# UDP, Data formats, Buffers, and Strings

Control systems and Computer Networks

Dr Alun Moon

Lecture 7.2

# UDP reception

- Send and Recieve functions are *blocking*
- Need to run in a *Thread* concurrently with other actions
- `buffer` is written into
- pointer to `SocketAddress`, for sending port and ip address

```
while(1){
    char buffer[1024]; /* 1k bytes */
    SocketAddress source;
    int len = udp.recvfrom( &source,
                            buffer, sizeof(buffer));
    buffer[len]='\0';
```

# Common Data Format

- Line oriented
- Key:Value pairs

- Email headers RFC822 `https://tools.ietf.org/html/rfc822`
- HTTP headers RFC7230 `https://tools.ietf.org/html/rfc7230`

```
Date: 26/02/2018
Pot1: 0.267
AccX: -0.01
```

Two tasks:

1. Split buffer into lines
2. Split line into *Key*:*Value* pairs

# UDP Data

## Array of bytes

- UDP Datgrams hold the data as an array of bytes

```java
byte[] payload;
DatagramPacket packet( payload, payload.length );
```

- Converting Bytes to Strings

```java
String message = new String( packet.getData() );
```

- Converting Strings to bytes

```java
packet.setData( message.getBytes() );
```

# Splitting the input

Java Strings have all you need for handling the message data,

- `split` returns an array of strings

  ```
  String[] lines = message.split("\n");
  ```

- `trim` gets rid of whitespace at the beginning and end of a string

  ```
  String clean = message.trim();
  String lines[] = message.trim().split("\n");
  ```

- arrays of lines can be iterated over

  ```
  for( String line : lines ) {...}
  ```

# Handling lines

- lines can be split on a colon delimeter

```
String[] pair = line.split(":");
```

- there should be 2 elements in the array, the first is the key.

```
String key = pair[0];
```

- the second is the value.

```
String value = pair[1];
```

- I use a hashtable to store the key-value pairs

# Java – String methods

```
DatagramPacket msg = new DatagramPacket(buffer, buffer.length)
socket.receive(msg);
String message = new String(msg.getData());
String[] lines = message.trim().split("\n");
for(String l : lines ) {
    if( l.length()>0 ) {
        String[] pair = l.trim().split(":");
        if(pair.length==2) datatable.put(pair[0], pair[1]);
    }
}
```

# C String handling is not quite so neat

A Quick review of strings in C:

- Strings are arrays of `char`

  ```
  char string[80];
  ```

- Pointers to `char` are also strings

  ```
  char *string;
  ```

- Strings are stored as ASCII values
  with a terminating byte value of zero

  ```
  "hello" === {104,101,108,108,111,0}
  ```

- Remember No Bounds Checks on Arrays

# UDP Datagrams

The MBED library just sends a block of data bytes/char

```
UDPSocket udp;
char *buffer;
int data_size;
udp.sendto( server, buffer, data_size );
```

How to find size of data? Numer of bytes to send?

- Use size of array declaration from compile time

  ```
  char data[256];
  int len = sizeof(data);
  ```

  `sizeof` is a compile time operator

- length of a string

  ```
  int len = strlen( datamessage );
  ```

# Building a Datagram message

For simple messages

Print to string

```
    sprintf( buffer, "pot:%f \n", value);
```

Slightly more complex

Print to string

```
    sprintf( buffer, "pot 1:%f \npot 2: %f\n", one, two);
```

Does it scale?

