

«بسمه تعالیٰ»



دانشکده مهندسی کامپیوتر

گزارش پروژه هشتم درس مدارهای واسط

عنوان:

طراحی مبدل انکوڈینگ 8B/10B به 64B/66B

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

گردآورندگان:

نیکا قادری

آریان افضلزاده

عاطفه قندھاری

استاد درس:

دکتر امین فصحتی

پاییز ۱۴۰۳

فهرست

3	مقدمه
3	اهداف پروژه
3	معرفی انکودینگ ها
4	سخت افزار
4	نرم افزار
5	شرح توابع به کار رفته در نرم افزار مبدل
5	Convert 8b/10b ()
6	Reverse Lowest 10 Bits ()
6	Encode Hamming ()
7	Decode Hamming ()
9	is Faulty ()
12	Inject Error ()
12	Print 10bit Binary ()
13	توضیح عملکرد توابع
13	تعریف متغیرها و ساختار داده
13	تابع () setup()
15	تابع () convert_8B_10B()
17	تابع () reverseLowest10Bits()
20	تابع () injectError()
21	تابع () isFaulty()
24	تابع () encodeHamming()
27	تابع () decodeHamming()
29	تابع () extractDataFromHamming()
30	تابع () loop()
36	نتیجه گیری
36	تصاویر نمونه پیاده سازی عملی
39	منابع

مقدمه

در عصر حاضر، سیستم‌های مخابراتی و انتقال داده به منظور افزایش بازده و کاهش خطا از روش‌های انکودینگ متعددی 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 پژوهش، 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 پژوهش

1. طراحی و پیاده‌سازی مبدل انکودینگ: توسعه سیستمی که داده‌های دریافت شده 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 عمده خطاهای 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.

سخت افزار

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 RMF 1/4W %5

نرم افزار

. Arduino IDE

ابزارهای شبیه‌سازی و تحلیل داده.

شرح توابع به کار رفته در نرم افزار 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 10/8 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 10/8 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 مقدار RD 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 چهار شیفت دهیم و با داده فعلی Or کنیم. سپس سطر متاثر 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 بیت‌ها 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 می‌گیرد (در قالب `uint16_t`) و آن را با استفاده از فرمت همینگ ۱۱/۵ کد گذاری می‌کند. ابتدا بیت‌های داده شیفت می‌خورد و هر بیت در مکان مناسب خود قرار می‌گیرد. سپس، با استفاده از بیت‌های داده، بیت‌های parity ساخته می‌شوند و به داده خروجی اضافه می‌شوند.

```

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 & 0b00000000100) << 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 کثارت هم گذاشتن این بیت های نیز می توانیم مکان این خطای 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 می‌گیرد، از آن عدد ده بیتی منتظر را استخراج می‌کند و سپس بررسی می‌کند که آیا در این عدد خطأ وجود دارد یا خیر.

این تشخیص خطأ با استفاده از انکوادینگ 10/8 بیت و ویژگی‌های آن انجام می‌شود. همانطور که دیدیم، تبدیل هشت بیت داده به ده بیت خروجی انکواد شده توسط یک جدول انجام می‌شود. 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، می‌توان مکانیزمی جهت تشخیص خطأ ساخت.

آنالیز جدول انکوادینگ توسط کد پایتون زیر انجام می‌شود:

```
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)[1:]
    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 رخداده است و به این صورت می توان از انکودينگ 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 می‌گیرد، یک بیت آن را به صورت شانسی انتخاب کرده و آن را معکوس می‌کند.
بدین صورت خطأ تزریق می‌شود.

نکته دیگر آن است که در این پیاده سازی، دو نوع خطأ 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 می‌کند و با است کردن مقدار isTransient، این مقدار را ثبت می‌کند.

