Report 2 - Communication Protocols

Pranav Mintri

December 2025

1 Universal Asynchronous Receiver/Transmitter:

1.1 Task 1:

Program (Both Arduinos):

```
String messageToTransmit="";
String messageToReceive="";
#include<SoftwareSerial.h>
SoftwareSerial Serial1(11, 10); //RX, TX
void setup() {
  Serial.begin(9600);
  Serial1.begin(9600);
}

void loop() {
  // Transmission:
  if(Serial.available()>0) {
    Serial.println("MintTree");
    char nextLetter=Serial.read();
    if(nextLetter == '\n') {
      Serial1.println(messageToTransmit);
      messageToTransmit="";
    }
    else {
      messageToTransmit+=nextLetter;
    }
  }
  // Retrieval:
  if(Serial1.available()>0) {
    Serial.println("SrinathPrabhu");
    char nextLetter=Serial1.read();
    if(nextLetter=='\n') {
      Serial.print(messageToReceive);
      messageToReceive="";
    }
    else {
      messageToReceive+=nextLetter;
    }
  }
  delay(100);
}
```

Explanation:

Pins 11 and 10 on the Arduino board were configured as the RX and TX pins for the new Serial communication line (as the existing RX and TX pins were already involved in communication with the computer). Both were set up to have a baud rate of 9600 Hz.

The logic in the transmission loop is that whenever the user is done writing, a newline character will get inserted, and on detecting that, the string `messageToTransmit` will get sent via UART to the second Arduino from the first.

The logic in the retrieval loop is that when the message is received, it will be a string of characters. Each of those characters will be read individually and combined into the string `messageToReceive`. When the newline character is detected, this string will get printed on the serial monitor.

1.2 Task 2:

Program (Board 1):

```
#include <SoftwareSerial.h>
SoftwareSerial Serial1(11, 10); // RX, TX

void setup() {
  Serial.begin(9600);
  Serial1.begin(9600);
}

void loop() {
  if (Serial.available() > 0) {
    String msg1 = Serial.readStringUntil('\n');
    int x = msg1.toInt();
    int y = x + 1;
    Serial1.println(y);          // Send as text with newline
  }

  if (Serial1.available() > 0) {
    String msg = Serial1.readStringUntil('\n');
    int z = msg.toInt();
    Serial.println(z);           // Print result
  }
}
```

Explanation:

The method `readStringUntil(char)` essentially does what we did in Task 1 - combining letters to form a string. The data inputted by the user is read as a string, which is then converted into an integer using the method `toInt()`. The number 1 is added to the number entered by the user, and transmitted to the second board.

When the second board returns some value through the Serial1 communication line, that number is again, first read character-wise, converted into a string, and then converted into an integer and displayed on the serial monitor.

Program (Board 2):

```
#include <SoftwareSerial.h>
SoftwareSerial Serial1(11, 10); // RX, TX

void setup() {
  Serial1.begin(9600);
}
```

```
void loop() {
  if (Serial1.available() > 0) {
    String msg2 = Serial1.readStringUntil('\n');
    int xc = msg2.toInt();
    int xc2 = xc * xc;
    Serial1.println(xc2);          // Send square back
  }
}
```

Explanation:

This program receives the number processed by Board 1 through the Serial1 communication line, squares it, and then sends it back to Board 1 through the Serial1 communication line.

One thing to note here is that the `println()` method differs from the `write()` method, in that the latter sends just one byte at a time by default, in the form of characters, whereas the former can send values in a human-readable format, multiple bytes at a time. `write()` can send multiple bytes, but the number of bytes needs to be specified inside the method, and the data needs to be in the form of an array of bytes. An integer, for example, would have to be broken down into its four constituent bytes (which we will have to do in Inter-Integrated Circuit and Serial Peripheral Interface later on).

2 Serial Peripheral Interface:

2.1 Task 1:

Program (Master):

```
#include<SPI.h>

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  SPI.begin(); // Start SPI as master.
  pinMode(SS, OUTPUT); // SS pin must be output.
  digitalWrite(SS, HIGH); // Deselect slave.
}

void loop() {
  // put your main code here, to run repeatedly:
  SPISettings myDeviceSettings(4000000, MSBFIRST, SPI_MODE0); // Define settings once
  SPI.beginTransaction(myDeviceSettings);
  digitalWrite(SS, LOW); // Select slave.
  String st = "ALL IS WELL";
  for(int i=0; i<st.length(); i++) {
    SPI.transfer(st[i]);
  }
  digitalWrite(SS, HIGH); // Deselect slave.
  SPI.endTransaction();
  delay(100);
}
```

Explanation:

The very first thing to be done is to set up SPI, and configure the board this code will be fed into, as the master. When the SS pin (Slave Select) is pulled HIGH, the node is deselected as the slave. (i.e. it becomes the master)

Now, the settings of the SPI communication protocol need to be specified. First, the frequency of the serial clock is defined. Here, it is 4 MHz. Then, `MSBFIRST` signifies the order in which bits will be sent in a byte, signifying that the most significant bit (MSB) will be sent first. `SPIMODE0` signifies the values of CPOL and CPHA. CPOL is clock polarity, and CPHA is clock phase. The following table will provide some clarity.

