Transmitting Images via LoRa

Max-Felix Müller

2021
January

Contents

1 Introduction **3**

2 Phase 1 **4**
2.1 Taking an Image 4
2.2 Viewing the Image with Python 5
2.2.1 Write a Binary File 5
2.2.2 Write into a Png Image 6
2.3 Dividing the Image into Messages 7
2.4 Actually Transmitting via LoRa 8

3 Phase 2 **9**
3.1 Protocol 9
3.1.1 Transmitter ID 9
3.1.2 Packet Length 9
3.1.3 Packet ID 9
3.1.4 CRC and Error Handling 9
3.1.5 Special Messages 10
3.1.6 Timeouts 10

4 Final Thoughts **11**

List of Figures

1 First transmitted Image 6
2 Second transmitted Image 7

1 Introduction

Using an EspCam the images can be streamed via a webserver. That option offers high data rates, but requires an active connection all the time. The goal of this project is to send single images via LoRa.

To do so a LoRa module is added to the EspCam. A second LoRa module is connected via a Teensy4.1 to the PC to which the received data is streamed. On the PC, a Python script will record the data and assemble the image.

The project is divided into two phases. In the first phase, the image shall be transmitted with no further requirements. Because the message length is limited, the image will have to be split in sections. This will be a one shot kind of transmission. When the receiver misses a message, part of the image will be lost.

In the second phase, a protocoll will be added. The protocoll enables messages between the transmitter and the receiver, allowing for error correction. This requires a connection in two ways. Because the module is the same on the transmitter and receiver, in this case it should always be possible to have two way communication if at least one way is confirmed working. Theoretically this does not have to be the case, when for example the receiver in one module is more sensitive or a module has higher transmit power.

2 Phase 1

2.1 Taking an Image

Surprisingly there are example codes for streaming videos via webservers, but none to just capture the raw image data. There is however a very nice entry on the ESP32 forum. Taking an image requires configuring the camera connections and image data structure.

First the camera is configured. Since it is not planned to change the camera, all options are hard coded.

Creating a camera configuration uses its own type definition.

```
camera_config_t config; // create a camera configuration
```

The led can be controlled via the camer. This allows using it as the flash for example.

```
config.ledc_channel = LEDC_CHANNEL_0; // led channel
config.ledc_timer = LEDC_TIMER_0; // led timer
```

Then the pins of the connector are configured. Summarized there are data, clock and some miscellaneous pins.

```
// pin definitions for CAMERA_MODEL_AI_THINKER
config.pin_d0 = 5; // data pin
config.pin_d1 = 18; // data pin
config.pin_d2 = 19; // data pin
config.pin_d3 = 21; // data pin
config.pin_d4 = 36; // data pin
config.pin_d5 = 39; // data pin
config.pin_d6 = 34; // data pin
config.pin_d7 = 35; // data pin
config.pin_xclk = 0; // crystal
config.pin_pclk = 22; // pixel clock
config.pin_vsync = 25; // vertical synchronization
config.pin_href = 23;
config.pin_sscb_sda = 26; // serial data
config.pin_sscb_scl = 27; // serial clock
config.pin_pwdn = 32; // power down
config.pin_reset = -1; // reset not available
```

Next some general configuration is done. The clock had to be reduced from 20 MHz to 5 MHz in order to avoid an error where the frame buffer was not received properly. Using the smallest resolution and a grayscale, the amount of data is reduced for now. Later those values can be increased, but the transmission will go longer.

```
config.xclk_freq_hz = 5000000; // set clock to 5 MHz
config.pixel_format = PIXFORMAT_GRAYSCALE;
// set the format to rgb with 8 bit per color
config.frame_size = FRAMESIZE_SVGA; // 800x600
config.fb_count = 1; // one frame buffer
```

Finally the camera can be initialized with the given configuration. If there would be a problem for example with the pinout, theis would throw an error.

```
esp_err_t err = esp_camera_init(&config);
// initialize the camera with the given settings, get an error code
if (err != ESP_OK) { // if the error code was not ok
  #ifdef SERIAL_DEBUG
  Serial.printf("Camera init failed with error 0x%x", err);
  // info message, print the error code
  #endif
  return false; // no success
}
```

If everything has worked out, a boolean true is returned.

```
return true; // camera initialized
```

After the initialization, an image can be taken by requesting the frame buffer. This will throw a timeout error, if the clock is too fast. Hence the reduction to 5 MHz.

```
fb = esp_camera_fb_get(); // get a pointer to the frame buffer
```

If the buffer is empty, the capture failed.

```
if (!fb) { // if the pointer is still a null pointer
  #ifdef SERIAL_DEBUG
  Serial.println("Camera capture failed"); // info message, capture failed
  #endif
  return; // abort
}
```

Otherwise the data can be sent via USB using Serial.write to transfer each byte. For 800x600 pixels the transmission with 9600 baud takes quite a while. There is no use in speeding it up, since the LoRa transmission will be even slower later on.

```
Serial.write(fb->buf, fb->len); // payload (image), payload length
```

2.2 Viewing the Image with Python

The code to view the image has been split into two parts. The first script is used to read the bytes coming in via USB and write them to a binary file as storage. The second script reads that binary file and interprets it as a grayscale image which is then stored as such.

