«بسمه تعالی»

دانشکده مهندسی کامپیوتر

گزارش پروژه هشتم درس مدارهای واسط

عنوان:

طراحی مبدل انکودینگ 64B/66B به 8B/10B

Design and Implementation of a 64B/66B to 8B/10B Encoding Converter

گردآورندگان:

نیکا قادری

آریان افضلی زاده

عاطفه قندهاری

استاد درس:

دکتر امین فصحتی

پاییز ۱۴۰۳

فهرست

مقدمه

در عصر حاضر، سیستم‌های مخابراتی و انتقال داده به منظور افزایش بازده و کاهش خطا از روش‌های انکودینگ متنوعی استفاده می‌کنند. دو نوع متداول از این انکودینگ‌ها 64B/66B و 8B/10B هستند که هر کدام ویژگی‌ها و کاربردهای خاص خود را دارند. در این پروژه، هدف طراحی و پیاده‌سازی یک مبدل برای تبدیل داده‌های کدگذاری شده با فرمت 64B/66B به 8B/10B با استفاده از برد آردوینو است. این مبدل ورودی داده‌های 64B/66B را دریافت کرده، آن را به فرمت 8B/10B تبدیل می‌کند و در خروجی ارسال می‌نماید. همچنین، بررسی‌های مختلفی برای تحلیل میزان تحمل این روش‌های کدگذاری در برابر اشکالات انجام می‌شود.

اهداف پروژه

1. طراحی و پیاده‌سازی مبدل انکودینگ: توسعه سیستمی که داده‌های دریافت‌شده با انکودینگ 64B/66B را پردازش کرده و آن را به فرمت 8B/10B تبدیل کند.
2. ارزیابی قابلیت تشخیص و تصحیح خطا: بررسی میزان توانایی این کدگذاری‌ها در تشخیص و تصحیح خطاهای احتمالی در داده‌های دریافتی.
3. تحلیل سناریوهای اشکال‌زایی: ایجاد و تزریق عمدی خطاها در داده‌ها برای تحلیل عملکرد سیستم در مواجهه با داده‌های نادرست.
4. ارائه روش‌هایی برای تصحیح خطا در سناریوهای مختلف.

معرفی انکودینگ‌ها

انکودینگ 64B/66B

این روش کدگذاری برای انتقال داده‌ها در شبکه‌های ارتباطی پرسرعت مانند Ethernet 10G استفاده می‌شود. در این روش، 64 بیت داده خام با افزودن 2 بیت اضافه به 66 بیت تبدیل می‌شود که شامل 2 بیت همگام‌سازی و 64 بیت داده است. این کدگذاری برای افزایش بهره‌وری انتقال و کاهش تعداد تغییرات بیت (Transition) طراحی شده است همچنین کاهش افزونگی و کارایی بالا از مزایای این روش است. این انکودینگ قابلیت تشخیص خطا دارد اما مکانیزم تصحیح خطا را ارائه نمی‌دهد.

انکودینگ 8B/10B

یک روش کدگذاری که به طور گسترده در پروتکل‌های سریال مانند PCIe ، SATA و Fibre Channel استفاده می‌شود. هر 8 بیت داده به 10 بیت کدگذاری می‌شود که افزونگی بیشتر نسبت به 64B/66B دارد. این روش به منظور حفظ تعادل DC و تشخیص خطا طراحی شده است.

سخت‌افزار

1) یک برد آردوینو ARDUINO UNO R3 DIP
2) برد برد الکترونیکی MB_ 102
3) کابل UNO
4) کابل فلت نری به نری 20 سانتی
5) ال ای دی سبز 5MM
6) مقاومت 5% RMF 1/4W

نرم‌افزار

. Arduino IDE
ابزارهای شبیه‌سازی و تحلیل داده.

شرح توابع به کار رفته در نرم‌افزار مبدل

در این بخش، توابع جانبی استفاده شده در خارج از loop و Setup را بررسی می کنیم و عملکرد آن ها را توضیح می دهیم.

Convert 8b/10b()

این تابع به عنوان ورودی خود یک عدد هشت بیتی (در قالب uint8_t) دریافت می کند و آن را به داده ای ده بیتی تبدیل می کند. انکودینگ 8/10 برای تبدیل داده ها از یک جدول استفاده می کند؛ اما قبل از پرداختن به جدول لازم است تا مفهومی به نام running disparity یا RD آشنا شویم. ویژگی انکودینگ 8/10 این است که داده های تولید ده در آن DC balance هستند. به همین دلیل، هنگام تبدیل این داده ها از بیت کنترلی RD استفاده می شود که نشان می دهد تا اینجا تعداد یک ها و صفر ها در خروجی تولید شده با هم برابر هستند یا یکی بیشتر از دیگری است. با توجه به داده ورودی، و RD فعلی، جدول یک داده ده بیتی و یک مقدارRD خروجی می دهد که همان RD بعدی و داده تبدیل شده هستند.

در نتیجه، برای جست و جو در جدول کافی است تا RD فعلی را هشت بیت به چپ شیفت دهیم و با داده فعلی Or کنیم. سپس سطر متناظر این مقدار را به دست آورده و RD و داده را از آن استخراج می کنیم. توضیحات داده شده در قالب کد زیر قابل نمایش است:

```
uint16_t convert_8B_10B(uint8_t byte) {
  uint16_t table_in = rd;
  table_in <<= 8;
  table_in |= byte;
  uint16_t encoded = lookup_8B_10B[table_in];
  rd = (encoded >> 10) & 0x01;
  return encoded & 0x3FF;
}
```

همچنین جدول استفاده شده در فایل lookup_tables.h قابل مشاهده است. به طور مثال، بخشی از محتویات آن در زیر آورده شده است. توجه کنید که بیت های داده در این جدول برعکس هستند و باید پس از استخراج، معکوس شوند.

```
const uint16_t lookup_8B_10B}=[512]
// Rjhgfiedcba       //RD(Pre) + Dx.y => RD(Post)
      0b00010111001,     // 00000000-D00.0[0]-
      0b00010101110,     // 00000001-D01.0[1]-
      0b00010101101,     // 00000010-D02.0[2]-
      0b11101100011,     // 00000011-D03.0[3]+
      0b00010101011,     // 00000100-D04.0[4]-
      0b11101100101,     // 00000101-D05.0[5]+
```

```
    0b11101100110,      // 00000110-D06.0[6]+
    0b11101000111,      // 00000111-D07.0[7]+
```

Reverse Lowest 10 Bits()

همانطور که از نام این تابع پیداست، وظیفه دارد تا محتوای خود را معکوس کند. اما از آنجایی که داده ما ده بیتی است، تنها کافی است این عملیات برای ده بیت کم ارزش انجام شود و به بقیه بیت ها کاری نداریم.

```
uint16_t reverseLowest10Bits(uint16_t value) {
  uint16_t reversed = 0;
  for (int i = 0; i < 10; i++) {
    reversed |= ((value >> i) & 1) << (9 - i);
  }
  return reversed;
}
```

Encode Hamming()

این تابع به عنوان ورودی خود یک عدد ده بیتی را می گیرد (در قالب uint16_t) و آن را با استفاده از فرمت همینگ 5/11 کد گذاری می کند. ابتدا بیت های داده شیفت می خورد و هر بیت در مکان مناسب خود قرار می گیرد. سپس، با استفاده از بیت های داده، بیت های parity ساخته می شوند و به داده خروجی اضافه می شوند.

```
uint16_t encodeHamming(uint16_t data) {
    uint16_t code = 0;

    // Set the data bits in the appropriate positions
    code |= (data & 0b0000000001) << 2; // Bit 3
    code |= (data & 0b0000000010) << 3; // Bit 5
    code |= (data & 0b0000000100) << 3; // Bit 6
    code |= (data & 0b0000001000) << 3; // Bit 7
    code |= (data & 0b0000010000) << 4; // Bit 9
    code |= (data & 0b0000100000) << 4; // Bit 10
    code |= (data & 0b0001000000) << 4; // Bit 11
    code |= (data & 0b0010000000) << 4; // Bit 12
    code |= (data & 0b0100000000) << 4; // Bit 13
    code |= (data & 0b1000000000) << 4; // Bit 14

    // Calculate parity bits
    // Position 1 (bit 1)
```

```
    uint8_t p1 = ((code >> 2) & 1) ^ ((code >> 4) & 1) ^ ((code
>> 6) & 1) ^ ((code >> 8) & 1) ^ ((code >> 10) & 1) ^ ((code >>
12) & 1) ^ ((code >> 14) & 1);
    code |= (p1 << 0);

    // Position 2 (bit 2)
    uint8_t p2 = ((code >> 1) & 1) ^ ((code >> 2) & 1) ^ ((code
>> 5) & 1) ^ ((code >> 6) & 1) ^ ((code >> 9) & 1) ^ ((code >>
10) & 1) ^ ((code >> 13) & 1) ^ ((code >> 14) & 1);
    code |= (p2 << 1);

    // Position 4 (bit 4)
    uint8_t p4 = ((code >> 3) & 1) ^ ((code >> 4) & 1) ^ ((code
>> 5) & 1) ^ ((code >> 6) & 1) ^ ((code >> 11) & 1) ^ ((code >>
12) & 1) ^ ((code >> 13) & 1) ^ ((code >> 14) & 1);
    code |= (p4 << 3);

    // Position 8 (bit 8)
    uint8_t p8 = ((code >> 7) & 1) ^ ((code >> 8) & 1) ^ ((code
>> 9) & 1) ^ ((code >> 10) & 1) ^ ((code >> 11) & 1) ^ ((code >>
12) & 1) ^ ((code >> 13) & 1) ^ ((code >> 14) & 1);
    code |= (p8 << 7);

    return code;}
```

Decode Hamming()

