CG1112

Engineering Principles and Practices II for CEG

Week 8 Studio 1

Communication Protocols

colinta[illegible]du.sg

NUS
National University
of Singapore

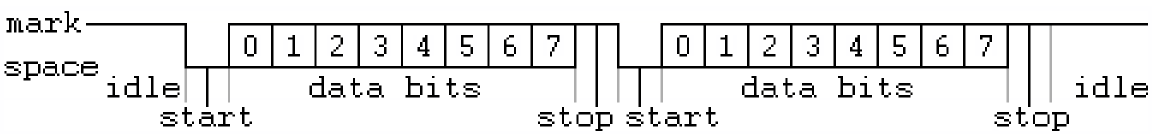
School of Computing

Introduction

- **In the previous studio you learnt how to establish communications with your Arduino using serial communications.**
- **In this lecture we will look at "communications protocols"**
  - A "protocol" is an agreement between machines on how to represent data.
- **In particular we will be looking at application layer protocols.**
  - In the previous studio we saw:
    - ✓**Physical layer protocols – Agreement on voltage levels, number of wires to use, how wires are to be connected, etc.**
    - ✓**Link layer protocols – How bits are placed together into units called "frames", what the layout of each frame should be, etc.**
  - Application layer protocols:
    - ✓**How we arrange instructions and data within a data structure for controlling behaviors of remote systems.**

Three things to do:

- **Decide on physical connection:**
  - Easy enough; we will connect the Arduino to the Pi using USB.
- **Decide on bit-level protocol:**
  - Decide baud rate.
  - Decide data length.
  - Decide # of parity bits.
  - Decide # of stop bits.
  - We will use 9600 8N1 as standard..



Serial Communication and Protocol Design

BUILDING A PROTOCOL

Assign an ID to each device

- **You need to be able to identify sensors (actuators) to read from (send data to).**

Device ID	Device
0	Sonar 1
1	Sonar 2
2	Touch Sensor 1
3	Touch Sensor 2
4	Buzzer
5	Tactile feedback motor
...	...

Create Packet Types

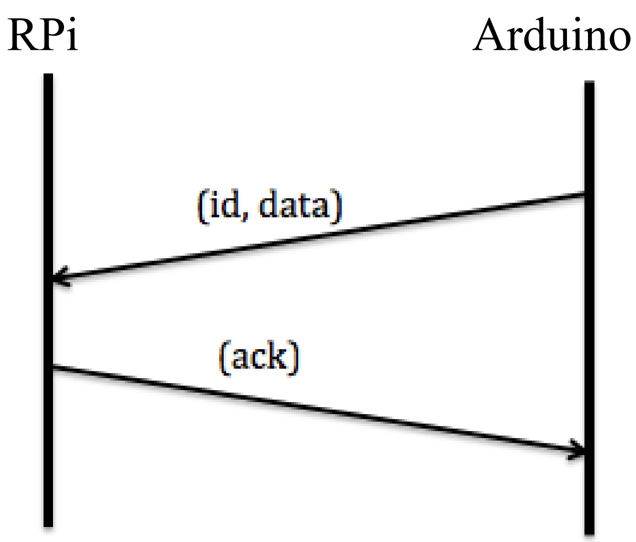
- **So both sides know what sort of packets are being sent (and the appropriate response)**

Packet Type	Packet Code
ACK	0
NAK	1
Hello	2
Read	3
Write	4
Data Response	5
...	...

Getting Data from the Arduino

- **Your Arduino may have data to send back to the Raspberry Pi (RPi):**
  - Compass readings.
  - Number of turns the wheels have made.
  - Etc?
- **You have two choices:**
  - Arduino can periodically send back data in "heartbeat packets".
    - ✓**These are packets sent back say once per second, containing data and status information.**
  - RPi can poll for data.
    - ✓**RPi sends a request for data.**
    - ✓**Arduino replies.**

