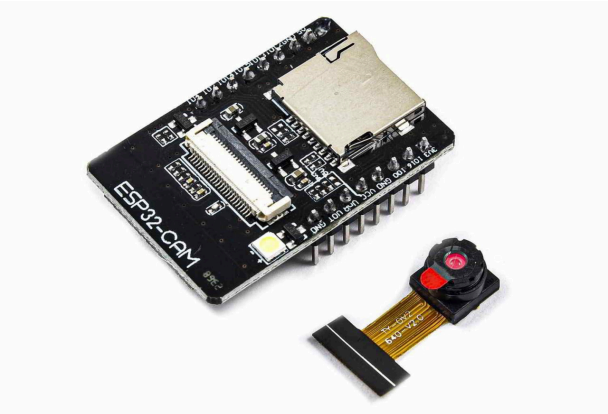
به نام خداوند بخشنده و مهربان

دوربین تصویربرداری و پخش زنده با انتقال داده به روش Blocking



پروژه‌ی درس مدارهای واسط - پائیز ۱۴۰۳

استاد: دکتر امین فصحتی

اعضای گروه: سعید فراتی کاشانی - مهدی علی‌نژاد - پوریا غفوری

فهرست

مقدمه

در دنیای امروز، نظارت و پایش از راه دور یکی از نیازهای اساسی در حوزه‌های امنیتی، هوشمندسازی و اتوماسیون محسوب می‌شود. پروژه حاضر یک سیستم ساده اما کارآمد برای ارسال و استریم تصاویر زنده از ماژول ESP32-CAM طراحی کرده است. این سیستم قادر است تصاویر را از طریق شبکه به یک سرور Flask ارسال کند و سپس سرور، تصویر را به یک چت تلگرام ارسال نماید.

این پروژه ترکیبی از اینترنت اشیا (IoT)، پردازش تصویر در سرور و پیام‌رسانی خودکار از طریق API تلگرام را ارائه می‌دهد. از این روش می‌توان برای نظارت بر محیط‌های مختلف مانند منازل، دفاتر، مزارع و هر جایی که نیاز به پایش تصویری از راه دور وجود دارد، استفاده کرد.

قالب اولیه

این بخش از گزارش، برای استفاده از ماژول ESP32-CAM جهت ارسال تصاویر به یک سرور مجازی خارجی (برای ارسال به تلگرام) و همچنین استریم زنده تصاویر از دوربین روی شبکه طراحی شده است. در ادامه عملکرد بخش‌های مختلف کد توضیح داده می‌شود. شایان ذکر است که در این بخش هنوز از تکنینک‌های Blocking استفاده نشده است.

تنظیمات و اتصال به WiFi

در ابتدای کد، کتابخانه‌های مورد نیاز مانند esp_camera.h و WiFi.h اضافه شده‌اند. سپس اطلاعات شبکه SSID و Password مشخص می‌شوند. کد تلاش می‌کند به شبکه WiFi متصل شود و در صورت موفقیت، آدرس IP تخصیص داده شده به ماژول را در Serial Monitor چاپ می‌کند.

ارسال تصویر به سرور

تابع sendPhotoToVPS مسئولیت ارسال تصویر گرفته شده از دوربین را به سرور VPS دارد. این تابع موارد زیر را انجام می‌دهد:

بررسی اتصال WiFi

- برقراری ارتباط با سرور از طریق پروتکل HTTP
- ارسال درخواست POST همراه با تصویر به سرور
- دریافت و نمایش پاسخ سرور

تصویر به صورت Multipart Form-Data ارسال شده و پس از دریافت پاسخ از سرور، اتصال بسته می‌شود.

راه‌اندازی سرور داخلی برای استریم تصاویر

تابع startCameraServer یک سرور HTTP روی پورت 80 اجرا می‌کند که امکان استریم زنده تصاویر را فراهم می‌سازد.

در handleClientStream، پس از برقراری ارتباط با یک کلاینت، دوربین به طور مداوم تصاویر را گرفته و با فرمت multipart/x-mixed-replace ارسال می‌کند. این روش به مرورگر امکان نمایش تصاویر پیوسته را بدون نیاز به بارگذاری مجدد صفحه می‌دهد.

```
void handleClientStream(WiFiClient client) {
  client.println("HTTP/1.1 200 OK");
            client.println("Content-Type:      multipart/x-mixed-replace;
boundary=frame");
  client.println();

  while (true) {
    camera_fb_t *fb = esp_camera_fb_get();
    if (!fb) {
      Serial.println("Camera capture failed");
      client.stop();
      return;
    }
```

```
    client.printf("--frame\r\nContent-Type: image/jpeg\r\nContent-Length:
%d\r\n\r\n", fb->len);
    client.write(fb->buf, fb->len);
    client.println("\r\n");

    esp_camera_fb_return(fb);

    if (!client.connected()) break;
  }
}
```