مکانیزم تصحیح خطای کد همینگ در این تابع پیاده سازی شده است. به این صورت که بررسی می کنیم آیا چهار بیت پریتی درست هستند یا نه. هر کدام که یک شد، یعنی خطایی رخ داده است. با کنار هم گذاشتن این بیت ها نیز می توانیم مکان این خطا را شناسایی کرده و آن را اصلاح کنیم.

```
uint16_t decodeHamming(uint16_t code) {

    uint16_t hamming = code;

    uint8_t p1 = ((code >> 0) & 1) ^ ((code >> 2) & 1) ^ ((code
>> 4) & 1) ^ ((code >> 6) & 1) ^ ((code >> 8) & 1) ^ ((code >>
10) & 1) ^ ((code >> 12) & 1) ^ ((code >> 14) & 1);
    uint8_t p2 = ((code >> 1) & 1) ^ ((code >> 2) & 1) ^ ((code
>> 5) & 1) ^ ((code >> 6) & 1) ^ ((code >> 9) & 1) ^ ((code >>
10) & 1) ^ ((code >> 13) & 1) ^ ((code >> 14) & 1);
```

```
    uint8_t p4 = ((code >> 3) & 1) ^ ((code >> 4) & 1) ^ ((code
>> 5) & 1) ^ ((code >> 6) & 1) ^ ((code >> 11) & 1) ^ ((code >>
12) & 1) ^ ((code >> 13) & 1) ^ ((code >> 14) & 1);
    uint8_t p8 = ((code >> 7) & 1) ^ ((code >> 8) & 1) ^ ((code
>> 9) & 1) ^ ((code >> 10) & 1) ^ ((code >> 11) & 1) ^ ((code >>
12) & 1) ^ ((code >> 13) & 1) ^ ((code >> 14) & 1);

    uint16_t syndrome = (p1 << 0) | (p2 << 1) | (p4 << 2) | (p8
<< 3);

    if (syndrome != 0) {
    int errorBit = syndrome - 1; // Calculate the position of
the error (0-indexed)
    hamming ^= (1 << errorBit);
    Serial.print("Hamming: Error corrected at bit ");
    Serial.println(errorBit);
    }

    return hamming;
}
```

Extract Data From Hamming()

ورودی این تابع یک عدد کد گذاری شده به فرمت همینگ است و خروجی آن، داده ده بیتی است که با حذف بیت های parity از ورودی استخراج می شود. با توجه به مکان هر بیت در فرمت کد گذاری شده، می توان داده خام را به صورت زیر بازیابی کرد:

```
uint16_t extractDataFromHamming(uint16_t hammingCode) {

    uint16_t data = 0;
    // Data bits are at positions 3, 5, 6, 7, 9, 10, 11, 12, 13,
14
    data |= ((hammingCode >> 2) & 1) << 0; // Bit 3 -> Bit 0
    data |= ((hammingCode >> 4) & 1) << 1; // Bit 5 -> Bit 1
    data |= ((hammingCode >> 5) & 1) << 2; // Bit 6 -> Bit 2
    data |= ((hammingCode >> 6) & 1) << 3; // Bit 7 -> Bit 3
    data |= ((hammingCode >> 8) & 1) << 4; // Bit 9 -> Bit 4
    data |= ((hammingCode >> 9) & 1) << 5; // Bit 10 -> Bit 5
    data |= ((hammingCode >> 10) & 1) << 6; // Bit 11 -> Bit 6
    data |= ((hammingCode >> 11) & 1) << 7; // Bit 12 -> Bit 7
    data |= ((hammingCode >> 12) & 1) << 8; // Bit 13 -> Bit 8
    data |= ((hammingCode >> 13) & 1) << 9; // Bit 14 -> Bit 9

    return data;
}
```

is Faulty()

این تابع به عنوان ورودی خود یک عدد 16 بیتی می گیرد، از آن عدد ده بیتی متناظر را استخراج می کند و سپس بررسی می کند که آیا در این عدد خطا وجود دارد یا خیر.

این تشخیص خطا با استفاده از انکودینگ 10/8 بیت و ویژگی های آن انجام می شود. همانطور که دیدیم، تبدیل هشت بیت داده به ده بیت خروجی انکود شده توسط یک جدول انجام می شود. این جدول، طوری طراحی شده است تا داده خروجی یک سری ویژگی خاص داشته باشد؛ برای مثال DC balance باشد. برای همین، با مطالعه مقادیر آن، می توان مکانیزمی جهت تشخیص خطا ساخت. آنالیز جدول انکودینگ توسط کد پایتون زیر انجام می شود:

```
import lookup_table

def max_consecutive_bits(number, bit='0'):
 binary_rep = bin(number)[:1]
 target = '0' if bit == '0' else '1'
 max_consecutive = max((len(segment) for segment in
binary_rep.split('1' if bit == '0' else '0')), default=0)
 return max_consecutive

def count_transitions(number):
 binary_rep = bin(number)[:1]
 zero_to_one = sum(1 for i in range(1, len(binary_rep)) if
binary_rep[i - 1] == '0' and binary_rep[i] == '1')
 one_to_zero = sum(1 for i in range(1, len(binary_rep)) if
binary_rep[i - 1] == '1' and binary_rep[i] == '0')
 return zero_to_one, one_to_zero

def max_difference_ones_zeros(number):
 binary_rep = bin(number)[:]
 count_ones = binary_rep.count('1')
 count_zeros = binary_rep.count('0')
 return abs(count_ones - count_zeros)

def run_tests:()
 table = lookup_table.lookup_8B_10B
 max_zeros = 0
 max_ones = 0
 max_zero_to_one = 0
 max_one_to_zero = 0
 max_diff_ones_zeros = 0
```

```
 for number in table:
  consecutive_zeros = max_consecutive_bits(number, bit='0')
  consecutive_ones = max_consecutive_bits(number, bit='1')
  zero_to_one, one_to_zero = count_transitions(number)
  difference_ones_zeros = max_difference_ones_zeros(number)

  max_zeros = max(max_zeros, consecutive_zeros)
  max_ones = max(max_ones, consecutive_ones)
  max_zero_to_one = max(max_zero_to_one, zero_to_one)
  max_one_to_zero = max(max_one_to_zero, one_to_zero)
  max_diff_ones_zeros = max(max_diff_ones_zeros,
difference_ones_zeros)

 print(f"Maximum number of consecutive zeros: {max_zeros}")
 print(f"Maximum number of consecutive ones: {max_ones}")
 print(f"Maximum number of 0-to-1 transitions:
{max_zero_to_one}")
 print(f"Maximum number of 1-to-0 transitions:
{max_one_to_zero}")
 print(f"Maximum difference between number of ones and zeros:
{max_diff_ones_zeros}")

if __name__ == "__main:"__
 run_tests()
```

با اجرای تست ها، به نتایج زیر می رسیم:

```
Maximum number of consecutive zeros: 4
Maximum number of consecutive ones: 5
Maximum number of 0-to-1 transitions: 5
Maximum number of 1-to-0 transitions: 5
Maximum difference between number of ones and zeros: 3
```

در نتیجه به طور مثال اگر تعداد یک های متوالی در عدد ورودی بیشتر از 5 عدد شد، متوجه می شویم که خطایی رخ داده است و به این صورت می توان از انکودینگ 10/8 برای تشخیص خطا استفاده کرد.

محتویات تابع `is Faulty` به صورت زیر نوشته شده تا تک تک این تست ها را اجرا کند و اگر جواب هر کدام مثبت شد، خطا را تشخیص داده و اعلام کند:

```
bool isFaulty(uint16_t value) {
    uint16_t data = extractDataFromHamming(value) & 3FF;
    int consecutiveZeros = 0, maxConsecutiveZeros = 0;
```

```
    int consecutiveOnes = 0, maxConsecutiveOnes = 0;
    int onesCount = 0, zerosCount = 0;
    int transitions0To1 = 0, transitions1To0 = 0;

    bool previousBit = (data >> 9) & 1;
    for (int i = 9; i >= 0; i--) {
        bool currentBit = (data >> i) & 1;

        if (currentBit == 0) {
            consecutiveZeros++;
            consecutiveOnes = 0;
        } else {
            consecutiveOnes++;
            consecutiveZeros = 0;
        }

        maxConsecutiveZeros = max(maxConsecutiveZeros,
consecutiveZeros);
        maxConsecutiveOnes = max(maxConsecutiveOnes,
consecutiveOnes);

        if (currentBit == 1) {
            onesCount++;
        } else {
            zerosCount++;
        }

        if (i < 9 && currentBit != previousBit) {
            if (previousBit == 0 && currentBit == 1) {
                transitions0To1++;
            } else if (previousBit == 1 && currentBit == 0) {
                transitions1To0++;
            }
        }

        previousBit = currentBit;
    }

    if (maxConsecutiveZeros > 4) return true;
    if (maxConsecutiveOnes > 5) return true;
    if (transitions0To1 > 5) return true;
    if (transitions1To0 > 5) return true;
    if (abs(onesCount - zerosCount) > 3) return true;

    // Added for error detection.
    bool foundInTable = false;
    for (int i = 0; i < 512; i++) {
```

```
        if ((lookup_8B_10B[i] & 0x3FF) == data) { // Compare
only the lower 10 bits
            foundInTable = true;
            break;
        }
    }
    if (!foundInTable) {
        return true;
    }
    return false;
}
```

Inject Error()

این تابع، یک داده 16 بیتی را به عنوان ورودی می گیرد، یک بیت آن را به صورت شانسی انتخاب کرده و آن را معکوس می کند.

بدین صورت خطا تزریق می شود.

نکته دیگر آن است که در این پیاده سازی، دو نوع خطا تعریف شده است. خطای دائمی یا permanent و خطای گذرا یا transient. در مورد تفاوت آن ها بعدتر صحبت خواهیم کرد. اینجا تابع با احتمال برابر، یک نوع را برای خطای تزریق شده انتخاب می کند و با ست کردن مقدار isTransient، این مقدار را ثبت می کند.

