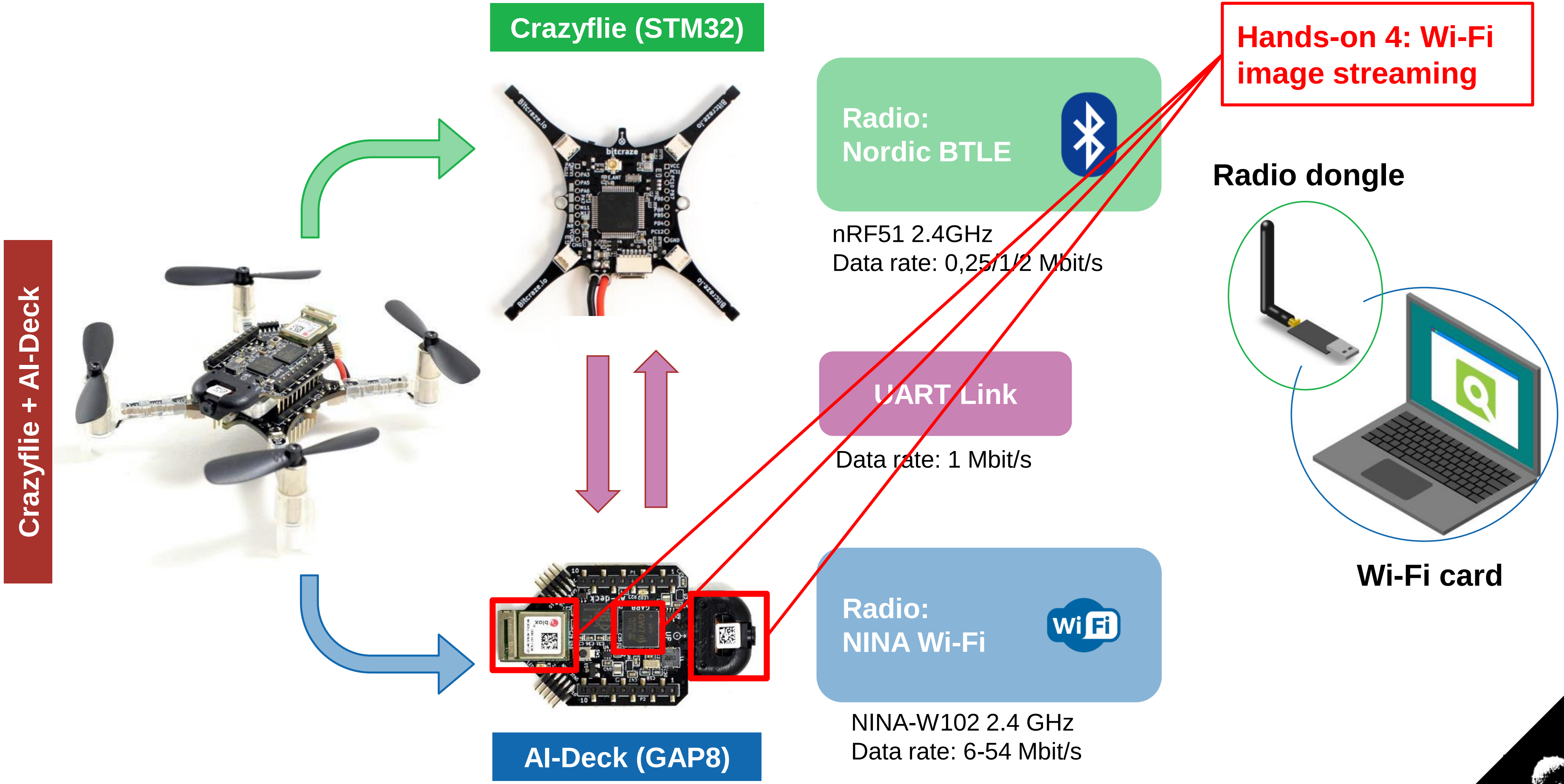
PULP PLATFORM
Open Source Hardware, the way it should be!

Bitcraze Workshop: *Hands-on Session 4*
Wi-Fi image streaming with AI-Deck

Lorenzo Lamberti, Hanna Müller, Vlad Niculescu, Manuele Rusci, Daniele Palossi

http://pulp-platform.org @pulp_platform https://www.youtube.com/pulp_platform

Hands-on session 4



Crazyflie (STM32)