تصویر 1 - راه‌اندازی سرور

اجرای دستورات از طریق پورت سریال

در تابع handleSerialCommand، ماژول ESP32-CAM می‌تواند دستورات دریافتی از پورت سریال را پردازش کند. اگر کاربر مقدار p را در Serial Monitor ارسال کند، دوربین یک عکس گرفته و از طریق تابع sendPhotoToVPS به سرور ارسال می‌کند.

```
void handleSerialCommand(void *parameter) {
  while (true) {
    if (Serial.available()) {
      char command = Serial.read();
      if (command == 'p') {
        camera_fb_t *fb = esp_camera_fb_get();
        if (fb) {
          sendPhotoToVPS(fb);
          esp_camera_fb_return(fb);
          Serial.println("Photo sent to VPS");
        } else {
          Serial.println("Failed to capture photo");
        }
      }
    }
    delay(10);
  }
}
```

تصویر 2 - اجرای دستورات

مقداردهی اولیه دوربین

در تابع setup، ابتدا ارتباط سریال و WiFi مقداردهی می‌شود. سپس پیکربندی سخت‌افزاری دوربین از طریق camera_config_t تنظیم شده و دوربین مقداردهی اولیه می‌شود. در صورت موفقیت، سرور HTTP داخلی راه‌اندازی شده و یک تسک FreeRTOS برای پردازش دستورات سریال ایجاد می‌شود.

حلقه اصلی برنامه

در loop، ماژول بررسی می‌کند که آیا کلاینتی به سرور متصل شده است یا خیر. در صورت اتصال، تابع handleClientStream اجرا می‌شود تا تصاویر زنده از دوربین به کلاینت ارسال شوند.

کد سرور Flask برای دریافت و ارسال تصاویر

این کد یک سرور Flask ایجاد می‌کند که تصاویر دریافت‌شده از ماژول ESP32-CAM را پردازش کرده و به یک چت تلگرامی ارسال می‌کند. در ادامه، عملکرد بخش‌های مختلف این کد توضیح داده می‌شود.

تنظیمات اولیه

ابتدا کتابخانه‌های مورد نیاز مانند Flask برای راه‌اندازی سرور، requests برای ارسال درخواست‌های HTTP و datetime برای ثبت زمان بارگیری می‌شوند. سپس یک نمونه از Flask ایجاد شده و دو متغیر TELEGRAM_TOKEN و CHAT_ID برای احراز هویت و ارسال پیام در تلگرام تعریف می‌شوند.

مسیر \send_photo برای دریافت تصویر

این مسیر از طریق متد POST تصاویر را دریافت می‌کند. فرآیند پردازش به شرح زیر است:

1. بررسی می‌شود که آیا فایل تصویر در درخواست وجود دارد یا خیر. اگر نباشد، یک پیام خطا (کد 400) بازگردانده می‌شود.
2. فایل دریافت شده در متغیر photo ذخیره می‌شود.
3. تاریخ و زمان دریافت تصویر ثبت شده و در کپشن تصویر استفاده می‌شود.
4. تصویر از طریق API تلگرام با استفاده از توکن TELEGRAM_TOKEN به چت مشخص‌شده در CHAT_ID ارسال می‌شود.
5. اگر ارسال موفق باشد، پاسخ "Photo sent" با کد 200 بازگردانده می‌شود، در غیر این صورت، پیام خطای دریافتی از تلگرام همراه با کد 500 ارسال خواهد شد.

کدگذاری 8b/10b

کدگذاری 8b/10b یک روش کدگذاری خطی است که در سیستم‌های ارتباطی سریال با سرعت بالا برای تضمین تعادل DC و محدودیت طول دنباله‌های یکسان استفاده می‌شود. در این روش، هر 8 بیت داده به یک کد 10 بیتی تبدیل می‌شود که به حفظ تعادل تعداد بیت‌های '0' و '1' کمک می‌کند و انتقال داده‌ها را قابل‌اعتمادتر می‌سازد. این کدگذاری در استانداردهایی مانند اترنت گیگابیتی، PCIe و SATA به کار می‌رود.

در کدگذاری 8b/10b، هشت بیت ورودی به دو بخش تقسیم می‌شود: 5 بیت پایین و 3 بیت بالا. بخش 5 بیتی به یک کد 6 بیتی و بخش 3 بیتی به یک کد 4 بیتی تبدیل می‌شود. این دو کد سپس ترکیب شده و یک کد 10 بیتی را تشکیل می‌دهند. این روش به حفظ تعادل DC و محدودیت طول دنباله‌های یکسان کمک می‌کند.

در کدگذاری 8b/10b، مفهوم "اختلاف جاری" (Running Disparity) اهمیت دارد. این اختلاف نشان‌دهنده تفاوت بین تعداد بیت‌های '1' و '0' در یک دنباله است. هدف این است که این اختلاف در طول زمان نزدیک به صفر باقی بماند تا تعادل DC حفظ شود. برای این منظور، برخی از ورودی‌های 8 بیتی ممکن است دو کد 10 بیتی معتبر داشته باشند و انتخاب بین آن‌ها به اختلاف جاری بستگی دارد.