# More complex messges

## build it a line at a time

`strcat` concatenates (joins) strings

`sprintf` writes formatted text to strings

Start with empty buffer

```c
char buffer[512], line[80];
buffer[0]='\0'; /* make buffer look like empty string */
```

Format each line and concatenate it with buffer

```c
  sprintf(line,"POT 1:%f\n",left.read());
  strcat(buffer,line);

  sprintf(line,"POT 2:%f\n",right.read());
  strcat(buffer,line);
```

# Splitting incoming messages in C

By “hand”

## Read line

1. point line-pointer to beginning of buffer

```
line = buffer;
```

2. move the buffer-pointer along, looking for the end of line character

```
while(*buffer!='\n')++buffer;
```

3. overwrite line ending with string terminator

```
*buffer='\0';
```

4. move buffer-pointer to first character of next line

```
buffer++;
```

## Steps 3 and 4 can be combined

```
*buffer++ = '\0'
```

# C String library to the rescue

The C string library has a function that performs a similar task `strtok`

## Using `strtok`

1. In the initial call to `strtok` supply a pointer to the initial buffer, and the list (string) of delimiters

```c
char line = strtok(buffer, "\n\r");
```

2. For subsequent calls, pass a NULL pointer to `strtok`, it "remembers" where it is in the original buffer.

```c
line = strtok(NULL, "\n\r");
```

## Nested `strtok`

Because `strtok` remembers where it is internally, it cant be nested, one for reading lines, and one for splitting key value pairs.

# Two solutions

## Use delimeter changes

1. find first *key* token `key = strtok(buffer,":");`
2. *value* is the rest of the line `value = strtok(NULL,"\n ");`
3. next *key* is `key = strtok(NULL,":");`
4. *value* is the rest of the line `value = strtok(NULL,"\n ");`

## Re-entrant `strtok_r`

There is a version of `strtok`, that uses an external parameter to remember its place, these can be nested.

1. Read line `line = strtok_r(buffer, "\n", &loc);`
   - split line `key = strtok(line,":");`
   - `value = strtok(NULL,":");`
2. read next line `line = strtok_r(NULL, "\n", &loc);`

# For loop

- strtok fits nicely (if clumbersome looking) into a for loop
- strtok returns NULL if there are no more tokens to be found.

```
for(
    line = strtok_r(buffer, "\r\n", &nextline);
    line;
    line = strtok_r(NULL, "\r\n", &nextline)
) {
    key = strtok(line, ":");
```

# Using Key/values

- Suppose I have a multi-line message.
- First key is a message id.
- different functions are needed for each message.
- Can't use `switch` statements
- `if  else if` with `strcmp` is messy

```
if( strcmp(key, "first-id")==0)
else
if (strcmp(key, "second-id")==0 )
else
```

# Look-up table and function pointers

- action is function that takes pointer to value (assume `strtok`)

```
typedef int (*action)(char *value);
```

- structure pairing string with action

```
typedef struct {
    char *key;
    action act;
} tableentry;
```

- Array of these

```
tableentry[] lookup = {
    "foo", function1
    "bar", somethingelse
};
```

# Tokenise message

- First token in buffer is “action name”

```
char *name = strtok(buffer, ":");
```

- Second in paramater (may be status)

```
char *state = strtok_r(NULL, '\n');
```

- Find entry in lookup table with name

```
int n = findintable(lookup, name);
```

- Call function

```
lookup[n].act(state);
```

- Subsequent calls to `strtok(NULL,...)` in “action” walk down buffer until exhausted.

# Table lookup made easy

- Provide a comparison function with the same semantics as `strcmp` ie

| | | |
|---|---|---|
| for `compare(A,B)` | -1 | A is before B |
| | 0 | A and B are equal in order |
| | 1 | A is after B |

```
int order(void *A, void *B) {
    tableentry *a=A , *b=B;
    return strcmp(a->name, b->name);
}
```

- sort the table into order.

```
qsort( lookup, /* array of entries */
       sizeof(lookup)/sizeof(tableentry), /* number of ent
       sizeof(tableentry),  /* size of each entry */
       order /* ordering function */
   );
```

# Find the entry

- create a placeholder

  ```
  tableentry key;
  ```

- set the name to the key `char *key`

  ```
  key.name = key;
  ```

- call search

  ```
  tableentry *result = bsearch( &key, /* pointer to key */
                  lookup, /* array of entries */
                  sizeof(lookup)/sizeof(tableentry), /* numb
                  sizeof(tableentry),  /* size of each entry
                  order /* ordering function */
         );
  ```

- call the function from the result

  ```
  result->act(state);
  ```