در نهایت این تابع به صورت زیر پیاده سازی می‌شود:

```

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 بازگشایی چاپ می‌کند. مزیت آن در این است که صفرهای قبل از عدد را که از نظر ریاضی بی ارزش هستند نیز چاپ می‌کند. 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 DC (Disparity Control) در کدگذاری 8B/10B است. مقدار آن 0 (تعادل منفی) یا 1 (تعادل مثبت) است.
- lookup_8B_10B[512] : یک آرایه از پیش محاسبه شده که شامل کدگذاری های استاندارد 8B/10B است. ورودی 8 بیتی گرفته می شود و مقدار 10 بیتی متناسب با آن از این جدول استخراج می شود.

```

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

```

- این ساختار داده برای ذخیره فریم های 64B/66B استفاده می شود. مقدار lowBits 64 بیت داده است و highBits دو بیت کنترلی رانگه می دارد.

تابع (setup)

هدف تابع

این تابع در مرحله‌ی راهاندازی (setup) آردوینو اجرا می شود و کارهای زیر را انجام می دهد:

- . تنظیم یک محدوده از پین ها (startPin و endPin) به عنوان خروجی (OUTPUT).
- . فعالسازی ارتباط سریال با نرخ baud 9600 برای ارسال داده به کامپیوتر یا مانیتور سریال.

کاربرد اصلی:

تنظیم پین‌های دیجیتال برای کنترل LED، 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 از شروع می‌شود.
2. تازمانی 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 به عنوان 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.

2. 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، RFID و ...

جمع‌بندی عملکرد `setup`

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

کاربردهای عملی:

- کنترل LED، 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 خط استفاده می‌شود.

به طور کلی می‌توان گفت:

1. ساخته می‌شود که شامل مقدار فعلی `rd` (بیت تعادل DC) 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 رمزگذاری شده 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 (`rd`) بازگردانده می‌شود.

نکات مهم:

- این روش از یک جدول `lookup_8B_10B` برای تبدیل سریع استفاده می‌کند که باعث کاهش پردازش می‌شود.
- مقدار `rd` کنترل 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.
متغیر `rd` نشان‌دهنده‌ی تعادل بین تعداد صفرها و یک‌ها در رشته کدگذاری شده‌ی 8B/10B (Running Disparity) است و در هر مرحله بروزرسانی 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 بت) | 00000000(7 بت) | 00000000(8 بت 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 بت) | 00000000(7 بت) | داده 8 بیتی]
```

۴. دریافت مقدار کدگذاری شده 10 بیتی از جدول `lookup_8B_10B`

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

مقدار `table_in` را به عنوان ایندکس (کلید) برای دسترسی به یک جدول از پیش محاسبه شده (`lookup_8B_10B`) استفاده می‌کند.

این جدول شامل تمام ترکیب‌های ممکن 8 بیت ورودی و 1 بیت `rd` است و مقدار معادل 10 بیت کدگذاری شده را بر می‌گرداند.

۵. بروزرسانی (`rd` (Running Disparity))

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

مقدار جدید `rd` را از بیت یازدهم (بیت شماره 10) مقدار کدگذاری شده 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 بازمی‌گردد.
برابر است با `0x3FF` که فقط 10 بیت پایین‌تر را نگه می‌دارد.

جمع‌بندی عملکرد تابع `convert_8B_10B`

1. مقدار `rd` را به عنوان بیت یازدهم در نظر می‌گیرد و آن را به 8 بیت داده‌ی ورودی اضافه می‌کند.
2. مقدار ترکیبی را به عنوان ایندکس برای دریافت مقدار کدگذاری شده 10 بیتی از `lookup_8B_10B` استفاده می‌کند.
3. مقدار جدید `rd` را به روزرسانی می‌کند تا برای کدگذاری بعدی استفاده شود.
4. مقدار 10 بیت نهایی را بر می‌گردد.

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 16 بیتی (`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 تبدیل Endianness، پردازش داده‌های باینری خاص، و تشخیص خطأ مفید است.