Crazyflie + AI-Deck

Radio: Nordic BTLE

nRF51 2.4GHz
Data rate: 0,25/1/2 Mbit/s

UART Link

Data rate: 1 Mbit/s

Radio: NINA Wi-Fi

NINA-W102 2.4 GHz
Data rate: 6-54 Mbit/s

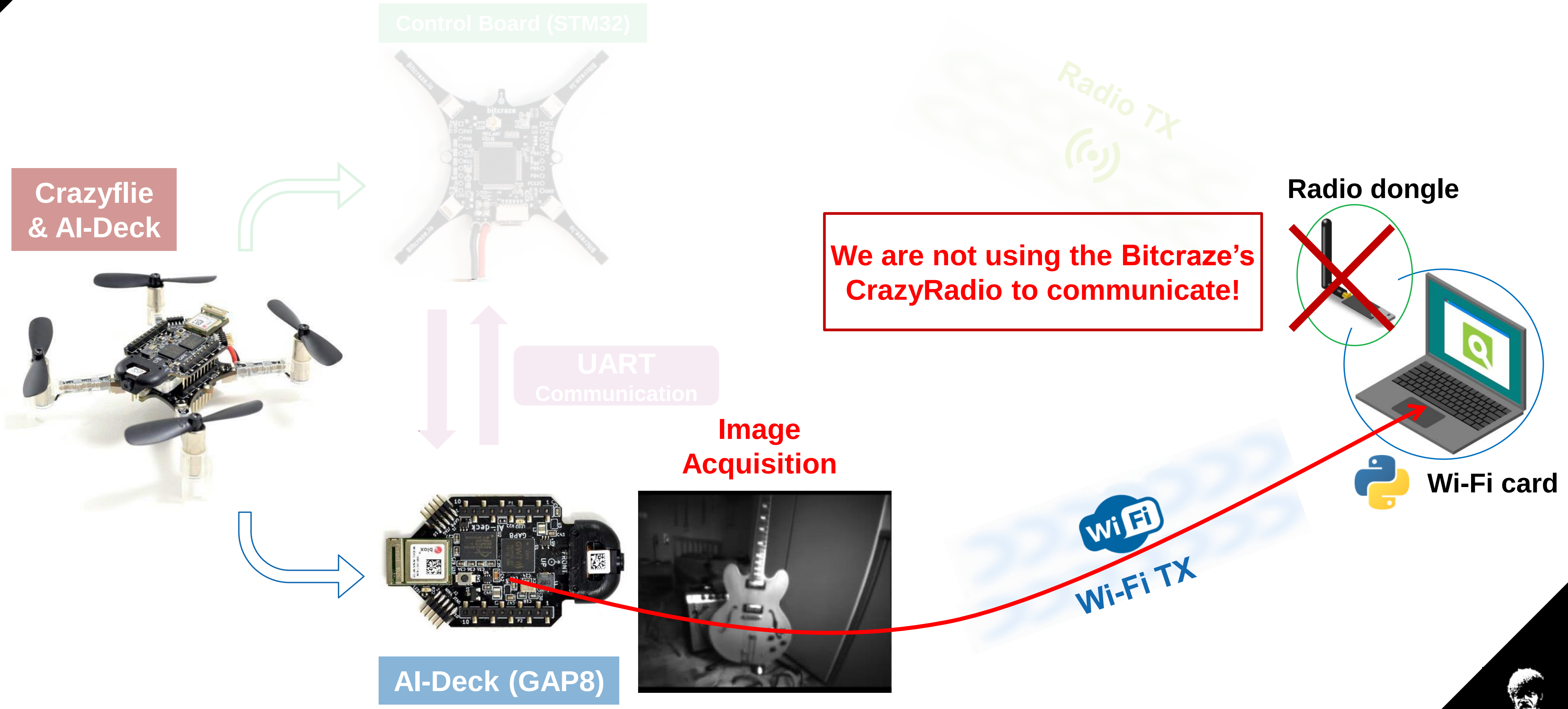
AI-Deck (GAP8)

Hands-on 4: Wi-Fi image streaming

Radio dongle

Wi-Fi card

Image streaming via Wi-Fi



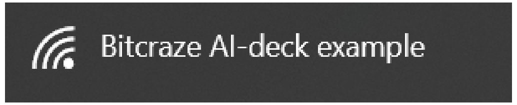
We are not using the Bitcraze's CrazyRadio to communicate!

Hands-on overview

The example is inside the Bitcraze GitHub repository, and it is called wifi_jpeg_streamer

Code: https://github.com/bitcraze/AIdeck_examples/blob/master/GAP8/test_functionalities/wifi_jpeg_streamer

Default Network SSID:

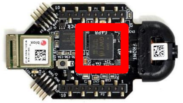


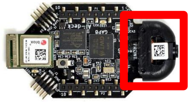
Bitcraze AI-deck example

- Create a Wi-Fi access-point with the NINA Wi-Fi module
- Establish a **point-to-point** Wi-Fi connection between laptop and AI-Deck

- **Acquisition of an image** ➡ ➡
- **Compression** (JPEG)
- **Wi-Fi transmission** of the image
- **Bonus task:** pre-processing the image before transmission

Wi-Fi Image streaming: Initial setup





AI-DeckV1: RGB Bayer
AI-DeckV2: Grayscale

FC: Fabric Controller
CL: Cluster (8-cores)

Input image buffer:
L2 memory allocation

QVGA format (320x240 pixel)
AEG: Auto-Exposure Gain

Initial Setup

Main loop

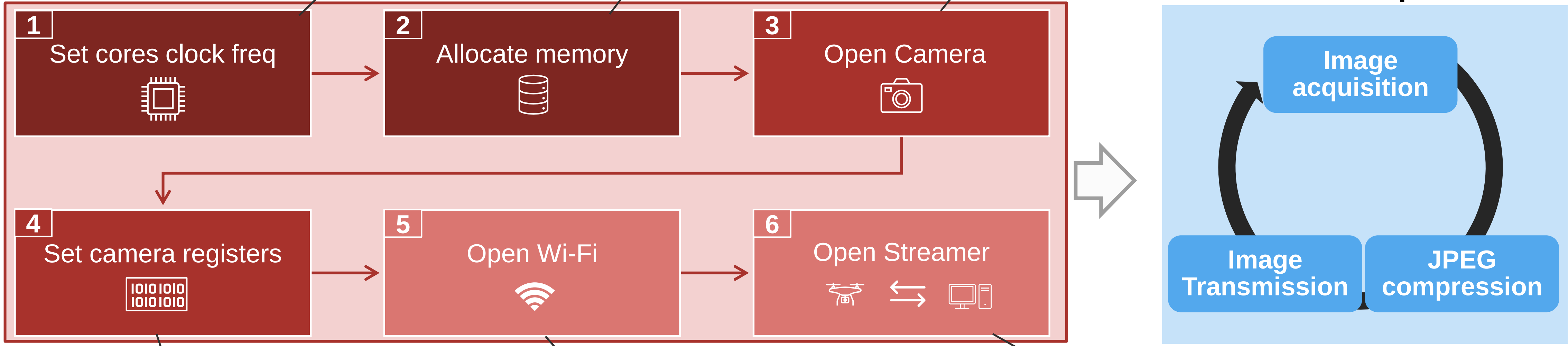
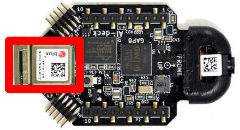