در کدگذاری 8b/10b، علاوه بر کدهای داده، کدهای کنترلی خاصی نیز تعریف شده‌اند که برای اهدافی مانند همگام‌سازی استفاده می‌شوند. این کدهای کنترلی به گیرنده کمک می‌کنند تا دنباله داده‌ها را به درستی تفسیر کند و همگام‌سازی را حفظ کند. همچنین در این کدگذاری از یک جدول نگاشت برای رمزگذاری و رمزگشایی استفاده می‌شود.

نحوه کدگذاری

در کدگذاری، این انکودینگ از Running Disparity استفاده می‌کند، این مفهوم به عنوان یک متغیر با نام rd تعریف می‌شود و از ابتدا صفر مقدار دهی شده است. همچنین یک آرایه در مورد نیاز است که در آن ایندکس هر عضو، داده‌ی 8 بیتی به همراه rd و مقدار آن عضو، داده‌ی 10 بیتی به همراه rd است.

1. **یافتن ایندکس در جدول کدگذاری:**
   - اگر rd مقدار true داشته باشد، **512** به مقدار input اضافه می‌شود تا نسخه مناسب از کدگذاری انتخاب شود.
   - در غیر این صورت، مقدار input بدون تغییر باقی می‌ماند.

```
int index = (*rd ? 512 : 0) + input;
```

تصویر 3 - یافتن ایندکس در جدول

2. **دریافت مقدار 11 بیتی:**
   - از **جدول tbl8b10b** مقدار رمزگذاری‌شده‌ی متناظر با input و rd دریافت می‌شود.

```
char *encodedStr = tbl8b10b[index];
```

تصویر 4 - دریافت مقدار 11 بیتی

3. **تبدیل مقدار 10 بیتی از رشته به عدد:**
   - مقدار دودویی (0 و 1) از رشته‌ی متنی استخراج و به یک عدد uint16_t تبدیل می‌شود.

```
uint16_t encoded = 0;

for (int i = 1; i <= 10; ++i) {

   encoded = (encoded << 1)
                | (encodedStr[i] == '1' ?
1 : 0);

}
```

تصویر 5 - تبدیل مقدار 10 بیتی از رشته به عدد

4. **به‌روزرسانی rd:**
   - مقدار rd بر اساس اولین بیت از رشته‌ی رمزگذاری‌شده تنظیم می‌شود.
   - این مقدار در مرحله بعدی رمزگذاری استفاده می‌شود.

```
int ones = 0, zeros = 0;

for (int i = 0; i < 10; i++) {

   if (encodedStr[i] == '1') ones++;

   else zeros++;

}

*rd = (ones > zeros);
```

تصویر 6 - به‌روزرسانی rd

5. **بازگرداندن مقدار رمزگذاری‌شده:**
   - مقدار 10 بیتی نهایی به عنوان خروجی تابع برگردانده می‌شود.

نحوه کدگشایی

1. **بررسی مقدار ورودی:**
   - مقدار ورودی باید حداکثر 10 بیت باشد (کوچک‌تر از 1024).
   - اگر مقدار ورودی نامعتبر باشد، برنامه متوقف می‌شود.

```
assert data_in <= 0x3FF, "Data in must be
maximum 10 bits"
```

تصویر 7 - بررسی مقدار ورودی

2. **جستجو در جدول رمزگشایی:**
   - مقدار **data_in** در جدول **dec_lookup** جستجو می‌شود.
   - اگر مقدار پیدا نشود، یعنی داده ورودی معتبر نیست و خطا نمایش داده می‌شود.

```
decoded = EncDec_8B10B.dec_lookup[data_in] if
       data_in < len(EncDec_8B10B.dec_lookup)
       else "DEC8b10bERR"

if decoded == "DEC8b10bERR":

   print(f"Error: {data_in:010b} is not a
valid 8b/10b code")

   raise Exception("Input to 8B10B Decoder is
not a 8B10B Encoded Word")
```

تصویر 8 - جستجو در جدول رمزگشایی

3. **تبدیل مقدار از رشته‌ی باینری به عدد:**
   - مقدار 8 بیتی اصلی از مقدار 10 بیتی استخراج می‌شود.

```
decoded_int = int(decoded, 2)
```

تصویر 9 - تبدیل مقدار از رشته‌ی باینری به عدد

4. **بررسی بیت کنترل (ctrl):**
   - بیت کنترل (Control Bit) بررسی می‌شود که نشان می‌دهد مقدار دریافتی یک کاراکتر داده‌ای یا کنترلی است.
   - اگر مقدار 1 باشد، یعنی داده‌ی کنترلی است.
   - اگر مقدار 0 باشد، یعنی داده‌ی معمولی است.

```
ctrl = (decoded_int >> 8) & 0x1

decoded_int &= 0xFF  # Extract 8-bit value
```

تصویر 10 - بررسی بیت کنترل

5. **برگرداندن مقدار نهایی:**
   - مقدار 8 بیتی اصلی و بیت کنترل به عنوان خروجی بازگردانده می‌شوند.