۱. تعریف متغیرها

```
uint16_t reversed = 0;
```

متغیر `reversed` مقدار معکوس شدهی 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:

: `(value >> i) & 1` .1

بیت شماره `i` از `value` را استخراج می‌کند.

: `<< (9 - i)` .2

این بیت را به موقعیت معکوس شدهی آن در `reversed` منتقل می‌کند.

: `reversed |= ...` .3

مقدار استخراج شده را در `reversed` قرار می‌دهد.

مثال عددی:

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

i	value	(i - 9) بیت i از reversed	جایگاه جدید (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 معکوس شده

```
return reversed;
```

مقدار 10 بیت معکوس شده 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 معکوس شده را بازمی‌گرداند.

کاربردهای عملی تابع:

- کدگذاری و رمزگذاری داده‌های سریالی (B/10B8)
- بررسی و تصحیح داده‌های باینری در مخابرات و پردازش سیگنال.
- تبدیل Big-Endian و Little-Endian و پردازش داده‌های Endianness

تابع `injectError()`

هدف تابع

این تابع به طور 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، تصویح خطا (ECC)، و پردازش سیگنال. آزمایش الگوریتم‌های تشخیص و تصویح 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 (Flip کردن بیت)

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

این خط بیت انتخاب شده را معکوس می‌کند:

نحوه‌ی عملکرد:

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

مثال عددی:

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

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

بیت شماره 3 از 1 به 0 تغییر کرد!

۳. تعیین نوع خطا (گذرا یا دائمی)

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

مقدار `isTransient` را به طور تصادفی `true` (گذرا) یا `false` (دائمی) تنظیم می‌کند.

خطای گذرا (transient error) :

خطا موقتی است و احتمال دارد در بازخوانی بعدی رفع شود.

معمولًا 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.

معمولًا ناشی از خرابی سخت‌افزاری یا سلول‌های معیوب حافظه است.

جمع‌بندی عملکرد injectError

1. یک بیت تصادفی را از 10 بیت پایین `data` انتخاب می‌کند.

2. بیت انتخاب شده را معکوس (flip) می‌کند تا خطا ایجاد شود.

3. تصادفی تعیین می‌کند که خطا گذرا (`true`) یا دائمی (`false`) باشد.

کاربردهای عملی تابع:

- شبیه‌سازی خطاهای ECC (مانند Hamming 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 10 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،

علامت گذاری 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 ۱۰ 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 خطأ 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 عموماً به دلیل نویز یا خطأ ایجاد می‌شوند.
- بیش از ۵ تغییر $0 \rightarrow 1$ یا $1 \rightarrow 0$: 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 داده شده 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 بررسی شده در این لیست نباشد، یعنی احتمالاً یک مقدار تصادفی یا خراب است.

نتیجه گیری

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 ۸B/۱۰B lookup موجود نباشد، مقدار `true` بر می‌گرداند که نشان می‌دهد داده نامعتبر است.

در غیر این صورت، مقدار `false` بازمی‌گردد که یعنی مقدار معتبر است.

تابع `encodeHamming()`

این تابع یک عدد ۱۰ بیتی را دریافت کرده و آن را به یک کد ۱۶ 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 ۱۱ بیت داده + ۴ بیت توازن ۱ + (Parity) بیت اضافه برای همخوانی با ۱۶ بیت دارد.

(به طور خلاصه می‌توان گفت:

۱. قرار دادن بیت‌های داده در موقعیت‌های مناسب:

مقدار `data` که 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.

۲. محاسبه بیت‌های توازن (parity bits) :

چهار بیت `p1, p2, p4, p8` بر اساس `XOR` بیت‌های داده محاسبه و در `code` ذخیره می‌شوند. این بیت‌ها به تشخیص و تصحیح خطای کمک می‌کنند.