در نهایت این تابع به صورت زیر پیاده سازی می شود:

```
void injectError(uint16_t &data, bool &isTransient) {
  int bitPosition = random(0, 15); // Random bit position
  data ^= (1 << bitPosition);     // Flip the bit
  isTransient = random(0, 2);     // Randomly decide if it's
transient (true) or permanent (false)
  // Modified this part to play with the transient rate.
  // isTransient = false;
}
```

Print 10bit Binary()

این تابع ده بیت کم ارزش ورودی خود را به صورت باینری چاپ می کند. مزیت آن در این است که صفر های قبل از عدد را که از نظر ریاضی بی ارزش هستند نیز چاپ می کند. این تابع و توابع مشابه آن (print8BitBinary و ...) جهت خوانایی بیشتر تعریف شده اند.

```
void print10BitBinary(uint16_t value) {
  for (int i = 9; i >= 0; i--) {
```

```
    Serial.print((value >> i) & 0x1); // Print each bit from
highest to lowest
  }
  Serial.println();
}
```

توضیح عملکرد توابع

تعریف متغیرها و ساختار داده

```
uint8_t rd = 0;
extern const uint16_t lookup_8B_10B[512];
```

- `rd` : این متغیر برای نگهداری اطلاعات مربوط به وضعیت تعادل (Disparity Control) DC در کدگذاری 8B/10B استفاده می‌شود. مقدار آن 0 (تعادل منفی) یا 1 (تعادل مثبت) است.
- `lookup_8B_10B[512]` : یک آرایه از پیش‌محاسبه‌شده که شامل کدگذاری‌های استاندارد 8B/10B است. ورودی 8 بیتی گرفته می‌شود و مقدار 10 بیتی متناسب با آن از این جدول استخراج می‌شود.

```
struct Data66b {
  uint64_t lowBits = 0;  // 64 بیت داده اصلی
  uint8_t highBits = 0;  // 2 بیت اضافه شده در انکودینگ 64B/66B
};
```

- این ساختار داده برای ذخیره فریم‌های 64B/66B استفاده می‌شود. مقدار `lowBits` شامل 64 بیت داده است و `highBits` دو بیت کنترلی را نگه می‌دارد.

تابع `setup()`

هدف تابع

این تابع در مرحله‌ی راه‌اندازی (setup) آردوینو اجرا می‌شود و کارهای زیر را انجام می‌دهد:

1. تنظیم یک محدوده از پین‌ها (`startPin` تا `endPin`) به عنوان خروجی(`OUTPUT`) .
2. فعال‌سازی ارتباط سریال با نرخ baud `9600` برای ارسال داده به کامپیوتر یا مانیتور سریال.

کاربرد اصلی:

تنظیم پین‌های دیجیتال برای کنترلLED ، موتور، رله، و سایر دستگاه‌های خروجی.

ارسال اطلاعات از آردوینو به مانیتور سریال (Serial Monitor) جهت دیباگ یا مشاهده داده‌ها.

1 .تنظیم پین‌های آردوینو به عنوان خروجی

```
for (int pin = startPin; pin <= endPin; pin++) {
    pinMode(pin, OUTPUT);
}
```

پین‌های دیجیتال از `startPin` تا `endpin` به عنوان خروجی تنظیم می‌شوند.

نحوه‌ی عملکرد حلقه `for` :

1. مقدار `pin` از `startPin`شروع می‌شود.
2. تا زمانی که `pin <= endpin` باشد، حلقه تکرار می‌شود.
3. در هر تکرار، `pinMode(pin, OUTPUT)` اجرا شده و پین تنظیم می‌شود.
4. مقدار `pin` افزایش می‌یابد(`pin++`) .

مثال عددی:

اگر `startPin = 5` و `endPin = 9` باشد، حلقه به این صورت اجرا می‌شود:

```
pinMode(5, OUTPUT);
pinMode(6, OUTPUT);
pinMode(7, OUTPUT);
pinMode(8, OUTPUT);
pinMode(9, OUTPUT);
```

نتیجه: پین‌های `5` تا `9` به‌عنوان خروجی(OUTPUT) تنظیم می‌شوند.

اگر این پین‌ها به LED متصل باشند، بعداً می‌توانیم آن‌ها را روشن و خاموش کنیم.

اگر این پین‌ها به موتور متصل باشند، می‌توانیم آن‌ها را فعال کنیم.

۲ .راه‌اندازی ارتباط سریال

```
Serial.begin(9600);
```

این خط ارتباط سریال بین آردوینو و کامپیوتر را فعال می‌کند.

مقدار `9600` نرخ انتقال داده (baud rate) را مشخص می‌کند.

کاربرد:

- امکان ارسال داده‌ها به مانیتور سریال در Arduino IDE .
- دیباگ کردن برنامه‌ها با چاپ اطلاعات در سریال مانیتور.
- ارتباط با کامپیوتر یا ماژول‌های سریال مثل بلوتوث HC-05 ، GPS، وRFID .

جمع‌بندی عملکرد `setup`

1. پین‌های دیجیتال `startPin` تا `endpin` را به‌عنوان خروجی تنظیم می‌کند.
2. ارتباط سریال با نرخ baud `9600` فعال می‌شود.

کاربردهای عملی:

- کنترلLED ، موتور، رله، و نمایشگرهای دیجیتال.
- ارسال داده به سریال مانیتور برای دیباگ.
- ارتباط با کامپیوتر یا ماژول‌های سریال.

تابع `convert_8B_10B()`

این تابع یک بایت 8 بیتی (`uint8_t byte`) را دریافت کرده و آن را به یک کد 10 بیتی مطابق با استاندارد 8B/10B تبدیل می‌کند. این استاندارد در اترنت گیگابیتی، PCIe، SATA، و سایر پروتکل‌های سریال برای افزایش تعادل صفر و یک (DC-Balance) و تصحیح خطا استفاده می‌شود.

به طور کلی می‌توان گفت:

1. table_in ساخته می‌شود که شامل مقدار فعلی rd (بیت تعادلDC) و byte (بایت 8 بیتی ورودی) است.
2. این مقدار به عنوان اندیس برای دسترسی به lookup_8B_10B استفاده می‌شود.
3. مقدار 10 بیتی رمزگذاری‌شده از lookup_8B_10B خوانده شده و در encoded ذخیره می‌شود.
4. مقدار rd (بیت تعادل جدید) از بیت 10 encoded استخراج می‌شود.
5. مقدار 10 بیت پایین‌تر (بدون rd) بازگردانده می‌شود.

نکات مهم:

- این روش از یک جدول lookup_8B_10B برای تبدیل سریع استفاده می‌کند که باعث کاهش پردازش می‌شود.
- مقدار rdکنترل تعادل DC را حفظ می‌کند که برای ارتباطات سریال حیاتی است.

مراحل اجرا:

۱ .تعریف و مقداردهی اولیه متغیرها

```
uint16_t table_in = rd;
```

متغیر table_in یک مقدار 16 بیتی است که مقدار rd را در خود ذخیره می‌کند.
rd (Running Disparity) نشان‌دهنده‌ی تعادل بین تعداد صفرها و یک‌ها در رشته کدگذاری‌شده‌ی 8B/10B است و در هر مرحله به‌روزرسانی می‌شود.

۲.شیفت table_in به چپ برای ایجاد فضای 8 بیتی

```
table_in <<= 8;
```

مقدار rd را 8 بیت به چپ شیفت می‌دهد تا جا برای مقدار جدید 8 بیتی ورودی (byte) باز شود. حالا table_in به شکل زیر است:

```
[rd (1 بیت)| 0000000 (7 بیت)| 00000000 (8 بیت داده ورودی)]
```

۳ .ترکیب مقدار جدید 8 بیتی با table_in

```
table_in |= byte;
```

مقدار byte را در 8 بیت پایانی متغیر table_in قرار می‌دهد. حالا table_in شامل rd و byte است:

```
[rd (1 بیت)| 0000000 (7 بیت)| داده 8 بیتی]
```

۴ .دریافت مقدار کدگذاری‌شده 10 بیتی از جدول lookup_8B_10B

```
uint16_t encoded = lookup_8B_10B[table_in];
```

مقدار table_in را به‌عنوان ایندکس (کلید) برای دسترسی به یک جدول از پیش محاسبه‌شده (lookup_8B_10B) استفاده می‌کند.
این جدول شامل تمام ترکیب‌های ممکن 8 بیت ورودی و 1 بیت rd است و مقدار معادل 10 بیت کدگذاری‌شده را برمی‌گرداند.

۵ .بروزرسانی rd (Running Disparity)

```
rd = (encoded >> 10) & 0x01;
```

مقدار جدید rd را از بیت یازدهم (بیت شماره 10) مقدار کدگذاری‌شده استخراج می‌کند.
rd تنظیم می‌شود تا برای تبدیل بعدی حفظ شود.

۶.بازگرداندن مقدار 10 بیتی نهایی

```
return encoded & 0x3FF;
```

مقدار 10 بیتی نهایی را استخراج می‌کند و بازمی‌گرداند.
0x3FFبرابر است با 0000001111111111که فقط 10بیت پایین‌تر را نگه می‌دارد.

جمع‌بندی عملکرد تابع convert_8B_10B

1. مقدار rd را به‌عنوان بیت یازدهم در نظر می‌گیرد و آن را به 8 بیت داده‌ی ورودی اضافه می‌کند.
2. مقدار ترکیبی را به‌عنوان ایندکس برای دریافت مقدار کدگذاری‌شده 10 بیتی از lookup_8B_10B استفاده می‌کند.
3. مقدار جدید rd را به‌روزرسانی می‌کند تا برای کدگذاری بعدی استفاده شود.
4. مقدار 10 بیت نهایی را برمی‌گرداند.

این روش باعث می‌شود که کدگذاری سریع و بهینه باشد، زیرا از جدول lookup_8B_10B برای تبدیل استفاده می‌کند و نیازی به انجام محاسبات اضافی نیست.