Periodic Push By Arduino

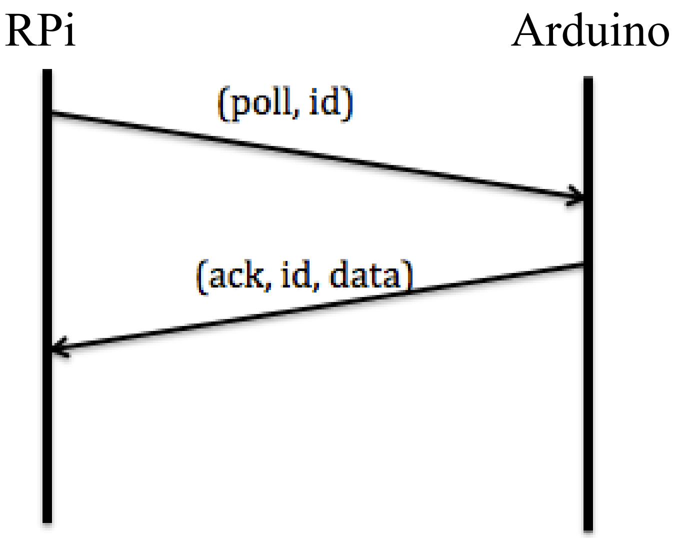


- **Arduino sends data whenever it is available.**
- **Often implemented as a "heartbeat" packet.**
- **RPi monitors and buffers data as it comes in.**

  +Arduino sends data whenever it is available.

  -RPi needs to buffer incoming data.

Periodic Poll by RPi



- **Arduino waits for poll packets from RPi.**
- **RPi requests data when it needs it.**

  +RPi decides when it needs the data and sends poll packet.

  -If RPi doesn't poll often enough, may lose data on Arduino (Arduino has only 2K of RAM)

Commanding the Arduino

- **You can use a variation of the "poll" packet to send commands to the Arduino:**
  - E.g. tell Arduino to move Vincent forward by 1 meter at 50% speed.
- **The Arduino responds immediately with an ACK packet.**
- **The RPi can either:**
  - Poll the Arduino continuously to check when it has finished the command, OR
  - The Arduino can send back a packet to inform the RPi that it is done.

Finding Checksums

- **Checksums are used to check that data is received correctly.**
- **Sender side:**
  - Compute checksum

    `checksum = b1 XOR b2 XOR b3 XOR b4 XOR…`
  - Attach to end of packet.
- **Receiver side:**
  - Compute checksum using data in packet, except checksum.
  - Compare against attached checksum.
  - If equal, reply with ACK, else reply with NAK.

Finding Checksums

- **For the most part:**
  - Serial comms is reliable.
  - Hence we don’t normally compute checksums (or send ACK for that matter).
- **However:**
  - Your set up is not going to be perfect. (headers and pins! C’mon)
  - If you are sending relatively large amounts of data, higher chance of errors.

Serializing Structures

- **The easiest way to implement the protocol packets is as structures.**
- **Example for our command packet:**

```
typedef tc
{
        int command;
        int speed;
        int distanceInMeters;
} TCommand;
```

- **Serializing: Converting a structure into a stream of bytes, because serial devices can only deal with streams of bytes.**
  - Get a pointer to the structure.
  - Copy into an array of char.
  - May want to include information on packet length and checksum.

Serializing Structures

```
typedef struct con
{
  unsigned char devCode;
  double maxValue;
  double minValue;

} TConfigPacket;

void sendConfig()
{
    TConfigPacket cfg;
    char buffer[64];
    cfg.devCode=deviceCode;
    cfg.minValue = minValue;
    cfg.maxValue = maxValue;
    unsigned len = serialize(buffer, &cfg, sizeof(cfg));
}
```

Serializing Structures

```
unsigned int serialize(char *buf, void *p, size_t size)
{
  char checksum = 0;
  buf[0]=size;
  memcpy(buf+1, p, size);
  for(int i=1; i<=size; i++)
  {
    checksum ^= buf[i];
  }
  buf[size+1]=checksum;
  return size+2;
}

void sendSerialData(char *buffer, int len)
{
  for(int i=0; i<len; i++)
    Serial1.write(buffer[i]);
}
```

Deserializing Structures

- **Deserialize: Convert a stream of bytes back to structures.**
  - Get a pointer to the structure.
  - Copy buffer of bytes to that pointer:
    - ✓ **May need to remove packet length and compute checksums first.**

Deserializing Structures

```
Void readConfig()
{
    char buffer[MAX_BUF_LEN];
    int len;
    TConfigPacket cfg;

    readSerial(buffer, &len);
    deserialize(&cfg, buffer);

    // Process cfg.
        …
}
```