۳. 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 & 0b000000000001) << 2; // Bit 3
code |= (data & 0b000000000010) << 3; // Bit 5
code |= (data & 0b000000000100) << 3; // Bit 6
code |= (data & 0b00000001000) << 3; // Bit 7
code |= (data & 0b00000010000) << 4; // Bit 9
code |= (data & 0b00001000000) << 4; // Bit 10
code |= (data & 0b00010000000) << 4; // Bit 11
code |= (data & 0b00100000000) << 4; // Bit 12
code |= (data & 0b01000000000) << 4; // Bit 13
code |= (data & 0b10000000000) << 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 ۱ 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 به دست 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 10 بیت داده اصلی بعد از کدگذاری و استخراج، تغییری نکرده‌اند. در صورت وجود اختلاف، یک پیام خطا نمایش داده می‌شود.

۵. بازگشت مقدار کدگذاری شده

```
return code;
```

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

جمع‌بندی عملکرد تابع encodeHamming

1. داده 10 بیتی را دریافت می کند.
2. آن را در موقعیت های مناسب در یک مقدار 16 بیتی قرار می دهد.
3. ۴ بیت توازن را با استفاده از عملیات XOR محاسبه می کند.
4. نتیجه کدگذاری شده را بازمی گردداند.
5. می تواند یک خطای تک بیتی 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 می کند. الگوریتم همینگ برای تشخیص و اصلاح یک خطای تک بیتی در داده استفاده می شود. در این تابع، از کد همینگ (15,11) استفاده شده است که از 15 بیت برای ارسال اطلاعات استفاده می کند که شامل 11 بیت 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 تک بیتی باشد. این مقدار در متغیر `hamming` ذخیره می شود تا در صورت لزوم اصلاح شود.

۲. محاسبه بیت های توازن(Parity Bits)

در کد همینگ، ۴ بیت توازن (p_1, p_2, p_4, p_8) برای بررسی خطا استفاده می شود. هر یک از این بیت ها XOR گروهی از بیت های داده را ذخیره می کنند.

```
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 خطأ باشد، این بیت‌های توازن تغییر کنند و به ما کمک کنند موقعیت خطأ را پیدا کنیم.

۳. محاسبه "سندروم خطأ"

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

یک عدد ۴ بیتی است که مقدار آن مشخص می‌کند کدام بیت از داده‌ها دچار خطأ شده است. syndrome
اگر مقدار syndrome برابر ۰ 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-موقعيت خطاي 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 خراب شده را مشخص می‌کند.
- با استفاده از عملیات XOR، مقدار آن بیت را برعکس می‌کنیم تا خطأ برطرف شود.
- موقعیت بیتی که اصلاح شده است در سریال مانیتور چاپ می‌شود.

۵. بازگرداندن مقدار تصحیح شده

```
return hamming;
```

پس از بررسی و اصلاح خطأ، مقدار اصلاح شده بازگردانده می‌شود.

نتیجه گیری کلی

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

تابع extractDataFromHamming()

هدف تابع

این تابع داده اصلی 10 بیتی را از یک کد همینگ 16 بیتی استخراج می‌کند.
کد همینگ شامل 10 بیت 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 خطأ.
استفاده در سیستم‌های تصحیح خطأ در مخابرات و حافظه‌ها (RAM، 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; // 0 بیت 3 → بیت 3
data |= ((hammingCode >> 4) & 1) << 1; // 1 بیت 5 → بیت 5
data |= ((hammingCode >> 5) & 1) << 2; // 2 بیت 6 → بیت 6
data |= ((hammingCode >> 6) & 1) << 3; // 3 بیت 7 → بیت 7
data |= ((hammingCode >> 8) & 1) << 4; // 4 بیت 9 → بیت 9
data |= ((hammingCode >> 9) & 1) << 5; // 5 بیت 10 → بیت 10
data |= ((hammingCode >> 10) & 1) << 6; // 6 بیت 11 → بیت 11
data |= ((hammingCode >> 11) & 1) << 7; // 7 بیت 12 → بیت 12
data |= ((hammingCode >> 12) & 1) << 8; // 8 بیت 13 → بیت 13
data |= ((hammingCode >> 13) & 1) << 9; // 9 بیت 14 → بیت 14
```

عملکرد:

1. بیت متناظر در hammingCode را می‌خواند.
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 خطأ 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 خطأ.
- 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 ۶۶ 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 (A0 تا A5) می‌خواند. مقدار آنالوگ به عددی بین ۰ تا 10^{23} 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 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.

