9 Framing Protocol: Byte Stuffing

Implement a Program in C which demonstrates byte-stuffing framing technique, where sender reads data, applies byte stuffing to the frame and sends it to receiver. Assume appropriate character for flag byte and stuff byte.

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

#define FLAG_BYTE 0x7E  // Flag byte
#define STUFF_BYTE 0x7D // Stuff byte

void byteStuffing(char *data, int size) {
    for (int i = 0; i < size; i++) {
        if (data[i] == FLAG_BYTE || data[i] == STUFF_BYTE) {
            printf("%02X %02X ", STUFF_BYTE, data[i] ^ 0x20);
        } else {
            printf("%02X ", data[i]);
        }
    }
}

int main() {
    char data[100];
    printf("Enter data to send: ");
    fgets(data, sizeof(data), stdin);
    data[strcspn(data, "\n")] = 0; // Remove trailing newline

    // Apply byte stuffing
    printf("Frame after byte stuffing: ");
    byteStuffing(data, strlen(data));
    printf("\n");

    return 0;
}
```

```
Enter data to send: Hello FLAG
Frame after byte stuffing: 48 65 6C 6C 6F 7D 5E 4C 41 47
```

10. Framing Protocol: Bit Stuffing
Implement a Program in C which demonstrates bit-stuffing framing technique, where sender reads data, applies bit stuffing to the frame and sends it to receiver.

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

#define FLAG_BYTE "01111110" // Flag sequence
#define STUFF_BYTE "011111110" // Stuff sequence

void bitStuffing(char *data, int size) {
   printf("%s ", FLAG_BYTE);
   for (int i = 0; i < size; i++) {
      printf("%c", data[i]);
      if (data[i] == '1') printf("%s", STUFF_BYTE);
   }
   printf("%s\n", FLAG_BYTE);
}

int main() {
   char data[100];
   printf("Enter data to send: ");
   fgets(data, sizeof(data), stdin);
   data[strcspn(data, "\n")] = 0; // Remove trailing newline

   // Apply bit stuffing
   printf("Frame after bit stuffing: ");
   bitStuffing(data, strlen(data));

   return 0;
}
```

```
Enter data to send: 101101
Frame after bit stuffing: 01111110 1010111111101101 01111110
```

---

11. Error Detection: LRC and Checksum

a. Split the message stream into sub-frames and find checksum by performing XOR operation on frame bits.
b. Perform LRC detection method while sending/receiving message at sender & receiver side.

```
#include <stdio.h>

// Function to calculate checksum by performing XOR operation on frame bits
char calculateChecksum(char *frame, int size) {
    char checksum = 0;
    for (int i = 0; i < size; i++) {
        checksum ^= frame[i];
    }
    return checksum;
}

// Function to perform LRC detection method
char calculateLRC(char **frames, int numFrames, int frameSize) {
    char lrc = 0;
    for (int i = 0; i < frameSize; i++)
        for (int j = 0; j < numFrames; j++)
            lrc ^= frames[j][i];
    return lrc;
}

int main() {
    // Example data
    char frames[3][8] = {"11011010", "10110110", "01101100"};
    int numFrames = 3, frameSize = 8;

    // Calculate and print checksum for each frame
    printf("Checksums:\n");
    for (int i = 0; i < numFrames; i++)
        printf("Frame %d: %d\n", i + 1, calculateChecksum(frames[i], frameSize));

    // Calculate and print LRC
    printf("\nLRC: %d\n", calculateLRC(frames, numFrames, frameSize));

    return 0;
}
```

```
Checksums:
Frame 1: 0
Frame 2: 1
Frame 3: 1

LRC: 0
```

---

12. Error Detection: CRC
Implement a Program in GNU C which determines whether a given Divisor is valid for CRC. The Divisor is to be entered in binary format from K/B. The sender encapsulates this data in a frame and transmits it to the receiver. The receiver applies the validation logic for CRC divisor and accordingly displays appropriate validation message on screen. For sender/receiver communication use IPC mechanism FIFO/Named Pipes in Linux/Unix environment. Use Bit-Wise operators in C wherever applicable. Consider a simplistic frame structure which encapsulates only the divisor information. All other fields of frame are to be neglected. Use appropriate data-types for various variables/frame structure. Take appropriate frame size.

```c
#include <stdio.h>
#include <stdbool.h>

// Function to validate CRC divisor
bool validateCRC(char *data, int dataSize, char *divisor, int divisorSize) {
   for (int i = 0; i < dataSize - divisorSize + 1; i++)
      if (data[i] == '1')
         for (int j = 0; j < divisorSize; j++)
            data[i + j] ^= divisor[j];
   for (int i = 0; i < dataSize; i++)
      if (data[i] == '1')
         return false;
   return true;
}

int main() {
   char data[] = "101010101";
   char divisor[] = "1011";
   bool isValid = validateCRC(data, 9, divisor, 4);
   printf("CRC divisor is %s.\n", isValid ? "valid" : "not valid");
   return 0;
}
```

```
CRC divisor is valid.
```

