Assignment 1
Control of Simultaneous Localization and Mapping (SLAM) Robot

Date Due: Tuesday 25th August 2020
Marks: 200 (20% of total assessment marks)
Penalty for Late Submission: 5% per day late

Outline

In this assignment, you will design, implement, test and document a C program to control a simple robot to be used in a Simultaneous Localization and Mapping (SLAM) application. SLAM is employed in self-driving cars, robotic vacuum cleaners and augmented reality applications.

The localization aspect of the application involves the robot tracking its own location and orientation as it explores an unknown environment. The mapping aspect involves the robot building a map of the environment based upon information from four touch/contact sensors and a single distance sensor, the latter of which can be rotated independently of the robot orientation.

The assignment does not use any robot hardware. Instead, the operation of the robot is simulated by a POSIX compliant program provided as a Cygwin executable which your controller program will interface to.

The assignment is divided into two parts:

- Part A (150 marks) involves manual control using the keyboard
- Part B (50 marks) involves autonomous control

Both parts must be implemented in the **same** C program per the requirements in this specification.

There are two deliverables:

1) A text readable and non-compressed/non-zipped Word or PDF **report** documenting your system design, implementation and testing. The report must include the following concise information:

System design	-	A state based diagram for Part A (manual) operation
	-	A state based diagram for Part B (autonomous) operation
	-	A brief explanation of significant design choices for Part A (manual) AND Part B (autonomous) (maximum 2 pages)
Testing results	-	The test cases you executed for Part A (manual) AND Part B (autonomous) and the associated results
Program listing	-	The commented C source code in an appendix

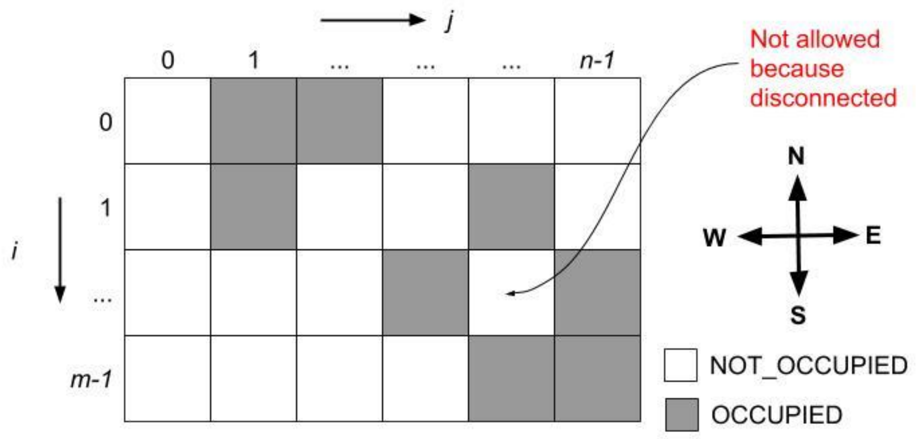
2) A **zip file** of the directory containing your Code::Blocks project and C source/header files. This must preserve the Code::Blocks project directory structure so that the project can be imported into Code::Blocks without manual tweaking. Remember to save not only the C source files, but the project itself from within Code::Blocks before you build your zip file.

System Specification

The environment to be explored is as follows:

- It comprises a rectangular 2D grid of *m* rows and *n* columns, where the rows are indexed *i*=0..*m-1* and the columns are indexed *j*=0..*n-1*
- Each cell of the grid can have a permanent occupancy status of either NOT_OCCUPIED or OCCUPIED as determined by a map that the simulator loads from a file at startup
- The robot can only successfully move to cells which have an occupancy status of NOT_OCCUPIED
- The robot always starts its mission at *i*=0 and *j*=0 so this cell always has an occupancy status of NOT_OCCUPIED in any valid map
- In order to limit the complexity of the robot design in autonomous mode, the maximum grid size that needs to be considered is *m=8* and *n=8*
- The set of cells with occupancy status NOT_OCCUPIED are connected in any valid map, which means it will always be possible to reach one such cell from another such cell if the correct sequence of instructions are presented
    - As a corollary, this also means there are no 'hollow' structures i.e. a valid map will not contain one or more cells with occupancy status NOT_OCCUPIED completely surrounded by cells with occupancy status OCCUPIED
- NORTH is the direction of decreasing row number *i*
- EAST is the direction of increasing column number *j*
- SOUTH is the direction of increasing row number *i*
- WEST is the direction of decreasing column number *j*

These concepts are illustrated below:



The robot has a simple design as follows:

- It uses a typical 2 motor control, driving a track on each side (left and right), in order to move and turn.