▪ مقدار تصادفی ۱ یا ۲ دریافت می‌کند. دلیل انتخاب این دو عدد برای دو بیت پرازش این است که اعداد

1 و 2 برای ورودی داده به کار می‌روند. (ونه بیت‌های کنترل یا خطای) به همین دلیل فرض شده که هنگام تولید داده‌های

66 بیتی، تنها بیت‌های داده را ورودی می‌گیریم.

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

۲.۱. تعریف آرایه‌ها برای بررسی خطاهای

```
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 guessedPermanent[8] = {false};

```

این آرایه‌ها وضعیت بیت‌های خطادار، تصحیح شده، و نسخه‌های رمزگذاری شده را نگه می‌دارند. در ادامه در مورد هر کدام از آن‌ها بیشتر توضیح می‌دهیم.

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 داده‌ها را به فرمت همینگ تبدیل کرده و آن‌ها را در encodedHamming و originalHamming ذخیره می‌کنیم 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 [i] encodedHamming ایجاد می‌شود.

۲.۴. بررسی مقدار خراب شده

```

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

```

با استفاده از تابع isFaulty، برنامه بررسی می‌کند که آیا مقدار دارای خطای خطا است یا خیر. توجه کنید که مکانیزم تشخیص خطا در این خط از همان انکوڈینگ ۸/۱۰ بیت استفاده می‌کند.

اصلاح خطاهای گذرا (Transient Errors)

حال که برنامه تعدادی خطای خطا را تشخیص داد، باید آن‌ها 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 خطاها 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 خطاها دائمی نیز دست نمی زنیم چون این خطاها باید باقی بمانند.

تابع زیر، با استفاده از تابع `transientErrors`، خطاها 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 خطاها مشکوک نمایش داده می شود. تعدادی از این خطاها دائمی بوده و در حال حاضر واقعاً مقداری اشتباه دارند و تعدادی از آن ها گذرا بوده و مقدار متاظر آن ها در مرحله قبلی اصلاح شده است.

۴.۲. بررسی و تصحیح خطاها دائمی (Permanent Errors)

در این بخش، مکانیزم تصحیح خطا پیاده سازی می شود. از آنجایی که انکودينگ 8/10 بیت یا 66/64 بیت اگرچه برای تشخیص خطا مفید هستند اما برای تصحیح خطا عملکرد مناسبی ندارند، از یک روش دیگر به نام کد همینگ استفاده می کیم. 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 خطأ، داده اصلاح شده را با داده اصلی مقایسه می کیم تا مطمئن شویم که در تبدیل این داده به کد همینگ اشتباہی رخ نداده است. سپس این خطای دائمی را چاپ می کنیم.

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

۱.۵. نمایش مقدار نهایی

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

ابتدا داده‌های ده بیتی به صورت باپنری چاپ می‌شوند.

۵.۲. ارسال داده‌های رمزگذاری شده 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 داده‌های انکود شده ده بیتی در LED‌ها نمایش داده شوند.

