We follow the same logic, but this time we specify the nrf52_bsim board.

```
west build -p -b nrf52_bsim samples/bluetooth/peripheral_hr
west build -p -b nrf52_bsim samples/bluetooth/central_hr
cp build/zephyr/zephyr.exe  ${BSIM_OUT_PATH}/bin/bs_nrf52_bsim_samples_bluetooth_
↪central_hr
```

Once compiled you can execute both the peripheral and central firmware, and(!) you have to start the bluetooth simulation. By starting each application in its own terminal, you can keep an eye on the output.

```
${BSIM_OUT_PATH}/bin/bs_nrf52_bsim_samples_bluetooth_central_hr -s=trial_sim -d=1

 zephyr/build/zephyr/zephyr.exe -s=trial_sim -d=0
 ${BSIM_OUT_PATH}/bin/bs_2G4_phy_v1 -s=trial_sim -D=2 -sim_length=10e6
```

12.3.2 the no-watch approach : simulation on a laptop

how to activate bluetooth?

VBOX running ubuntu (first deactivate driver in windows) (CTRL home – select usb – (intel in my case)) – this lets you select the integrated bluetooth module of your laptop

```
hciconfig hci0 down

west build -p -b native_posix_64 samples/bluetooth/peripheral_hr
./build/zephyr/zephyr.exe --bt-dev=hci0
```

Now you can connect your smartphone to the posix_64 bluetooth device!

Or, with a second bluetooth interface (eg dongle) .. code-block:: console

> bluetoothctl [bluetooth]# devices Device C6:78:40:29:EC:31 Zephyr Heartrate Sensor Device C9:16:85:ED:B6:4E DS-D6 b64e Device C8:B7:89:A9:B0:C9 Espruino-107 b0c9 Device 00:1A:7D:DA:71:0B posix_64
>
> [bluetooth]# info 00:1A:7D:DA:71:0B Device 00:1A:7D:DA:71:0B (public) Name: posix_64 Alias: posix_64 Paired: no Trusted: no Blocked: no Connected: no LegacyPairing: no UUID: Device Information (0000180a-0000-1000-8000-00805f9b34fb) UUID: Current Time Service (00001805-0000-1000-8000-00805f9b34fb)

If you have a smartphone, you can download the nrf utilities app from nordic.

12.4 Bluez

With Bluez on linux you can investigate the bluetoothservices, using bluetoothctl:

```
#bluetoothctl
[bluetooth]#scan on


[NEW] Device 60:7C:9E:92:50:C1 Zephyr Peripheral Sample Long
once you see your device
[blueooth]#connect 60:7C:9E:92:50:C1 (the device mac address as displayed)

then you can already see the services
```

12.4.1 using bluez to write something to the device

Characteristic /org/bluez/hci0/dev_74_71_4B_D5_18_21/service001f/char0023 00002a38-0000-1000-8000-00805f9b34fb Body Sensor Location

```
[dsd6]# select-attribute /org/bluez/hci0/dev_74_71_4B_D5_18_21/service001f/char0023
write <data=0x1>
```

same thing with the app from nordic, you could try to connect and display value of e.g. heart rate

12.5 using Python to read out bluetoothservices

In this repo you will find a python script : readbat.py In order to use it you need bluez on linux and the python *bluepy* module.

It can be used in conjunction with the peripheral bluetooth demo. It just reads out the battery level, and prints it.

```
import binascii
from bluepy.btle import UUID, Peripheral

temp_uuid = UUID(0x2A19)

p = Peripheral("60:7C:9E:92:50:C1", "random")

try:
   ch = p.getCharacteristics(uuid=temp_uuid)[0]
   print binascii.b2a_hex(ch.read())
finally:
    p.disconnect()
```

CHAPTER

THIRTEEN

BLUETOOTH NOTIFICATION

As mentioned earlier, I searched for ways to transfer data to, and(!) from the device. Bluetooth data exchange might be a standard feature of Android, Apple ...

On a linuxbox there are ways to do this as well!

In the zephyr samples, I found a sample for Indication. I created a sample for Notification. (samples/bluetooth/Peripheral-notification)

13.1 What?

The BLE standard defines two ways to transfer data from the server to the client: notification and indication.

Notification don't need acknowledgement, so it is faster. Hence, server does not know if the message reached the client....

13.2 How?

Notification has to be enabled.

Linuxbox (bluez) writes "enable" to the watch notification characteristic.

Once Nofication is enabled the watch sends continuously messages : the maximum data payload size defined by the specification in each message is 20 bytes.

13.3 Bluez

on linux you could use "bluetoothctl"

There is an example for heartrates.

```
#bluetoothctl

connect MAC-device

menu gatt
select-attribute 00002a37-0000-1000-8000-00805f9b34fb
notify on

if all goes well, the watch starts sending heartrates
```

(continues on next page)

(continued from previous page)

```
in case it stops:

notify off
notify on
```

13.4 A word on attributes

The data that the server exposes is structured as attributes.

Attribute type (Universally Unique Identifier or UUID) This is a :

- 16-bit number
- or 128-bit number

The 16-bit number is converted to 128-bit by means of a 128-bit base UUID: 00000000-0000-1000-8000-00805F9B34FB.

These 16-bit numbers are standardised. On each device the temperature measurement value is 0x2A1C.

Because I create a custom service to exchange data I use a custom attribute type, also sometimes referred to as vendor-specific UUID. BLE_CHARACTERISTIC_UUID= “12345678-1234-5678-1234-56789abcdef1”

13.5 Attribute Handle

This is a 16-bit value that the server assigns to each of its attributes — think of it as an address. This value is used by the client to reference a specific attribute and is guaranteed by the server to uniquely identify the attribute during the life of the connection between two devices. The range of handles is 0x0001-0xFFFF, where the value of 0x0000 is reserved.

13.6 Python

The scripts are provided. (scan+not.py) But here is a short explanation on how to enable notifications.

On linux one could use the Gatttool from the commandprompt.

Another option is Python!

In this manual the module Bluepy is used : https://github.com/IanHarvey/bluepy.

(custom service) BLE_SERVICE_UUID =”12345678-1234-5678-1234-56789abcdef0” BLE_CHARACTERISTIC_UUID= “12345678-1234-5678-1234-56789abcdef1”

ch = ble_service.getCharacteristics()[0] ch.valHandle (this gets you the handle to the notification characteristic

Writing x01 is the protocol for all BLE notifications. # However .... you need to write it to the handle + 1

dev.writeCharacteristic(ch.valHandle+1, b”x01x00”, withResponse=True)

CHAPTER

FOURTEEN

FIRMWARE OVER THE AIR (FOTA)

14.1 Wireless Device Firmware Upgrade

14.1.1 Overview

In order to perform a FOTA (firmware over the air) update on zephyr you need 2 basic components:

- MCUboot (a bootloader)
- SMP Server (a bluetooth service)

14.2 MCUboot with zephyr

Clone MCUBOOT for zephyr from github. Install additional packages required for development with mcuboot:

```
cd ~/mcuboot  # or to your directory where mcuboot is cloned
pip3 install --user -r scripts/requirements.txt
```

To build MCUboot, create a build directory in boot/zephyr, and build it as follows:

```
cd boot/zephyr
mkdir build && cd build
cmake -GNinja -DBOARD=pinetime ..
ninja
```

After building the bootloader, the binaries should reside in *build/zephyr/zephyr.{bin,hex,elf}*.

