```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char add_parity_bit(char data[], int len) {
    int count = 0;
    for (int i = 0; i < len; i++) {
        if (data[i] == '1') {
            count++;
        }
    }
    char parity = (count % 2 == 0) ? '0' : '1';
    return parity;
}

int check_parity(char data[], int len, char received_parity) {
    char calculated_parity = add_parity_bit(data, len);
    return (calculated_parity == received_parity);
}

unsigned char calculate_checksum(char *data) {
    unsigned char checksum = 0;
    while (*data) {
        checksum += *data++;
    }
    return checksum;
}

int check_checksum(char *data, unsigned char received_checksum) {
```

```
    unsigned char calculated_checksum = calculate_checksum(data);
    return (calculated_checksum == received_checksum);
}

void xor(char *data, const char *generator, int pos) {
    for (int i = 0; i < strlen(generator); i++) {
        data[pos + i] = data[pos + i] == generator[i] ? '0' : '1';
    }
}

void crc_remainder(char *data, const char *generator) {
    int data_len = strlen(data);
    int generator_len = strlen(generator);

    for (int i = 0; i <= data_len - generator_len; i++) {
        if (data[i] == '1') {
            xor(data, generator, i);
        }
    }
}

int crc_check(char *data, const char *generator) {
    crc_remainder(data, generator);
    for (int i = strlen(data) - strlen(generator) + 1; i < strlen(data); i++) {
        if (data[i] == '1') {
            return 0;
        }
    }
    return 1;
```

```
}

void hamming_encode(char *data, char *encoded_data) {
    int n = strlen(data);
    int r = 0;

    while ((1 << r) < (n + r + 1)) {
        r++;
    }

    int j = 0, k = 0;
    for (int i = 1; i <= n + r; i++) {
        if (i == (1 << j)) {
            encoded_data[i - 1] = '0';
            j++;
        } else {
            encoded_data[i - 1] = data[k++];
        }
    }

    for (int i = 0; i < r; i++) {
        int x = 1 << i;
        int parity = 0;
        for (int j = x; j <= n + r; j++) {
            if (j & x) {
                parity ^= (encoded_data[j - 1] - '0');
            }
        }
        encoded_data[x - 1] = parity + '0';
```

```
    }

    encoded_data[n + r] = '\0';
}

void hamming_decode(char *encoded_data, char *decoded_data) {
    int n = strlen(encoded_data);
    int r = 0;
    while ((1 << r) < n) {
        r++;
    }

    int error_position = 0;
    for (int i = 0; i < r; i++) {
        int x = 1 << i;
        int parity = 0;
        for (int j = x; j <= n; j++) {
            if (j & x) {
                parity ^= (encoded_data[j - 1] - '0');
            }
        }
        if (parity != 0) {
            error_position += x;
        }
    }

    if (error_position) {
        encoded_data[error_position - 1] = encoded_data[error_position - 1] == '0' ? '1' : '0';
    }
```

```
    int j = 0;
    for (int i = 1; i <= n; i++) {
        if (i != (1 << j)) {
            *decoded_data++ = encoded_data[i - 1];
        } else {
            j++;
        }
    }
    *decoded_data = '\0';
}

int reed_solomon_encode(char *data, int data_len, unsigned char *encoded_data, int nsym) {
    memcpy(encoded_data, data, data_len);
    return data_len + nsym;
}

int reed_solomon_decode(unsigned char *encoded_data, int encoded_len, char *decoded_data, int
nsym) {
    memcpy(decoded_data, encoded_data, encoded_len - nsym);
    return encoded_len - nsym;
}

void display_options() {
    printf("Select Error Detection Algorithm:\n");
    printf("1. Parity Check\n");
    printf("2. Checksum\n");
    printf("3. CRC\n");
```

```
    printf("Select Error Correction Algorithm:\n");
    printf("1. Hamming Code\n");
    printf("2. Reed-Solomon Code\n");
}

void introduce_error(char *data) {
    int error_pos;
    printf("Enter position to introduce error (0 to %lu): ", strlen(data) - 1);