تابع reverseLowest10Bits()

هدف تابع

این تابع مقدار 10 بیت پایینی یک عدد 16بیتی (uint16_t value)را دریافت کرده و ترتیب بیت‌های آن را معکوس می‌کند.

کاربرد اصلی:
در کدینگ 8B/10B و سایر روش‌های رمزگذاری داده‌ها، برخی از عملیات نیاز دارند که بیت‌های داده را معکوس کنند، مثلاً هنگام ارسال یا دریافت داده در پروتکل‌های سریالی.
این تابع در مواردی مثل تبدیلEndianness ، پردازش داده‌های باینری خاص، و تشخیص خطا مفید است.

۱ .تعریف متغیرها

```
uint16_t reversed = 0;
```

متغیر `reversed` مقدار معکوس‌شده‌ی 10 بیت پایینی را ذخیره خواهد کرد. مقدار اولیه‌ی آن صفر است.

۲ .حلقه‌ی `for` برای جابجایی بیت‌ها

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

یک حلقه‌ی 10 مرحله‌ای که بیت‌های 10 بیت پایین `value`را از کم‌ارزش‌ترین بیت (LSB) تا پرارزش‌ترین بیت (MSB) استخراج کرده و جای آن‌ها را معکوس می‌کند.

۳ .معکوس کردن موقعیت بیت‌ها

```
reversed |= ((value >> i) & 1) << (9 - i);
```

این خط کد مقدار بیت `i` را گرفته و در موقعیت `(9 - i)` قرار می‌دهد:

نحوه‌ی عملکرد:

1. `(value >> i) & 1`:
   بیت شماره `i` از `value` را استخراج می‌کند.
2. `<< (9 - i)`:
   این بیت را به موقعیت معکوس‌شده‌ی آن در `reversed` منتقل می‌کند.
3. `reversed |= ...`:
   مقدار استخراج‌شده را در `reversed` قرار می‌دهد.

مثال عددی:

فرض کنید مقدار `value = 0b1100101100` (10 بیت) باشد. در هر مرحله داریم:

i	بیت i از value	جایگاه جدید (9 - i)	reversed (بعد از اعمال تغییر)
0	0	9	0000000000
1	0	8	0000000000
2	1	7	0000001000
3	1	6	0000011000
4	0	5	0000011000
5	1	4	0000011010
6	0	3	0000011010
7	0	2	0000011010
8	1	1	0000011011
9	1	0	0000011011

مقدار خروجی معکوس شده: `0b0011010110`

۴ .بازگرداندن مقدار معکوس‌شده

```
return reversed;
```

مقدار 10 بیت معکوس‌شده را برمی‌گرداند.

جمع‌بندی عملکرد تابع `reverseLowest10Bits`

1. یک عدد 16 بیتی دریافت می‌کند.
2. 10 بیت پایین آن را جدا کرده و معکوس می‌کند.
3. مقدار 10 بیتی معکوس‌شده را بازمی‌گرداند.

کاربردهای عملی تابع:

- کدگذاری و رمزگذاری داده‌های سریالی (مانند 8B/10B)
- بررسی و تصحیح داده‌های باینری در مخابرات و پردازش سیگنال.
- تبدیل Endianness و پردازش داده‌های Little-Endian و Big-Endian .

تابع injectError()

هدف تابع

این تابع به‌طور تصادفی یک خطا در داده‌های 10 بیتی ایجاد می‌کند .همچنین تعیین می‌کند که خطا گذرا (transient) یا دائمی (permanent) باشد.

کاربرد اصلی:
شبیه‌سازی خطاها در سیستم‌های مخابراتی، تصحیح خطا(ECC) ، و پردازش سیگنال.
آزمایش الگوریتم‌های تشخیص و تصحیح خطا مانند Hamming Code یا 8B/10B Encoding .

۱ .انتخاب یک بیت تصادفی برای ایجاد خطا

```
int bitPosition = random(0, 10);
```

تابع random(0, 10) یک عدد تصادفی بین 0 تا 9 انتخاب می‌کند.
این عدد مشخص می‌کند که کدام بیت از 10 بیت پایین data قرار است دچار خطا شود.

۲ .ایجاد خطا (Flipکردن بیت)

```
data ^= (1 << bitPosition);
```

این خط بیت انتخاب‌شده را معکوس می‌کند:

نحوه‌ی عملکرد:

1. (1 << bitPosition) :
   مقدار 1را به موقعیت bitPosition شیفت می‌دهد.
2. data ^= (1 << bitPosition) :
   این مقدار را با عملگر XOR روی data اعمال می‌کند.
   اگر بیت 0 باشد1 می‌شود، و اگر 1 باشد 0 می‌شود.

مثال عددی:
فرض کنید مقدار data = 0b1100101100 و bitPosition = 3 باشد:

1. (1 << 3) = 0b0000001000
2. 1100101100 XOR 0000001000 = 1100100100

بیت شماره 3 از 1 به 0 تغییر کرد !

۳ .تعیین نوع خطا (گذرا یا دائمی)

```
isTransient = random(0, 2);
```

مقدار isTransient را به‌طور تصادفی true (گذرا) یا false (دائمی) تنظیم می‌کند.

خطای گذرا(transient error) :
خطا موقتی است و احتمال دارد در بازخوانی بعدی رفع شود.
معمولاً ناشی از تداخل الکترومغناطیسی یا نویز است.
در DRAM(حافظه‌های پویا) و سیستم‌های مخابراتی رایج است.

خطای دائمی(permanent error) :
خطا همیشه باقی می‌ماند مگر اینکه اصلاح شود.
معمولاً ناشی از خرابی سخت‌افزاری یا سلول‌های معیوب حافظه است.

جمع‌بندی عملکرد injectError

1. یک بیت تصادفی را از 10بیت پایین data انتخاب می‌کند.
2. بیت انتخاب‌شده را معکوس (flip) می‌کند تا خطا ایجاد شود.
3. تصادفی تعیین می‌کند که خطا گذرا (true) یا دائمی (false) باشد.

کاربردهای عملی تابع:

- شبیه‌سازی خطاها در حافظه‌های ECC (مانندHamming Code)
- تست الگوریتم‌های تصحیح خطا در سیستم‌های مخابراتی (مثل 8B/10B)
- بررسی پایداری داده‌ها در حافظه‌های DRAM و حافظه‌های غیر فرار.

تابع isFaulty()

این تابع وظیفه دارد که یک مقدار ۱۰ بیتی را بررسی کند و مشخص کند که آیا این مقدار مشکوک به خطا است یا نه. در واقع، تابع سعی دارد الگوهای نامعمول را در داده‌های 10 بیتی شناسایی کند و آن‌هایی که از نظر قوانین کدینگ 8B/10B معتبر نیستند، علامت‌گذاری کند. این کار از طریق چندین معیار انجام می‌شود:

۱ .استخراج ۱۰ بیت پایین و مقداردهی اولیه

```
uint16_t data = value & 0x3FF;
```

تابع مقدار value را دریافت می‌کند، اما فقط ۱۰ بیت پایین آن را نگه می‌دارد (چون مقدار دریافتی یک uint16_t یعنی ۱۶ بیتی است).

۲ .شمارش ویژگی‌های بیت‌های 10 بیتی

تابع چند ویژگی داده را محاسبه می‌کند:

1. بیشترین تعداد صفرهای پشت‌سرهم(maxConsecutiveZeros)
2. بیشترین تعداد یک‌های پشت‌سرهم(maxConsecutiveOnes)
3. تعداد کلی بیت‌های ۱(onesCount)
4. تعداد کلی بیت‌های ۰(zerosCount)
5. تعداد تغییرات از ۰ به ۱(transitions0To1)
6. تعداد تغییرات از ۱ به ۰(transitions1To0)

این ویژگی‌ها از طریق یک حلقه for محاسبه می‌شوند که بیت‌های مقدار ۱۰ بیتی را از بیشترین مرتبه به کمترین مرتبه پردازش می‌کند:

```
for (int i = 9; i >= 0; i--) {
    bool currentBit = (data >> i) & 1;

    if (currentBit == 0) {
        consecutiveZeros++;
        consecutiveOnes = 0;
    } else {
        consecutiveOnes++;
        consecutiveZeros = 0;
    }

    maxConsecutiveZeros = max(maxConsecutiveZeros, consecutiveZeros);
    maxConsecutiveOnes = max(maxConsecutiveOnes, consecutiveOnes);

    if (currentBit == 1) {
        onesCount++;
    } else {
        zerosCount++;
    }

    if (i < 9 && currentBit != previousBit) {
        if (previousBit == 0 && currentBit == 1) {
            transitions0To1++;
```

```
        } else if (previousBit == 1 && currentBit == 0) {
            transitions1To0++;
        }
    }

    previousBit = currentBit;
}
```

۳ .بررسی شرایط نامعتبر بودن داده

بعد از محاسبه ویژگی‌های بالا، تابع بررسی می‌کند که آیا مقدار داده شده مشکوک به خطا است یا نه. این بررسی از طریق چندین شرط انجام می‌شود:

```
if (maxConsecutiveZeros > 4) return true;
if (maxConsecutiveOnes > 5) return true;
if (transitions0To1 > 5) return true;
if (transitions1To0 > 5) return true;
if (abs(onesCount - zerosCount) > 3) return true;
```

این بررسی‌ها به چه دلیل انجام می‌شوند؟

- بیش از ۴ صفر پشت سر هم :این موضوع ممکن است نشان‌دهنده یک خطای سخت‌افزاری باشد.
- بیش از ۵ یک پشت سر هم :در کدینگ 8B/10B، چنین توالی‌هایی نادر هستند و معمولاً به دلیل نویز یا خطا ایجاد می‌شوند.
- بیش از ۵ تغییر $0 \to 1$ یا $1 \to 0$: تغییرات بیش از حد زیاد در ۱۰ بیت نشان می‌دهد که داده احتمالاً خراب شده است.
- تفاوت زیاد بین تعداد ۱ و ۰ : اگر تعداد بیت‌های ۱ و ۰ بیش از حد متفاوت باشند (بیش از ۳ عدد اختلاف)، این مقدار احتمالاً نامعتبر است.