2.2.1 Write a Binary File

The serial port is opened.

```
usb = serial.Serial('/dev/ttyUSB0', 9600) # open serial port
```

When starting the ESP it prints some debug messages via serial. To clear those, the input buffer can be cleared before starting to read bytes. Then the script sends a 0x00 to start the ESP code.

```
usb.reset_input_buffer() # clear the startup data from the espcam
usb.write(b'0') # write something to start execution
```

A for loop is executed as many times as there are pixels in the image. For each pixel a byte is read and stored into the binary file.

```
with open(f"{PATH}image.bin", "wb") as file: # open the binary file
    for _ in tqdm(range(800 * 600)): # read one byte for each pixel
        received = usb.read() # read the byte
        file.write(received) # write it to the file
```

2.2.2 Write into a Png Image

The binary file is read as a numpy array.

```
image = np.fromfile(f"{PATH}image.bin", dtype='uint8')
# read the binary image into a numpy array
```

The array is then interpreted as a grayscale image. To do so, the size has to be set. Also notice, that the ESP was hold upside down when taking the image. That is compensated here by rotating the image.

```
cv2.imwrite(f"{PATH}image.png", cv2.rotate(image[:480000].reshape(600,800), 1))
# rotate the image and store it in a file
```

This is the first image. It is blurry because the lens was moved...

Figure 1: First transmitted Image

2.3 Dividing the Image into Messages

Dividing the message into multiple messages is done by limiting the number of bytes send per write() function. To compensate for the bytes already sent, a offset is added to the buffer address for each iteration.

```
#define MESSAGE_LENGTH 1000 // 480000 / 1000 = 480 messages
for (uint64_t offset = 0; offset < 800*600; offset += MESSAGE_LENGTH) {
  // split all pixel data into segments
  Serial.write(fb->buf + offset, MESSAGE_LENGTH);
  // payload (image), payload length
}
```

To prove the image is still complete, the messages have to be assembled back in the Python script. Here is a second image to prove that. Actually, while there is no additional information in each message, the Python script does not have to be changed.

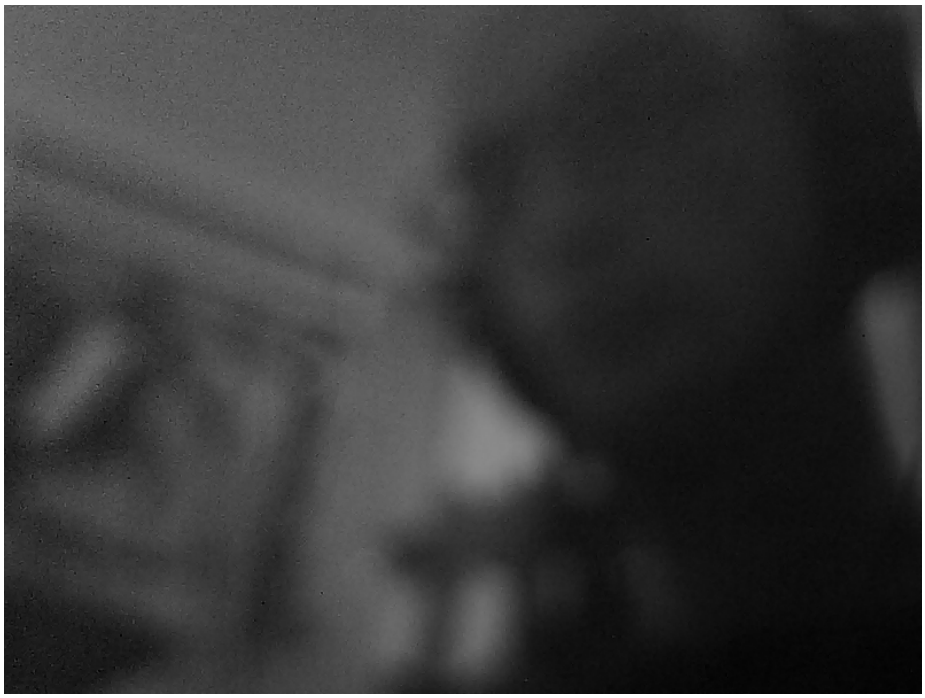

Figure 2: Second transmitted Image

2.4 Actually Transmitting via LoRa

All data transmission up to this point has been done via serial. Using a baudrate of 9600 baud, the transmission of a grayscale 800x600 image took roughly 8 minutes. Now, using the EBYTE modules, the data will be transmitted via LoRa from the EspCam to a Teensy and from there to the PC.