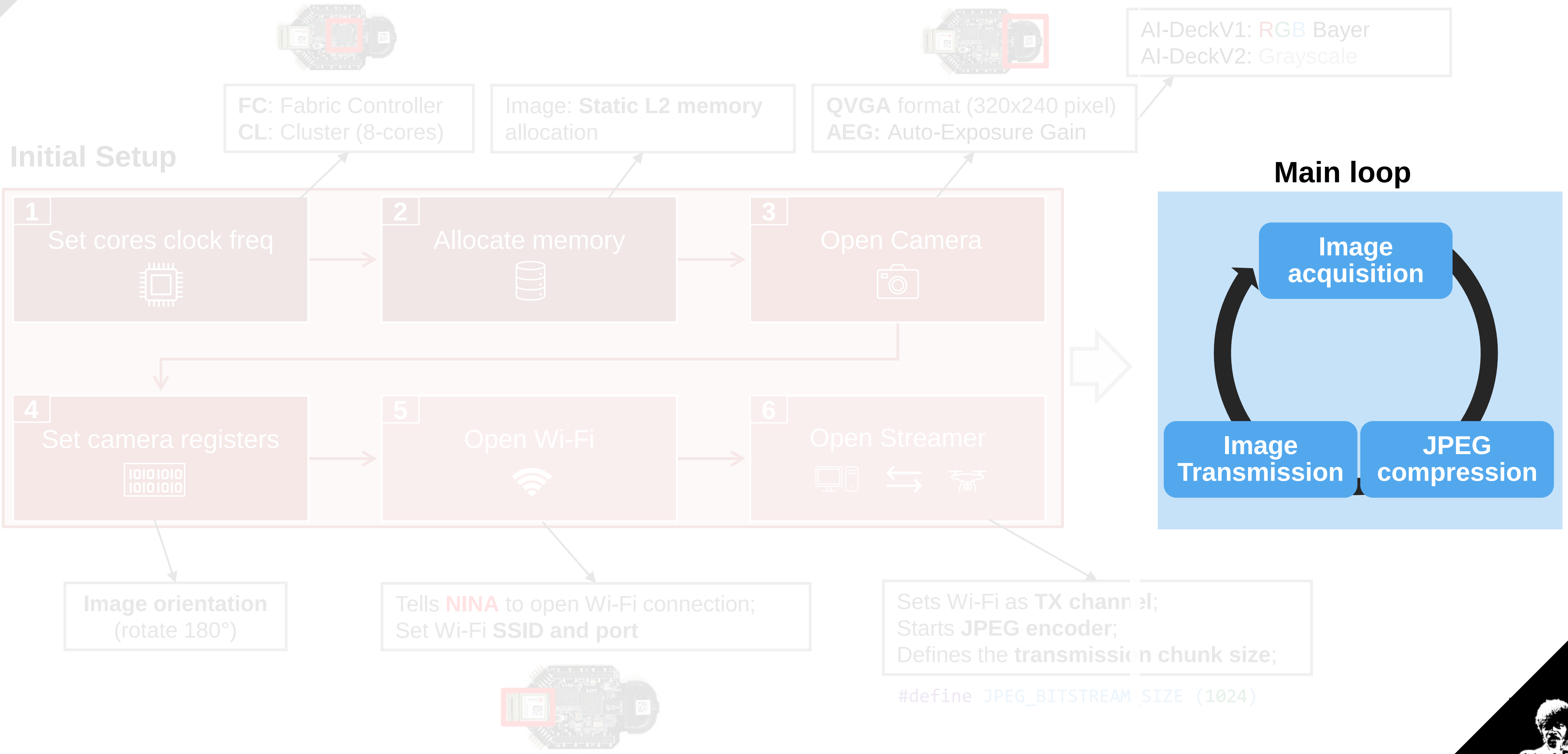
Image orientation
(rotate 180°)

Tells **NINA** to open Wi-Fi access-point;
Set Wi-Fi **SSID** and **port**

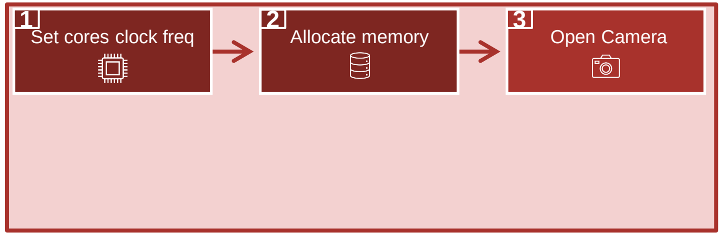
Sets Wi-Fi as **TX channel**;
Starts **JPEG encoder**;

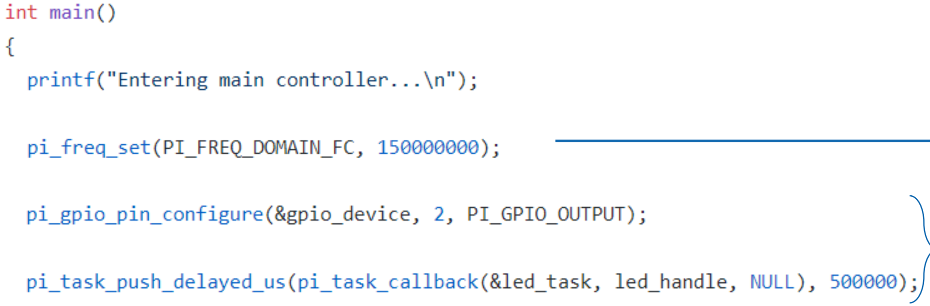


Wi-Fi Image streaming: Initial setup



Code inspection: Initial setup

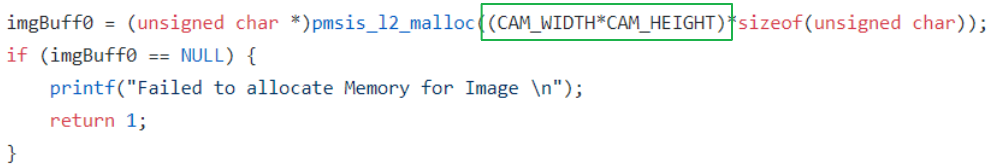




```
int main()
{
  printf("Entering main controller...\n");

  pi_freq_set(PI_FREQ_DOMAIN_FC, 150000000);

  pi_gpio_pin_configure(&gpio_device, 2, PI_GPIO_OUTPUT);

  pi_task_push_delayed_us(pi_task_callback(&led_task, led_handle, NULL), 500000);
```