SPI Mode	CPOL	CPHA	Meaning (CPOL)	Meaning (CPHA)
0	0	0	The serial clock is low in the idle state.	Data is sampled along the first edge (rising edge).
1	0	1	The serial clock is low in the idle state.	Data is sampled along the second edge (falling edge).
2	1	0	The serial clock is high in the idle state.	Data is sampled along the first edge (falling edge).
3	1	1	The serial clock is high in the idle state.	Data is sampled along the second edge (rising edge).

We begin the transfer process by first starting the transaction with the settings we configured earlier. Further, in order to select the slave (since there is only one slave in this case), we pull the SS pin LOW. In order to transfer the string ”ALL IS WELL”, we transfer each character of the string, byte-by-byte. Then, we deselect the slave by pulling the SS pin HIGH, and end the transaction.

Program (Slave):

```
#include<SPI.h>
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  SPI.begin();
  SPCR |= _BV(SPE); // Enable SPI in slave mode
  // SPCR = SPI Control Register (controls the configuration of the SPI hardware)
  // SPE = SPI Enable bit inside SPCR (when set to 1, it turns on the SPI hardware,
  and when cleared (0), SPI is disabled)
  // _BV(x) = Bit Value
  // This statement sets the SPE bit to 1, without changing the other bits, and
  enables the SPI hardware.
  SPI.attachInterrupt(); // Enable SPI interrupt
}

ISR(SPI_STC_vect) {
  received = SPDR; // SPDR = SPI Data Register, this gets the received byte.
  Serial.write(received); // Prints character directly.
}

void loop() {
  // put your main code here, to run repeatedly:
}
```

Explanation:

The comments in the code provide a very detailed description of the workings of the code.

Essentially, when the slave receives data, it prints the data it receives character-by-character, on the serial monitor. This data is stored in the SPI Data Register (SPDR).

Nothing has to be put into the loop, because the slave only acts when it receives data from the master. (i.e. it does not do anything repeatedly)

2.2 Task 2:

Program (Master): (Note: This code is faulty. It needs to be corrected.)

```
#include<SPI.h>
int SS_A = 10, SS_B = 9;
SPISettings settings(4000000, MSBFIRST, SPI_MODE0);
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  SPI.begin();
  pinMode(SS_A, OUTPUT);
  pinMode(SS_B, OUTPUT);
  digitalWrite(SS_A, HIGH);
  digitalWrite(SS_B, HIGH);
}

void loop() {
  // put your main code here, to run repeatedly:
  if(Serial.available()>0) {
    String st = Serial.readStringUntil('\n');
    int32_t x = st.toInt();
    if(x%2==0) {
      SPI.beginTransaction(settings);
      digitalWrite(SS_A, LOW); // Selecting Slave 1
      byte b0 = (x >> 24) & 0xFF;
      byte b1 = (x >> 16) & 0xFF;
      byte b2 = (x >> 8) & 0xFF;
      byte b3 = x & 0xFF;
      byte r0 = SPI.transfer(b0);
      byte r1 = SPI.transfer(b1);
      byte r2 = SPI.transfer(b2);
      byte r3 = SPI.transfer(b3);
      digitalWrite(SS_A, HIGH);
      SPI.endTransaction();
      int32_t numberReceived = ((int32_t)r0 << 24) |
                     ((int32_t)r1 << 16) |
                     ((int32_t)r2 << 8)  |
                     (int32_t)r3;
      Serial.println(numberReceived);
    }
    else {
      SPI.beginTransaction(settings);
      digitalWrite(SS_B, LOW); // Selecting Slave 2
      byte b0 = (x >> 24) & 0xFF;
      byte b1 = (x >> 16) & 0xFF;
      byte b2 = (x >> 8) & 0xFF;
      byte b3 = x & 0xFF;
      byte r0 = SPI.transfer(b0);
      byte r1 = SPI.transfer(b1);
      byte r2 = SPI.transfer(b2);
      byte r3 = SPI.transfer(b3);
      digitalWrite(SS_B, HIGH);
      SPI.endTransaction();
      int32_t numberReceived = ((int32_t)r0 << 24) |
                      ((int32_t)r1 << 16) |
                      ((int32_t)r2 << 8)  |
                      (int32_t)r3;
      Serial.println(numberReceived);
    }
  }

}
```

Explanation:

Since there are two slaves in this setup, two separate slave select (SS) pins need to be configured, one for each slave. This is exactly what has been done in `void setup()`.