نحوه تشخیص خطا

در این کدگذاری تشخیص خطا به این صورت است که تعدادی از داده‌های جدول کدگشا هیچوقت مورد استفاده قرار نمی‌گیرند از سمت کدگذار و از این طریق کدگشا می‌تواند در برخی موارد متوجه بروز خطا بشود. کد زیر به هدف آنالیز درصد خطا زده شده است. این کد 256 داده را کدگذاری می‌کند، یک بیت رندوم از آن را تغییر می‌دهد و به کدگشا می‌دهد و تعداد داده‌هایی که خطای آنها تشخیص داده می‌شود را می‌شمارد.
بعد از یک بار اجرای آن متوجه شدیم که حدود 30 درصد داده‌هایی که دچار خطا شده بودند، به درستی تشخیص داده شده بودند و 70 درصد باقی مانده، خطای آن‌ها تشخیص داده نشد و به عنوان داده صحیح در نظر گرفته شدند.

```
    @staticmethod
    def dec_8b10b(data_in, verbose=False):
        assert data_in <= 0x3FF, "Data in must be maximum 10 bits"
        if verbose:
            print(f"Decoding input: {data_in:010b} ({hex(data_in)})")
              decoded  =  EncDec_8B10B.dec_lookup[data_in]  if  data_in  <
len(EncDec_8B10B.dec_lookup) else "DEC8b10bERR"
        if decoded == "DEC8b10bERR":
            print(f"Error: {data_in:010b} is not a valid 8b/10b code")
```

```
                raise Exception("Input to 8B10B Decoder is not a 8B10B
Encoded Word")
        decoded_int = int(decoded, 2)
        ctrl = (decoded_int >> 8) & 0x1
        decoded_int &= 0xFF  # Extract 8-bit value
        if verbose:
            print(f"Decoded: {decoded_int:02X} - Control: {ctrl}")
        return ctrl, decoded_int
    @staticmethod
    def encode(data_in, disparity=0, verbose=False):
              encoded_value  =  EncDec_8B10B.enc_lookup[data_in  +  512  *
disparity]
        if verbose:
            print(f"encoding value {data_in} to {encoded_value}")
        return int(encoded_value[1:], 2), disparity
if __name__ == "__main__":
    test = [random.randint(0, 255) for _ in range(256)]
    decoded = []
    catch = 0
    for data in test:
        value_as_number = data
        encoded = EncDec_8B10B.encode(value_as_number)[0]
        bit_to_flip = 1 << random.randint(0, 9)
        faulty_encoded = encoded ^ bit_to_flip  # Flip one bit
        try:
            decoded = EncDec_8B10B.dec_8b10b(faulty_encoded)[1]
        except Exception as e:
            catch += 1
    print(catch)
```

تصویر 11 - بررسی تشخیص خطا

کدگذاری 64b/66b

کدگذاری 64b/66b یکی از روش‌های کدگذاری خطی است که در شبکه‌های پرسرعت مانند 10GbE (اترنت ۱۰ گیگابیتی) و سایر پروتکل‌های پرسرعت مانند SATA، PCIe و Infiniband استفاده می‌شود که ما در این پروژه نیز برای ارسال عکس‌های خود از ESP به سرور از آن استفاده کردیم.
از مزایای این روش می‌توان به موارد زیر اشاره کرد:

1. **افزایش بهره‌وری پهنای باند:** در روش 8b/10b، هر ۸ بیت داده به ۱۰ بیت کدگذاری می‌شود که بازدهی ۸۰٪ دارد، اما در 64b/66b، هر ۶۴ بیت داده به ۶۶ بیت کدگذاری می‌شود که بازدهی ۹۶.۹۷٪ دارد. این افزایش بازدهی برای لینک‌های پرسرعت حیاتی است.
2. **کاهش پیچیدگی سخت‌افزاری:** کدگذاری 8b/10b نیازمند جداول جستجو و منطق پیچیده‌ای برای نگه‌داشتن تعادل DC و جلوگیری از الگوهای نامطلوب است، درحالی‌که 64b/66b از یک روش ساده‌تر با افزودن دو بیت پیشوند (sync bits) استفاده می‌کند.
3. **کاهش نرخ خطا و حفظ تعادل DC:** کدگذاری 64b/66b تعادل تعداد صفر و یک‌ها را حفظ می‌کند، که باعث کاهش احتمال ایجاد پریودهای طولانی از صفرها یا یک‌ها می‌شود. این موضوع باعث بهبود عملکرد Clock Recovery و کاهش Jitter در ارتباطات سریال می‌شود. البته دقت کنید که این کدگذاری به طور کامل تعادل DC را تضمین نمی‌کند.
4. **افزایش مقاومت در برابر خطا:** دو بیت افزوده‌شده در ابتدای هر فریم ۶۶ بیتی نقش مهمی در تشخیص خطاها و همگام‌سازی فریم‌ها دارند. این دو بیت معمولاً مقدار 10 (برای دستورات کنترلی) یا 01 (برای ارسال داده) دارند که به گیرنده اجازه می‌دهد تا سرآغاز هر فریم را تشخیص دهد.