1. Set the core frequency of the main GAP8's core (FC = Fabric Controller)

We configure the LED GPIO (LED#2) to "output mode"
so that we can control it.
Then we start the blinking task: `led_handle()`

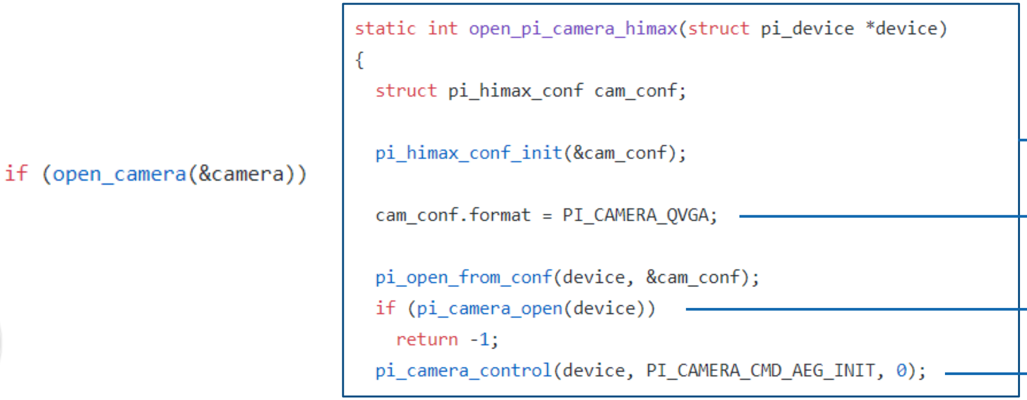


```
  imgBuff0 = (unsigned char *)pmsis_l2_malloc((CAM_WIDTH*CAM_HEIGHT)*sizeof(unsigned char));
  if (imgBuff0 == NULL) {
      printf("Failed to allocate Memory for Image \n");
      return 1;
  }
```

2. Allocate the memory for the image (QVGA format)

- CAM_WIDTH = 320
- CAM HEIGHT = 240

We use the L2 memory (512Kb), which is enough for storing an image.
In GAP8 you must specify the target memory for the malloc
(L2 in this case).



```
 if (open_camera(&camera))
```

```
static int open_pi_camera_himax(struct pi_device *device)
{
  struct pi_himax_conf cam_conf;

  pi_himax_conf_init(&cam_conf);

  cam_conf.format = PI_CAMERA_QVGA;

  pi_open_from_conf(device, &cam_conf);
  if (pi_camera_open(device))
    return -1;
  pi_camera_control(device, PI_CAMERA_CMD_AEG_INIT, 0);
```

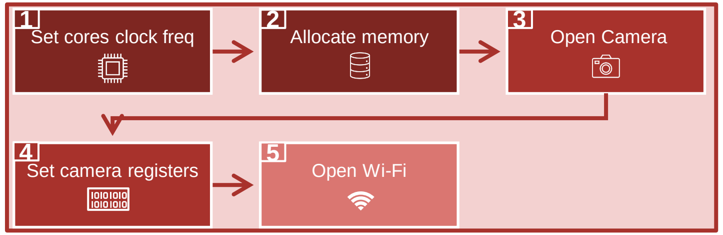
3. Open the camera

We specify the format between QVGA and QQVGA

Camera is opened

The AEG= auto-exposure-gain is activated

Code inspection: Initial setup



1 Set cores clock freq

2 Allocate memory

3 Open Camera

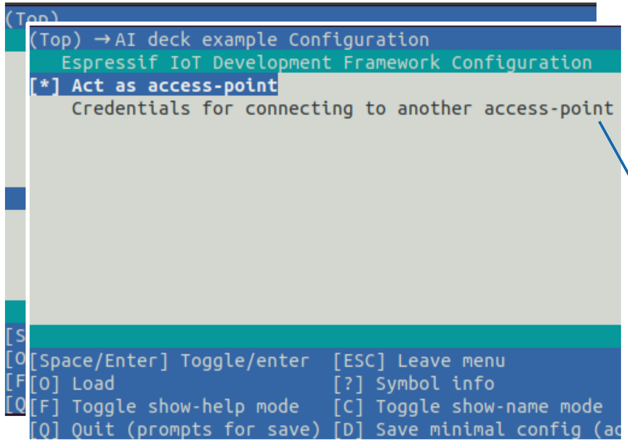
4 Set camera registers

5 Open Wi-Fi

```
pi_camera_reg_set(&camera, IMG_ORIENTATION, &set_value);
```

4. Set the camera registers to rotate the image by 180° (the image is upside-down by default).

```
if (open_wifi(&wifi))
```



```
(Top) → AI deck example Configuration
    Espressif IoT Development Framework Configuration
[*] Act as access-point
    Credentials for connecting to another access-point

[Space/Enter] Toggle/enter   [ESC] Leave menu
[O] Load                     [?] Symbol info
[F] Toggle show-help mode    [C] Toggle show-name mode
[Q] Quit (prompts for save)  [D] Save minimal config (ac
```

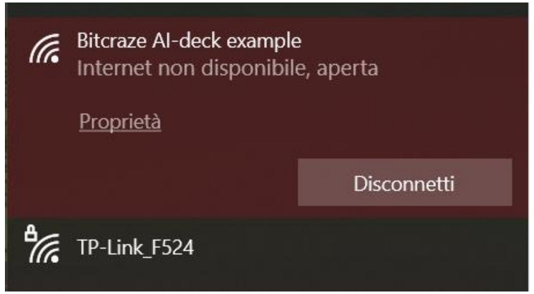
5. Open Wi-Fi

We open the Wi-Fi connection of the NINA Wi-Fi on-board module.