This image can be flashed as a normal application.

Some additional configuration is required to build applications for MCUboot.

This is handled internally by the Zephyr configuration system and is wrapped in the *CONFIG_BOOTLOADER_MCUBOOT* Kconfig variable, which must be enabled in the application's *prj.conf* file.

The Zephyr *CONFIG_BOOTLOADER_MCUBOOT* configuration option [documentation](http://docs.zephyrproject.org/reference/kconfig/CONFIG_BOOTLOADER_MCUBOOT.html) provides additional details regarding the changes it makes to the image placement and generation in order for an application to be bootable by MCUboot.

In order to upgrade to an image (or even boot it, if *MCUBOOT_VALIDATE_PRIMARY_SLOT* is enabled), the images must be signed.

To make development easier, MCUboot is distributed with some example keys. It is important to stress that these should never be used for production, since the private key is publicly available in this repository. See below on how to make your own signatures.

Images can be signed with the *scripts/imgtool.py* script. It is best to look at *samples/zephyr/Makefile* for examples on how to use this.

Since the bootloader is already in place, you cannot flash your application.bin to 0x00000.

Eg. in openocd : program application.bin 0x0c000. (which corresponds to the flash layout of slot 0)

These images can also be marked for upgrade, and loaded into the secondary slot, at which point the bootloader should perform an upgrade.

14.3 Partitions

```
have a look at boards/arm/pinetime/pinetime.dts
```

14.3.1 Defining partitions for MCUboot

The first step required for Zephyr is making sure your board has flash partitions defined in its device tree. These partitions are:

- *boot_partition*: for MCUboot itself
- *image_0_primary_partition*: the primary slot of Image 0
- *image_0_secondary_partition*: the secondary slot of Image 0
- *scratch_partition*: the scratch slot

The flash partitions are defined in the pinetime boards folder, in a file named *boards/arm/pinetime/pinetime.dts*.

14.3.2 Using NOR flash in partitions

The flash space on the Nordic nrf52 is 512K. Basically with the partitioning you end up with less space for your program.

As the pinetime has an extra spi nor flash chip, we can use this.

The flash-layout can be modified so as 1 chunk is on system flash and 1 chunk is on SPI NOR flash. This way the space for your firmware remains almost the same.

```
chosen {
        zephyr,flash = &flash0;
        //zephyr,flash = &flash1;
        zephyr,code-partition = &slot0_partition;
};


Rename the SPI JEDEC NOR Flash definition to : flash1

flash1: mx25r6435f@1 {
        compatible = "jedec,spi-nor";
        reg = <1>;
        spi-max-frequency = <80000000>;
        label = "MX25R64";
        jedec-id = [0b 40 16];
        size = <67108864>;
        has-be32k;
```

(continues on next page)

(continued from previous page)

```
        erase-block-size = <4096>;
        write-block-size = <4>;
};



 &flash0 {
        /*
         * For more information, see:
         * http://docs.zephyrproject.org/latest/guides/dts/index.html#flash-partitions
         */
        partitions {
                compatible = "fixed-partitions";
                #address-cells = <1>;
                #size-cells = <1>;

                boot_partition: partition@0 {
                        label = "mcuboot";
                        reg = <0x00000000 0xc000>;
                };
                slot0_partition: partition@c000 {
                        label = "image-0";

                change the size of partition 0 from 0x32000 to 0x64000

                        reg = <0x0000C000 0x64000>;
                };

                move slot1 partition to &flash1

                //slot1_partition: partition@3e000 {
                //      label = "image-1";
                //      reg = <0x0003e000 0x32000>;
                //};
                scratch_partition: partition@70000 {
                        label = "image-scratch";
                        reg = <0x00070000 0xa000>;
                };
                storage_partition: partition@7a000 {
                        label = "storage";
                        reg = <0x0007a000 0x00006000>;
                };
        };
 };

 &flash1 {
        partitions {
                compatible = "fixed-partitions";
                #address-cells = <1>;
                #size-cells = <1>;
                slot1_partition: partition@3e000 {
                        label = "image-1";
                        reg = <0x00000000 0x64000>;
                };
        };
 };
```

14.4 Signing an application

In order to improve the security, only signed images can be uploaded.

There is a public and private key. The Bootloader is compiled with the public key. Each time you want to upload firmware, you have to sign it with a private key.

NOTE: it is important to keep the private key hidden

14.4.1 Generating a new keypair

Generating a keypair with imgtool is a matter of running the keygen subcommand:

```
$ ./scripts/imgtool.py keygen -k mykey.pem -t rsa-2048
```

14.4.2 Extracting the public key

The generated keypair above contains both the public and the private key. It is necessary to extract the public key and insert it into the bootloader.

```
$ ./scripts/imgtool.py getpub -k mykey.pem
```

This will output the public key as a C array that can be dropped directly into the *keys.c* file.

14.4.3 Example

sign the compiled zephyr.bin firmware with the root-rsa-2048.pem, private key:

```
imgtool.py sign --key ../../root-rsa-2048.pem \
    --header-size 0x200 \
    --align 8 \
    --version 1.2 \
    --slot-size 0x60000 \
    ../mcuboot/samples/zephyr/build/ds_d6/hello1/zephyr/zephyr.bin \
    signed-hello1.bin
```

14.5 SMP Server Sample

14.5.1 Overview

This sample application implements a Simple Management Protocol (SMP) server. SMP is a basic transfer encoding for use with the MCUmgr management protocol.

This sample application supports the following mcumgr transports by default:

- Shell
- Bluetooth

14.5.2 Requirements

In order to communicate with the smp server sample installed on your pinetime, you need mcumgr.

Here is a procedure to install mcumgr on a raspberry pi (or similar)

It is written in the go-language. You need to adapt the path : PATH=$PATH:/root/go/bin.

14.5.3 Building and Running

The sample will let you manage the pinetime over bluetooth. (via SMP protocol)

There are slot0 and slot1 which can both contain firmware.

Suppose you switch from slot0 to slot1, you still want to be able to communicate.

So both slots need smp_svr software!

