PiADC Project

Bent Buttwill

The goal of this project is to get used to the environment the Raspberry Pi brings along. With this in mind the project uses almost all of the topics taught in the lecture ”Data Systems for Experimental Physics” PHYS391 UiB.

One simple use case for the Raspberry Pi, hereafter named RPi, with its 40 GPIO (General-purpose input/output) pins, is to use it as a data acquisition device. Although none the its pins can read analog signals, in connection with an ADC chip (Analog-digital converter) one can add this function to its repertoire. To communicate with the ADC chip the RPi can take advantage of its SPI (Serial Peripheral Interface) pins. Together with its networking abilities the RPi can be used for data acquisition in physical experiments or for home automation.

The project will include 3 binary files compiled from C. Two of which will be running on the RPi and one on a PC. The structure is represented in Fig. 1. As one can see on the RPi and and the PC there are programs running which are establishing the networking between each other. This is done by using sockets.

The server side is running on the pi and uses the following code to open a socket and wait for a client to connect. The execution of the program will halt in line 16 and will continue after a client wants to connect. It is important to note that the server as well as the client are using the same byte order. For example the value for `server_address.sin_port` was given in network byte order. The function `htons()` turns the bytes from host into network byte order. This has to be done on both sides, so that both machines read the port number in the same way.

```
//Socket setup
int server_sockfd;
int server_len, client_len;

char buffo[100], buff[100];
struct sockaddr_in server_address;
struct sockaddr_in client_address;
server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
server_address.sin_family = AF_INET;
```

```c
server_address.sin_addr.s_addr = INADDR_ANY;
server_address.sin_port = htons(9734);
server_len = sizeof(server_address);
bind(server_sockfd, (struct sockaddr *)&server_address,
   server_len);

//listen to port and wait
listen(server_sockfd, 5);

//client connected
client_len = sizeof(client_address);  // can this be wrong ??
client_sockfd = accept(server_sockfd, (struct sockaddr *)&
   client_address, &client_len);
printf("Connected\n");
```

Listing 1: Server side socket initialization.

The client side tries to establish connect with the server. Upon successful connection the rest of the code will be executed.

```c
//Socket setup
int result, len;
struct sockaddr_in address;
sockfd = socket(AF_INET, SOCK_STREAM, 0);
address.sin_family = AF_INET;
address.sin_addr.s_addr = inet_addr("192.168.0.142");
address.sin_port = htons(9734);
len = sizeof(address);
result=connect(sockfd, (struct sockaddr *) &address, len);
if (result == -1) {
  perror("ERROR:␣couldn't␣connect!");
  exit(1);
}
```

Listing 2: Client side socket initialization.

After successful connecting both the server and the client will start their send and receive threads and will then be trapped in infinite loops.

```c
//Start threads
pthread_t recieveThread;
pthread_t sendThread;
pthread_create(&recieveThread, NULL, thread_recieveData, NULL);
pthread_create(&sendThread, NULL, thread_sendSignal, NULL);
int gnuplotCreated;

for(;;) {
  //IF ladder for choosing what to do upon user input
  char input[10];
  fgets(input, 10, stdin);
```

```
    if(strcmp(input, "s\n") == 0) {
      sendState = 1;
    } else if(strcmp(input, "q\n") == 0) {
      kill(pid, SIGINT);
      break;
    } else if(strcmp(input, "p\n") == 0 && !gnuplotCreated) {
      gnuplotCreated = 1;
      pid = fork();
      if(pid == 0) {
        execlp("gnuplot", "gnuplot", "plot.p", NULL);
      }
    }

  }
  printf("Closing␣Socket!\n");
  close(sockfd);
  exit(0);
```

Listing 3: Client starting send/recieve threads and sitting endlessly in for-loop, waiting for input of user.

```
  //Start threads
  pthread_t readSHMthread;
  pthread_t LEDthread;
  pthread_create(&readSHMthread, NULL, thread_read_shm, NULL);
  pthread_create(&LEDthread, NULL, thread_TurnOnLED, NULL);


  for(;;) {
  }

  close(client_sockfd);
```

Listing 4: Server starting send/recieve threads and waiting in for-loop endlessly.

The recieving thread on the client is reading from the socket and storing the data in output file. The `fflush(file)` command is later important as it writes to the file without closing it.

```
void *thread_recieveData(void *arg) {
  int recieve;
  FILE *file = fopen("recieve.d", "w");
  for(;;) {
    read(sockfd, &recieve, sizeof(recieve));
    fprintf(file, "%d\n", recieve);
    fflush(file);
  }
  fclose(file);
}
```

Listing 5: Receiving thread on client.

The thread for sending a signal to the RPi loops indefinetly too and writes the letter *s* to the socket when the `sendState` gets set to 0 from the `main`.

```
void *thread_sendSignal(void *arg) {
  for(;;) {
    if(sendState) {
      printf("%d\n", sendState);
      sendState = 0;
      write(sockfd, "s", 10);
    }
  }
}
```

Listing 6: Sending thread on client.

For the server the sending and receiving threads are some what more interesting.
The sending thread is reading the data from shared memory guided by semaphores, to ensure the accessing of the memory only happens one at a time.

```
//THREAD// read the sheared memory and send to pc.c
void *thread_read_shm(void *arg) {
  //shared memory initialization
  void *shared_memory_point = (void *) 0;
  struct shm_structure *shmB;
  int shmid;

  shmid = shmget((key_t)1243, sizeof(struct shm_structure), 0666
     | IPC_CREAT);
  shared_memory_point = shmat(shmid, (void *)0, 0);
  shmB = (struct shm_structure *)shared_memory_point;

  //initialize semaphore
  sem_id = semget(   (key_t) 1133 ,  1  ,  0666 | IPC_CREAT );

  for(;;) {
    //get data from shared memory, then send to client
    (void) semaphore_p();
    write(client_sockfd, &shmB->data, sizeof(data));
    usleep(100);
    (void) semaphore_v();
  }
}
```

Listing 7: Thread for reading from shared memory, sending to client.