The configuration of NINA is loaded. To change it, you must modify the configuration and flash NINA

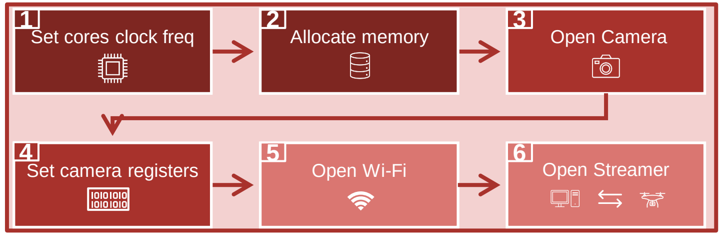
```
cd AIdeck_examples/NINA/firmware/
make menuconfig
(then follow instructions to flash NINA)
```

Instead of opening an access-point, you can also chose to connect to an existing one



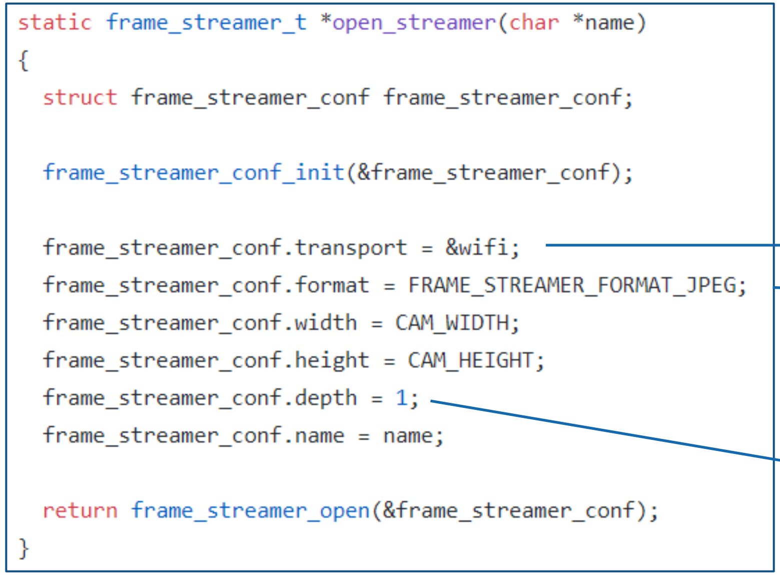
Now the "***Bitcraze AI-deck example***" SSID will appear in the Wi-Fi connections available.
We can **connect to it** with our Laptop (point-to-point).

Code inspection: Initial setup



```
streamer1 = open_streamer("camera");
```

6. Open the streamer



```
static frame_streamer_t *open_streamer(char *name)
{
  struct frame_streamer_conf frame_streamer_conf;

  frame_streamer_conf_init(&frame_streamer_conf);

  frame_streamer_conf.transport = &wifi;
  frame_streamer_conf.format = FRAME_STREAMER_FORMAT_JPEG;
  frame_streamer_conf.width = CAM_WIDTH;
  frame_streamer_conf.height = CAM_HEIGHT;
  frame_streamer_conf.depth = 1;
  frame_streamer_conf.name = name;

  return frame_streamer_open(&frame_streamer_conf);
}
```

We select Wi-Fi to stream images

We choose the image format

- **FRAME_STREAMER_FORMAT_JPEG**: enables the JPEG encoder
- **FRAME_STREAMER_FORMAT_RAW**: does not enable the JPEG encoder and streams raw images

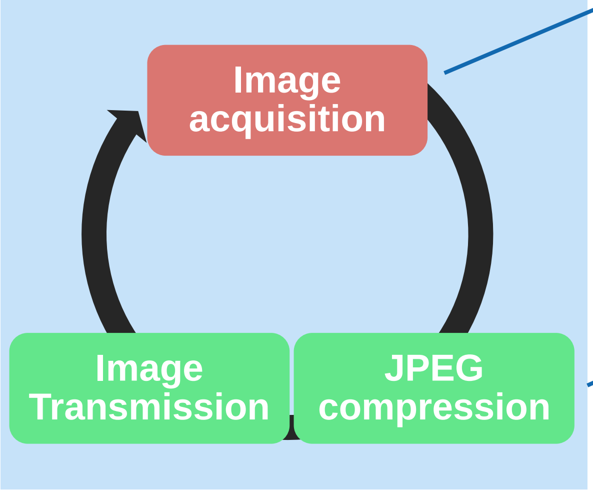
Image channels: Grayscale=1, RGB =3.
(But the Bayer RGB sensor AI-DeckV1 still uses one channel !)

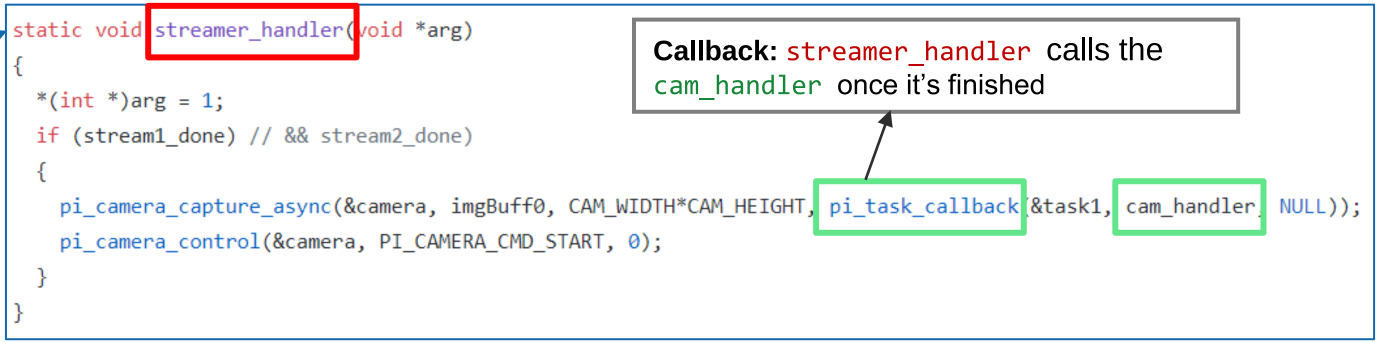
Hand-shaking between GAP8 and NINA Wi-Fi Module and the JPEG encoder is started.