`void loop()` begins by checking whether the user has entered an even integer or not. int32_t is meant to imply a 32-bit (or a 4-byte) integer. If the user has entered an even integer, we begin the transfer of the integer via SPI, with `beginTransaction()`. Note that the settings were defined beforehand. Then, we select Slave 1 by pulling SS_A LOW.

```
byte b0 = (x >> 24) & 0xFF;
byte b1 = (x >> 16) & 0xFF;
byte b2 = (x >> 8) & 0xFF;
byte b3 = x & 0xFF;
```

The above piece of code is meant to split the entered integer into its four constituent bytes. b0 represents the most significant byte of the integer, whereas b3 represents the least significant byte of the integer.

```
byte r0 = SPI.transfer(b0);
byte r1 = SPI.transfer(b1);
byte r2 = SPI.transfer(b2);
byte r3 = SPI.transfer(b3);
```

The above piece of code is meant to transfer those bytes to Slave 1, while also receiving bytes in response (only when a master sends/requests data does a slave respond, in SPI). Note that these responses are not the bytes of the integer, but simply the last bytes present in the SPI Data Register. **This is the fault in the code. I had initially assumed (incorrectly) that r0, r1, r2 and r3**

were bytes of the integer sent back by the slave.

When a transaction is complete, the SS_A pin must be pulled HIGH. After that, the integer is reconstructed from r0, r1, r2 and r3, and printed on the serial monitor.

In case the number is odd, the process is the same - the only difference is that the second slave is selected, instead of the first.

Program (Slave 1):

```
#include<SPI.h>
byte integerReader[4];
int index=0;
int receivedNumber = 0;
byte replyBytes[4];
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  pinMode(MISO, OUTPUT);
  SPI.begin();
  SPCR |= _BV(SPE); // Enable SPI in slave mode
  // SPCR = SPI Control Register (controls the configuration of the SPI hardware)
  // SPE = SPI Enable bit inside SPCR (when set to 1, it turns on the SPI hardware,
  and when cleared (0), SPI is disabled)
  // _BV(x) = Bit Value
  // This statement sets the SPE bit to 1, without changing the other bits, anmd
  enables the SPI hardware.
  SPI.attachInterrupt();
}
ISR(SPI_STC_vect) {
  integerReader[index]=SPDR;
  SPDR = replyBytes[index % 4];
  index++;
  if(index==4) {
    receivedNumber = ((int32_t)integerReader[0] << 24) |
                     ((int32_t)integerReader[1] << 16) |
                     ((int32_t)integerReader[2] << 8)  |
                     (int32_t)integerReader[3];
    receivedNumber = receivedNumber - 1;
    index=0;
    replyBytes[0] = (receivedNumber >> 24) & 0xFF;
    replyBytes[1] = (receivedNumber >> 16) & 0xFF;
    replyBytes[2] = (receivedNumber >> 8)  & 0xFF;
    replyBytes[3] = receivedNumber & 0xFF;

    // Load the corresponding reply byte into SPDR
  }
}
void loop() {
  // put your main code here, to run repeatedly:

}
```

Explanation:

We shall directly move to the explanation of the ISR method, as the contents of `void setup()` and the absence of any content in `void loop()` has been explained in the explanation of the first task.

With every byte the slave receives from the master, ISR is activated, and data is taken from the SPI Data Register (SPDR) and fed into the array `replyBytes`. When four bytes are received (i.e. `index` = 4), the received number is operated on, and then loaded back into `replyBytes`.

The four bytes are converted into a 32-bit integer by:

```
receivedNumber = ((int32_t)integerReader[0] << 24) |
                     ((int32_t)integerReader[1] << 16) |
                     ((int32_t)integerReader[2] << 8)  |
                     (int32_t)integerReader[3];
```

`integerReader[0]` is the most significant byte, whereas `integerReader[3]` is the least significant byte.

`index` is set to 0. In the next iteration of ISR, elements of the array `replyBytes` are loaded into the SPDR, for the master to read.