First the receiver code is written. For now a Teensy is used that simply relays the received bytes from the LoRa module to the PC.

```
if (LoRa.available()) { // if there are incoming bytes
  uint8_t input = LoRa.GetByte();
  // get the next byte from the module
  USB.write(input); // write it to the laptop
}
```

The initialization of the module is done by giving it a reference to the serial port and the pins for the configuration and auxiliary connections.

```
EBYTE LoRa = EBYTE(&LORA_S, 4, 3, 2);

void setup() {
  LORA_S.begin(9600); // open the serial port
  LoRa.init(); // initialize the module
}
```

The Python receive script was slightly modified for the changed port of the Teensy versus the EspCam.

```
usb = serial.Serial('/dev/ttyACM0', 9600) # open serial port
```

After also including and initializing the EBYTE code on the EspCam there is only one change left to be done. Instead of writing the frame buffer out to the serial port, instead it is written to the EBYTE LoRa module. This works nearly the same, also requiring a pointer and the length that shall be written.

```
for (uint64_t offset = 0; offset < 800*600; offset += MESSAGE_LENGTH) {
  // split all pixel data into segments
  LoRa.SendStruct(fb->buf + offset, MESSAGE_LENGTH);
  // send a piece of the image data via the lora module
}
```

Using LoRa to transmit the image lowers the data rate even more, goring from 8 minutes to about 1 hour and 30 minutes. Well, now it is clear why LoRa is not used to transmit image data. Anyhow, the protocoll that would have been used is documented here.

During the transmission an error occured, forcing an abort. A change in the Python script still allows viewing the image if it is not complete. To do so, the data is extended with zeros.

```
if image.shape[0] < 480000: # if the image was not complete
    print("Image not complete!") # notify the user
    trailingzeros = np.zeros(480000 - image.shape[0], dtype='uint8')
    # create an array with the missing zeros
    image = np.append(image, trailingzeros) # extend with zeros
```

But actually the image is bad enough not to be shown here.

3 Phase 2

To ensure a reliable data transmission over larger distances, a protocoll with error correction and packet identification is required. Instead of using an existing protocoll, a new one is specifically created fulfilling the needs for this project. These are:

- Timeouts
- Transmitter identification
- Acknowledge of each packet
- Error checking with repeated transmission, no error correction

3.1 Protocol

The protocoll uses communication messages and data messages. Communication messages mainly serve the flow control. Data messages include the image data.

A communication message starts with the Transmitter ID, followed by a byte containing the message. These bytes can be found under Special Messages. Finally a CRC is sent to check for transmission errors.

The data message hold the Transmitter ID, Packet ID, a data block and the CRC. The standard length of a data block is 100 bytes, but the size can be changed using the Special Message.

Finally there is a Acknowledge or Not Acknowledge message, containing the Packet ID, ack or nack and a CRC. Acknowledge is 0xAA and Not Acknowledge is 0x55.

3.1.1 Transmitter ID

To identify different transmitters, each one has its unique id. It can be freely chosen between 0 and 255 and is one byte.

3.1.2 Packet Length

The standard length of a data block is 100 bytes, but the size can be changed using the Special Message. Sending a 0x06 followed by a two byte length of the data blocks for the next image transmission.

3.1.3 Packet ID

Each packet is numbered with an eight bit number. This allows flow control and checks for packet loss.

3.1.4 CRC and Error Handling

The CRC is a one byte field. To calculate it, add all other bytes of the message.

If the CRC does not match, discard the whole message and ask for it to be resent using the not acknowledge message. There is no error correction.

3.1.5 Special Messages

- 0x00 - Request an image
- 0x01 - Image ready
- 0x02 - Receive ready
- 0x03 - Image done
- 0x04 - Image complete
- 0x05 - Image repeat
- 0x06 - Block length change
- 0xFF - Abort transmission

3.1.6 Timeouts

Timeouts are used to give a point of continuation when no packet was received for a while.

The timeout on the transmitter side is 30s if no acknowledge or not acknowledge message was received. After that, the message will be repeated.

The receiver has a 30s timeout, after which he will discard the image and request a new one.

4 Final Thoughts

I really thought it would be a good idea to transmit images via LoRa. Although I knew about the low data rate, I did not realize how slow it really is.

Let's do some quick math. A 800x600 image in grayscale takes 8 minutes. For a colored image using RGB565 (5 bit red, 6 bit green and 5 bit blue) the time would double, even triple for RGB888. That is 4 hours 30 minutes. Increasing the resolution to the maximum is 1600x1200 which is double in both dimensions, so quadruple the pixels. A grayscale image at that resolution would therefore take about 6 hours and in full color, the time would go up to 18 hours for a single picture.

At least now I know...

The protocoll, as it is described here, is probably useless. I imagine there being some flaws which I could have found if I actually implemented it. That will have to wait for another opportunity.