نحوه کدگذاری

برای کدگذاری ما یک تابع C نوشتیم که در فایل encoder64b66b.c موجود است. این تابع یک ورودی 64 بیتی را دریافت کرده و بر اساس الگوریتم کدگذاری 64b/66b، عملیات‌های مورد نیاز را بر روی آن اعمال می‌کند.

در این نوع کدگذاری، در ابتدای بیت‌های ارسالی مقدار 01 را قرار می‌دهیم چون فقط می‌خواهیم عکس ارسال کنیم که همان داده‌های ما است.

در ادامه الگوریتمی که برای کدگذاری 64b/66b پیدا کردیم را در تابع C پیاده‌سازی کردیم که نحوه پیاده‌سازی آن را می‌توانید در کد مشاهده کنید و پیچیدگی خاصی ندارد و واضح است. به طور کلی پیاده‌سازی آن بر اساس مقادیر بیت‌های قبلی است تا بتواند از آمدن صفر یا یک‌های زیاد پشت سر یکدیگر جلوگیری کند.

نکته قابل توجه در این پیاده‌سازی این است که چون ما پایگاه داده‌ای برای ذخیره‌سازی 66 بیت نداشتیم، آن را به آرایه‌ای از اعداد مبنای 16 شکاندیم تا بتوانیم آن‌ها را ذخیره و ارسال کنیم.

در انتها این کد را داخل تابع ارسال عکس در main.ino صدا زدیم تا قبل از ارسال عکس، کدگذاری را بر روی آن اعمال کرده و سپس عکس را ارسال کند.

به طور دقیق‌تر، با تصاویر کد در ادامه آن را توضیح می‌دهیم:

1. در ابتدا به یک متغیر با نام **scrambler_state** نیاز داریم تا وضعیت‌های قبلی را در آن ذخیره کنیم تا بر اساس آن کدگذاری را انجام دهیم تا تلاش کنیم که تعادل DC را حفظ کنیم و از توالی زیاد صفر و یک‌ها پشت یکدیگر جلوگیری کنیم.

```
static uint64_t scrambler_state = (1ULL << 58) - 1;
```

تصویر 12 - متغیری برای نگهداری حالات قبلی

2. در ادامه تابع اصلی کدگذاری خود را با نام encode_64b66b را ساختیم تا این عملیات را انجام دهد. یک متغیر scrambled_data تعریف شده است تا مقدار 64 بیت کدگذاری شده ما را در خود ذخیره کند (این مقدار بدون header است به همین دلیل 66 بیت نیست). سپس الگوریتم آن را پیاده‌سازی کردیم که به ازای هر بیت داخل حلقه اجرا می‌شود. به طور خلاصه حلقه هر بیت از داده را استخراج کرده و بر اساس مقادیر

قبلی، مقدار جدیدی را به آن اختصاص می‌دهد و آن را داخل scrambled_data ذخیره کند. دقت کنید که هرگاه بیت جدیدی را محاسبه می‌کنیم، مقدار scrambled_state بر اساس آن به روز می‌شود تا در بیت‌های بعدی اثرگذار باشد و بتوانیم تا حدی تعادل DC را حفظ کنیم.

```
void encode_64b66b(uint64_t data, uint8_t encoded[9]) {
    uint64_t scrambled_data = 0;

    for (int i = 63; i >= 0; i--) {
        uint8_t input_bit = (data >> i) & 1;

        uint8_t feedback = ((scrambler_state >> 38) & 1) ^ ((scrambler_state >> 57) & 1);

        uint8_t scrambled_bit = input_bit ^ feedback;

        scrambler_state = (scrambler_state << 1) | scrambled_bit;
        scrambler_state &= (1ULL << 58) - 1;

        scrambled_data = (scrambled_data << 1) | scrambled_bit;
    }
```

تصویر 13 - تابع اصلی کدگذاری 64b/66b

3. در انتها نیز چون پایگاه داده‌ای برای اعداد 66 بیتی نداشتیم، آن را داخل یک آرایه ذخیره کردیم که هر خانه آرایه 1 بایت را برای ما ذخیره می‌کرد. دقت کنید که در خانه نهم (ایندکس هشت) صرفا دو بیت MSB آن را استفاده می‌کنیم و کاری با 6 بیت LSB آن نداریم چون در مجموع آرایه ما توانایی ذخیره‌سازی 72 بیت داده را دارد اما ما با 66 بیت آن کار داریم.