Program (Slave 2):

```
#include<SPI.h>
byte integerReader[4];
int index=0;
int receivedNumber = 0;
byte replyBytes[4];
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  pinMode(MISO, OUTPUT);
  SPI.begin();
  SPCR |= _BV(SPE); // Enable SPI in slave mode
  // SPCR = SPI Control Register (controls the configuration of the SPI hardware)
  // SPE = SPI Enable bit inside SPCR (when set to 1, it turns on the SPI hardware,
  and when cleared (0), SPI is disabled)
  // _BV(x) = Bit Value
  // This statement sets the SPE bit to 1, without changing the other bits, anmd
  enables the SPI hardware.
  SPI.attachInterrupt();
}
ISR(SPI_STC_vect) {
  integerReader[index]=SPDR;
  SPDR = replyBytes[index % 4];
  index++;
  if(index==4) {
    receivedNumber = ((int32_t)integerReader[0] << 24) |
                     ((int32_t)integerReader[1] << 16) |
                     ((int32_t)integerReader[2] << 8)  |
                     (int32_t)integerReader[3];
    receivedNumber = receivedNumber + 1;
    index=0;
    replyBytes[0] = (receivedNumber >> 24) & 0xFF;
    replyBytes[1] = (receivedNumber >> 16) & 0xFF;
    replyBytes[2] = (receivedNumber >> 8)  & 0xFF;
    replyBytes[3] = receivedNumber & 0xFF;

    // Load the corresponding reply byte into SPDR
  }
}
void loop() {
  // put your main code here, to run repeatedly:

}
```

Explanation:

This code is extremely similar to that of the first slave, with the only difference being the operation carried out on `receivedNumber`.

3 Inter-Integrated Circuit:

3.1 Task 1:

Program (Master):

```
#include<Wire.h>
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  Wire.begin();
}

void loop() {
  // put your main code here, to run repeatedly:
  Wire.beginTransmission(4); // transmits to device #4
  Wire.write("ALL IS WELL");
  Wire.endTransmission();
}
```

Explanation:

The Wire library has been included for implementing Inter-Integrated Circuit.

`Wire.beginTransmission(int)` requires an argument - the address of the device to which data is to be transmitted. In this case, we are transmitting data to device 4.

`Wire.write()` can only contain either byte-sized arguments (`byte` or `char`), or other arguments with a specified fixed number of bytes. In this case, the string ”ALL IS WELL” contains 11 characters, and thus has a fixed number of bytes, which is implicitly specified as well. A random string (like a string taken as input from the user on a serial monitor) cannot be fed as an argument into the `Wire.write()` method.

`Wire.endTransmission()`, as the name suggests, is the method which is to be written when the transaction is over.

Program (Slave):

```
#include<Wire.h>
void setup() {
  // put your setup code here, to run once:
  Wire.begin(4);
  Wire.onReceive(receiveEvent);
  Serial.begin(9600);
}

void loop() {
  // put your main code here, to run repeatedly:
```

```
  delay(500);
}
void receiveEvent() {
  String st = "";
  while(Wire.available()>0) {
    char c = Wire.read();
    st = st+c;
  }
  Serial.println(st);
}
```

Explanation:

The slave works differently than the master. Let us examine this step-by-step.

In `void setup()`, we begin by specifying the address of the slave in `Wire.begin()`. This also serves as an indication to the device this code will be flashed into, that that device is a slave.

`Wire.onReceive()` is a method that is activated when a slave device receives something via I2C. Slave devices are activated only when they are requested for data, or when they are given data by the master. Therefore, in this case, on receiving some data from the master, the method `receiveEvent()` is invoked.

In `void receiveEvent()`, we are reading the string sent by the master character-by-character and then printing the complete string on the serial monitor.

Note that ideally, in `void receiveEvent(int)`, providing an integer argument (`int howMany` is the convention) is a good practice, as that integer argument is used to specify exactly how many bytes are to be read, leading to a safer and more structured process. Since this code is simple, we did not bother with that integer argument, but when the programs start getting more complicated (Task 2 and Task 3), we shall resort to this convention for easier debugging, if things happen to go wrong.

In `void loop()`, a delay has been given. This is not required and can be removed, as the loop stays idle anyway.