Code inspection: Wi-Fi images transmission

```
pi_camera_control(&camera, PI_CAMERA_CMD_STOP, 0);
pi_camera_capture_async(&camera, imgBuff0, CAM_WIDTH*CAM_HEIGHT, pi_task_callback(&task1, cam_handler, NULL));
```

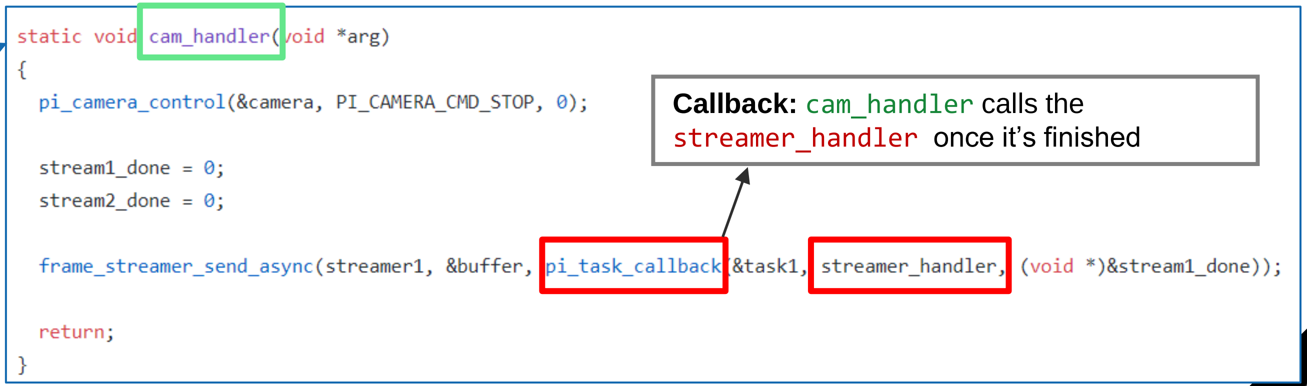
First image acquisition **starts the Main Loop**





```
static void streamer_handler(void *arg)
{
  *(int *)arg = 1;
  if (stream1_done) // && stream2_done)
  {
    pi_camera_capture_async(&camera, imgBuff0, CAM_WIDTH*CAM_HEIGHT, pi_task_callback(&task1, cam_handler, NULL));
    pi_camera_control(&camera, PI_CAMERA_CMD_START, 0);
  }
}
```

Callback: `streamer_handler` calls the `cam_handler` once it's finished



```
static void cam_handler(void *arg)
{
  pi_camera_control(&camera, PI_CAMERA_CMD_STOP, 0);

  stream1_done = 0;
  stream2_done = 0;

  frame_streamer_send_async(streamer1, &buffer, pi_task_callback(&task1, streamer_handler, (void *)&stream1_done));

  return;
}
```

Callback: `cam_handler` calls the `streamer_handler` once it's finished

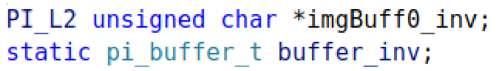
Hands on the code!!

Image manipulation before TX

We can manipulate the images before sending them via Wi-Fi:

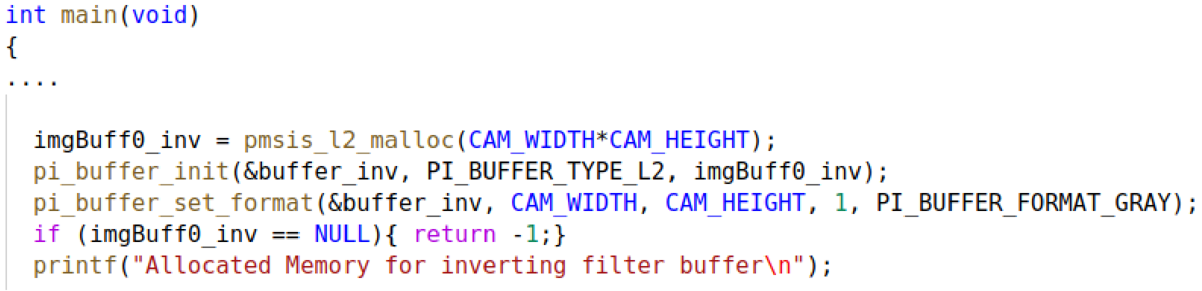
- We will be applying the same `inverting()` kernel that we used in the Hands-on session 2! →

`inverting()` inverts black & white in the image



```
PI_L2 unsigned char *imgBuff0_inv;
static pi_buffer_t buffer_inv;
```

→ Define a buffer as a global variable



```
int main(void)
{
....
   imgBuff0_inv = pmsis_l2_malloc(CAM_WIDTH*CAM_HEIGHT);
   pi_buffer_init(&buffer_inv, PI_BUFFER_TYPE_L2, imgBuff0_inv);
   pi_buffer_set_format(&buffer_inv, CAM_WIDTH, CAM_HEIGHT, 1, PI_BUFFER_FORMAT_GRAY);
   if (imgBuff0_inv == NULL){ return -1;}
   printf("Allocated Memory for inverting filter buffer\n");
```

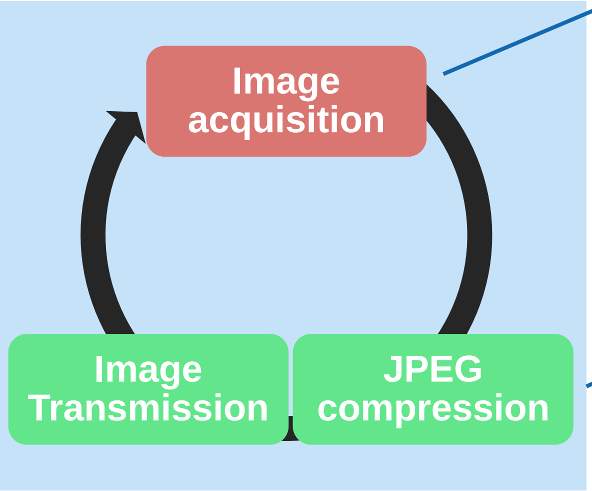
→ We allocate the memory for another image in the L2 memory