۵.۳. نشان دادن خطاهای تصحیح شده 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

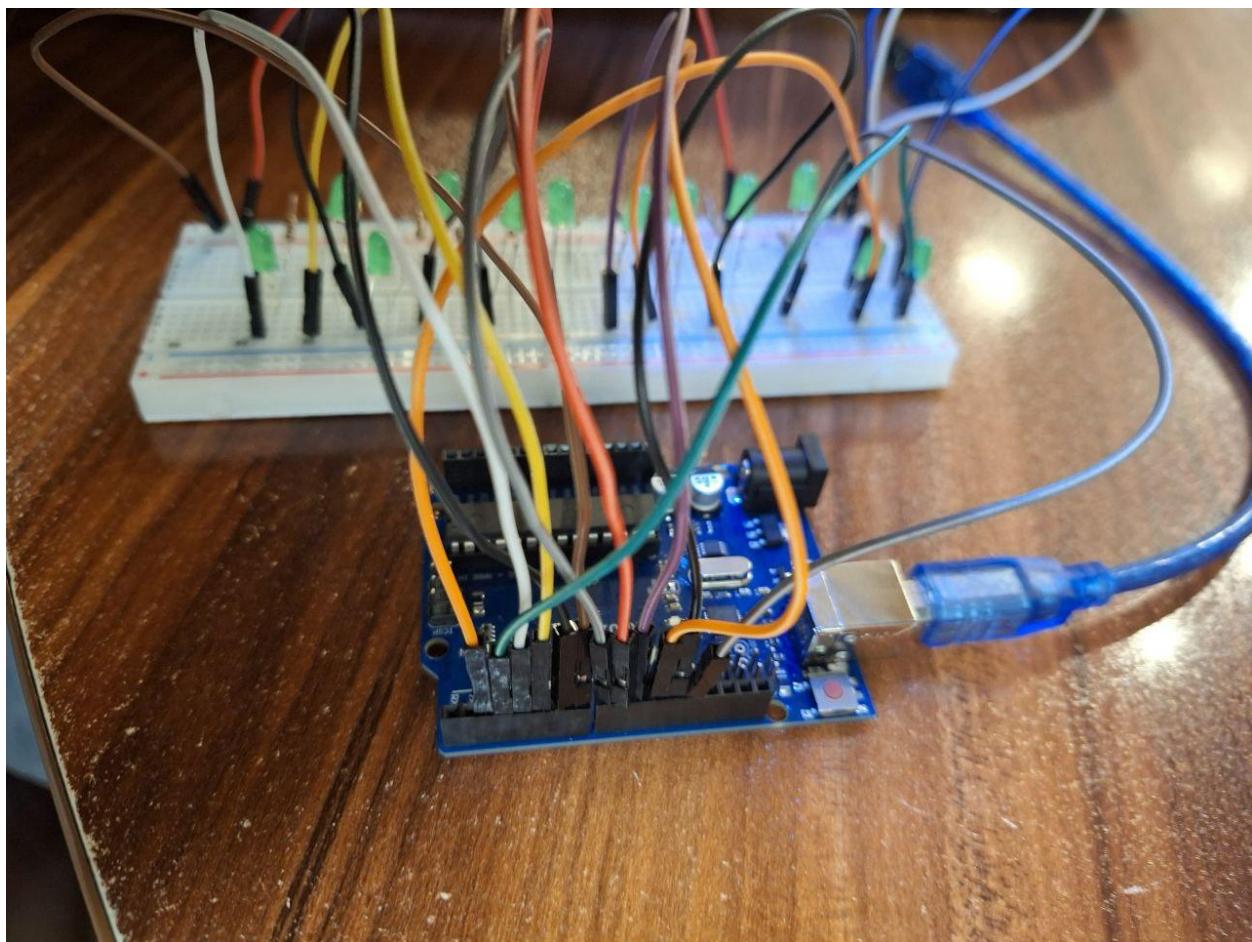
```