3.2 Task 2:

Program (Master):

```
#include<Wire.h>
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  Wire.begin();
}

void loop() {
  // put your main code here, to run repeatedly:
  Wire.requestFrom(4, 4); // requesting 4 bytes from device #4
  int32_t value1=0;
  byte b0A, b1A, b2A, b3A, b0B, b1B, b2B, b3B;
  if(Wire.available() >= 4) {
    b0A = Wire.read();
    b1A = Wire.read();
    b2A = Wire.read();
    b3A = Wire.read();
    value1 = ((int32_t)b0A << 24) |
                    ((int32_t)b1A << 16) |
                    ((int32_t)b2A << 8)  |
                    (int32_t)b3A;
    Wire.beginTransmission(6);
    Wire.write(b0A);
    Wire.write(b1A);
    Wire.write(b2A);
    Wire.write(b3A);
    Wire.endTransmission();
  }
  Wire.requestFrom(6, 4); // requesting 4 bytes from device #6
  int32_t value2=0;
  if(Wire.available() >= 4) {
    b0B = Wire.read();
    b1B = Wire.read();
    b2B = Wire.read();
    b3B = Wire.read();
    value2 = ((int32_t)b0B << 24) |
                    ((int32_t)b1B << 16) |
                    ((int32_t)b2B << 8)  |
                    (int32_t)b3B;
    Wire.beginTransmission(4);
    Wire.write(b0B);
    Wire.write(b1B);
    Wire.write(b2B);
    Wire.write(b3B);
    Wire.endTransmission();
  }
  int32_t x = value1 + value2;
  int32_t x2 = x*x;
  Serial.println(x2);
  delay(500);
}
```

Explanation:

First, in `void loop()`, as the comment states, we are requesting 4 bytes from device 4, using the method `Wire.requestFrom(int deviceNumber, int numberOfBytes)`. Only when a master requests data from a slave or sends data to it, does the slave respond. Thus, this command is necessary.

Moving further down - if the number of available bytes equals or exceeds four, we start reading those bytes. 4 bytes are converted into an integer. Post this, we transmit the integer to device 6 (broken into 4 bytes).

Once this transmission is completed, we request data from device 6, and transmit the integer to device 4 in the same fashion as it was done when the data was requested from device 4 and given to device 6.

This transmission of integers (from one slave to another) is therefore done, by relaying the message through the master. The master itself though, must also display the square of the sum of the integers provided by devices 4 and 6. This is done using standard Serial commands.

Program (Slave 1):

```
#include <Wire.h>

void setup() {
  Serial.begin(9600);
  Wire.begin(4);                 // join I2C bus as slave with address 4
  Wire.onRequest(requestEvent); // callback when master requests data
  Wire.onReceive(receiveEvent); // callback when master sends data
}

void loop() {
  delay(500);
}

// Called when master requests data
void requestEvent() {
  int32_t value = 1234; // example value to send
  byte b0 = (value >> 24) & 0xFF;
  byte b1 = (value >> 16) & 0xFF;
  byte b2 = (value >> 8) & 0xFF;
  byte b3 = value & 0xFF;
  Wire.write(b0);
  Wire.write(b1);
  Wire.write(b2);
  Wire.write(b3);
}

// Called when master sends data
void receiveEvent(int howMany) {
  if (howMany >= 4) {
    byte b0B = Wire.read();
    byte b1B = Wire.read();
    byte b2B = Wire.read();
    byte b3B = Wire.read();
    int32_t value2 = ((int32_t)b0B << 24) |
                     ((int32_t)b1B << 16) |
                     ((int32_t)b2B << 8)  |
                     (int32_t)b3B;
    Serial.println(value2); // print the integer received from master
  }
}
```

Explanation:

Here, the slave is both requested for data by the master, and receives data from the master. Hence, both methods - `void requestEvent()` and `void receiveEvent(int howMany)` - need to be defined.

The delay in the loop is not required and can be removed, as the program will never enter the loop.

In `void requestEvent()`, a 32-bit integral number `value` has been declared, whose value can be hard-coded. This integer is broken into its constituent bytes, and transferred via I2C to the master.

In `void receiveEvent()`, if the number of bytes to be transferred from the master to the slave (defined by `howMany`) is equal to or greater than four, the four bytes are read by the slave, and are converted into a 32-bit integer. That value is printed on the serial monitor. In other words, the integer sent by device 6 is displayed on the serial monitor of device 4.