Step 1: Build smp_svr

`smp_svr` can be built for the nRF52 as follows:

NOTE: to perform a firmware update over the air, you have to build a second sample

Step 2: Sign the image

Using MCUboot's `imgtool.py` script, sign the `zephyr.(bin|hex)` file you built in Step 3. In the below example, the MCUboot repo is located at `~/src/mcuboot`.

```
~/src/mcuboot/scripts/imgtool.py sign \
     --key ~/src/mcuboot/root-rsa-2048.pem \
     --header-size 0x200 \
     --align 8 \
     --version 1.0 \
     --slot-size <image-slot-size> \
     <path-to-zephyr.(bin|hex)> signed.(bin|hex)
```

The above command creates an image file called `signed.(bin|hex)` in the current directory.

Step 3: Flash the smp_svr image

Upload the bin-file from Step 2 to image slot-0. For the pinetime, slot-0 is located at address `0xc000`.

```
in openocd : program zephyr.bin 0xc000
```

Step 4: Run it!

Note: If you haven't installed `mcumgr` yet, then do so by following the instructions in the mcumgr_cli section of the Management subsystem documentation.

The `smp_svr` app is ready to run. Just reset your board and test the app with the `mcumgr` command-line tool's `echo` functionality, which will send a string to the remote target device and have it echo it back:

```
sudo mcumgr --conntype ble --connstring ctlr_name=hci0,peer_name='Zephyr' echo hello
hello
```

Step 5: Device Firmware Upgrade

Now that the SMP server is running on your pinetime, you are able to communicate with it using *mcumgr*.

You might want to test "OTA DFU", or Over-The-Air Device Firmware Upgrade.

To do this, build a second sample (following the steps below) to verify it is sent over the air and properly flashed into slot-1, and then swapped into slot-0 by MCUboot.

```
* Build a second sample
* Sign the second sample
* Upload the image over BLE
```

Now we are ready to send or upload the image over BLE to the target remote device.

```
sudo mcumgr --conntype ble --connstring ctlr_name=hci0,peer_name='Zephyr' image␣
↪upload signed.bin
```

If all goes well the image will now be stored in slot-1, ready to be swapped into slot-0 and executed.

Note: At the beginning of the upload process, the target might start erasing the image slot, taking several dozen seconds for some targets. This might cause an NMP timeout in the management protocol tool. Use the `-t <timeout-in-seconds` option to increase the response timeout for the `mcumgr` command line tool if this occurs.

List the images

We can now obtain a list of images (slot-0 and slot-1) present in the remote target device by issuing the following command:

```
sudo mcumgr --conntype ble --connstring ctlr_name=hci0,peer_name='Zephyr' image list
```

This should print the status and hash values of each of the images present.

Test the image

In order to instruct MCUboot to swap the images we need to test the image first, making sure it boots:

```
sudo mcumgr --conntype ble --connstring ctlr_name=hci0,peer_name='Zephyr' image test
↪<hash of slot-1 image>
```

Now MCUBoot will swap the image on the next reset.

Reset remotely

We can reset the device remotely to observe (use the console output) how MCUboot swaps the images:

```
sudo mcumgr --conntype ble --connstring ctlr_name=hci0,peer_name='Zephyr' reset
```

Upon reset MCUboot will swap slot-0 and slot-1.

You can confirm the new image and make the swap permanent by using this command:

```
sudo mcumgr --conntype ble --connstring ctlr_name=hci0,peer_name='Zephyr' image␣
↪confirm
```

Note that if you try to send the very same image that is already flashed in slot-0 then the procedure will not complete successfully since the hash values for both slots will be identical.

CHAPTER

FIFTEEN

SAMPLES

15.1 Battery OSWatch Framework

15.1.1 Todo

Not functional yet... WIP (work in progress)

15.1.2 Overview

This is an opensource watch framework. The same software can run in simulation (SDL) on linux, on pinetime and on the ds_d6 oled watch.

This frame can be used if you need power management (PM).

The application keeps track of the battery, and is frugal on energy.

15.1.3 Requirements

no requirements since it can run without board (simulation)

15.1.4 Building and Running

west build -p -b native-posix_64 oswatch

15.1.5 References

15.2 OSWatch bluetooth setting parameters

15.2.1 Overview

This is an opensource watch framework. The same software can run in simulation (SDL) on linux, on the pinetime and on the ds_d6 oled watch.

15.2.2 Bluetooth

This application adds bluetoothfunctionality to update 6 parameters. One should be able to read out and set these parameters.

15.2.3 Requirements

no requirements since it can run without board (simulation)

15.2.4 Building and Running

west build -p -b native-posix_64 oswatch-bt

15.2.5 References

15.3 Setting up an alarm/calendar event

15.3.1 Overview

This sample sets up a calendar event.

Time and date (calendar event) are read from a gatt-server.

Once the event is due, the watch blinks the screen and vibrates.

15.3.2 Requirements

A bluetooth gatt-server. (this is supplied : using bluez on linux)

Included is a python-script : gatt-calendar-server.py, for use with bluez on linux.

```
python3 gatt-calendar-server.py
```

you need to connect to the watch in order, to have it read the time.

```
bluetoothctl
scan on (gets you list of bluetoothdevices)
connect D5:A0:A2:6D:A0:D5 (pick the pinetime - macaddress)
```

15.3.3 Building and Running

west build -p -b pinetime_devkit1 oswatch-calendar

15.3.4 References

15.4 Current Time Setting OSWatch Framework

15.4.1 Overview

This is an opensource watch framework. The same software can run in simulation (SDL) on linux, on pinetime and on the ds_d6 oled watch.

This sample reads/sets the time, using bluetooth CTS.

15.4.2 Requirements

A bluetooth CTS server. This can be a smartphone.

Included is a python-script : gatt-cts-server.py, for use with bluez on linux.

```
python3 gatt-cts-server.py
```

you need to connect to the watch in order, to have it read the time.

```
bluetoothctl
scan on (gets you list of bluetoothdevices)
connect D5:A0:A2:6D:A0:D5 (pick the pinetime - macaddress)
```

15.4.3 Building and Running

west build -p -b native-posix_64 oswatch-cts

15.4.4 References

15.5 FOTA OSWatch Framework

15.5.1 Todo

This is not functional yet

15.5.2 Overview

This is an opensource watch framework. The same software can run in simulation (SDL) on linux, on pinetime and on the ds_d6 oled watch.

This is a demo, when you want to implement firmware updates over the air.

Zephyr has a procedure in place.

Look for MCUBOOT definitions in the source code

15.5.3 Requirements

no requirements since it can run without board (simulation)

15.5.4 Building and Running

west build -p -b native-posix_64 oswatch-fota

15.5.5 References

15.6 Basic OSWatch Framework

The application keeps track of time and displays time and date. The initial time is the time of build.

Switching between screens can be done by multiple presses on button.

On the native_posix application 2 serial ports are available. UART_1 connected to pseudotty: /dev/pts/2 UART_0 connected to pseudotty: /dev/pts/3

By connecting to UART_0 with : minicom -D /dev/pts/3, you have access to the zephyr-shell. (just type "help" to get overview)

15.6.1 Overview

This is an opensource watch framework. The same software can run in simulation (SDL) on linux, on the pinetime and on the ds_d6 oled watch.