delay(300000);
```

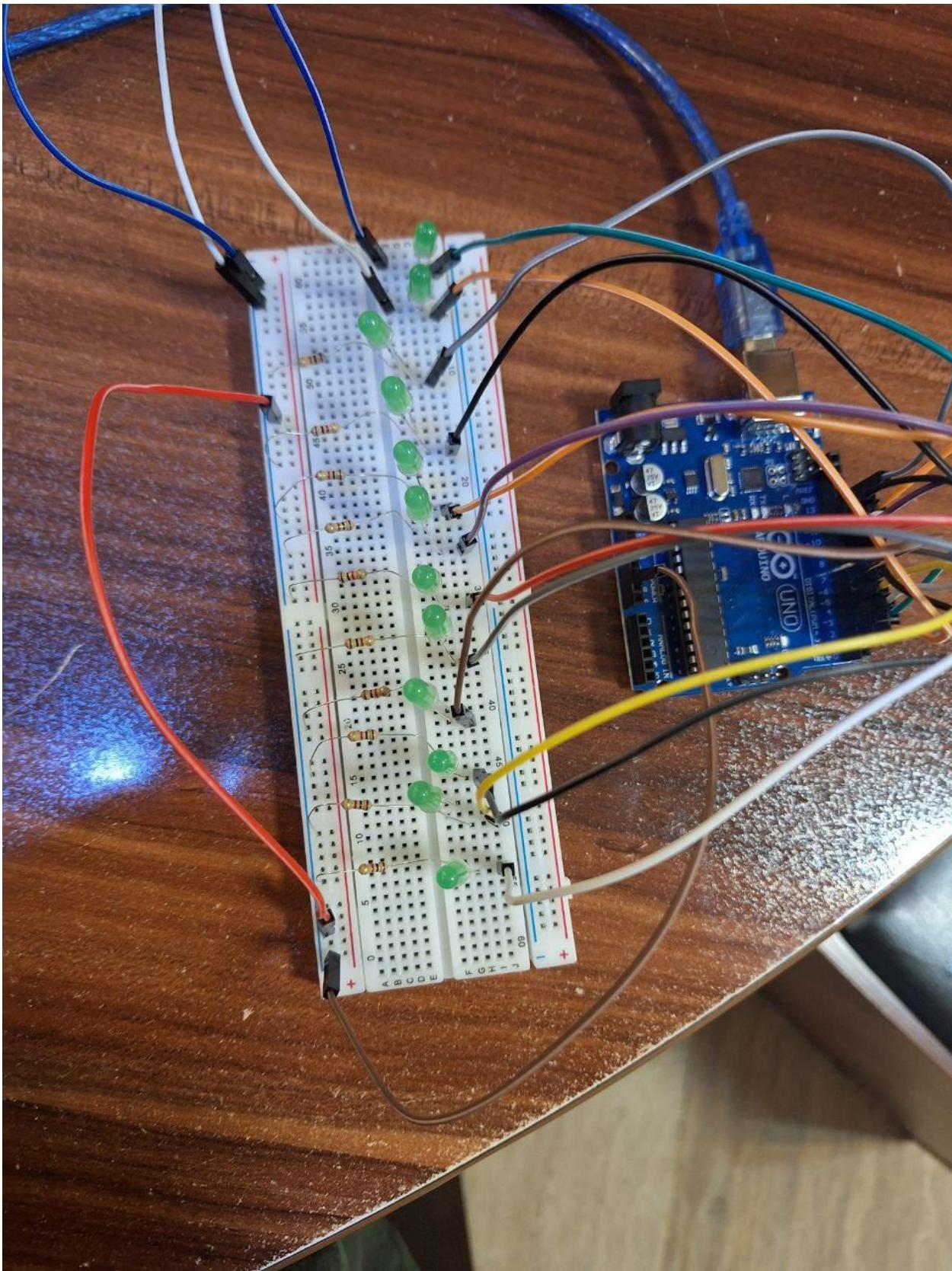
۵ دقیقه تأخیر برای اجرای مجدد (loop) به منظور شبیه‌سازی یک سیستم زمان‌بندی شده.