Program (Slave 2):

```
#include <Wire.h>

void setup() {
  Serial.begin(9600);
  Wire.begin(6);                // join I2C bus as slave with address 6
  Wire.onRequest(requestEvent); // callback when master requests data
  Wire.onReceive(receiveEvent); // callback when master sends data
}

void loop() {
  delay(500);
}

// Called when master requests data
void requestEvent() {
  int32_t value = 1234; // example value to send
  byte b0 = (value >> 24) & 0xFF;
  byte b1 = (value >> 16) & 0xFF;
  byte b2 = (value >> 8) & 0xFF;
  byte b3 = value & 0xFF;
  Wire.write(b0);
  Wire.write(b1);
  Wire.write(b2);
  Wire.write(b3);
}

// Called when master sends data
void receiveEvent(int howMany) {
  if (howMany >= 4) {
    byte b0A = Wire.read();
    byte b1A = Wire.read();
    byte b2A = Wire.read();
    byte b3A = Wire.read();
    int32_t value1 = ((int32_t)b0A << 24) |
                     ((int32_t)b1A << 16) |
                     ((int32_t)b2A << 8)  |
                     (int32_t)b3A;
    Serial.println(value1); // print the integer received from master
  }
}
```

Explanation:

This program is extremely similar to that of Slave 1. The only difference is that the device addresses have been inverted.

Here, the value sent by device 4 will be displayed on the serial monitor of device 6.

3.3 Task 3:

Program (Master 1):

```
#include<Wire.h>
bool roleChangeRequested=false;
int32_t a, b;
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  Wire.begin();
  Wire.requestFrom(0x06, 4);
  a = 0, b = 0;
  byte b0A, b1A, b2A, b3A, b0B, b1B, b2B, b3B;
  if(Wire.available()>=4) {
    b0A = Wire.read();
    b1A = Wire.read();
    b2A = Wire.read();
    b3A = Wire.read();
    a = ((int32_t)b0A << 24) |
                ((int32_t)b1A << 16) |
                ((int32_t)b2A << 8)  |
                (int32_t)b3A;
  }
  Wire.requestFrom(0x07, 4);
  if(Wire.available()>=4) {
    b0B = Wire.read();
    b1B = Wire.read();
    b2B = Wire.read();
    b3B = Wire.read();
    b = ((int32_t)b0B << 24) |
                ((int32_t)b1B << 16) |
                ((int32_t)b2B << 8)  |
                (int32_t)b3B;
    Wire.beginTransmission(0x05);
    Wire.write('s');
    Wire.endTransmission();
    Wire.end();
    Wire.begin(0x04);
    Wire.onReceive(receiveEvent);
  }
}
void receiveEvent(int howMany) {
  if (Wire.available()) {
    byte control = Wire.read();
    if (control == 's') {   // example control byte
```

```
      roleChangeRequested = true;
    }
  }
}
void loop() {
  // put your main code here, to run repeatedly:
  if(roleChangeRequested==true) {
    while(digitalRead(SDA_PIN)==LOW || digitalRead(SCL_PIN) == LOW) {}
    Wire.end();
    Wire.begin();
    Wire.beginTransmission(0x00);
    int32_t aplusb = a+b;
    byte b0 = (aplusb >> 24) & 0xFF;
    byte b1 = (aplusb >> 16) & 0xFF;
    byte b2 = (aplusb >> 8) & 0xFF;
    byte b3 = aplusb & 0xFF;
    Wire.write(b0);
    Wire.write(b1);
    Wire.write(b2);
    Wire.write(b3);
    Wire.endTransmission();
  }
}
```

Explanation:

Master 1 starts as a master, and requests 4 bytes from 0x06, with the `requestFrom()` method. If there exist four or more bytes for the master to read, it will read those bytes, and convert four of them into a 32-bit integer.

Then, the master requests 4 bytes from 0x07, and converts them into an integer.

The master then transmits a control byte (in this case, a character 's') to 0x05, so that it can trigger the slave to convert itself into a master. Since 0x05 is going to become a master, this node (whose code has been given) needs to end its reign as a master, and convert into a slave. Hence, `Wire.end()` is used, and then `Wire.begin(0x04)` is used. (0x04 is this node's new slave address) Now, since it's a slave, it will be provided data by the new master (erstwhile 0x05) and get triggered to execute the method `void receiveEvent(int howMany)`.

Just like how this master sent a control byte 's' to 0x05 to initiate 0x05's role change from a slave to a master, 0x05, after becoming a master, will also send a control byte to the slave 0x04 (this node) to initiate its conversion into a master once again. When the device 0x04 receives this control byte, the variable `roleChangeRequested` is turned true. The value of `roleChangeRequested` changing will result in `void loop()` getting executed.

The role of this node, when it turns into a master again, will be to transmit the sum of a (the number which is hard coded into this node), and b (the number which is received from 0x07) to every node in the system (broadcasting). This function (broadcasting) is disabled by default, and it can be done by transmitting to the 0x00 address.

One important detail about `void loop()` to mention here is that when the variable `roleChangeRequested` is set to true, it does not necessarily mean that the data register is empty at that point of time, or that no slave has been selected. We must wait to ensure that that is done. This is why we have added the `while(digitalRead(SDA_PIN)==LOW || digitalRead(SCL_PIN)==LOW)` condition.