۴ .بررسی مقدار در جدول کدینگ 8B/10B

در نهایت، تابع مقدار ۱۰ بیتی را در یک جدول معتبر از مقادیر کدینگ 8B/10B جستجو می‌کند. اگر مقدار داده‌شده در این جدول پیدا نشود، یعنی مقدار نامعتبر است.

```
bool foundInTable = false;
for (int i = 0; i < 512; i++) {
    if ((lookup_8B_10B[i] & 0x3FF) == data) {
        foundInTable = true;
        break;
    }
}
return !foundInTable;
```

- کدینگ 8B/10B فقط به تعداد خاصی از مقادیر ۱۰ بیتی اجازه استفاده می‌دهد.
- اگر مقدار بررسی‌شده در این لیست نباشد، یعنی احتمالاً یک مقدار تصادفی یا خراب است.

نتیجه‌گیری

تابع `isFaulty` داده‌ی ۱۰ بیتی را از چندین جهت تحلیل می‌کند تا ببیند آیا مقدار دریافتی به احتمال زیاد دچار خطا شده است یا خیر.
اگر داده دارای صفات غیرعادی باشد یا در جدول 8B/10B lookup موجود نباشد، مقدار `true` برمی‌گرداند که نشان می‌دهد داده نامعتبر است.
در غیر این صورت، مقدار `false` بازمی‌گردد که یعنی مقدار معتبر است.

تابع `encodeHamming()`

این تابع یک عدد 10بیتی را دریافت کرده و آن را به یک کد 16 بیتی همینگ (Hamming Code) تبدیل می‌کند. این نوع کدگذاری برای تشخیص و تصحیح خطای تک‌بیتی در داده‌ها استفاده می‌شود. در اینجا از کد همینگ (15,11) استفاده شده که 11 بیت داده + 4 بیت توازن 1 + (Parity) بیت اضافه برای همخوانی با 16 بیت دارد.

(به طور خلاصه می‌توان گفت:

1. قرار دادن بیت‌های داده در موقعیت‌های مناسب:
   مقدار dataکه ۱۰ بیت است، در مکان‌های مشخصی از متغیر code ذخیره می‌شود. برخی بیت‌ها برای بیت‌های توازن (parity bits)در نظر گرفته شده‌اند و مقدارشان در این مرحله تعیین نمی‌شود.
2. محاسبه بیت‌های توازن(parity bits) :
   چهار بیت p1 , p2, p4, p8بر اساس XOR بیت‌های داده محاسبه و در code ذخیره می‌شوند.
   این بیت‌ها به تشخیص و تصحیح خطا کمک می‌کنند.
3. بررسی صحت کد تولید شده:
   کد تولید شده بررسی می‌شود که آیا در بازخوانی مقدار اصلی تغییر نکرده است یا نه.

چیدمان بیت‌ها در ۱۶ بیت همینگ:

```
Bit Positions:  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16

Contents:       p1 p2 d1 p4 d2 d3 d4 p8 d5 d6  d7  d8  d9 d10 -  -
```

مراحل اجرا:

۱ .مقداردهی اولیه

```
uint16_t code = 0;
```

یک متغیر 16 بیتی (code) مقداردهی اولیه می‌شود تا داده‌های 10 بیتی و بیت‌های توازن در آن ذخیره شوند.

۲ .جایگذاری 10 بیت داده در موقعیت مناسب

```
code |= (data & 0b0000000001) << 2; // Bit 3
code |= (data & 0b0000000010) << 3; // Bit 5
code |= (data & 0b0000000100) << 3; // Bit 6
code |= (data & 0b0000001000) << 3; // Bit 7
code |= (data & 0b0000010000) << 4; // Bit 9
code |= (data & 0b0000100000) << 4; // Bit 10
code |= (data & 0b0001000000) << 4; // Bit 11
code |= (data & 0b0010000000) << 4; // Bit 12
code |= (data & 0b0100000000) << 4; // Bit 13
code |= (data & 0b1000000000) << 4; // Bit 14
```

وظیفه این بخش:

- داده‌ی 10 بیتی ورودی را در مکان‌های مناسب قرار می‌دهد.
- بیت‌های ۱، ۲، ۴ و ۸ رزرو شده‌اند برای بیت‌های توازن و مقدار داده در بیت‌های دیگر قرار می‌گیرد.

۳ .محاسبه بیت‌های توازن(Parity Bits)

کد همینگ از ۴ بیت توازن (Parity) استفاده می‌کند تا امکان تشخیص خطا را فراهم کند. این بیت‌ها از طریق XOR گروهی از بیت‌های داده محاسبه می‌شوند.

بیت توازن ۱ (bit 1)

```
uint8_t p1 = ((code >> 2) & 1) ^ ((code >> 4) & 1) ^ ((code >> 6) & 1) ^
((code >> 8) & 1) ^ ((code >> 10) & 1) ^ ((code >> 12) & 1) ^ ((code >> 14) &
1);
code |= (p1 << 0);
```

وظیفه: XOR تعدادی از بیت‌های داده را محاسبه کرده و نتیجه را در بیت 1 ذخیره می‌کند.

بیت توازن ۲ (bit 2)

```
uint8_t p2 = ((code >> 1) & 1) ^ ((code >> 2) & 1) ^ ((code >> 5) & 1) ^
((code >> 6) & 1) ^ ((code >> 9) & 1) ^ ((code >> 10) & 1) ^ ((code >> 13) &
1) ^ ((code >> 14) & 1);
code |= (p2 << 1);
```

وظیفه: مقدار XOR دیگری از بیت‌های داده را در بیت 2 ذخیره می‌کند.

بیت توازن ۴ (bit 4)

```
uint8_t p4 = ((code >> 3) & 1) ^ ((code >> 4) & 1) ^ ((code >> 5) & 1) ^
((code >> 6) & 1) ^ ((code >> 11) & 1) ^ ((code >> 12) & 1) ^ ((code >> 13) &
1) ^ ((code >> 14) & 1);
code |= (p4 << 3);
```

وظیفه: در بیت 4 مقدار توازن جدید را قرار می‌دهد.

بیت توازن ۸ (bit 8)

```
uint8_t p8 = ((code >> 7) & 1) ^ ((code >> 8) & 1) ^ ((code >> 9) & 1) ^
((code >> 10) & 1) ^ ((code >> 11) & 1) ^ ((code >> 12) & 1) ^ ((code >> 13)
& 1) ^ ((code >> 14) & 1);
code |= (p8 << 7);
```

وظیفه: مقدار بیت 8 را از XOR گروه دیگری از بیت‌های داده به‌دست آورده و مقدارش را ثبت می‌کند.

۴ .بررسی صحت کدگذاری

```
// if ((extractDataFromHamming(code) & 0x3FF) != data) {
//   Serial.println("Critical error");
// }
```

این خط کد کامنت شده است، اما اگر فعال شود، مقدار تولیدشده توسط `encodeHamming` را بررسی می‌کند تا اطمینان حاصل شود که 10بیت داده اصلی بعد از کدگذاری و استخراج، تغییری نکرده‌اند. در صورت وجود اختلاف، یک پیام خطا نمایش داده می‌شود.

۵ .بازگشت مقدار کدگذاری‌شده

```
return code;
```

مقدار 16 بیتی هیمنگ که شامل 10 بیت داده و 4 بیت توازن است، بازگردانده می‌شود.

جمع‌بندی عملکرد تابع `encodeHamming`

1. داده 10 بیتی را دریافت می‌کند.
2. آن را در موقعیت‌های مناسب در یک مقدار 16 بیتی قرار می‌دهد.
3. ۴ بیت توازن را با استفاده از عملیات XOR محاسبه می‌کند.
4. نتیجه کدگذاری‌شده را بازمی‌گرداند.
5. می‌تواند یک خطای تک‌بیتی را در مرحله دیکدینگ تصحیح کند.

تابع `decodeHamming()`

این تابع یک مقدار کد شده با همینگ (Hamming Code) را دریافت کرده و در صورت وجود خطا، آن را تصحیح می‌کند. الگوریتم همینگ برای تشخیص و اصلاح یک خطای تک‌بیتی در داده استفاده می‌شود. در این تابع، از کد همینگ (15,11) استفاده شده است که از 15بیت برای ارسال اطلاعات استفاده می‌کند که شامل 11بیت داده و 4 بیت توازن (Parity) است.

۱ .مقداردهی اولیه

```
uint16_t hamming = code;
```

مقدار `code` همان مقدار 15 بیتی همینگ است که از قبل کد شده است و ممکن است دارای یک خطای تک‌بیتی باشد. این مقدار در متغیر `hamming` ذخیره می‌شود تا در صورت لزوم اصلاح شود.

۲ .محاسبه بیت‌های توازن(Parity Bits)

در کد همینگ، ۴ بیت توازن (`p1`, `p2`, `p4`, `p8`) برای بررسی خطا استفاده می‌شود. هر یک از این بیت‌ها XOR گروهی از بیت‌های داده را ذخیره می‌کنند.

```
uint8_t p1 = ((code >> 0) & 1) ^ ((code >> 2) & 1) ^ ((code >> 4) & 1) ^
((code >> 6) & 1) ^ ((code >> 8) & 1) ^ ((code >> 10) & 1) ^ ((code >> 12) &
1) ^ ((code >> 14) & 1);

uint8_t p2 = ((code >> 1) & 1) ^ ((code >> 2) & 1) ^ ((code >> 5) & 1) ^
((code >> 6) & 1) ^ ((code >> 9) & 1) ^ ((code >> 10) & 1) ^ ((code >> 13) &
1) ^ ((code >> 14) & 1);

uint8_t p4 = ((code >> 3) & 1) ^ ((code >> 4) & 1) ^ ((code >> 5) & 1) ^
((code >> 6) & 1) ^ ((code >> 11) & 1) ^ ((code >> 12) & 1) ^ ((code >> 13) &
1) ^ ((code >> 14) & 1);

uint8_t p8 = ((code >> 7) & 1) ^ ((code >> 8) & 1) ^ ((code >> 9) & 1) ^
((code >> 10) & 1) ^ ((code >> 11) & 1) ^ ((code >> 12) & 1) ^ ((code >> 13)
& 1) ^ ((code >> 14) & 1);
```

عملکرد بیت‌های توازن:

- `p1` مقدار XOR برخی از بیت‌های خاص را نگه می‌دارد.
- `p2` مقدار XOR دیگری از گروه‌های خاص را محاسبه می‌کند.
- `p4` و `p8` نیز مقادیر مشابهی را محاسبه می‌کنند.

