KF5011 Assignment

Dr Alun Moon

Semester 2 — 2017/18

Contents

List of Tables

Part I
Assignment Details and Administration

1 Key Dates and Submission Mechanism

Table 1: Key Dates for assignment

Submission Deadline	Monday 30th April 2018
Demonstrations and Marking	Tuesday 1st May to Friday 18th May

1.1 Submission mechanism

The submission mechanism is via Github Classrooms and Blackboard.
There are separate assignment parts, each a software project:

Step 1. Follow the links in table 2, these should give you a private Github repository that you and I have access to.

Step 2. Use the repository for your work, remember to commit and push code regularly (see appendix B).

Step 3. Submit the URL through the Blackboard assignment(s) (Paste URL into the *Write Submission* text box)

Table 2: Assignment URLs

Game Controller	`https://classroom.github.com/a/VRh3OnkL`
Game Dashboard	`https://classroom.github.com/a/APDfS3Bb`

1.2 Academic Integrity Statement

This is an Individual Assignment. You must adhere to the university regulations on academic conduct. Formal inquiry proceedings will be instigated if there is any suspicion of plagiarism or any other form of misconduct in your work. Refer to the Universitys Assessment Regulations for Northumbria Awards if you are unclear as to the meaning of these terms. The latest copy is available on the University website.

2 Summative assessment

Summative assessment is by a portfolio assignment covering the development of a complex networked control system, including elements such as; input, output, feedback loops, networking, data logging, and security. It will assess all of the modules MLOs.

2.1 Learning Outcomes

2.1.1 Level 5 : Application

- using and applying knowledge
- using problem solving methods
- Manipulating
- Designing
- Experimenting

2.1.2 Knowledge & Understanding

1. Demonstrate knowledge and critical understanding of the interaction between physical systems, computer hardware and software, including control theories, network protocols, and cybersecurity architecture and operations.
2. Apply principles of design and implementation of stack models, network protocols, control systems, and security.

2.1.3 Intellectual/Professional skills & abilities

1. Design, implement, test, document and evaluate a networked embedded control system

2. Apply software development tools and best practice to produce, test and debug software for small networked control systems using specifications for embedded devices and associated hardware

2.1.4 Personal Values Attributes (Global/Cultural awareness, Ethics, Curiosity) (PVA)

1. Demonstrate independent, critical and reflective thinking and practice in the development of a networked control system, and engagement with appropriate professional and technical literature to support and communicate such development.

Part II
Assignment Tasks

3 Overview

You are to write programs as described in 3.2 and 3.3, and 3.4. The details are in sections 4, 5, and 6. You will be provided with a server that models the flight dynamics and propulsion, this is communicated with via UDP using the protocol described in appendix A.1

3.1 Lunar Lander

In the classic Lunar lander games the player controls the decent of a Lunar Lander spacecraft onto the surface of the moon. [1] [2]

3.2 Game Controller

You are to write a program (C++/Mbed) for the K64F platform and application shield to implement a game controller. This takes input from the player to control the game. It can display some limited information on the display and indicate states using the LEDs.

3.3 Game Dashboard

You are provided with a limited interface on the server, mainly for monitoring the server and debugging. The display on the applications board is black and white with a limited size. To provide a better user interface you are to write a dashboard program that runs on a PC and displays information communicated to it from the K64F.

3.4 Data Logging

You should be able to log important game data during a game. This can be analysed and plotted in a post game analysis.

[1] `https://en.wikipedia.org/wiki/Lunar_Lander_(video_game_genre)`
[2] `https://en.wikipedia.org/wiki/Lunar_Lander_(1979_video_game)`

4 Game Controller

4.1 Requirements

4.1.1 Inputs

Using the sensors and input devices on the FRDM+K64F and Applications shield, monitor the user's activity as inputs to the game. You can use any combination of Accelerometer, buttons, magnetometer, and temperature sensors as you wish.

4.1.2 Outputs

Some feedback to the user can be given using the LEDs, the LCD display, and the speaker. Examples could include

contact light indicates that the lander has touched down successfully

proximity warning to indicate you are close to the ground

fuel warning for when your fuel is running out

4.1.3 Communications

The K64F needs to manage it's communications with the Lander server, the dashboard, and the data logging service. Communications with the Lander server are by UDP Datagrams, the format of the messages and responses are given in appendix A.1

4.1.4 Control Logic

The K64F needs to take the requested actions from the user, and the current state of the lander. From these it needs to generate the outputs presented to the user, and the control signals it needs to communicate to the server.

4.2 Programming Hints

Here are some hints[3] and guidance on how to write a good, robust solution.

Use a good layer model for hardware. Having functions that provide a logical view of a device (such as `ispressed()`) aid in writing the program and hiding implementation details (such as the different wiring of the switches).

Clean Modular Separation of Function. Using separate functions, even source files, or libraries. Keeps functional requirements separate and allows you concentrate on each bit of functionality without the clutter of the other parts.