Deserializing Structures

```
unsigned int deserialize(void *p, char *buf)
{
  size_t size = buf[0];
  char checksum = 0;

  for(int i=1; i<=size; i++)
    checksum ^= buf[i];

  if(checksum == buf[size+1])
  {
    memcpy(p, buf+1, size);
    return PACKET_OK;
  }
  else
  {
    printf("CHECKSUM ERROR\n");
    return PACKET_BAD_CHECKSUM;
  }
}
```

Serializing / Deserializing Structures

- **Our serializing / deserializing algorithms may not work in the real world because of two complications:**
  - Endianness
  - Differing Data Types
- **In addition:**
  - The representation shown earlier is not efficient. We would use a separate data structure to store the data to be serialized, the checksum and the size of the data.
  - See serialize.zip for an example.

Complication #1
Endianness

How to store multibyte word (object > 1 byte), 2 general schemes, Eg: 0x87654321 (4 byte int)

- **Big Endian**: higher order bytes at lower addresses

	87	65	43	21
Address	a	a+1	a+2	a+3

- **Little Endian**: lower order bytes at lower addresses

	21	43	65	87
Address	a	a+1	a+2	a+3

x86: little endian ; Sparc: big endian ; powerpc: configurable

Complication #1
Endianness

- **As it turns out:**
  - The Arduino is little endian.
  - The Pi is little endian.
- **Our job here is done. ;)**
- **Note though:**
  - If your destination machine has a different endianness, our algorithm will not work.
  - Serializing / Deserializing will be considerably more troublesome.
    - ✓ **Convert all data to a standard endianness.**
    - ✓ **At the destination, convert it back to the native endianness.**
  - In such cases consider using established libraries like MAVLINK and protobuf.

NUS National University of Singapore

School of Computing

Complication #2
Different Data Sizes

- **We want to send over a data structure consisting of:**
  - A single character, initialized to ‘a’.
  - An integer, initialized to 1.
- **As it turns out, the following code segments do not work. The sample printout on the Pi is shown on the next page:**

Arduino Side

```
void setup() {
  // put your setup code here, to run once:

  Serial.begin(115200);
}

typedef struct tm
{
  int x;
  char c;
} TTest;

void loop() {
  TTest data;

  data.x=1;
  data.c='a';

  // put your main code here, to run repeatedly:

  Serial.write((char *) &data, sizeof(TTest));
  delay(500);

}
```

RPi Side

```
#include <stdio.h>
#include <termios.h>
#include <string.h>
#include <stdint.h>
#include "serial.h"

typedef struct tm
{
  int x;
  char c;
} TTest;

int main()
{
  char buffer[MAX_BUFFER_LEN];
  int n;

  memset(buffer, 0, MAX_BUFFER_LEN);

  startSerial("/dev/ttyACM0", B115200, 8, 'N', 1, 5);
  while(1)
  {
    n = serialRead(buffer);
    TTest *data = (TTest *) buffer;
    printf("Read %d bytes: x is %d, c is %c\n", n, data->x, data->c);
  }
}
```

NUS National University of Singapore School of Computing

Complication #2 Different Data Sizes

Intended Output (RPi)

```
ATTEMPTING TO CONNECT TO SERIAL. ATTEMPT # 1 of 5.
Read 3 bytes: x is 1, c is a
Read 3 bytes: x is 1, c is a
Read 3 bytes: x is 1, c is a
Read 3 bytes: x is 1, c is a
Read 3 bytes: x is 1, c is a
Read 3 bytes: x is 1, c is a
Read 3 bytes: x is 1, c is a
Read 3 bytes: x is 1, c is a
Read 3 bytes: x is 1, c is a
Read 3 bytes: x is 1, c is a
Read 3 bytes: x is 1, c is a
Read 3 bytes: x is 1, c is a
Read 3 bytes: x is 1, c is a
```