This Basic application serves as a chassis, where accessories can be supplemented (bluetooth, fota, powermanagement, . . .) These application carry the name oswatch-bt, oswatch-pm, oswatch-fota, oswatch-full.

Why implement powermanagent if you do not have a battery? Why implement firmware over the air, if you can update the firmware with a debugprobe.

15.6.2 Remark

This basic time keeping application, might need adjustment. In that case you will still need CTS (bluetooth functionality).

15.6.3 Requirements

no requirements since it can run without board (simulation)

15.6.4 Building and Running

west build -p -b native-posix_64 oswatch

15.6.5 References

15.7 HRS3300 Heart Rate Sensor

15.7.1 Overview

A sensor application that demonstrates how to poll data from the hrs3300 heart rate sensor.

15.7.2 Building and Running

This project configures the hrs3300 sensor on the pinetime_devkit1 board to enable the green LED and measure the reflected light with a photodiode. The raw ADC data prints to the console. Further processing (not included in this sample) is required to extract a heart rate signal from the light measurement.

Sample Output

for this you will need a Segger JLink Console

```
rtt:~$ sensor get HRS3300 18 (ir -- this switches off the sensor)
rtt:~$ sensor get HRS3300 19 (red -- this switches on the sensor)
rtt:~$ sensor get HRS3300 20 (green -- get a value)
```

15.8 Bluetooth: Central / Heart-rate Monitor

15.8.1 Overview

Similar to the Central sample, except that this application specifically looks for heart-rate monitors and reports the heart-rate readings once connected.

15.8.2 Requirements

- BlueZ running on the host, or
- A board with BLE support

15.8.3 Building and Running

This sample can be found under **:zephyr_file:`samples/bluetooth/central_hr`** in the Zephyr tree.

See bluetooth samples section for details.

15.9 Bluetooth: transferring data to the watch

15.9.1 Overview

On a linux box with bluez installed, a python script is launched (example-gatt-server)

The linuxbox becomes a bluetooth gatt server.

This script offers a service and a bluetooth characteristic. (the battery service is used for this purpose)

In the python script example-gatt-server 10 bytes of data are advertised (battery service).

The watch scans for this characteristic, once it is connected and reads out the variables and puts them in a 10 byte arry.

Hence a transfer from the server to the watch took place!

Later on we can use this to set a timer in the watch or an alarm, or ....

You'll need to:

- run Bluez,
- start the script,
- and then connect to the watch, using bluetoothctl.

The mechanism is the same as with the CTS time setting.

15.9.2 Requirements

- BlueZ running on the host, or
- A board with BLE support

15.9.3 Building and Running

See bluetooth samples section for details.

15.10 Bluetooth: getting data from the watch

15.10.1 Overview

The watch transmits 240 indexed values. It uses bluetoothnotification. There is no confirmation of reception. (a bit like UDP network traffic)

This method is fast, and could be used to transmit measurement values in real time.

A custom service and characteristic is used.

A phyton script is supplied :

- the scrips scans for a specific device (since bluetoothmac is variable : a bit like DHCP in netwerk traffic)
- once the device is found, it switches on "notifications" and listens for incoming messages.

15.10.2 Requirements

- BlueZ running on the host, or
- A board with BLE support

15.10.3 Building and Running

See bluetooth samples section for details.

15.11 Bluetooth: transferring data to the watch

15.11.1 Overview

The same technique as in the cts-time setting is used. A python script on linux, running bluez, reads 3 parameters.

The watch can read these parameters over bluetooth.

This sketch is used as a building block.

The gatt server python script is supplied. (values-gatt-server) It prompts to input 3 parameters (integers) You'll need to:

- run Bluez,
- start the script,
- and then connect to the watch, using bluetoothctl.

15.11.2 Requirements

- BlueZ running on the host, or
- A board with BLE support

15.11.3 Building and Running

See bluetooth samples section for details.

15.12 Pinetime Button Sample

15.12.1 Overview

The button on the pinetime uses 2(!) GPIO ports.

This sample application displays “a LED”; Long Press on the button and the LED changes “color” Short Press toggles the LED.

Several messages are printed to the serial port. In order to view these you will need a segger JLink probe (RTT).

15.12.2 Requirements

pinetime segger jlink debugprobe

15.12.3 Building and Running

west build -p -b pintime_devkit1 samples/button/pinebutton

15.12.4 References

15.13 SDL Button Timer Sample

15.13.1 Overview

Use a timer to distinguish between long press ,short press and double-click.

15.13.2 Requirements

This sample uses the native_posix solution, so no need for a real board. You’ll need to have a SDL library installed.

15.13.3 Building and Running

west build -p -b native_posix_64 samples/sdlbutton2

15.13.4 References

15.14 LittlevGL SDL Button Sample

15.14.1 Overview

This sample application displays “a LED” and “a button” Long Press on the button and the LED changes “color”

15.14.2 Requirements

This sample uses the native_posix solution, so no need for a real board. You'll need to have a SDL library installed.

15.14.3 Building and Running

west build -p -b native_posix_64 samples/sdlbutton

15.14.4 References

15.15 Character frame buffer

15.15.1 Overview

This sample displays character strings using the Character Frame Buffer (CFB) subsystem framework.

15.15.2 Building and Running

build the application: west build -p -b ds_d6 samples/cfb

on unix : #minicom -b 115200 -D /dev/ttyACM1

you get a shell and you can type help

to display something on the screen : cfb init cfb invert cfb print 0 0 "hello world"

15.15.3 POSIX

west build -p -b native_posix_64 samles/cfb

connect to serial port : minicom -D /dev/pts/1

15.15.4 Pinetime

Problem : does not display a thing ....

15.16 Character Framebuffer Shell Module Sample

15.16.1 Overview

This is a simple shell module that exercises displays using the Character Framebuffer subsystem.

—not working on posix—-

15.16.2 Building and Running

Build the sample app by choosing the target board, for example:

Shell Module Command Help

```
cfb - Character Framebuffer shell commands
Options:
        -h, --help  :Show command help.
Subcommands:
        init        :[none]
        get_device  :[none]
        get_param   :<all, height, width, ppt, rows, cols>
        get_fonts   :[none]
        set_font    :<idx>
        invert      :[none]
        print       :<col: pos> <row: pos> <text>
        scroll      :<dir: (vertical|horizontal)> <col: pos> <row: pos>
                     <text>
        clear       :[none]
```

init: should be called first to initialize the display.

Command example (ds_d6):

```
uart:~$ cfb init
Framebuffer initialized: SSD1306
Display Cleared
```

get_device: prints the display device name.

Command example (ds_d6 board):

```
uart:~$ cfb get_device
Framebuffer Device: SSD16XX
```

get_param: get the display parameters where height, width and ppt (pixel per tile) are in pixels and the number of rows and columns. The row position is incremented by a multiple of the ppt.