- Each of the two track motors can be independently operated at one of three settings:
    - FORWARD (single speed)
    - OFF
    - BACK (single speed)

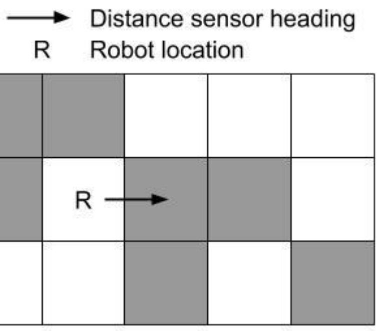
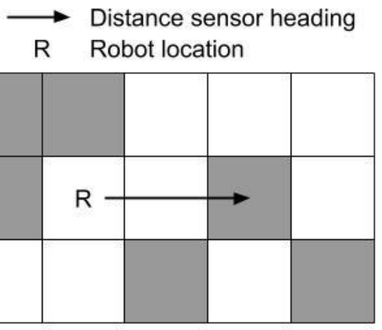
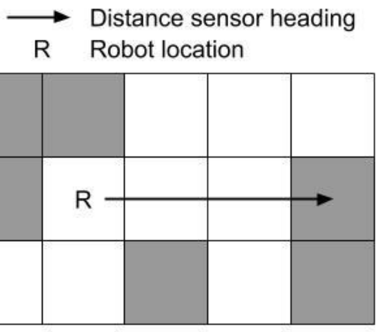
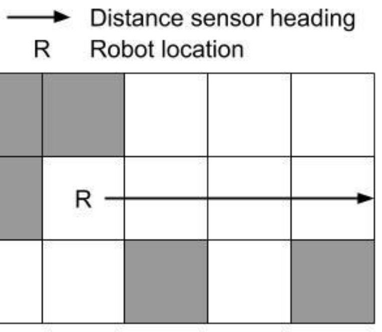
- The controller sets both track motors to FORWARD to drive the robot forward by 1 grid cell (subject to there being no obstacle in its path). For example, if the initial robot location is (i,j) and its initial heading is EAST before the instruction to move forward, the location will be (i,j+1) after the instruction has completed execution provided the occupancy status of cell (i,j+1) is NOT_OCCUPIED.

- The controller sets both track motors to BACKWARD to drive the robot backward by 1 grid cell (subject to there being no obstacle in its path). For example, if the initial robot location is (i,j) and its initial heading is EAST before the instruction to move backward, the location will be (i,j-1) after the instruction has completed execution provided the occupancy status of cell (i,j-1) is NOT_OCCUPIED.

- The controller sets the left track motor to FORWARD and the right track motor to BACKWARD to turn the robot clockwise on the spot by 90 degrees (as viewed from above) without changing its grid cell. For example, if the initial robot location is (i,j) and its initial heading is EAST before the instruction to turn clockwise, the location will still be (i,j) and the robot heading will be SOUTH after the instruction has completed execution.

- The controller sets the left track motor to BACKWARD and the right track motor to FORWARD to turn the robot anticlockwise on the spot by 90 degrees (as viewed from above) without changing its grid cell. For example, if the initial robot location is (i,j) and its initial heading is EAST before the instruction to turn anticlockwise, the location will still be (i,j) and the robot heading will be NORTH after the instruction has completed execution.

- A single distance sensor is attached to the robot with an associated motor to change its heading. The distance sensor has a range of 2 grid cells in the direction of its heading and therefore reports one of the following values:

Distance Sensor Value	Description	Example
0	Adjacent grid cell in the direction of the sensor heading is OCCUPIED	
1	Adjacent grid cell in the direction of the sensor heading is NOT_OCCUPIED, but the next grid cell along is OCCUPIED	
2	Adjacent 2 grid cells in the direction of the sensor heading are NOT_OCCUPIED, but the next grid cell along is OCCUPIED	
NO_OBJECT_IN_RANGE	Adjacent 3 grid cells in the direction of the sensor heading are NOT_OCCUPIED. The occupancy status of the next (i.e. 4th) grid cell along is indeterminate from this sensor value.	

- The distance sensor motor can be operated at one of three settings:
    - CLOCKWISE (single speed)
    - OFF
    - ANTICLOCKWISE (single speed)

- The controller sets the distance sensor motor to CLOCKWISE to turn the distance sensor clockwise by 90 degrees relative to the robot (as viewed from above) and to ANTICLOCKWISE to turn the distance sensor anticlockwise by 90 degrees relative to the robot (as viewed from above). For example, if the initial robot location is (*i*,*j*), its initial heading is EAST and the distance sensor heading is SOUTH before the instruction to turn the distance sensor clockwise, the location will still be (*i*,*j*), the robot heading will still be EAST and the distance sensor heading will be WEST after the instruction has completed execution.