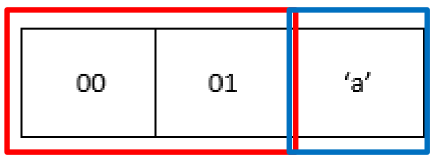
Actual Output (RPi)

```
ATTEMPTING TO CONNECT TO SERIAL. ATTEMPT # 1 of 5.
Read 6 bytes: x is 23134209, c is
Read 3 bytes: x is 23134209, c is
Read 3 bytes: x is 23134209, c is
Read 3 bytes: x is 23134209, c is
Read 3 bytes: x is 23134209, c is
Read 3 bytes: x is 23134209, c is
Read 3 bytes: x is 23134209, c is
Read 3 bytes: x is 23134209, c is
Read 3 bytes: x is 23134209, c is
Read 3 bytes: x is 23134209, c is
Read 3 bytes: x is 23134209, c is
Read 3 bytes: x is 23134209, c is
```

- **We can see that the output is obviously wrong. BUT WHY?**

Complication #2
Different Data Sizes

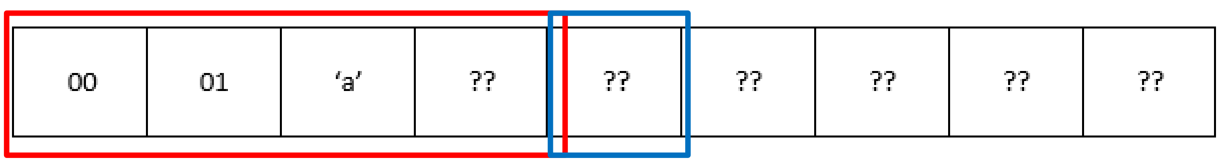
- **Turns out:**
  - Ints on the Arduino are 16 bits wide, but ints on the RPi are 32 bits wide!
  - The diagram below shows the consequences:



What the Arduino sent

Our integer x=1

Our character c='a'



What the Pi THINKS the Arduino sent

Where the Pi thinks x is. We end up with garbage

Where the Pi thinks c is.

Complication #2
Different Data Sizes

- **Fortunately there are two solutions:**
  - Switch to Arduino Due, which has a 32-bit ARM architecture like the Pi.
    - ✓**However the hardware registers and how we access them is completely different.**
    - ✓**Despair, fail CG1112, drop out and spend the rest of our lives scraping dead animals off the streets.**
  - Use standardized integer types
  - Replace:
    - *–int with int32_t*
    - *–unsigned int with uint32_t*
    - *–long with int64_t*
    - *–unsigned long with uint64_t*
    - ✓**Must remember to #include <stdint.h>**
  - Floats are not affected.
    - ✓**Both the Arduino and Pi use 32-bit IEEE 754 format in little endian.**

NUS National University of Singapore School of Computing

Complication #2
Different Data Sizes

- **Our corrected code (with additional integer y and a floating point z thrown in!):**

Arduino Side

```
void setup() {
  // put your setup code here, to run once:

  Serial.begin(115200);
}

typedef struct tm
{
  int32_t x;
  int32_t y;
  float z;
  char c;
} TTest;

void loop() {
  TTest data;

  data.x=1;
  data.y=2;
  data.z = 3.141592654;
  data.c='a';

  // put your main code here, to run repeatedly:

  Serial.write((char *) &data, sizeof(TTest));
  delay(500);

}
```

RPi Side

```
#include <stdio.h>
#include <termios.h>
#include <string.h>
#include <stdint.h>
#include "serial.h"

typedef struct tm
{
  int32_t x;
  int32_t y;
  float z;
  char c;
} TTest;

int main()
{
  char buffer[MAX_BUFFER_LEN];
  int n;

  memset(buffer, 0, MAX_BUFFER_LEN);

  startSerial("/dev/ttyACM0", B115200, 8, 'N', 1, 5);
  while(1)
  {
    n = serialRead(buffer);
    TTest *data = (TTest *) buffer;
    printf("Read %d bytes: x is %d, y is %d, z is %f, c is %c\n",
        n, data->x, data->y, data->z, data->c);
  }
}
```