    scanf("%d", &error_pos);
    if (error_pos >= 0 && error_pos < strlen(data)) {
        data[error_pos] = (data[error_pos] == '0') ? '1' : '0';
    } else {
        printf("Invalid position!\n");
    }
}

int main() {
    int detection_choice, correction_choice;
    display_options();

    printf("Enter your choice for error detection (1-3): ");
    scanf("%d", &detection_choice);
    printf("Enter your choice for error correction (1-2): ");
    scanf("%d", &correction_choice);

    char detection_data[100], correction_data[100];
    printf("Enter data for error detection: ");
    scanf("%s", detection_data);
    strcpy(correction_data, detection_data);
```

```
int introduce_error_option;
printf("Do you want to introduce an error? (1 for Yes, 0 for No): ");
scanf("%d", &introduce_error_option);
int error_detected = 0;
if (detection_choice == 1) {

   int parity_len = strlen(detection_data);
   char parity = add_parity_bit(detection_data, parity_len);
   printf("Encoded Data with Parity Bit: %s%c\n", detection_data, parity);

   if (introduce_error_option) {
   introduce_error(detection_data);
    printf("Data with Error: %s\n", detection_data);
    }

   if (!check_parity(detection_data, parity_len, parity)) {
      printf("Parity Check: Failed\n");
      error_detected = 1;
   } else {
      printf("Parity Check: Passed\n");
   }
} else if (detection_choice == 2) {
   unsigned char checksum = calculate_checksum(detection_data);
   printf("Checksum: %d\n", checksum);

   if (introduce_error_option) {
   introduce_error(detection_data);
    printf("Data with Error: %s\n", detection_data);
    }
```

```
    if (!check_checksum(detection_data, checksum)) {
        printf("Checksum Check: Failed\n");
        error_detected = 1;
    } else {
        printf("Checksum Check: Passed\n");
    }
} else if (detection_choice == 3) {
    char crc_generator[] = "10011";
    char crc_encoded_data[100];
    strcpy(crc_encoded_data, detection_data);
    strcat(crc_encoded_data, "0000"); // Append zero bits
    crc_remainder(crc_encoded_data, crc_generator);
    printf("Encoded Data with CRC: %s\n", crc_encoded_data);

    if (introduce_error_option) {
    introduce_error(detection_data);
     printf("Data with Error: %s\n", detection_data);
     }
    if (!crc_check(crc_encoded_data, crc_generator)) {
        printf("CRC Check: Failed\n");
        error_detected = 1;
    } else {
        printf("CRC Check: Passed\n");
    }
}
 if (error_detected) {
    printf("Error detected! Applying correction technique...\n");
    if (correction_choice == 1) {
        char hamming_encoded_data[100];
```

```
        hamming_encode(correction_data, hamming_encoded_data);
        printf("Encoded Data with Hamming Code: %s\n", hamming_encoded_data);
        char hamming_decoded_data[100];
        hamming_decode(hamming_encoded_data, hamming_decoded_data);
        printf("Corrected Data: %s\n", hamming_decoded_data);
      } else if (correction_choice == 2) {
        int nsym = 10;
        unsigned char rs_encoded_data[255];
        int rs_encoded_length = reed_solomon_encode(correction_data, strlen(correction_data),
rs_encoded_data, nsym);
        printf("Encoded Data with Reed-Solomon Code: ");
        for (int i = 0; i < rs_encoded_length; i++) {
          printf("%02X ", rs_encoded_data[i]);
        }
        printf("\n");
        for (int i = 0; i < rs_encoded_length; i++) {
          printf("%02X ", rs_encoded_data[i]);
        }
        printf("\n");
        char rs_decoded_data[255];
        int rs_decoded_length = reed_solomon_decode(rs_encoded_data, rs_encoded_length,
rs_decoded_data, nsym);
        printf("Corrected Data: %.*s\n", rs_decoded_length, rs_decoded_data);
      }
  } else {
      printf("No error detected.\n");
  }
return 0;
}
```