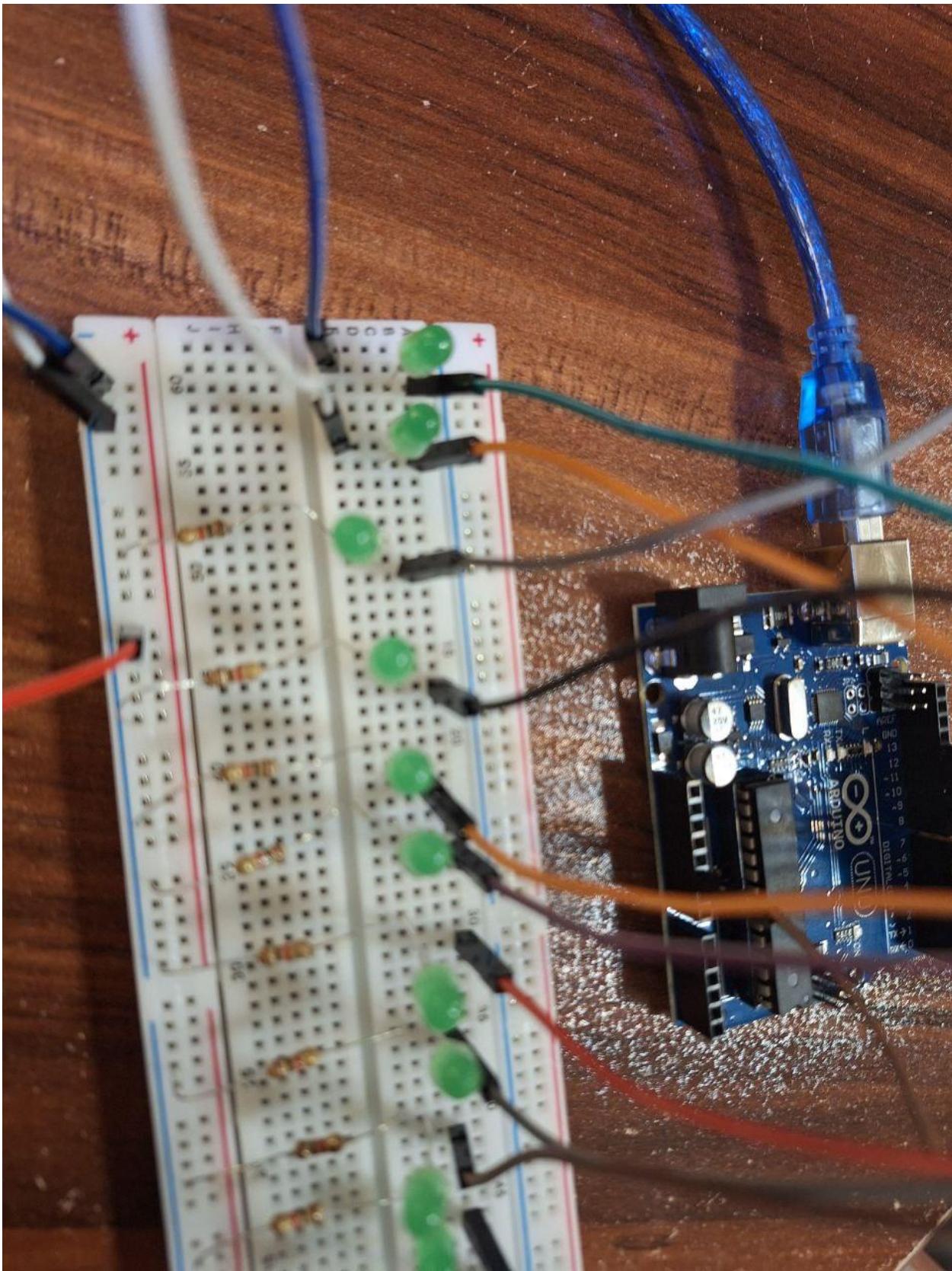
نتیجه گیری

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

تصاویر نمونه پیاده‌سازی عملی







مراجع

- [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