Command example (ds_d6 board):

```
uart:~$ cfb get_param all
param: height=120
param: width=250
param: ppt=8
param: rows=15
param: cols=250
```

get_fonts: print the index, height and width in pixels of the static defined fonts presented in the system.

Command example (ds_d6 board):

```
uart:~$ cfb get_fonts
idx=0 height=32 width=20
idx=1 height=24 width=15
idx=2 height=16 width=10
```

set_font: choose the font to be used by passing the font index. Only one font can be used at a time.

Command example (ds_d6 board):

```
uart:~$ cfb set_font 0
Font idx=0 height=32 widht=20 set
```

invert: invert the pixel color of the display.

Command example (ds_d6 board):

```
uart:~$ cfb invert
Framebuffer Inverted
```

print: pass the initial column and row positions and the text in double quotation marks when it contains spaces. If text hits the edge of the display the remaining characters will be displayed on the next line. The previous printed text will be overwritten.

Command example (ds_d6 board):

```
uart:~$ cfb print 60 5 ZEPHYR
```

scroll: pass the scroll direction, vertical or horizontal, the initial column and row positions, and the text to be displayed in double quotation marks when it contains spaces. If the text hits the edge of the display, the remaining characters will be displayed in the next line. The text will scroll until it hits the display boundary, last column for horizontal and last row for vertical direction. The text passed with the scroll command will be moved vertically or horizontally on the display.

Command example (ds_d6 board):

```
uart:~$ cfb scroll vertical 60 5 ZEPHYR
```

clear: clear the display screen.

Command example (ds_d6 board):

```
uart:~$ cfb clear
Display Cleared
```

15.17 LittlevGL Basic Sample

15.17.1 Overview

This sample application displays “Hello World” in the center of the screen.

15.17.2 Requirements

- Desay D6 OLED SSD1306
- Linux box
- Pinetime

15.17.3 Building and Running

west build -p -b ds_d6 -d build-lvgl samples/lvgl

or

west build -p -b native_posix_64 samples/lvgl

run : ./build/zephyr/zephyr.exe

west build -p -b pinetime_devkit1 samples/lvgl

run : ./build/zephyr/zephyr.exe

15.17.4 References

15.18 Lightsensor

This is a modified sample app to read the pah8001 Heartrate sensor from the Desay D6 smartwatch. It is used as a light sensor, uses the I2C protocol and is based on the max44009. In order to function it needs the modified driver.

Notice : in Kconfig MAX44009NEW was defined to avoid confusion with existing driver max44009 (this way the original does not get selected)

15.19 Testing the timer refresh

15.19.1 Overview

This sample application displays “a blinking LED”. A timer triggers the LED on/off state.

15.19.2 Requirements

This sample uses the native_posix solution, so no need for a real board. You’ll need to have a SDL library installed.

15.19.3 Building and Running

west build -p -b native_posix_64 samples/timer

15.19.4 References

CHAPTER

SIXTEEN

MENUCONFIG

16.1 Zephyr is like linux

TIP: the pinetime specific drivers are located under Modules

Note: to get a feel, compile a program, for example

```
west build -p -b pinetime samples/bluetooth/peripheral -D CONF_FILE="prj.conf"
```

the pinetime contains an external 32Kz crystal now you can have a look in the configurationfile (and modify if needed)

```
$ west build -t menuconfig
```

```
    Modules  --->
    Board Selection (nRF52832-MDK)  --->
    Board Options  --->
    SoC/CPU/Configuration Selection (Nordic Semiconductor nRF52 series MCU)  --->
    Hardware Configuration  --->
    ARM Options  --->
    Architecture (ARM architecture)  --->
    General Architecture Options  --->
[ ] Floating point  ----
    General Kernel Options  --->
    Device Drivers  ---> **************SELECT THIS ONE**************************
    C Library  --->
    Additional libraries  --->
[*] Bluetooth  --->
[ ] Console subsystem/support routines [EXPERIMENTAL]  ----
[ ] C++ support for the application  ----
    System Monitoring Options  --->
    Debugging Options  --->
[ ] Disk Interface  ----
    File Systems  --->
-*- Logging  --->
    Management  --->
    Networking  --->
```

```
[ ] IEEE 802.15.4 drivers options  ----
(UART_0) Device Name of UART Device for UART Console
[*] Console drivers  --->
[ ] Net loopback driver  ----
[*] Serial Drivers  --->
```

(continues on next page)

(continued from previous page)

```
    Interrupt Controllers  --->
    Timer Drivers  --->
-*- Entropy Drivers  --->
[*] GPIO Drivers  --->
[ ] Shared interrupt driver  ----
[ ] SPI hardware bus support  ----
[ ] I2C Drivers  ----
[ ] I2S bus drivers  ----
[ ] PWM (Pulse Width Modulation) Drivers  ----
[ ] Enable board pinmux driver  ----
[ ] ADC drivers  ----
[ ] Watchdog Support  ----
[*] Hardware clock controller support  ---> <<<<<<<<<<<<<<<<<SELECT THIS ONE<<<<<<<<<<
[ ] Precision Time Protocol Clock driver support
[ ] IPM drivers  ----
    Max compiled-in log level for ipm (Info)  --->
[ ] Flash hardware support  ----
[ ] Sensor Drivers  ----
```

```
    Max compiled-in log level for clock control (Info)  --->
[*] NRF Clock controller support  ---> <<<<<<<<<<<<<<<<<<<<SELECT THIS ONE<<<<<<<<<<
```

CHAPTER

SEVENTEEN

DEBUGGING

17.1 debugging

17.1.1 Segger JLink

```
The pinetime smartwatch does not have a serial port.
A way around this is to use a Segger Jlink debug-probe and enable logging and shell␣
↪over RTT:

in prj.cfg:

CONFIG_LOG=y
CONFIG_USE_SEGGER_RTT=y
CONFIG_SHELL=y
CONFIG_SHELL_BACKEND_RTT=y

start the debugger :
west debug
and enter : continue

This starts up in the background "Segger Jlink Processes"
```

Note: What happens when you type : west debug? You could type this on the command prompt.

```
(this start the server)
JLinkGDBServer -select usb -port 2331 -if swd -speed 4000 -device nRF52832_xxAA -
↪silent -singlerun
(this starts the debug session)
~/zephyr-sdk/arm-zephyr-eabi/bin/arm-zephyr-eabi-gdb /root/zephyrproject/app/build/
↪zephyr/zephyr.elf -ex target :2331 -ex halt -ex reset -ex load
telnet 127.0.0.1 19021 (putty on linux) start the rtt console and shows the debug-log
```

17.1.2 Black Magic

```
The ds-D6 smartwatch has a serial port.


The blackmagicprobe can lauch a debugger : west debug --runnerblackmagicprobe.
The probe has a serial port /dev/ttyACM1 (linux : minicom -b 115200 -D /dev/ttyACM1)
```