در ابتدا مقدار header را به خروجی اضافه می‌کنیم که مقدار 01 را در MSB آن قرار می‌دهیم. در ادامه نیز هر بیت داده را در جایگاه درست خود در خروجی قرار می‌دهیم و کار تابع ما پایان می‌یابد و داده کدگذاری شده آماده است.

```c
memset(encoded, 0, 9);

encoded[0] = 0x40 | ((scrambled_data >> 58) & 0x3F);

encoded[1] = (scrambled_data >> 50) & 0xFF;
encoded[2] = (scrambled_data >> 42) & 0xFF;
encoded[3] = (scrambled_data >> 34) & 0xFF;
encoded[4] = (scrambled_data >> 26) & 0xFF;
encoded[5] = (scrambled_data >> 18) & 0xFF;
encoded[6] = (scrambled_data >> 10) & 0xFF;
encoded[7] = (scrambled_data >> 2)  & 0xFF;
encoded[8] = (scrambled_data & 0x03) << 6;
```

تصویر 14 - چسباندن header و تشکیل خروجی نهایی

نحوه کدگشایی

برای این کار یک کد پایتون نوشتیم که کاملا برعکس کد C کار می‌کند تا بتواند داده‌های دریافتی را کدگشایی کرده و داده اصلی را از آن استخراج کند. این کد روند تشخیص خطای بسیار ساده‌ای دارد که header را بررسی می‌کند که اگر مقادیر معتبر 01 نبود، می‌تواند متوجه شود که خطا رخ داده است.

روند این کد نیاز به توضیح ندارد زیرا برعکس کد سی می‌باشد. در ادامه این کد را داخل کد پایتون سرور صدا می‌زنیم تا داده‌های دریافتی را کد گشایی کرده و عکس مورد نظر ما را تشکیل دهد.

در ادامه به توضیح کد این بخش می‌پردازیم:

1. برای عملیات کدگشایی، تابع decode_64b66b را داریم که ابتدا header را استخراج کرده و بررسی می‌کند که مقدار معتبری دارد یا خیر.

```python
def decode_64b66b(encoded):

    header = (encoded[0] >> 6) & 0b11
    if header != 0b01:
        raise ValueError("Invalid sync header")
```

تصویر 15 - تابع کدگشا و تشخیص خطا

2. در ادامه، تابع داده‌های اصلی را از آرایه ورودی استخراج کرده و داخل متغیر scrambled_data ذخیره می‌کند تا عملیات کدگشایی را آغاز کند. در انتها نیز متغیر descrambler_state را محاسبه می‌کند تا بر اساس آن عملیات را انجام داده و داده‌های اصلی را داخل متغیر original_data ذخیره کند.

```
scrambled_data = 0
scrambled_data |= (encoded[0] & 0x3F) << 58
scrambled_data |= encoded[1] << 50
scrambled_data |= encoded[2] << 42
scrambled_data |= encoded[3] << 34
scrambled_data |= encoded[4] << 26
scrambled_data |= encoded[5] << 18
scrambled_data |= encoded[6] << 10
scrambled_data |= encoded[7] << 2
scrambled_data |= (encoded[8] >> 6) & 0b11

descrambler_state = (1 << 58) - 1
original_data = 0
```

تصویر 16 - استخراج داده‌ها

3. در انتها نیز دقیقا برعکس عملیات کدگذاری را انجام می‌دهیم تا داده‌های اصلی را از داده کدگذاری شده به دست آوریم و داخل original_data ذخیره کنیم.

```
for i in range(63, -1, -1):
    scrambled_bit = (scrambled_data >> i) & 1

    feedback = ((descrambler_state >> 38) & 1) ^ ((descrambler_state >> 57) & 1)

    original_bit = scrambled_bit ^ feedback

    descrambler_state = ((descrambler_state << 1) | scrambled_bit) & ((1 << 58) - 1)

    original_data = (original_data << 1) | original_bit
```