Use Threads, Events, and EventQueues You'll need several threads to handle some of the concurrent tasks, and allow for waits on the IO.

1. **Communications thread** for handling the UDP communications. The UDP send and receive functions are *blocking* so thread(s) allow other code to continue.

Skeleton

```
Thread comms;

void communicate(void) {
    while(1){
        udpsocket.sendto(server,...);
        udpsocker.recvfrom(&source,...);
    }
}

int main() {
    comms.start(communicate);
}
```

[3]Programming hints are gives as code fragments. Each may have a `main` function, you'll have onnly one `main`. They may be other code missing that is assumed, such as preparing and decoding messages.

2. **Hardware thread** for polling the sensors on the application shield. Use an `EventQueue` and `call_every()` to set up a periodic poll of the devices.

Skeleton

```
Thread worker;
EventQueue queue ;

void hardware_poll(void) {
    /* scan hardware, for position, motion etc */
}

int main() {
    queue.call_every(500, hardware_poll);
    worker.start(
        callback(&queue,
            &EventQueue::dispatch_forever ));
}
```

For the sample rate consider how responsive you want to be to the users inputs.

3. **Control thread** for periodically updating the system. Use an `EventQueue` and `call_every()` to set up a periodic tick for requlary updating the state.

4. **Display thread** updating the LCD takes time, a thread can update the display as data is available, independently of the rest of the system. It can run:

    - Continuously as a thread

Skeleton

```
Thread display;
display.start(display_update);
void display_update() {
    C12832 lcd(D11, D13, D12, D7, D10);
    while(1){
        /* Do LCD stuff */
    }
}
```

- Continuously with a wait

```
void display_update() {
    C12832 lcd(D11, D13, D12, D7, D10);
    while(1){
        /* Do LCD stuff */
        wait( /*some time*/ );
    }
}
```

- Periodically from an event queue

```
C12832 lcd(D11, D13, D12, D7, D10);
Thread events;
EventQueue queue;
int main(){
    queue.call_every(20, display_update);
    events.start(callback(
                &queue,
                &EventQueue::dispatch_forever));
}

void display_update() {
        /* Do LCD stuff */
        /* this function runs ONCE each event */
}
```

5 Dashboard

The dashboard is able to provide a more detailed user interface than can be provided for on the LCD display.

5.1 Requirements

The Dashboard should run on a PC or other computer connected to the K64F via the off-campus network (assuming you are in a Pandon Lab). There are some options for the communications, you are free to choose which ever works for you.

UDP Datagrams (**recommended**) UDP Datagrams will work once you have a network between the two. There is an overhead in that you'll have to hardwire IP addresses into the code and recompile it for different IP address combinations.

PC Serial (**not recommended!**) Using the serial connection via `Serial pc(USBTX, USBRX);`. The downside of this, is that this is the channel used by the serial monitor, so you'd loose that for debugging.

USB Serial Device The connection created by the `USBSerial comm;` library uses the other USB connection on the K64F board and is seen by the PC as a second serial device, this can be used in parallel with the Serial Monitor.

5.2 Asynchronous Communications

The Dashboard need only receive messages from the K64F. You may have ideas on features to add that does need communications in return – feel free to do so.

5.3 Protocol

You will need to use a Protocol that describes the format of the data as sent in the UDP Datagrams. If you end up writing the Dashboard in Java, or Python, then as the MBED/K64F code is in C++, there is a danger of miscommunicating. A protocol allows you to describe the format of the data, independently from the programming languages, and you'll have a better chance of success.

The Protocol will also state the Port Number that the Dashboard listens on. The choice is up to you, and you'll need to make sure that the Dashboard listens on the right port, and the controller sends to that port on the PC's IP address.

5.4 Skeleton Java UDP Class

```
public class UDPServer extends Thread {
    protected DatagramSocket socket = null;
    public static void main(String[] args) {
        UDPServer server = new UDPServer();
        server.start();
    }
    public UDPServer()  throws UnknownHostException, SocketException {
        socket = new DatagramSocket(
                    /* Port number */,
                    InetAddress.getLocalHost()/*<-- gets the PC IP address*/
        );
        /* confirm socket port and ip */
        System.err.println("Listening on port:"+socket.getLocalPort()+
                           " at ip:"+socket.getLocalAddress());
    }
    public void run() {
        while(true){
            byte[] buffer = new byte[8*1024];/* what ever size you need */
            DatagramPacket packet = new DatagramPacket(buffer,buffer.length);
            try {
                socket.receive(packet);
                /* you now have the packet data in the array of bytes */
                /* parse the data */
                /* update variables and display */
            }catch(IOException e){}
        }
    }
}
```

6 Data Logging

Part III
Marking Criteria

7 Game Controller

1.1 Over | 0 | 1 | 2 | 3 | 4 | 5 | 5%

Part IV
Appendicies

A Lander Server

The lander Server is a Java program that models the dynamics of the lander and the lunar surface. It is packaged as a `jar` file and executed using

```
java -jar LunarLander.jar
```

A.1 Communications Protocol

The server listens for UDP Packets on port 20769

A.1.1 Message Format

Messages to the Server and replies are formatted as Key:Value pairs as is done in email headers (RFC822[4]) or HTTP Headers (RFC7230[5]).