After initialization of the shared memory and the semaphore in the first code block this thread also gets trapped in an endless for loop. The function `semaphore_p()` asks linux if it can access the shared memory and waits for approval. When this thread gets the token from the system the execution can continue. Then the data gets copied from the shared memory into the socket and then send to the PC. After sleeping for about $100\mu s$ the semaphore gets released so that the other program can access the shared memory again. The structure of the shared memory has to be exactly defined. This is done by calling the following.

```
//shared memory structure
//TODO put into own header file
struct shm_structure {
  int data;
};
```

Listing 8: The structure used for the shared memory.

The receiving part on the RPi is also running continuously and checks if the incoming character is correct. Upon recieval the LED on GPIO pin 5 turn on for 1 second.

```
void *thread_TurnOnLED(void *arg) {
  int gpio_pin = 5;
  char recieve[10];
  for(;;) {
    recieve[0] = 'a';
    read(client_sockfd, recieve, 10);
    if(strcmp(recieve, "s") == 0) {
      //GPIO blink LED for 1 sec
      printf("Turning␣LED␣on␣and␣off!\n");
      setup_io();
      INP_GPIO(gpio_pin);
      OUT_GPIO(gpio_pin);

      GPIO_SET = 1 << gpio_pin;
      sleep(1);
      GPIO_CLR = 1 << gpio_pin;
    }
  }

}
```

Listing 9: Upon receiving the correct character the LED on GPIO pin 5 turns on for 1 second.

First the GPIO pins have to be configured. For this the addresses in memory are going to be mapped to virtual memory that is accessible for this program. This happens in the `setup_io();` function which is included in the `GPIO.h` header file. To put a specific GPIO pin into output mode one has to set it into input mode first and then into output mode. This is done via `INP_GPIO(gpio_pin)` and `OUT_GPIO(gpio_pin)` respectively. With the commands `GPIO_SET = 1 << gpio_pin` and `GPIO_CLR = 1 << gpio_pin` one can set the GPIO pin into the high and low state respectively. This is done by bit shifting a 1 bit to the correct position in memory.

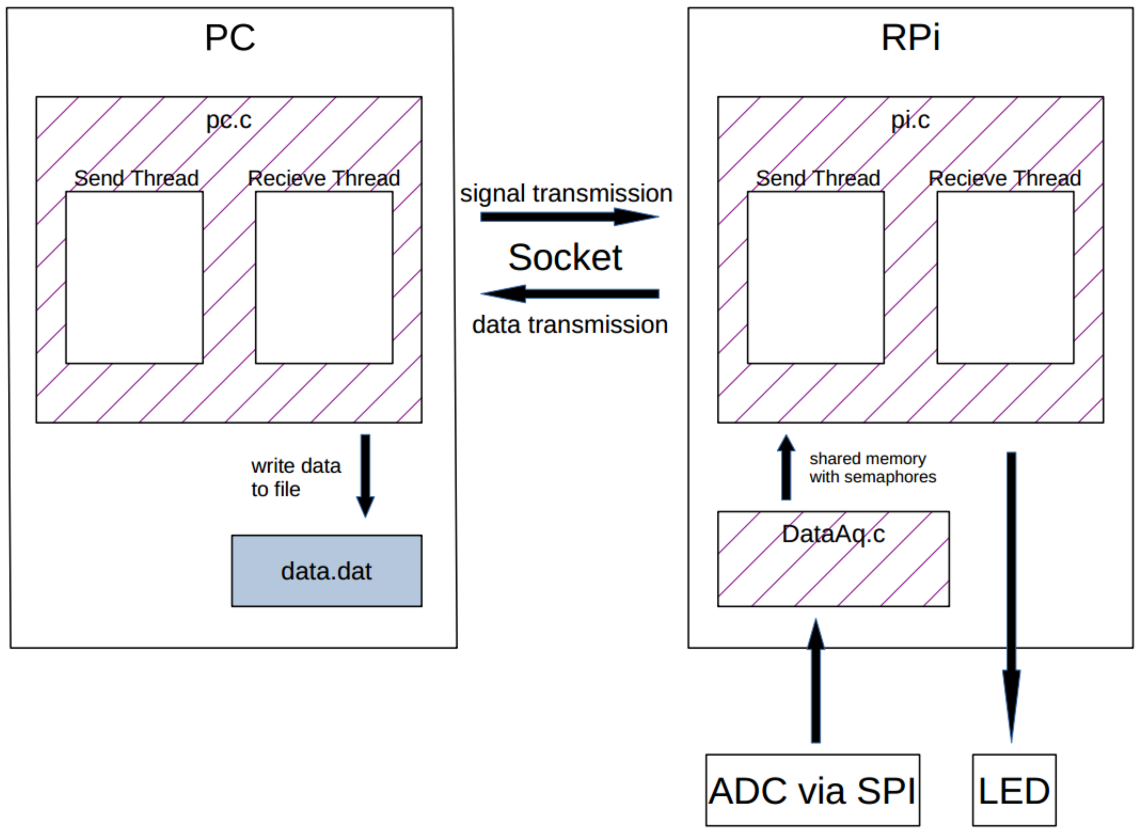

Figure 1: Structure of this project. Binary files are represented hatched.