تصویر 17 - کدگشایی نهایی

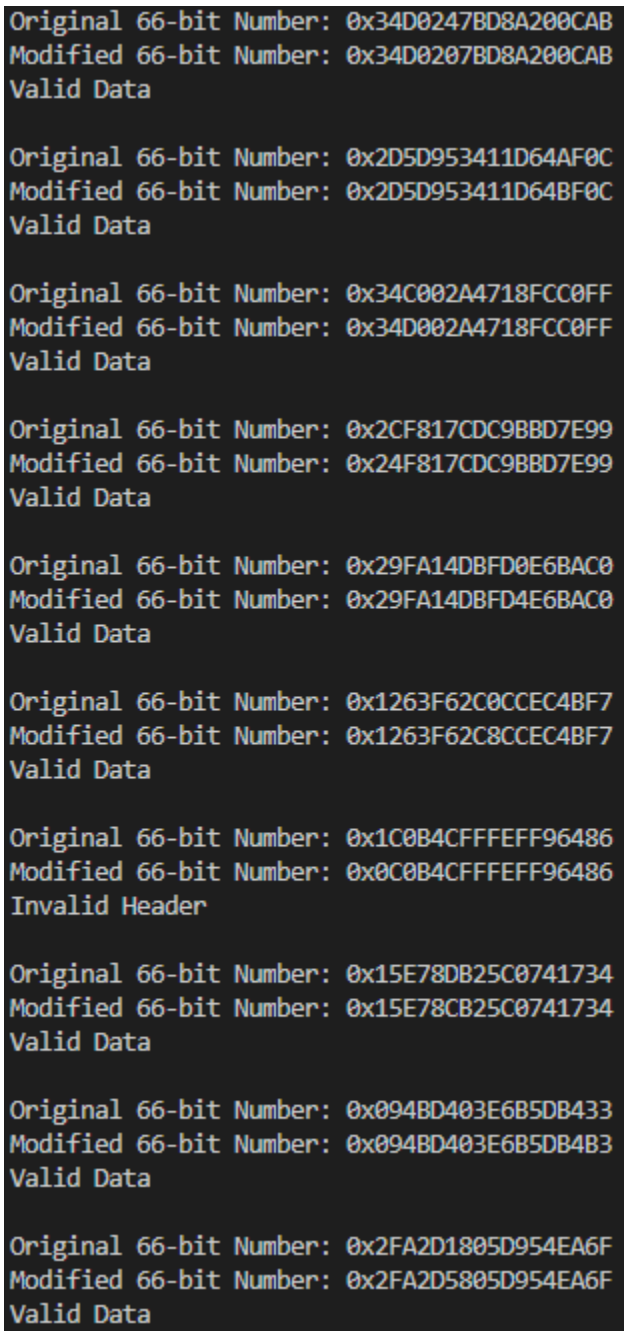
نکته قابل توجه در کدهای کدگذاری و کدگشایی که زدیم این است که چون کدهای کدگذار باید در Board اجرا می‌شدند، به زبان C نوشته شدند اما کدهای کدگشا که باید در سرور اجرا می‌شدند، به زبان پایتون نوشته شده‌اند.

نحوه تشخیص خطا

این روش کد گذاری الگوریتم تشخیص خطای خاصی ندارد چون مقادیر بیت‌های خروجی آن می‌تواند هر مقداری داشته باشد و صرفا تلاش می‌کند که تعداد 0 یا 1 متوالی را کاهش دهد اما آن را تضمین نمی‌کند. پس از این راه نمی‌توان خطای آن را تشخیص داد.
تنها راه تشخیص خطا در این روش، استفاده از بیت‌های header آن است که باید بررسی کنیم که این بیت‌ها مقادیر معتبر 01 و 10 را داشته باشند اما در کد ما چون فقط با داده کار داریم و مقدار آن را برابر با 01 قرار داده‌ایم، می‌توانیم بررسی کنیم که اگر مقدار header چیزی غیر از 01 بود، خطا رخ داده است.

بررسی تشخیص خطا

طبق مطالبی که در بالا بیان شد، می‌توان گفت که این انکودینگ در تئوری سیستم تشخیص خطای خاصی ندارد و سیستم تشخیص خطای آن در سیستم‌های پیشرفته‌تر و لایه‌های بالاتر صورت می‌گرفته که از چارچوب این پروژه خارج است. در نتیجه با توجه به اینکه نحوه تشخیص خطا در این پیاده‌سازی اولیه تنها توسط 2 بیت از میان 66 بیت صورت می‌گیرد، پس می‌توان گفت که بررسی تشخیص خطا در آن بی‌فایده است چون می‌توان گفت که هیچگاه این انکودینگ نمی‌تواند خطای داخل داده‌ها را تشخیص دهد.
به طور مثال در تصویر زیر این کار را 10 بار انجام دادیم و هر دفعه به طور رندوم یک بیت از 66 بیت ارسالی را تغییر دادیم و اگر این تغییر باعث می‌شد که مقدار Header غیرمعتبر شود، خطای Invalid Header را چاپ می‌کرد. همان‌طور که مشخص است در 10 دفعه اجرای الگوریتم، صرفا یک بار توانستیم که این خطا را تشخیص دهیم که این مقدار مورد قبول نیست و سربار پیاده‌سازی آن نسبت به بازدهی‌ای که می‌دهد نمی‌صرفد.