17.1.3 STM32 - Raspberry - OpenOCD

```
If you do not have a Segger debug probe nor a serial port, you can put a value in␣
↪memory at a fixed location.
With openocd you can peek at this memory location.
If you own a raspberry pi or an orange pi, you can use the GPIO header.
Another cheap option is an stm32 debug probe.
```

Note: #define MY_REGISTER (*(volatile uint8_t*)0x2000F000)

in the program you can set values: MY_REGISTER=1; MY_REGISTER=8;

this way you know till where the code executes

```
#telnet 127.0.0.1 4444
```

programming

```
once your telnet sessions started:
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
Open On-Chip Debugger
>mdw 0x2000F000 0x1

the last byte shows the value of your program trace value
```

17.2 debugging Posix

start gdb on the command prompt

```
(gdb) file build/zephyr/zephyr.elf
Reading symbols from build/zephyr/zephyr.elf...done.
(gdb) b main
Breakpoint 1 at 0x5909: file /root/work/zephyr/boards/posix/native_posix/main.c, line␣
↪56.
(gdb) run
```

Debug oswatch-bt (which needs and argument –bt-dev=hci0) Set the breakpoint in the function connected

```
(gdb) file build/zephyr/zephyr.elf
Reading symbols from build/zephyr/zephyr.elf...done.
(gdb) b connected
```

(continues on next page)

(continued from previous page)

```
Breakpoint 1 at 0x498a: file /root/work/app/oswatch-bt/src/bt.c, line 140.
(gdb) r --bt-dev=hci0
```

17.3 The black magic probe

17.3.1 probes in zephyr

You can program the nrf52832 with a debuggerprobe. The standard-setup is jlink (segger).

/root/zephyrproject/zephyr/boards/arm/id107plus/board.cmake (adapt the runner here)

in our case : instead of jlink specify : blackmagicprobe

The cool thing about this probe that it has a serial port (3.3V) and a debug (upload) port on the same usb-port.

:: /dev/ttyACM1 is serial port (pb6 pb7)

minicom -b 115200 -D /dev/ttyACM1

/dev/ttyACM0 is used as debugger/uploading

west debug –runner blackmagicprobe west flash –runner blackmagicprobe

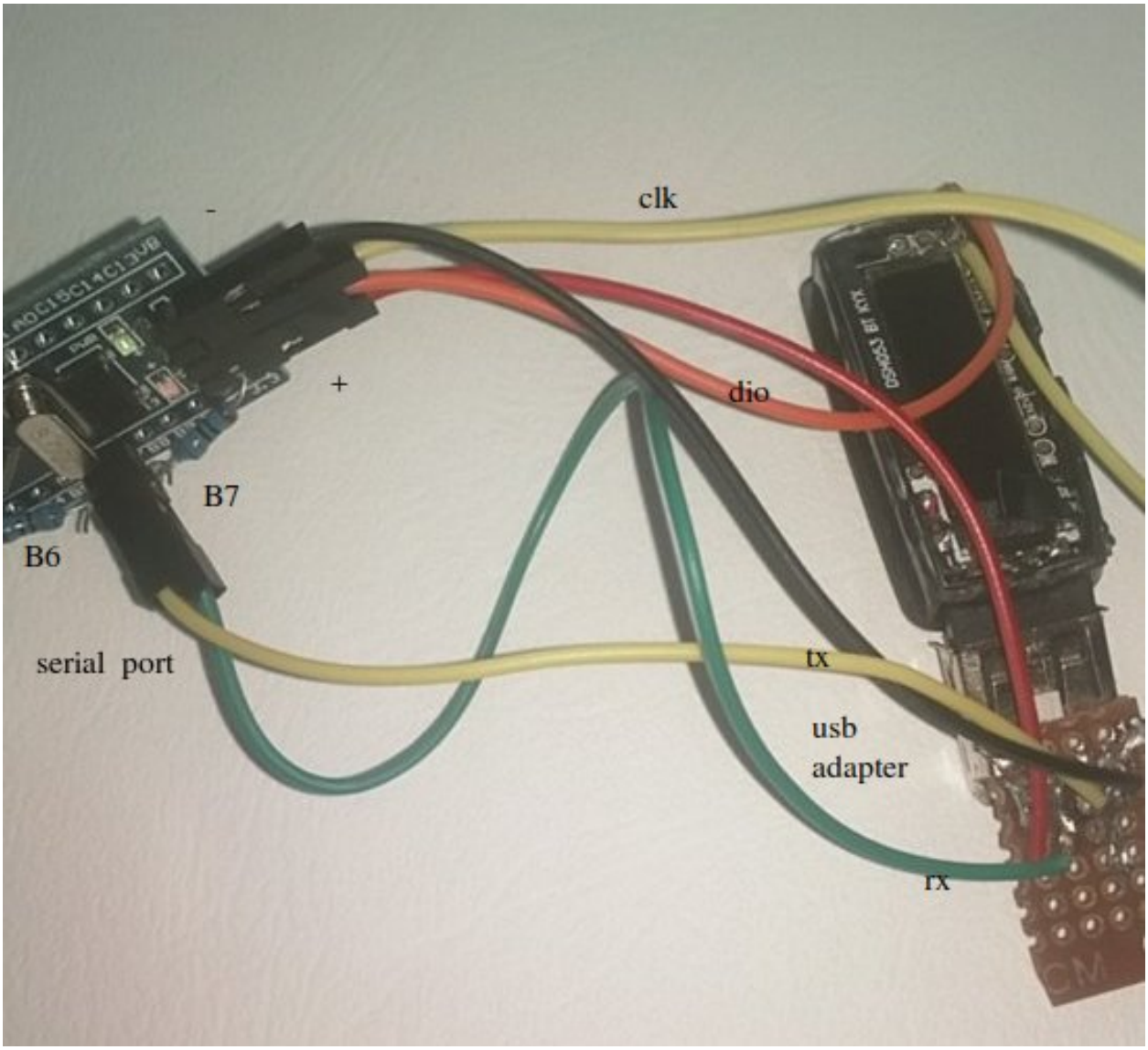
17.3.2 howto setup a blackmagicprobe

You can buy this probe and support the developers. (make this world a better place)

I bought a “cheapo” “blue pill” stm32 board for future projects . . . soldered a 1.8K resistor between 3.3K and PA12

downloaded from https://jeelabs.org/docs/software/bmp/ - blackmagic.bin (79 ko) - blackmagic_dfu.bin (7 ko)

in jlink : loadbin blackmagic_dfu.bin 0x8000000 (specify jlink no options . . .) switch boot0 or boot1 or whatever connect usb in linux dfu-util -v -R -d 0483:df11 -s 0x08002000 -D blackmagic.bin (uploading in jlink was a problem cause memory restrictions)



(removed boot0 and boot1 connectors on the stm afterwards)

plugged it in the USB port and it pops up (had to enable it first in virtual box usb : black sphere technologies .....)