هدف این است که اگر مقدار دریافتی در یک بیت دارای خطا باشد، این بیت‌های توازن تغییر کنند و به ما کمک کنند موقعیت خطا را پیدا کنیم.

۳ .محاسبه "سندروم خطا"

```
uint16_t syndrome = (p1 << 0) | (p2 << 1) | (p4 << 2) | (p8 << 3);
```

`syndrome` یک عدد ۴ بیتی است که مقدار آن مشخص می‌کند کدام بیت از داده‌ها دچار خطا شده است.
اگر مقدار `syndrome` برابر `0` باشد، یعنی داده صحیح است و نیازی به تصحیح ندارد.

۴ .تصحیح خطای تک‌بیتی (در صورت وجود)

```
if (syndrome != 0) {
    int errorBit = syndrome - 1; // 0 موقعیت خطای-indexed
    hamming ^= (1 << errorBit);
    Serial.print("Hamming: Error corrected at bit ");
    Serial.println(errorBit);
}
```

عملکرد این بخش:

- اگر مقدار `syndrome` صفر نباشد، یعنی یک خطای تک‌بیتی وجود دارد.
- مقدار `syndrome` موقعیت دقیق بیت خراب‌شده را مشخص می‌کند.
- با استفاده از عملیات `XOR`، مقدار آن بیت را برعکس می‌کنیم تا خطا برطرف شود.
- موقعیت بیتی که اصلاح شده است در سریال مانیتور چاپ می‌شود.

۵ .بازگرداندن مقدار تصحیح‌شده

```
return hamming;
```

پس از بررسی و اصلاح خطا، مقدار اصلاح‌شده بازگردانده می‌شود.

نتیجه‌گیری کلی

این تابع از کد همینگ (15,11) برای تصحیح یک خطای تک‌بیتی استفاده می‌کند.
در صورت بروز خطا، مقدار آن را با استفاده از بیت‌های توازن پیدا کرده و آن را اصلاح می‌کند.
اگر خطا وجود نداشته باشد، مقدار اصلی بدون تغییر بازگردانده می‌شود.
از عملیات XOR برای بررسی بیت‌های داده و تشخیص خطا استفاده می‌شود.

تابع `extractDataFromHamming()`

هدف تابع

این تابع داده اصلی 10 بیتی را از یک کد همینگ 16 بیتی استخراج می‌کند.
کد همینگ شامل 10بیت داده + 5 بیت توازن 1 + (Parity) بیت اضافی است .تابع، این بیت‌های توازن را نادیده می‌گیرد و فقط بیت‌های داده‌ای را استخراج می‌کند.

کاربرد اصلی:
بازگردانی داده اصلی از کد همینگ پس از بررسی یا تصحیح خطا.
استفاده در سیستم‌های تصحیح خطا در مخابرات و حافظه‌هاRAM) ، DDR، .(SSD

۱ .استخراج بیت‌های داده‌ای

```
uint16_t data = 0;
```

یک متغیر 16 بیتی برای ذخیره داده خالص (10 بیت اصلی) ایجاد می‌شود.

۲ .نگاشت بیت‌های کد همینگ به داده اصلی

کد همینگ 16 بیتی این ساختار را دارد:

```
P1 P2 D1 P4 D2 D3 D4 P8 D5 D6 D7 D8 D9 D10
```

هدف :حذف بیت‌های توازن (`P1, P2, P4, P8`) و استخراج بیت‌های داده.

نگاشت بیت‌ها در `extractDataFromHamming`

کد همینگ ← داده اصلی

```
data |= ((hammingCode >> 2) & 1) << 0;  // بیت 3 → بیت 0
data |= ((hammingCode >> 4) & 1) << 1;  // بیت 5 → بیت 1
data |= ((hammingCode >> 5) & 1) << 2;  // بیت 6 → بیت 2
data |= ((hammingCode >> 6) & 1) << 3;  // بیت 7 → بیت 3
data |= ((hammingCode >> 8) & 1) << 4;  // بیت 9 → بیت 4
data |= ((hammingCode >> 9) & 1) << 5;  // بیت 10 → بیت 5
data |= ((hammingCode >> 10) & 1) << 6; // بیت 11 → بیت 6
data |= ((hammingCode >> 11) & 1) << 7; // بیت 12 → بیت 7
data |= ((hammingCode >> 12) & 1) << 8; // بیت 13 → بیت 8
data |= ((hammingCode >> 13) & 1) << 9; // بیت 14 → بیت 9
```

عملکرد:

1. بیت متناظر در `hammingCode` را می‌خواند `(hammingCode >> n) & 1`
2. آن را در جای درست در `data` قرار می‌دهد `(<< m)`
3. همه بیت‌ها را با OR `|=` در `data` ذخیره می‌کند.

جمع‌بندی عملکرد `extractDataFromHamming`

1. بیت‌های داده‌ای (10 بیت اصلی) را از کد همینگ 16 بیتی جدا می‌کند.
2. بیت‌های تصحیح خطا را حذف می‌کند.
3. خروجی تابع مقدار اصلی داده را باز می‌گرداند.

کاربردهای عملی:

- دیکد کردن داده‌ها در مخابرات و حافظه‌های ECC.
- بازسازی داده اصلی پس از بررسی و تصحیح خطا.
- استفاده در سیستم‌های ایمنی و صنعتی برای حفاظت از داده‌ها.

تابع `loop()`

این تابع وظیفه‌ی اجرای چرخه‌ی اصلی پردازش داده‌ها را بر عهده دارد:

۱ .خواندن داده‌های آنالوگ و تبدیل آن به یک مقدار 66 بیتی

۱.۱ .خواندن مقدار آنالوگ و ذخیره در data_66b.lowBits

```
for (int i = 0; i < 6; i++) {
    uint64_t analogValue = analogRead(A0 + i);
```

این حلقه مقدار آنالوگ را از ۶ پین (A0تا A5) می‌خواند. مقدار آنالوگ به عددی بین ۰ تا ۱۰۲۳ تبدیل می‌شود.

```
    for (int bitPos = 0; bitPos < 10; bitPos++) {
      int bitValue = (analogValue >> bitPos) & 1;
      int targetBitPos = i * 10 + bitPos;
      if (bitValue == 0x01) {
        data_66b.lowBits |= (uint64_t)((uint64_t)0x01 <<
targetBitPos);
      }
    }
```

- مقدار آنالوگ (۱۰ بیتی) را به صورت بیت به بیت بررسی می‌کند.
- بیت‌ها را در موقعیت مناسب در data_66b.lowBits تنظیم می‌کند.

۱.۲ .اضافه کردن ۶ بیت پایانی و مقدار highBits

```
uint8_t last_six_bits = analogRead(A0) & 0x3F;
data_66b.lowBits |= last_six_bits << (60);
data_66b.highBits = random(1, 3);
```

- ۶ بیت پایانی مقدار آنالوگ پین A0 را دریافت کرده و به ۶ بیت پایانی lowBits اضافه می‌کند.
- highBits مقدار تصادفی ۱ یا ۲ دریافت می‌کند. دلیل انتخاب این دو عدد برای دو بیت پرارزش این است که اعداد 1 و 2 برای ورودی داده به کار می روند. (و نه بیت های کنترل یا خطا) به همین دلیل فرض شده که هنگام تولید داده های 66 بیتی، تنها بیت های داده را ورودی می گیریم.

۲ .رمزگذاری داده‌ها با 8B/10B و کد همینگ

۲.۱ .تعریف آرایه‌ها برای بررسی خطاها

```
uint16_t encodedValues[8];
bool transientErrors[8] = {0};
bool suspectedFaults[8] = {0};
uint16_t originalValues[8];
uint16_t originalHamming[8];
```

```
uint16_t encodedHamming[8];
bool guessedTransient[8] = {false};
bool guessedPermenant[8] = {false};
```

این آرایه‌ها وضعیت بیت‌های خطادار، تصحیح‌شده، و نسخه‌های رمزگذاری‌شده را نگه می‌دارند. در ادامه در مورد هر کدام از آن ها بیشتر توضیح می دهیم.

۲.۲ .تبدیل هر ۸ بیت به ۱۰ بیت (8B/1B) و کدگذاری همینگ

```
for (int i = 0; i < 8; i++) {
    uint8_t byte = (data_66b.lowBits >> (i * 8)) & 0xFF;
    uint16_t encoded = convert_8B_10B(byte);
    encoded = reverseLowest10Bits(encoded);
    originalValues[i] = encoded;
    originalHamming[i] = encodeHamming(encoded);
    encodedHamming[i] = encodeHamming(encoded);
```

- هر ۸ بیت از `lowBits` استخراج شده و با `convert_8B_10B()` به ۱۰ بیت تبدیل می‌شود.
- بیت‌ها با استفاده از تابع `reverseLowest10Bits` معکوس می‌شوند.
- در آخر، این داده ها را به فرمت همینگ تبدیل کرده و آن ها را در originaHamming و encodedHamming ذخیره می کنیم تا بعد از تزریق خطا، بتوان مقدار درست را بازیابی کرد.