تصویر 18 - بررسی تشخیص خطا در انکودینگ 64b/66b

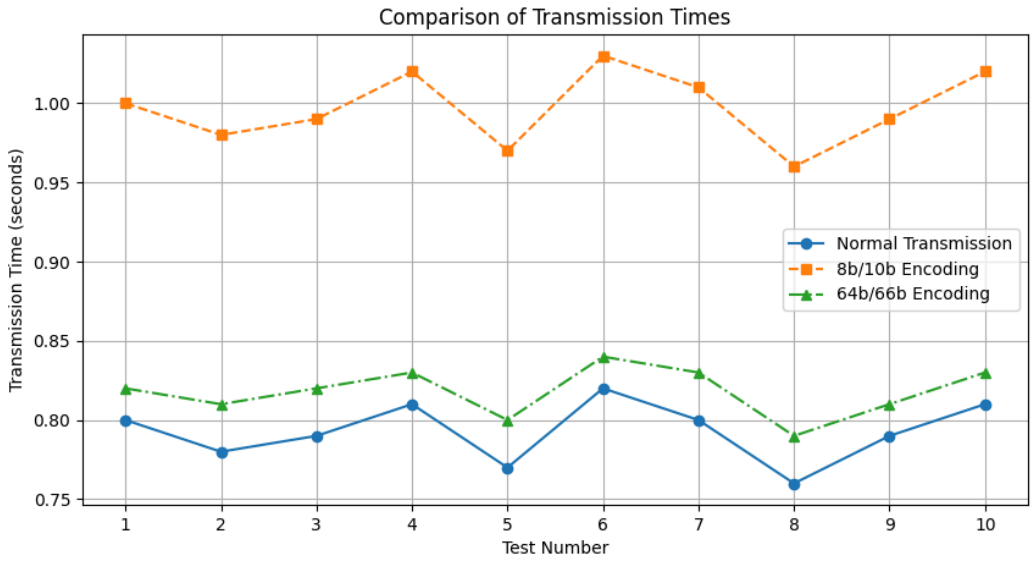
مقایسه روش‌های 8b/10b و 64b/66b

1. در روش 64b/66b نرخ بهره‌وری بیشتری داریم. همان‌طور که در مزایای 64b/66b بیان کردیم، نرخ بهره‌وری در روش 64b/66b برابر با 96.97% است با اینکه این مقدار در 8b/10b تنها برابر با 80% است و پهنای باند بیشتری را هدر می‌دهد.
2. 8b/10b بیشتر در سرعت‌های پایین استفاده می‌شود (مانند SATA، USB 3.0 و PCIe Gen 1/2) و هدف از آن حفظ تعادل DC و قابلیت اطمینان بالاتر است. اما 64b/66b بیشتر در سرعت‌های بالا (مانند 100GbE، 10GbE و PCIe Gen 3+) استفاده می‌شود و هدف اصلی آن بهره‌وری بیشتر از خطوط است.
3. 8b/10b به دلیل حفظ تعادل، می‌تواند خطاهای تک‌بیتی را تشخیص دهد اما 64b/66b به دلیل اینکه قابلیت خاصی ندارد که بر اساس آن خطا را تشخیص دهد، صرفا می‌تواند بر اساس مقادیر header خطا را تشخیص داده و گزارش دهد.
4. کد گذاری 8b/10b پیچیدگی پیاده‌سازی کمتری دارد، چون مقادیر محتمل آن تعدادشان کم است، می‌توانیم از یک جدول برای مپ کردن آن استفاده کنیم. در 64b/66b اما به دلیل تعداد زیاد حالات، نمی‌توانیم جدولی طراحی کنیم و باید الگوریتم آن را به صورت سخت‌افزاری با استفاده از گیت‌ها پیاده‌سازی کنیم.

در ادامه به مقایسه‌ی سرعت ارسال در این سه حالت می‌پردازیم. جدول زیر با توجه به زمان ارسال برای تصویر 100 کیلوبایتی در 10 تست متفاوت آماده شده است. به طور تقریبی همان‌طور که انتظار می‌رفت، انکودینگ 8b/10b حدود 25% افزایش زمان ارسال دارد، زیرا این روش باعث افزایش حجم داده تا 25% می‌شود. انکودینگ 64b/66b فقط 2-5% افزایش در زمان ارسال دارد، زیرا این روش کارآمدتر بوده و سربار کمتری دارد. در شرایطی که تأخیر مهم است، انکودینگ 64b/66b انتخاب بهتری نسبت به 8b/10b است.

شماره تست	زمان ارسال (64b/66b)	زمان ارسال (8b/10b)	زمان ارسال (حالت عادی)
1	0.82 ثانیه	1.00 ثانیه	0.80 ثانیه
2	0.81 ثانیه	0.98 ثانیه	0.78 ثانیه
3	0.82 ثانیه	0.99 ثانیه	0.79 ثانیه
4	0.83 ثانیه	1.02 ثانیه	0.81 ثانیه
5	0.80 ثانیه	0.97 ثانیه	0.77 ثانیه
6	0.84 ثانیه	1.03 ثانیه	0.82 ثانیه
7	0.82 ثانیه	1.01 ثانیه	0.80 ثانیه
8	0.79 ثانیه	0.96 ثانیه	0.76 ثانیه
9	0.81 ثانیه	0.99 ثانیه	0.79 ثانیه
10	0.83 ثانیه	1.02 ثانیه	0.81 ثانیه

جدول 1 - مقایسه زمان‌های عادی و بلاکینگ‌ها



تصویر 19 - مقایسه زمانی روش عادی و بلاکینگ‌ها بر روی نمودار