NUS National University of Singapore
School of Computing

Complication #2
Different Data Sizes

Output (RPi side)

```
ATTEMPTING TO CONNECT TO SERIAL. ATTEMPT # 1 of 5.
Read 42 bytes: x is 1, y is 2, z is 3.141593, c is a
Read 13 bytes: x is 1, y is 2, z is 3.141593, c is a
Read 13 bytes: x is 1, y is 2, z is 3.141593, c is a
Read 13 bytes: x is 1, y is 2, z is 3.141593, c is a
Read 13 bytes: x is 1, y is 2, z is 3.141593, c is a
Read 13 bytes: x is 1, y is 2, z is 3.141593, c is a
Read 13 bytes: x is 1, y is 2, z is 3.141593, c is a
Read 13 bytes: x is 1, y is 2, z is 3.141593, c is a
Read 13 bytes: x is 1, y is 2, z is 3.141593, c is a
Read 13 bytes: x is 1, y is 2, z is 3.141593, c is a
Read 13 bytes: x is 1, y is 2, z is 3.141593, c is a
Read 13 bytes: x is 1, y is 2, z is 3.141593, c is a
```

Bonus Complication

- **One bonus complication:**
  - ▪The serial receive doesn't guarantee that you will receive the entire packet at one time.
  - ▪You may receive fragments, which can cause errors as shown below:

```
Read 13 bytes: x is 1, y is 2, z is 3.141593, c is a
Read 13 bytes: x is 1, y is 2, z is 3.141593, c is a
Read 1 bytes: x is 1, y is 2, z is 3.141593, c is a
Read 12 bytes: x is 33554432, y is -620756992, z is 22168995540103331840.000000, c is a
Read 3 bytes: x is 33554433, y is -620756992, z is 22168995540103331840.000000, c is a
Read 10 bytes: x is 512, y is 1225775872, z is 22179890394827640128.000000, c is a
```

  - ▪Solution:
    - ✓**Test number of bytes received. If it is == size of data structure, accept. Otherwise buffer and add in subsequent bytes that arrive.**

Bonus Complication

- **We do two things:**
  - We pad the TTest structure so that its size in bytes is divisible by 4
  - This is to match what the compiler does on the Pi side:
    - ✓ **We currently have 2 int32_t, 1 float and 1 char.**
    - ✓ **This totals 4 + 4 + 4 + 1 = 13 bytes.**
    - ✓ **We add in a dummy of 3 bytes:**

```
typedef struct tm
{
  int32_t x;
  int32_t y;
  float z;
  char c;
  char dummy[3];
} TTest;
```

Bonus Complication

- **Now we write the code to re-assemble the packet fragments:**

```
typedef enum
{
  PACKET_OK = 0,
  PACKET_INCOMPLETE = 1,
  PACKET_ERROR = 2

} TResult;

TResult assemble(char *packetBuffer, const char *dataBuffer, int len, size_t dataLen)
{
  static int count=0;

  if(count + len > dataLen)
  {
    count=0;
    return PACKET_ERROR;
  }
  else
  {
    int i;
    for(i=0; i<len; i++)
      packetBuffer[count++]=dataBuffer[i];
  }

  if(count == dataLen)
  {
    count=0;
    return PACKET_OK;
  }
  else
    return PACKET_INCOMPLETE;

}
```

Bonus Complication

- **Then we change our main to call assemble:**

```
int main()
{
  char buffer[MAX_BUFFER_LEN];
  char dataBuffer[MAX_BUFFER_LEN];
  int n;

  memset(buffer, 0, MAX_BUFFER_LEN);

  startSerial("/dev/ttyACM0", B115200, 8, 'N', 1, 5);
  while(1)
  {
    n = serialRead(buffer);

    TResult result = assemble(dataBuffer, buffer, n, sizeof(TTest));

    if(result == PACKET_OK)
    {
      TTest *data = (TTest *) dataBuffer;
      printf("Read %d bytes: x is %d, y is %d, z is %f, c is %c\n",
          n, data->x, data->y, data->z, data->c);
    }
    else
      if(result == PACKET_ERROR)
        printf("\n\t*** PACKET ERROR ***\n\n");
      else
        printf("Incomplete packet. Expecting %d more bytes.\n", sizeof(TTest) - n);

  }
}
```

Full Serialization / Deserialization Code

- **Please see the serialize.zip file for the full serialization/deserialization code.**
- **As this is a rather complicated topic:**
  - Please use the code in this file for your studios.
  - However you MUST understand how it works.
    - ✓**In particular we use the Tcomms data structure to store the data to be sent, the length of the data, and a checksum.**
    - ✓**We also add a magic number. When we receive a packet, we check the magic number to ensure that this is valid packet.**