17.4 pseudo

17.4.1 simulation on NATIVE_POSIX_64

It is possible to use two serial ports on the virtual posix board. One can display log message, while the other one lets you interact with the shell.

```
CONFIG_UART_CONSOLE=y
CONFIG_UART_CONSOLE_ON_DEV_NAME="UART_0"
CONFIG_UART_NATIVE_POSIX_PORT_1_ENABLE=y
CONFIG_UART_NATIVE_POSIX_PORT_1_NAME="UART_1"
```

(continues on next page)

(continued from previous page)

```
CONFIG_SHELL=y
CONFIG_UART_SHELL_ON_DEV_NAME="UART_0"
```

```
root@osboxes:~/work/app# ./build/zephyr/zephyr.exe
UART_1 connected to pseudotty: /dev/pts/2
UART_0 connected to pseudotty: /dev/pts/3


minicom -D /dev/pts/3
```

```
(you should see log messages)
```

17.5 Bsim

this is a method of debugging a bluetooth enabled application, without a bluetooth device

17.5.1 debugging on nrf52_bsim

You can compile the oswatch-bt sample for the bsim board.

```
west build -p -b nrf52_bsim oswatch-bt
gdb

file build/zephyr/zephyr.elf
b connected (breakpoint on function connected)
r -s=trial_sim -d=1
```

17.6 Segger RTT (Real Time Transfer)

17.7 Serial data without a serial port

Pinetime does not have UART pins but UART-like connection can be achieved using RTT (Real Time Transfer) feature of Segger JLink debugger. RTT data can be accessed using the SEGGER tool (JLinkRTTViewer) or by using telnet connection to active debug session. Second method is recommended since it gives better throughput and allows bidirectional communication using Zephyr Shell (with RTT as backend). PuTTY can be used to telnet to debug session.

Prerequisites: - JLink debugger, for example one of Nordic Semiconductor Development Kits. - Setup: https://wiki.pine64.org/index.php/PineTime#Using_JLink_programmer_and_nrfjprog_tools

Following steps needs to be taken to run RTT shell in the application: 1. Install PuTTY and Setup RTT session. On Linux, copy *misc/rtt_shell/rtt* to *~/putty/sessions*. On Windows, execute *misc/rtt_shell/putty-rtt.reg*. 2. Enable logging and shell over RTT: ` CONFIG_LOG=y CONFIG_USE_SEGGER_RTT=y CONFIG_SHELL=y CONFIG_SHELL_BACKEND_RTT=y `

3. Build and flash application.

4. Start debug session ` west debug (will start gdb) continue (in gdb console) ` 5. Start PuTTY RTT session, prompt should be printed.

Resources: - Zephyr shell documentation: https://docs.zephyrproject.org/latest/reference/shell/index.html - Zephyr logger documentation: https://docs.zephyrproject.org/latest/reference/logging/index.html

CHAPTER

EIGHTEEN

HACKING STUFF

18.1 hacking the pinetime smartwatch

```
The pinetime is preloaded with firmware.
This firmware is secured, you cannot peek into it.
```

Note: The pinetime has a swd interface. To be able to write firmware, you need special hardware. I use a stm-link which is very cheap(2$). You can also use the GPIO header of a raspberry pi. (my repo: https://github.com/najnesnaj/openocd is adapted for the orange pi)

To flash the software I use openocd : example for stm-link usb-stick

```
# openocd -s /usr/local/share/openocd/scripts -f interface/stlink.cfg -f target/nrf52.
↪cfg
```

example for the orange-pi GPIO header (or raspberry)

> # openocd -f /usr/local/share/openocd/scripts/interface/sysfsgpio-raspberrypi.cfg -c 'transport select swd' -f /usr/local/share/openocd/scripts/target/nrf52.cfg -c 'bindto 0.0.0.0'

once you started the openocd background server, you can connect to it using:

```
#telnet 127.0.0.1 4444
```

programming

```
once your telnet sessions started:
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
Open On-Chip Debugger
> program zephyr.bin

target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x00001534 msp: 0x20004a10
** Programming Started **
auto erase enabled
using fast async flash loader. This is currently supported
only with ST-Link and CMSIS-DAP. If you have issues, add
"set WORKAREASIZE 0" before sourcing nrf51.cfg/nrf52.cfg to disable it
target halted due to breakpoint, current mode: Thread
xPSR: 0x61000000 pc: 0x2000001e msp: 0x20004a10
```

(continues on next page)

(continued from previous page)

```
wrote 24576 bytes from file zephyr.bin in 1.703540s (14.088 KiB/s)
** Programming Finished **

And finally execute a reset :
>reset
```

removing write protection see: *howto flash your zephyr image*

18.2 scanning the I2C_1 port

```
The pinetime does not have a serial port.
I do not have a segger debugging probe.
A way around this, it to put a value in memory at a fixed location.
With openocd you can peek at this memory location.
```

18.2.1 Building and Running

In this repo under samples you will find an adapted i2c scanner program.

```
west build -p -b pinetime samples/drivers/i2c_scanner
```

Note: #define MY_REGISTER (*(volatile uint8_t*)0x2000F000)

in the program you can set values: MY_REGISTER=1; MY_REGISTER=8;

this way you know till where the code executes

```
#telnet 127.0.0.1 4444
```

Peeking

```
once your telnet sessions started:
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
Open On-Chip Debugger
>mdw 0x2000F000 0x1
0x2000f000: 00c24418
```

Note::

this corresponds to 0x18, 0x44 and 0xC2 (which is endvalue of scanner, so it does not detect touchscreen, which should be touched first. . . .)

18.3 howto flash your zephyr image

Once you completed your `west build` , your image is located under the build directory

```
   $ cd ~/work/pinetime/zephyr/build/zephyr
   here you can find zephyr.bin which you can flash



I use Openocd to flash.
Just connect : telnet 127.0.0.1 4444

.. code-block:: console

    program zephyr.bin
```

18.4 howto remove the write protection

:: the PineTime watch is read/write protected (at least the one I got) executing the following : nrf52.dap apreg 1 0x0c shows 0x0

Mind you, st-link does not allow you to execute that command, for this you will need a J-link.

There is a workaround using the GPIO of a raspberry pi or an Orangepi. (in this case you won't need an external programmer at all) (You can find an example for the orange pi in my repo :https://github.com/najnesnaj/openocd.) You have to reconfigure Openocd with the –enable-cmsis-dap option.