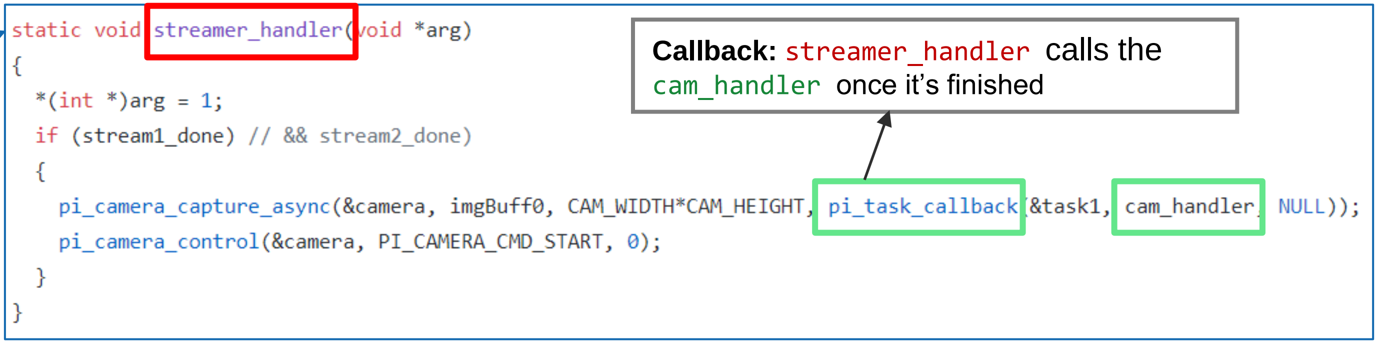
Image manipulation before TX

We keep the very same loop for transmission that we saw before, **but we manipulate the image with the `inverting()` function right before sending it**

`inverting()` inverts black & white in the image

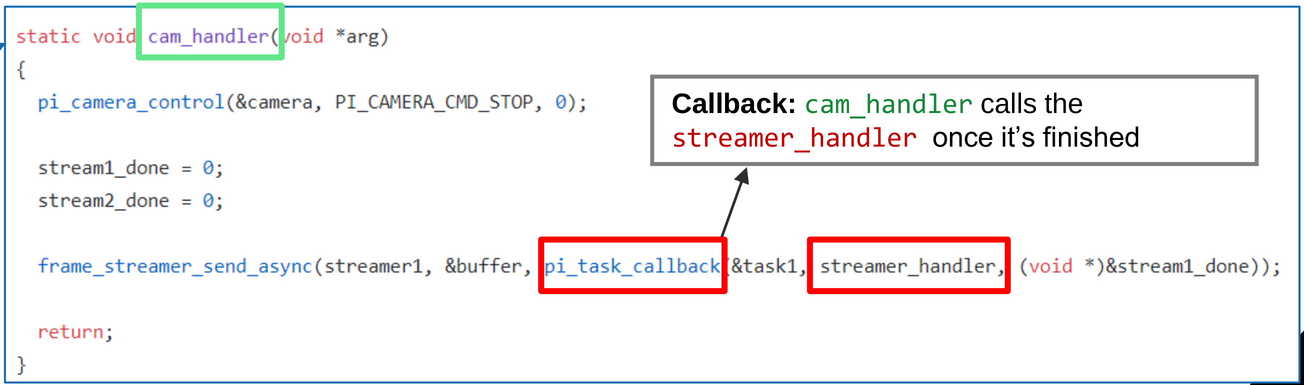
Main loop





```
static void streamer_handler(void *arg)
{
  *(int *)arg = 1;
  if (stream1_done) // && stream2_done)
  {
    pi_camera_capture_async(&camera, imgBuff0, CAM_WIDTH*CAM_HEIGHT, pi_task_callback(&task1, cam_handler, NULL));
    pi_camera_control(&camera, PI_CAMERA_CMD_START, 0);
  }
}
```

Callback: `streamer_handler` calls the `cam_handler` once it’s finished



```
static void cam_handler(void *arg)
{
  pi_camera_control(&camera, PI_CAMERA_CMD_STOP, 0);

  stream1_done = 0;
  stream2_done = 0;

  frame_streamer_send_async(streamer1, &buffer, pi_task_callback(&task1, streamer_handler, (void *)&stream1_done));

  return;
}
```

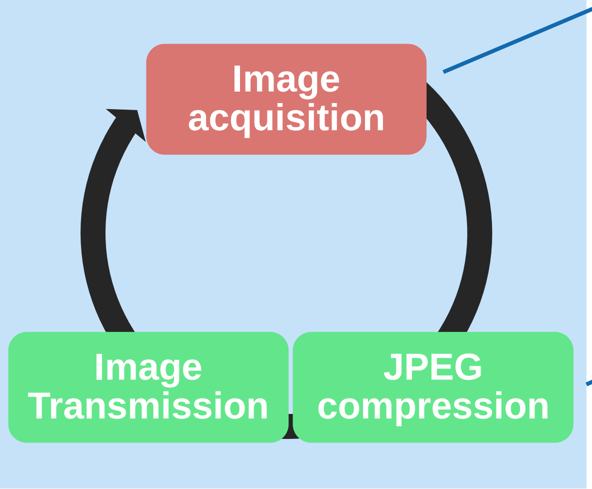
Callback: `cam_handler` calls the `streamer_handler` once it’s finished

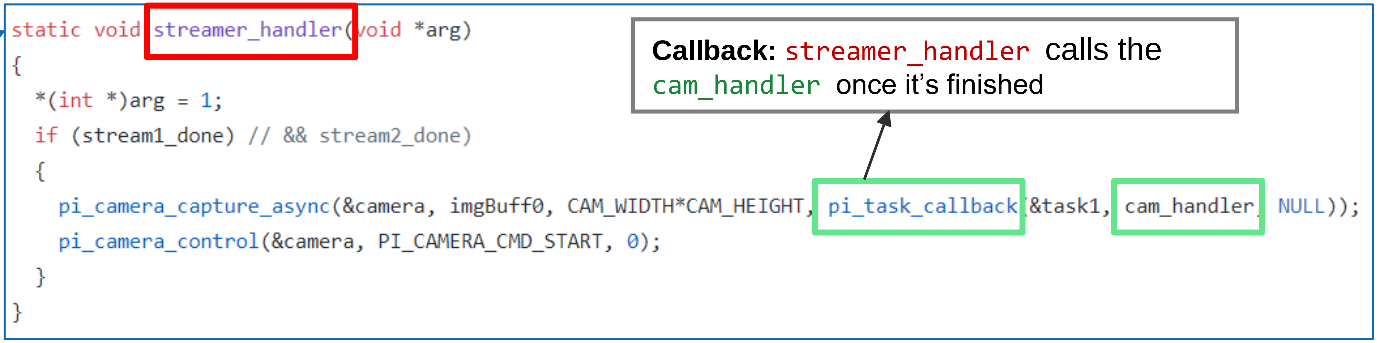
Image manipulation before TX

We keep the very same loop for transmission that we saw before, **but we manipulate the image with the `inverting()` function right before sending it**