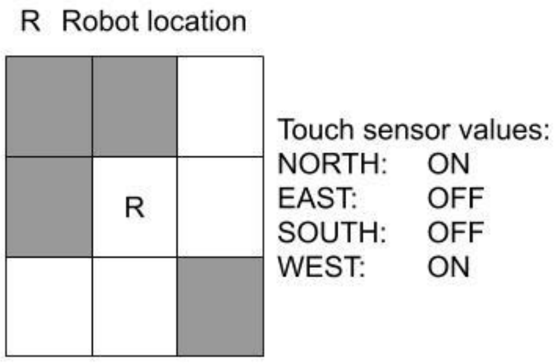
- When the controller turns the robot by 90 degrees, the distance sensor is also turned by 90 degrees in the same sense. For example, if the initial robot location is (*i*,*j*), its initial heading is EAST and the distance sensor heading is SOUTH before the instruction to turn the robot clockwise, the location will still be (*i*,*j*), the robot heading will be SOUTH and the distance sensor heading will be WEST after the instruction has completed execution.

- Simultaneous moving of the robot (forward/backward), turning of the robot (clockwise/anticlockwise) and/or turning of the distance sensor (clockwise/anticlockwise) is not supported. Only one of these three operations is supported at any one time.

- After successfully completing a robot move (by 1 grid cell), a robot turn (by 90 degrees) or a distance sensor turn (by 90 degrees), the <u>simulator</u> sets both track motors to OFF and the distance sensor motor to OFF. It signals that execution is complete for the associated instruction, so it is not necessary for the controller to explicitly stop the motors.

- The robot always starts its mission at $i=0$ and $j=0$ with a heading of EAST and the distance sensor also has an initial heading of EAST.

- The robot is equipped with four touch/contact sensors, one each for the directions of NORTH, EAST, SOUTH and WEST. Each touch sensor reports OFF if the adjacent grid cell in its associated direction is NOT_OCCUPIED and ON if the adjacent grid cell in its direction is OCCUPIED. This is independent of the current robot heading i.e. if the robot is turned clockwise or anticlockwise to change its heading, the values of the four touch sensors will not change. An example is illustrated below.



- Touch and distance sensor values reported by the simulator are only updated after the current instruction (i.e. move robot, turn robot, turn distance sensor) has finished execution so that all motors are stopped.

- Any notional grid cell outside the current map limits (*i*=0..*m-1* and *j*=0..*n-1*) is regarded as having an occupancy status of OCCUPIED for the purposes of determining distance sensor and touch sensor values.

- Assume the robot never runs out of battery charge.

- The following parameters are provided for information (these are not required for manual control, however they could conceivably be used to optimise the algorithm for autonomous control):

Duration for robot to move one grid cell:	2.5 secs
Duration for robot to turn 90 degrees (robot turn):	1.5 secs
Duration for distance sensor to turn 90 degrees (distance sensor turn):	1.0 secs

Map Files

The simulator loads the map from a file with name “map.txt” at startup. For expediency, the map file also indicates whether the mission should be manual or autonomous by a single character ‘M’ or ‘A’ at the start of the file.

Several sample map files are provided (note that your program may be tested with map files other than those provided). To use a specific map file, copy it, rename it to “map.txt” and place it in the same directory as the simulator executable. Be careful that the actual final name of the file is “map.txt” and not “map.txt.txt” which can be an issue if Windows is configured to hide file extensions. You can check the true name of a file by examining its properties. The simulator will only read a file with name “map.txt”.

Your controller program must not read the map file that configures the environment, this is only for the simulator to read; the controller must discover and build the map based upon sensor values it obtains, for both Part A and Part B operation.

It is possible for you to build your own maps for use by the simulator as the map file format is self explanatory for the most part. The occupancy status of the grid cells is coded in hexadecimal on a row-by-row basis with OCCUPIED = 1, NOT_OCCUPIED = 0.

Part A (Manual Control) Design

The application to be designed must correctly control the robot in response to user input from the keyboard. You are required to design, develop and test a **state driven implementation** for the robot control to meet the following requirements. Some utility functions, described later in this specification, are provided to help you in this task.

1. Control the robot based on the following key presses which can be obtained via a utility function described later:

Key	Action Required
‘f’ or ‘F’	**F**orward move by 1 grid cell (if not blocked)
‘b’ or ‘B’	**B**ackward move by 1 grid cell (if not blocked)
‘c’ or ‘C’	**C**lockwise turn of robot by 90 degrees
‘a’ or ‘A’	**A**nticlockwise turn of robot by 90 degrees
‘1’	**C**lockwise turn of distance sensor by 90 degrees
‘2’	**A**nticlockwise turn of distance sensor by 90 degrees
‘m’ or ‘M’	**M**ap print (print the map as it is currently known)
‘q’ or ‘Q’	**Q**uit simulation

2. Track the robot location, robot heading and distance sensor heading, updating these after each successful robot move, robot turn or distance sensor turn. Note that you are not expected to track fractional moves or turns; all tracked co-ordinates should be integers and all tracked headings should be from the set {NORTH, EAST, SOUTH, WEST}.You can (manually) confirm that you are tracking correctly by correlating the printout from your controller window with the printout from the simulator window.