۲.۳ .شبیه‌سازی خطاهای تصادفی (٪۸۰ احتمال ایجاد خطا)

```
if (random(0, 10) < 8) {
      injectError(encodedHamming[i], transientErrors[i]);
}
```

با ٪۸۰ احتمال، یک خطای تصادفی در `encodedHamming[i]` ایجاد می‌شود.

۲.۴ .بررسی مقدار خراب‌شده

```
suspectedFaults[i] = isFaulty(encodedHamming[i]);
```

با استفاده از تابع isFaulty، برنامه بررسی می‌کند که آیا مقدار دارای خطا است یا خیر. توجه کنید که مکانیزم تشخیص خطا در این خط از همان انکودینگ 10/8 بیت استفاده می کند.

اصلاح خطاهای گذرا(Transient Errors)

حال که برنامه تعدادی خطا را تشخیص داد، باید آن ها را تصحیح کند. اما قبل از آن، لازم است تا خطاهای گذرا رفع شوند.

همانطور که گفته شد، هر خطایی که تزریق می شود یا دائمی است یا گذرا. ویژگی خطاهای گذرا این است که پس از مدتی رفع می شوند، و به اصطلاح دائمی نیستند. برای پیاده سازی این مفهوم، در فاصله بین *تشخیص خطا* و *تصحیح خطا*، این خطاها را رفع می کنیم تا مفهوم گذرا بودن ای نوع خطا قابل مشاهده باشد. به خطاهای دائمی نیز دست نمی زنیم چون این خطاها باید باقی بمانند.

تابع زیر، با استفاده از تابع transientErrors، خطاهای گذرا را پیدا کرده و مقدار اصلی آن ها را – که در آرایه originalHamming ذخیره شده – بر می گرداند. سپس اطلاعات خطا و اندیس داده را پرینت می کند.

```
Serial.println("Correcting transient errors...");
for (int i = 0; i < 8; i++) {
  if (transientErrors[i]) {
    encodedHamming[i] = originalHamming[i];
    transientErrors[i] = false;
    Serial.print("Transient error corrected at index ");
    Serial.println(i);
  }
}
```

۴ .بررسی مقادیر مشکوک و تصحیح خطاها با کد همینگ

۴.۱ .نمایش مقادیر مشکوک

```
Serial.println("Current suspected values are:");
for (int i = 0; i < 8; i++) {
  if (suspectedFaults[i]) {
    Serial.print("Index ");
    Serial.println(i);
  }
}
```

فهرست خطاهای مشکوک نمایش داده می‌شود. تعدادی از این خطاها دائمی بوده و در حال حاظر واقعا مقداری اشتباه دارند و تعدادی از آن ها گذرا بوده و مقدار متناظر آن ها در مرحله قبلی اصلاح شده است.

۴.۲ بررسی و تصحیح خطاهای دائمی(Permanent Errors)

در این بخش، مکانیزم تصحیح خطا پیاده سازی می شود. از آنجایی که انکودینگ 10/8 بیت یا 66/64 بیت اگرچه برای تشخیص خطا مفید هستند اما برای تصحیح خطا عملکرد مناسبی ندارند، از یک رش دیگر به نام کد همینگ استفاده می کنیم. کافی است تا کد همینگی که روی آن خطا تزریق شده را دیکود کرده و آن را به تابع decodeHamming بدهیم. این تابع مقدار اصلاح شده ورودی را می دهد. اگر این مقدار با ورودی تابع برابر نبود، یعنی خطایی رخ داده است و کد همینگ ان را اصلاح کرده است. در نتیجه متوجه می

شویم خطای رخ داده از نوع دائمی بوده است. البته قبل از چاپ اطلاعات خطا، داده اصلاح شده را با داده اصلی مقایسه می کنیم تا مطمئن شویم که در تبدیل این داده به کد همینگ اشتباهی رخ نداده است. سپس این خطای دائمی را چاپ می کنیم.

حال اگر خروجی تابع decodeHamming با ورودی آن یکی باشد، یعنی کد همینگ خطایی را شناسایی نکرده است. پس نتیجه می گیریم خطایی که رخ داده، گذرا بوده است و در قبل از این که اسقدام به تصحیح خطا کنیم، خود به خود برطرف شده است. بنابراین این نتیجه را نیز چاپ می کنیم و تصحیح خطا کامل می شود.

```
Serial.println("Analyzing faults with Hamming code...");
for (int i = 0; i < 8; i++) {
  if (suspectedFaults[i]) {
    uint16_t correctedHamming =
decodeHamming(encodedHamming[i]);
    if (correctedHamming != encodedHamming[i]) {
      uint16_t correctedRawData =
extractDataFromHamming(correctedHamming);

    if (correctedRawData != originalValues[i]) {
        Serial.println("Decoding hamming is not correct");
      }

      guessedPermenant[i] = true;
      Serial.print("Permanent fault detected at index ");
      Serial.print(i);
      Serial.print(". Original value: ");
      print10BitBinary(encodedValues[i]);
      Serial.print("Corrected value: ");
      print10BitBinary(correctedRawData);
    } else {
      guessedTransient[i] = true;
      Serial.print("False alarm or transient error at index ");
      Serial.println(i);
    }
  }
}
```

۵ .نمایش داده‌های نهایی و ارسال به پین‌ها

۵.۱ .نمایش مقدار نهایی

```
Serial.println("Final data:");
for (int i = 0; i < 8; i++) {
    print10BitBinary(originalValues[i]);
```

ابتدا داده‌های ده بیتی به صورت باینری چاپ می‌شوند.

۵.۲ .ارسال داده‌های رمزگذاری‌شده به پین‌های دیجیتال

```
for (int pin = startPin + 2; pin <= endPin; pin++) {
    int delta = pin - startPin - 2;
    if ((originalValues[i] >> delta) & 1) {
        digitalWrite(pin, HIGH);
    }
    else {
        digitalWrite(pin, LOW);
    }
}
```

بیت‌های هر مقدار به ترتیب به پین‌های دیجیتال ارسال می‌شوند تا داده های انکود شده ده بیتی در LED ها نمایش داده شوند.

۵.۳ .نشان دادن خطاهای تصحیح‌شده با LED

```
digitalWrite(3, LOW);
digitalWrite(2, LOW);
if (guessedPermenant[i] == true) {
    digitalWrite(3, HIGH);
}
if (guessedTransient[i] == true) {
    digitalWrite(2, HIGH);
}
delay(60000);
```

- اگر خطای دائمی اصلاح شده باشد، `digitalWrite(3, HIGH)` فعال می‌شود.
- اگر خطای گذرا اصلاح شده باشد، `digitalWrite(2, HIGH)` فعال می‌شود.
- هر مقدار ۶۰ ثانیه بررسی می‌شود.

۶ .تأخیر برای اجرای چرخه‌ی بعدی

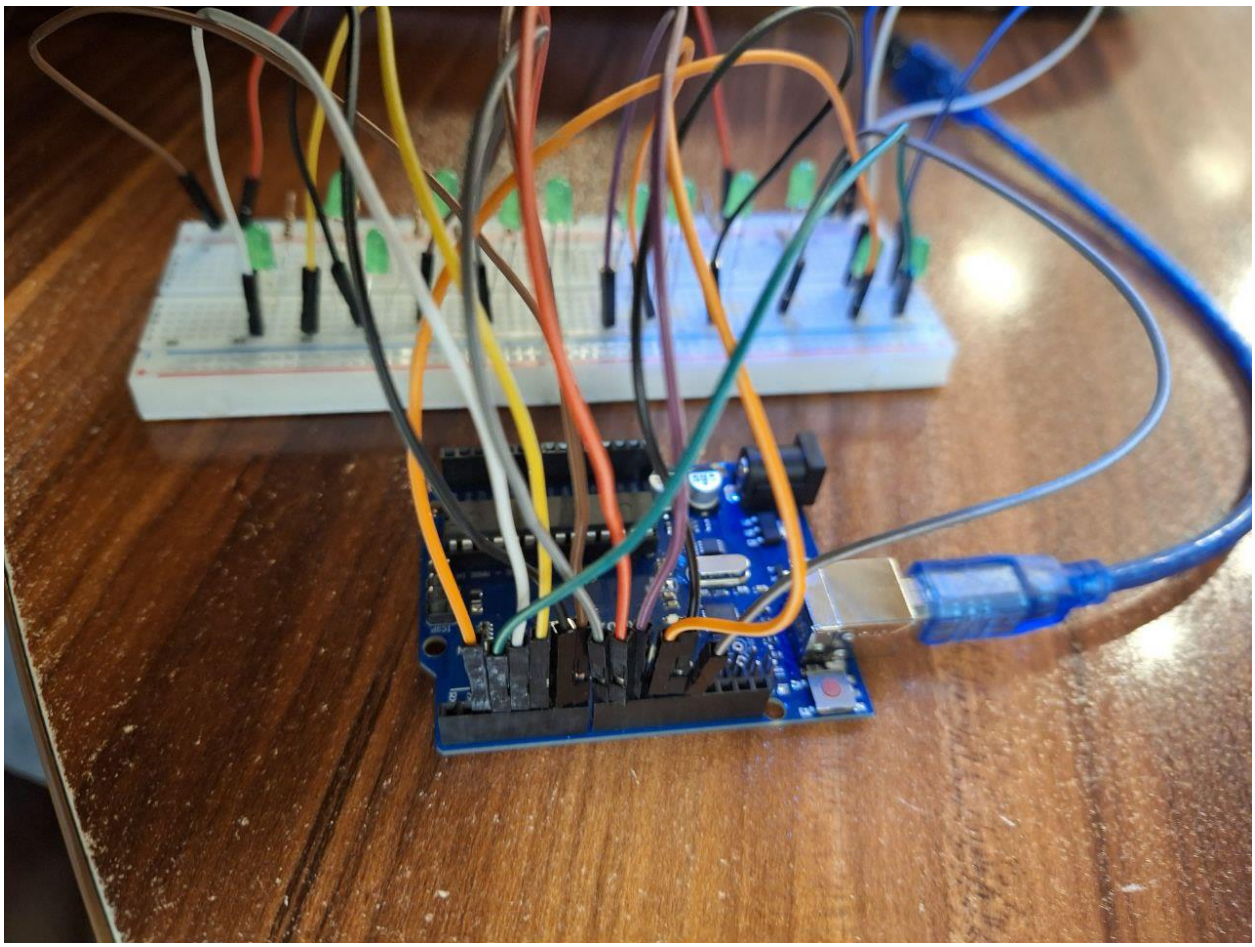
```
delay(300000);
```

۵ دقیقه تأخیر برای اجرای مجدد `loop()` به منظور شبیه‌سازی یک سیستم زمان‌بندی‌شده.