→ `inverting()` inverts black & white in the image

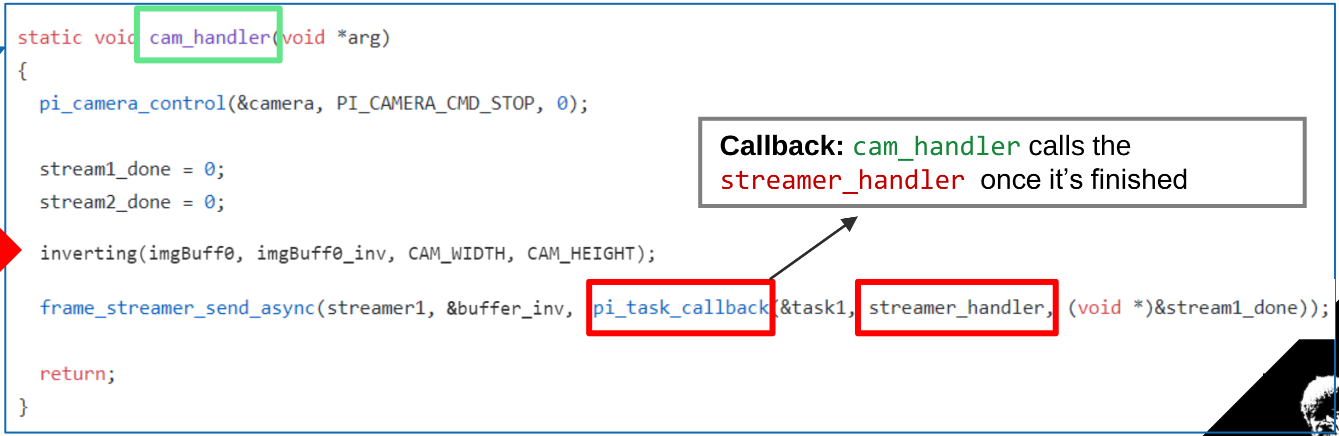
Main loop





```
static void streamer_handler(void *arg)
{
  *(int *)arg = 1;
  if (stream1_done) // && stream2_done)
  {
    pi_camera_capture_async(&camera, imgBuff0, CAM_WIDTH*CAM_HEIGHT, pi_task_callback(&task1, cam_handler, NULL));
    pi_camera_control(&camera, PI_CAMERA_CMD_START, 0);
  }
}
```

Callback: `streamer_handler` calls the `cam_handler` once it's finished

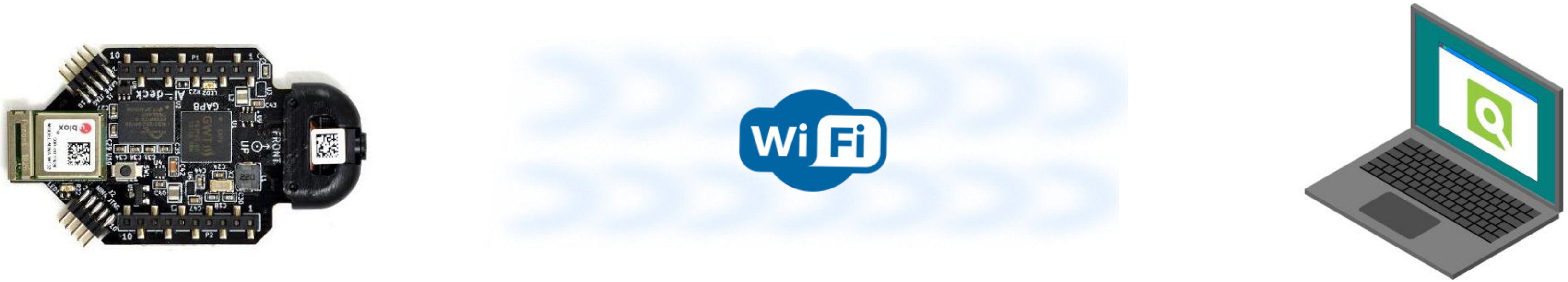


```
static void cam_handler(void *arg)
{
  pi_camera_control(&camera, PI_CAMERA_CMD_STOP, 0);

  stream1_done = 0;
  stream2_done = 0;

  inverting(imgBuff0, imgBuff0_inv, CAM_WIDTH, CAM_HEIGHT);

  frame_streamer_send_async(streamer1, &buffer_inv, pi_task_callback(&task1, streamer_handler, (void *)&stream1_done));

  return;
}
```

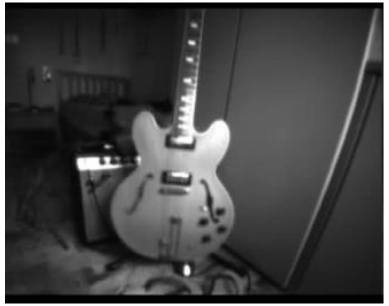
Callback: `cam_handler` calls the `streamer_handler` once it's finished

Image manipulation before TX

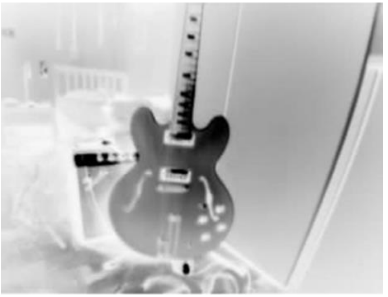
This is the behavior that we will experience



~~`inverting()`~~ **(Deactivated)**



`inverting()` **(Activated)**



Hands on the code!!

Thank you for your attention