---

13. Error Correction: Hamming Code
Implement a Program in GNU C which demonstrates the Hamming Code method.

```c
#include <stdio.h>

// Function to calculate the parity bit for a given position
int calculateParityBit(int data[], int dataSize, int position) {
    int parity = 0;
    for (int i = position - 1; i < dataSize; i += position * 2) {
        for (int j = 0; j < position && i + j < dataSize; j++) {
            parity ^= data[i + j];
        }
    }
    return parity;
}

// Function to generate Hamming Code
void generateHammingCode(int data[], int dataSize) {
    int parityPositions[32], parityIndex = 0;
    for (int i = 1; i <= dataSize; i *= 2) {
        parityPositions[parityIndex++] = i;
    }
    for (int i = 0; i < parityIndex; i++) {
        int position = parityPositions[i];
        data[position - 1] = calculateParityBit(data, dataSize, position);
    }
}

// Function to detect and correct errors in received Hamming Code
void detectAndCorrectErrors(int received[], int dataSize) {
    int errorPosition = 0;
    for (int i = 0; i < dataSize; i++) {
        if (received[i]) {
            errorPosition += (1 << i);
        }
    }
    if (errorPosition > 0) {
        printf("Error detected and corrected at position %d.\n", errorPosition);
        received[errorPosition - 1] ^= 1;
    } else {
        printf("No error detected.\n");
    }
}

int main() {
    int data[] = {1, 0, 1, 1}; // Example data
    int dataSize = sizeof(data) / sizeof(data[0]);

    // Generate Hamming Code
    generateHammingCode(data, dataSize);

    // Simulate error by flipping one bit
    data[2] ^= 1;

    // Detect and correct errors
    detectAndCorrectErrors(data, dataSize);

    // Print corrected data
    printf("Corrected data: ");
    for (int i = 0; i < dataSize; i++) {
        printf("%d ", data[i]);
    }
    printf("\n");

    return 0;
}
```

```
Error detected and corrected at position 3.
Corrected data: 1 0 0 1
```

---

14. Congestion control protocols: Leaky Bucket The leaky bucket is an algorithm used in packet switched computer networks and telecommunications networks. It can be used to check that data transmissions, in the form of packets, conform to defined limits on bandwidth and burstiness (a measure of the unevenness or variations in the traffic flow). It can also be used as a scheduling algorithm to determine the timing of transmissions that will comply with the limits set for the bandwidth and burstiness: see network scheduler. The leaky bucket algorithm is also used in leaky bucket counters, e.g. to detect when the average or peak rate of random or stochastic events or stochastic processes exceed defined limits.

```
#include <stdio.h>
#include <stdbool.h>

#define BUCKET_CAPACITY 10 // Bucket capacity
#define OUTPUT_RATE 2      // Output rate
```

```
// Function to simulate the leaky bucket algorithm
void leakyBucket(int packets[], int numPackets) {
    int bucketLevel = 0;
    for (int i = 0; i < numPackets; i++) {
        // Add packet to the bucket
        bucketLevel += packets[i];

        // If bucket overflows, remove excess packets
        if (bucketLevel > BUCKET_CAPACITY)
            bucketLevel = BUCKET_CAPACITY;

        // Transmit packets at the output rate
        if (bucketLevel >= OUTPUT_RATE) {
            printf("Transmitting %d packets\n", OUTPUT_RATE);
            bucketLevel -= OUTPUT_RATE;
        } else {
            printf("Bucket underflow, waiting for more packets\n");
        }
    }
}

int main() {
    // Example packets
    int packets[] = {3, 5, 2, 4, 1};
    int numPackets = sizeof(packets) / sizeof(packets[0]);

    // Simulate leaky bucket algorithm
    leakyBucket(packets, numPackets);

    return 0;
}
```

```
Transmitting 2 packets
Transmitting 2 packets
Bucket underflow, waiting for more packets
Transmitting 2 packets
Bucket underflow, waiting for more packets
```

---

15. Congestion control protocols: Token Bucket Unlike leaky bucket, token bucket allows saving, up to maximum size of bucket n. This means that bursts of up to n packets can be sent at once, giving faster response to sudden bursts of input. An important difference between two algorithms: token bucket throws away tokens when the bucket is full but never discards packetswhile leaky bucket discards packets when the bucket is full. Let token bucket capacity

be C (bits), token arrival rate ρ (bps), maximum output rate M (bps), and burst length S (s) – During burst length of S (s), tokens generated are ρS (bits), and output burst contains a maximum of C + ρS (bits) – Also output in a maximum burst of length S (s) is M · S (bits), thus C + ρS = MS or S = C M − ρ Token bucket still allows large bursts, even though the maximum burst length S can be regulated by careful selection of ρ and M. One way to reduce the peak rate is to put a leaky bucket of a larger rate (to avoid discarding packets) after the token bucket.

```
#include <stdio.h>

#define TOKEN_BUCKET_CAPACITY 10 // Token bucket capacity
#define TOKEN_ARRIVAL_RATE 2     // Token arrival rate
#define MAX_OUTPUT_RATE 5        // Maximum output rate
#define BURST_LENGTH 3           // Burst length

// Function to simulate the token bucket algorithm
void tokenBucket() {
    int tokens = 0;
    for (int i = 0; i < BURST_LENGTH; i++) {
        // Generate tokens
        tokens += TOKEN_ARRIVAL_RATE;
        if (tokens > TOKEN_BUCKET_CAPACITY)
            tokens = TOKEN_BUCKET_CAPACITY;

        // Calculate output burst
        int outputBurst = (tokens < MAX_OUTPUT_RATE) ? tokens : MAX_OUTPUT_RATE;
        tokens -= outputBurst;

        // Output burst
        printf("Output burst: %d bits\n", outputBurst);
    }
}

int main() {
    // Simulate token bucket algorithm
    tokenBucket();

    return 0;
}
```

```
Output burst: 5 bits
Output burst: 5 bits
Output burst: 4 bits
```