Unlock the chip by executing the command: > nrf52.dap apreg 1 0x04 0x01

18.5 howto configure gateway

Once you completed your `west build` , your image is located under the build directory

```
$ cd ~/work/pinetime/zephyr/build/zephyr
here you can find zephyr.bin which you can flash
```

```
I use an orange pi single board computer.
The pinetime watch is attached to this.
My development is done on a laptop.
```

```
How can you copy from one environment (laptop) to another (SBC) without typing␣
↪password?
```

On the laptop :

```
ssh-keygen -b 8092 -t rsa -C "fota gw access key" -f ~/.ssh/orange
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):          (LEAVE EMPTY!)
Enter same passphrase again:
Your identification has been saved in /root/.ssh/orange.
Your public key has been saved in /root/.ssh/orange.pub.
```

(continues on next page)

(continued from previous page)

```
The key fingerprint is:
SHA256:xCM5Fk1LAVjEWqrM6LKM8Y6+Y12ONt6eV8vDa/KdRUM fota gw access key
The key's randomart image is:
+---[RSA 8092]----+
|     ==++.       |
|B*B.o+. +ooo     |
+----[SHA256]-----+

(the standard port is 22 and not 9988 which is my custom port)

copy the certificate to the SBC (which name is orange in my case):
ssh-copy-id -p 9988 -i ~/.ssh/orange.pub root@orange


create config file :  ~/.ssh/config
      Host orange
      HostName orange
      User root
      Port 9988
      IdentityFile ~/.ssh/orange
```

```
Now you can copy without a password :
scp build/zephyr/zephyr.bin orange:/usr/src
```

w

18.6 howto use 2 openocd sessions

Once you completed your `west build` , your image is located under the build directory

```
   $ cd ~/work/pinetime/zephyr/build/zephyr
   here you can find zephyr.bin which you can flash



I use Openocd to flash.
Just connect : telnet 127.0.0.1 4444

.. code-block:: console

    program zephyr.bin
```

18.6.1 Suppose you have 2 microcontrollers

Just connect : telnet 127.0.0.1 7777 for the second.

18.6.2 Howto setup a second openocd session on a different port?

In this case an ST-LINK/V2 an in-circuit debugger and programmer is used.

```
openocd -c 'telnet_port 7777' -c 'tcl_port 6667' -c 'gdb_port 3332' -s /usr/local/
↪share/openocd/scripts -f interface/stlink.cfg  -f target/nrf52.cfg
```

18.6.3 Howto use the GPIO header of a Single Board computer

This works really well, and does not require a seperate programmer.

```
openocd -f /usr/local/share/openocd/scripts/interface/sysfsgpio-raspberrypi.cfg -c
↪'transport select swd' -f /usr/local/share/openocd/scripts/target/nrf52.cfg  -c
↪'bindto 0.0.0.0'
```

18.7 howto generate pdf documents

sphinx cannot generate pdf directly, and needs latex

```
apt-get install latexmk
apt-get install texlive-fonts-recommended
apt-get install xzdec
apt-get install cmap
apt-get install texlive-latex-recommended
apt-get install texlive-latex-extra
```

18.8 The Movie

https://youtu.be/A-kYHdrr-Mg

or

oswatchclip.mp4

18.8.1 editing :

Kdenlive

18.8.2 script :

LibreOffice Impress

18.8.3 Camera :

SimpleScreenRecorder

18.8.4 Voice over :

espeak -v en+f5 -a 100 -p 50 -s 110 -f slide2.txt -w slide2.wav

18.8.5 Music by Beethoven:

fluidsynth -a pulseaudio -o audio.alsa.device=hw:1 -m alsa_seq -l -i /usr/share/sounds/sf2/FluidR3_GM.sf2 moonlight_sonata.mid

this one works best : timidity moonlight_sonata.mid -Ow -o out.wav

CHAPTER

NINETEEN

BEHIND THE SCENE

19.1 Touchscreen

In my first release I created a touchscreendriver that read the x-y coordinates from the pinetime screen.

Meanwhile a rudimentary touchscreen driver exist in Zephyr. (KSCAN_TOUCH)

It integrates with LVGL.

I managed to adapt the driver from Focaltech, so it now supports the Pinetime as well.

However it is not capable of detecting events like “slide up/down”, but point and click is usable.

19.1.1 Overview

Touchscreen Hynitron

19.1.2 Requirements

19.1.3 Building and Running

19.1.4 Todo

-support more touchscreen events

19.1.5 References

19.2 placing a button on the screen

```
This sample is not really important, but it will teach you that you need to set LVGL_
↪CONFIG values, in order    to be able to use LVGL functions.
```

19.2.1 Building and Running

CHAPTER

TWENTY

ABOUT

I got a pinetime development kit very early. I would like to thank the folks from https://www.pine64.org/ for the kit.

The Nordic nrf52832 is an Arm based, 32bit microcontroller with a lot of flash, RAM memory and(!) bluetooth! It is a good platform to explore the wonderful world of opensource RTOS's. I choose the Zephyr platform, because it already contained the display driver, and the bluetooth functionality. Furthermore it can be updated wireless.

I started of with adapting simple samples, so they would run on the pinetime watch. Meanwhile I learned how to adapt the board definition file. Gradually this morphed into a pinetime-toolkit. A few experts contributed and improved this kit so it became usable.

I took a long break from this project, zephyr evolved and a project for creating an open source watch, the hypnos-firmware flourished.

The drivers for the heartrate sensor, touch panel that I adapted, became obsolete. The pinetime board definition file is now part of the zephyr RTOS distribution.

Recently I discovered that the standard focaltech touchscreen driver could be adapted easily so it would run on the pinetime and integrate nicely with the LVGL-gui. There were several OLED ssd1306 watches on my desk and I wondered if I could run Zephyr and LVGL (graphical stuff) on them as well. (yes, we can!) I played around with the "native_posix_64" board, which is a virtual board that allows you to run firmware on your computer instead of on the watch (no need for flashing).

This gave me the idea for an opensource-watch-framework. Start of with simple samples, and add some more functionality, until finally you end up with a smartwatch. The framework is build around three smartwatches, the pinetime, the ds_d6 and the posix_sdl. It should work on other platform as well, with minor modifications. I only recently got the idea for out-of-tree development. Just put all the modified drivers, board definitions and application in one place outside of the zephyr-tree.

```
A word of warning: this is work in progress.
You're likely to have a better skillset then me.
You are invited to add the missing pieces and to improve what's already there.
```

CHAPTER

TWENTYONE

AUTHOR

Some 20 years ago I attended a course on how to brew beer. Real beer, starting with barley, wheat, yeast, brew kettles ...

I bought some brewing equipment and started brewing my own beer. Apart from a stupid idea, (why brew beer in a country that has plenty of breweries), I soon found out that brewing is labor intensive.

Instead of quitting right away (the most sensible thing to do), I tried to reduce the effort. I switched from a gas heated kettle to an electric one.

The next step was to control the temperature. I got an arduino and a temp-sensor. This got me interested in microcontrollers.

It appealed to my childish nature, that I could get quick results with little coding. Blinking a led, really lightens up my day! And a beer of course.

```
Guess, even for a technically skilled audiance, story-telling,  renders a manual less␣
↪boring ...
```