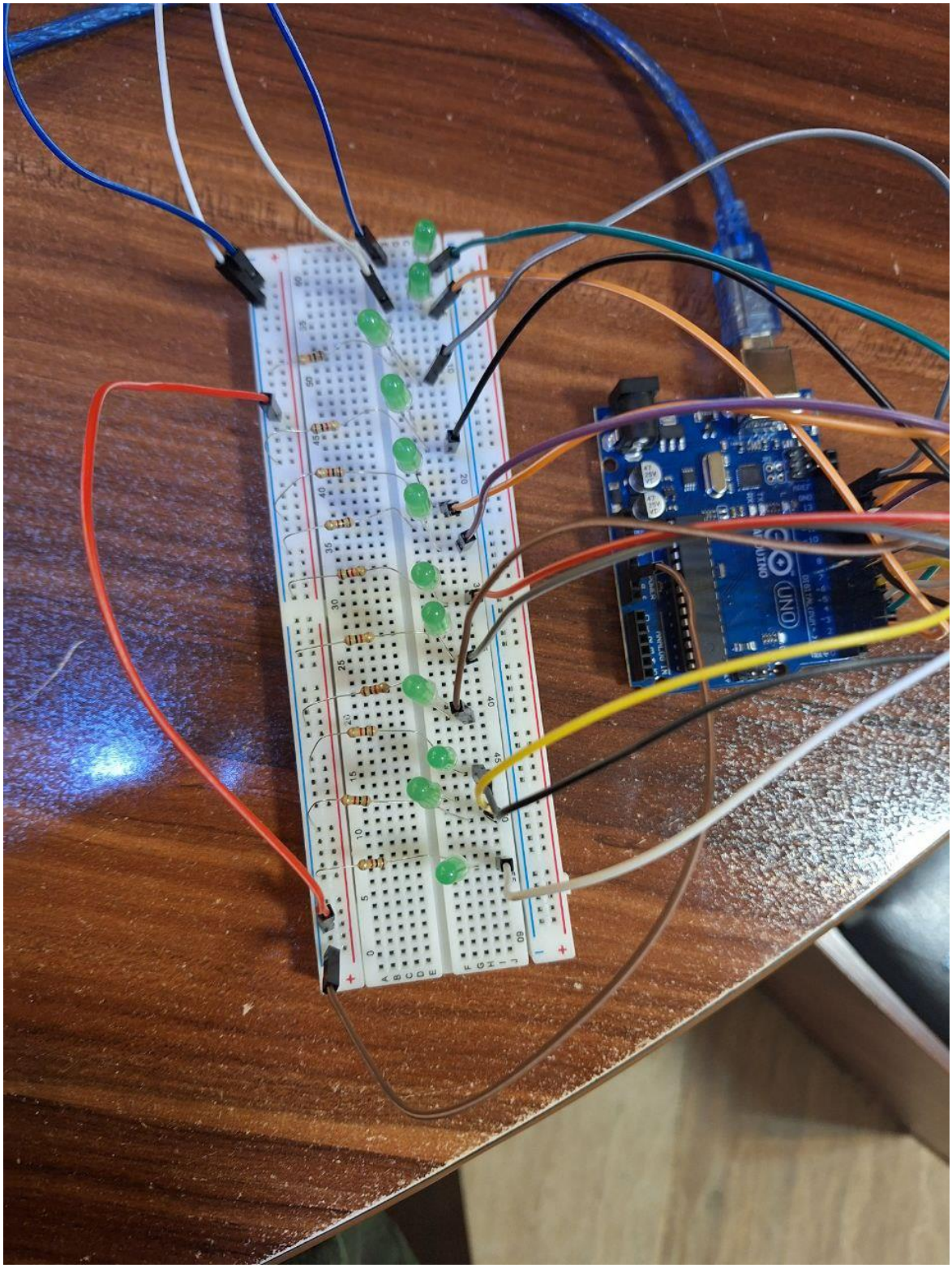
نتیجه‌گیری

پروژه طراحی مبدل انکودینگ 64B/66B به 8B/10B با هدف بررسی و مقایسه توانایی‌های این انکودینگ‌ها در شناسایی خطا، راهکاری عملی برای بهبود کارایی سیستم‌های مخابراتی ارائه می‌دهد. همچنین، توسعه الگوریتم‌های تصحیح خطا می‌تواند گامی مؤثر در افزایش قابلیت اطمینان این سیستم‌ها باشد.

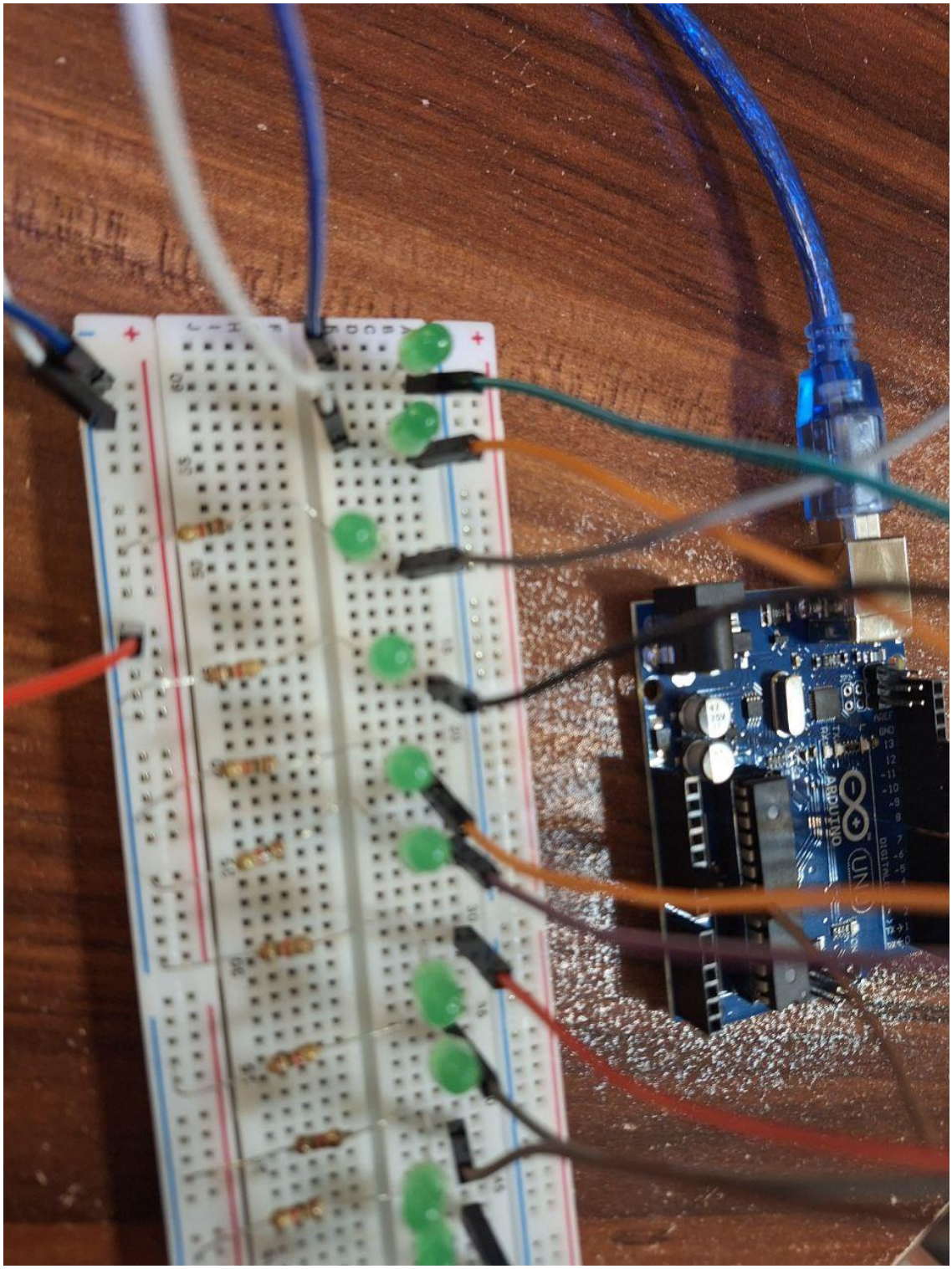
تصاویر نمونه پیاده‌سازی عملی



نمای کلی برد Arduino در هنگام اتصال



اتصالات Breadboard و LEDها و مقاومت‌ها



نمایی دیگر از جزئیات برد Arduino و مدار اصلی

منابع

[1] https://users.cs.fiu.edu/~downeyt/cop3402/hamming.html

[2] https://www.cs.cornell.edu/courses/cs3410/2013sp/lab/tables/dy_encoding_table.pdf

[3] https://www.ieee802.org/3/bn/public/mar13/hajduczenia_3bn_04_0313.pdf

[4] https://www.intel.com/content/www/us/en/docs/programmable/683617/21-1/64b-66b-encoder-and-transmitter-state.html

[5] https://cdrdv2-public.intel.com/654653/agx_52004.pdf