Program (Master 2):

```
#include<Wire.h>
bool roleChangeRequested=false;
void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  Wire.begin(0x05);
  Wire.onReceive(receiveEvent);
}
void receiveEvent(int howMany) {
  if (Wire.available()) {
    byte control = Wire.read();
    if (control == 's') {   // example control byte
      roleChangeRequested = true;
    }
  }
}
void loop() {
  // put your main code here, to run repeatedly:

  if(roleChangeRequested) {
    while(digitalRead(SDA_PIN)==LOW || digitalRead(SCL_PIN) == LOW) {}
    Wire.end();
    Wire.begin();
    Wire.requestFrom(0x06, 4);
    int32_t c = 0, d = 0;
    byte b0C, b1C, b2C, b3C, b0D, b1D, b2D, b3D;
    if(Wire.available()>=4) {
      b0C = Wire.read();
      b1C = Wire.read();
      b2C = Wire.read();
      b3C = Wire.read();
      c = ((int32_t)b0C << 24) |
                 ((int32_t)b1C << 16) |
                 ((int32_t)b2C << 8)  |
                  (int32_t)b3C;
    }
    Wire.requestFrom(0x07, 4);
    if(Wire.available()>=4) {
      b0D = Wire.read();
      b1D = Wire.read();
      b2D = Wire.read();
      b3D = Wire.read();
      d = ((int32_t)b0D << 24) |
                  ((int32_t)b1D << 16) |
                  ((int32_t)b2D << 8)  |
                  (int32_t)b3D;
    }
    int32_t cplusd = c+d;
    byte b0 = (cplusd >> 24) & 0xFF;
    byte b1 = (cplusd >> 16) & 0xFF;
    byte b2 = (cplusd >> 8) & 0xFF;
    byte b3 = cplusd & 0xFF;
    Wire.beginTransmission(0x00);
    Wire.write(b0);
    Wire.write(b1);
    Wire.write(b2);
    Wire.write(b3);
    Wire.endTransmission();
    roleChangeRequested=false;
    Wire.beginTransmission(0x04);
    Wire.write('s');
    Wire.endTransmission();
    Wire.end();
    Wire.begin(0x05);
  }

}
```

Explanation:

This node starts off as a slave, with the address 0x05. On receiving a control byte from Master 1 (0x04), `roleChangeRequested` turns true, and 0x05 becomes a master in `void loop()`.

Once again, we wait for the SDA (Serial Data) and SCL (Serial Clock) pins to be pulled high. Then, 0x05 becomes a master, and it requests four bytes from the slave 0x06. Those bytes are then read and converted into an integer c.

In a similar way, and integer d is read from 0x07 as well. c and d are added, and the sum is broadcasted to all devices in the network. Then, the node converts itself from a master to a slave, after sending a control byte to slave 0x04, prompting it to convert itself from a slave to a master.

Program (Slave 1): (Note: There is a slight change that needs to be made in this code.)

```
#include<Wire.h>
int32_t a = 0, c = 0, aplusb = 0, cplusd = 0, b = 0, d = 0;
bool goBack = false;

void setup() {
  Serial.begin(9600);
  Wire.begin(0x06);                  // Slave 1 address
  Wire.onReceive(receiveEvent);      // Register receive handler
  Wire.onRequest(requestEvent);      // Register request handler
}

void receiveEvent(int howMany) {
  if (howMany >= 8) { // expecting 8 bytes: aplusb and cplusd
    byte b0AB = Wire.read();
    byte b1AB = Wire.read();
    byte b2AB = Wire.read();
    byte b3AB = Wire.read();
    aplusb = ((int32_t)b0AB << 24) | ((int32_t)b1AB << 16) | ((int32_t)b2AB << 8) | (int32_t

    byte b0CD = Wire.read();
    byte b1CD = Wire.read();
    byte b2CD = Wire.read();
    byte b3CD = Wire.read();
    cplusd = ((int32_t)b0CD << 24) | ((int32_t)b1CD << 16) | ((int32_t)b2CD << 8) | (int32_t

    goBack = true;
  }
}

void requestEvent() {
  // Send a and c when master requests
  byte b0A = (a >> 24) & 0xFF;
  byte b1A = (a >> 16) & 0xFF;
  byte b2A = (a >> 8) & 0xFF;
  byte b3A = a & 0xFF;
  Wire.write(b0A); Wire.write(b1A); Wire.write(b2A); Wire.write(b3A);

  byte b0C = (c >> 24) & 0xFF;
  byte b1C = (c >> 16) & 0xFF;
  byte b2C = (c >> 8) & 0xFF;
  byte b3C = c & 0xFF;
  Wire.write(b0C); Wire.write(b1C); Wire.write(b2C); Wire.write(b3C);
}
```

```
void loop() {
  if (goBack) {
    b = aplusb - a;
    d = cplusd - c;
    Serial.print("a = "); Serial.print(a);
    Serial.print(", b = "); Serial.print(b);
    Serial.print(", c = "); Serial.print(c);
    Serial.print(", d = "); Serial.println(d);
    goBack = false;
  }
}
```