Query messages have the message name as the first key, with a single question mark (ascii 63) as the value

Command messages that set values or cause actions, have the message name as the first key and a corresponding exclamation mark (ascii 33) as the value.

Reply messages returned in responce to the above, have the message name as the key and as equals sign (ascii 61) as the value.

Request State This message requests the current state of the Lander:

Message send

```
state:?
```

[4] `https://tools.ietf.org/html/rfc822`
[5] `https://tools.ietf.org/html/rfc7230`

Reply from server

```
state:=
x:45.6
y:10.3
O:5
y':-1
O':+0.01
```

The keys and values are:

`x`	x position
`y`	y position
`O`	θ orientation (capital O)
`x'`	$\dot{x}$ horizontal velocity (x single-quote(ascii 39))
`y'`	$\dot{y}$ vertical velocity (y single-quote(ascii 39))
`O'`	$\dot{\theta}$ rate of rotation (capital O single-quote(ascii 39)))

Request condition This message requests the general condition of the Lander.

Message send

```
condition:?
```

Reply from server

```
condition:=
fuel:98%
altitude: 20.7
contact:flying|down|crashed
```

The keys and values are:

`fuel`	percentage of fuel remaining
`altitude`	the radar altitude above the ground
`contact`	A keyword describing the general state of the Lander
	`flying` moving above the terrain
	`down` in contact with the ground in a successful landing
	`crashed` in contact or underground in a failed landing

Command Engines This message sets the requested levels on the engines

Message send

```
command:!
main-engine: 100
rcs-vertical: 1.0
rcs-horizontal: -0.7
rcs-roll:  +0.5
```

The keys and values are:

`main-engine`	percentage throttle setting	0...100
`rcs-vertical`	the strength of a vertical thrust	-1...1
`rcs-horizontal`	the strength of a horizontal thrust	-1...1
`rcs-roll`	the strength of the rotational thrust +ve is anti-clockwise	-1...1,

Reply from server

```
command:=
```

Request Terrain This message requests the terrain below the lander from its mapping radar

Message send

```
terrain:?
```

Reply from server

```
tarrain:=
points:10
data:[
12.0  , 23.0
34.2 , 23.0
]
```

The keys and values are:

`points`	The number of data points
`data`	an array of (x, y) pairs of ground coordinates

The ground coordinates are supplied one (x, y) pair per line of text. The coordinates are absolute and measured in the same system as the lander

state.

If the Lander's (x, y) position matches a ground (x, y) position, it has reached that point on the ground.

The altitude reported should be the difference between y coordinates of the lander and the ground directly below it.

B Git usage guide

Once you have a clone of the assignment repository you can follow the workflow below. These are shell commands and work for the Linux, Mac OSX, and Windows (Command line) versions of git.

1. Clone the repository – create a local working copy
   `git clone https://github.com/your-git-id/repository-name.git`
   or if you prefer the ssh version
   `git clone git@github.com:your-git-id/repository-name.git`
2. Work on your code...
3. Check which files have changed and/or been added
   `git status`
4. Add the files that you want to keep the changes for
   `git add file1 file2.cpp etc`
   These files will be uploaded back to the github server below
5. Commit the changes, add a note explaining the work you have done
   `git commit -m"message goes here in quotes"`
6. Push the changes back to the github server
   `git push`

Whether you use

Github desktop `https://desktop.github.com/`

GitKraken `https://www.gitkraken.com/`

Atom's Github integration `https://github.atom.io/`

The basic workflow is the same, the details of each interface and front end vary.

Table 3: Lander position and orientation

x	horizontal position	m
y	vertical position	m
θ	orientation	$^\circ$

Table 4: Lander states

x	horizontal position	m
y	vertical position	m
θ	orientation	$^\circ$
$\dot{x}$	horizontal velocity	$\mathrm{m\,s^{-1}}$
$\dot{y}$	vertical velocity (-ve is down)	$\mathrm{m\,s^{-1}}$
$\dot{\theta}$	rotation rate	$^\circ\,\mathrm{s^{-1}}$

C Lander Model

The Lander moves in 2D, it's position is fully described by three values (table 3). The position is measures from a datum such that all values of y are positive. An orientation of 0 is upright, positive values of θ is a tilt left, and negative a tilt right. The model maintains these three values and the rate that they are changing, for a total of 6 states (table 4). The controls are the Main-engine throttle, and the RCS thrusters. The Main engine fires alone the line of the Landers orientation, the RCS thrusters can be fired to control rotation and small movements sideways and up-down.