Explanation:

The slave 0x06, on receiving data from whoever the master is among Master 1 and Master 2 (Here, Master 1 (0x04) will be the master at first, and then Master 2 (0x05) will be the master later. Then, Master 1 will become the master once again.) will execute the method `void receiveEvent(int howMany)`.

It will receive both sums, first the sum of c and d from Master 2 (broadcasted by it), and then the sum of a and b from Master 1 (broadcasted by it).

This order is wrongly written in `void receiveEvent(int howMany)`.

Also, then, the boolean variable `goBack`, which was initally false, will be turned true, prompting `void loop()` to get executed. Since a and c were hard-coded into the slave's code, a, b, c and d can all be computed. They are computed, and printed on the serial monitor of the slave.

Also, Slave 1 sends a to Master 1, and c to Master 2, on being requested to send data.

Program (Slave 2): (Note: There is a slight change that needs to be made in this code.)

```
#include <Wire.h>

int32_t b = 0, d = 0, aplusb = 0, cplusd = 0, a = 0, c = 0;
bool goBack = false;

void setup() {
  Serial.begin(9600);
  Wire.begin(0x07);                 // Slave 2 address
  Wire.onReceive(receiveEvent);     // Register receive handler
  Wire.onRequest(requestEvent);     // Register request handler
}

void receiveEvent(int howMany) {
  if (howMany >= 8) { // expecting 8 bytes: aplusb and cplusd
```

```
    byte b0AB = Wire.read();
    byte b1AB = Wire.read();
    byte b2AB = Wire.read();
    byte b3AB = Wire.read();
    aplusb = ((int32_t)b0AB << 24) | ((int32_t)b1AB << 16) | ((int32_t)b2AB << 8) | (int32_t

    byte b0CD = Wire.read();
    byte b1CD = Wire.read();
    byte b2CD = Wire.read();
    byte b3CD = Wire.read();
    cplusd = ((int32_t)b0CD << 24) | ((int32_t)b1CD << 16) | ((int32_t)b2CD << 8) | (int32_t

    goBack = true;
  }
}

void requestEvent() {
  // Send b and d when master requests
  byte b0B = (b >> 24) & 0xFF;
  byte b1B = (b >> 16) & 0xFF;
  byte b2B = (b >> 8) & 0xFF;
  byte b3B = b & 0xFF;
  Wire.write(b0B); Wire.write(b1B); Wire.write(b2B); Wire.write(b3B);

  byte b0D = (d >> 24) & 0xFF;
  byte b1D = (d >> 16) & 0xFF;
  byte b2D = (d >> 8) & 0xFF;
  byte b3D = d & 0xFF;
  Wire.write(b0D); Wire.write(b1D); Wire.write(b2D); Wire.write(b3D);
}

void loop() {
  if (goBack) {
    a = aplusb - b;
    c = cplusd - d;
    Serial.print("a = "); Serial.print(a);
    Serial.print(", b = "); Serial.print(b);
    Serial.print(", c = "); Serial.print(c);
    Serial.print(", d = "); Serial.println(d);
    goBack = false;
  }
}
```

Explanation:

The slave 0x07, on receiving data from whoever the master is among Master 1 and Master 2 (Here, Master 1 (0x04) will be the master at first, and then Master

2 (0x05) will be the master later. Then, Master 1 will become the master once again.) will execute the method `void receiveEvent(int howMany)`.

It will receive both sums, first the sum of c and d from Master 2 (broadcasted by it), and then the sum of a and b from Master 1 (broadcasted by it).

This order is wrongly written in `void receiveEvent(int howMany)`.

Also, then, the boolean variable `goBack`, which was initally false, will be turned true, prompting `void loop()` to get executed. Since b and d were hard-coded into the slave's code, a, b, c and d can all be computed. They are computed, and printed on the serial monitor of the slave.

Also, Slave 2 sends b to Master 1, and d to Master 2, on being requested to send data.