3. Update the map of the environment after each successful robot move, robot turn or distance sensor turn based upon the readings of the touch sensors and distance sensor that are provided by the simulator (note: you only need to print the map on a key press of ‘m’ or ‘M’). You will not be able to gain visibility of all grid cells because some will be surrounded by grid cells with an occupancy status of OCCUPIED; for Part A operation, you may leave the occupancy status of such cells as unknown.

4. Print at least the following information to the controller window on a regular basis (e.g. 2 times per second), but not so fast that the output is unreadable in real time:
   - simulation time
   - (i,j) integer co-ordinates of robot
   - robot heading
   - four touch sensor values (OCCUPIED = 1, NOT_OCCUPIED = 0)
   - distance sensor heading
   - distance sensor value (0, 1, 2 or NO_OBJECT_IN_RANGE = -1)
   - current state according to your state machine

   Note that this printing is already done for you by default in the example source code for the controller; it is stated here so that you do not remove this functionality.

Part B (Autonomous Control) Design

The application to be designed must autonomously control the robot to explore its environment and build a complete map of it, return the robot to its starting point and print the final map. You may use the touch sensors and/or the distance sensor in any way you see fit, and you may employ any exploration algorithm you choose. You do not necessarily need to visit each and every grid cell with occupancy status NOT_OCCUPIED to achieve the mission. In fact, as there are marks for performance (i.e. time taken to build the map and return to the origin) as well as functionality, the number of grid cells you visit should be minimised. So while a random exploration of the environment may gain full marks for functionality, it is unlikely to score well in terms of performance.

In general, this is a very difficult problem to solve because of the variety of environmental features you may find (e.g. tunnels, dead-ends) and the fact that you will not be able to gain direct visibility of the occupancy status of some grid cells as they are surrounded by grid cells with an occupancy status of OCCUPIED. The following simplifications make the problem more tractable for this assignment:

- the maximum grid size that needs to be considered is $m=8$ and $n=8$
- after completely exploring the environment, you may set the occupancy status of any cells which are hidden to OCCUPIED without performing any calculations to ensure such cells are not reachable

It is recommended that you use a wall following algorithm to explore the environment i.e. the robot follows the perimeters of walls or standalone obstacles. Consider each possible scenario of how the perimeter of a wall or obstacle may change from one grid cell to the next, then evaluate whether the robot should move forward or turn before moving forward in order to follow the perimeter. Do not forget to consider dead-ends! You will find this algorithm can be implemented very efficiently as a state machine.

You are required to design, develop and test a **state driven implementation** for the robot autonomous control to meet the following requirements:

1. The robot must start and complete its mission at $i=0$ and $j=0$.

2. Track the robot location, robot heading and distance sensor heading, updating these after each successful robot move, robot turn or distance sensor turn. Note that you are not expected to track fractional moves or turns; all tracked co-ordinates should be integers and all tracked headings should be from the set {NORTH, EAST, SOUTH, WEST}.

3. Update the map of the environment based upon the readings of the touch sensors and distance sensor that are provided by the simulator after successfully executing instructions.

4. After each successful move (not turn) of the robot and the associated map update, print the currently known map contents to the screen so its path can be visualised as it explores its environment.

5. Just as in Part A, print at least the following information to the controller window on a regular basis (e.g. 2 times per second), but not so fast that the output is unreadable in real time:
   - simulation time
   - (i,j) integer co-ordinates of robot
   - robot heading
   - four touch sensor values (OCCUPIED = 1, NOT_OCCUPIED = 0)
   - distance sensor heading
   - distance sensor value (0, 1, 2 or NO_OBJECT_IN_RANGE = -1)
   - current state according to your state machine

6. You do not need to respond to any key presses except for ‘q’ or ‘Q’ to exit.

7. At the end of the mission, if some grid cells still have an unknown occupancy status, update the occupancy status to OCCUPIED, print the final map and wait for the user to press ‘q’ or ‘Q’ to exit.

Implementation

You are required to implement a single **state-based** real time control program for Part A (manual control) and Part B (autonomous control) using the C standard library and C POSIX library to execute on a Windows compatible PC with a Cygwin environment. Stated-based operation is required, NOT input based.

You will be provided with the following in addition to this specification:

- ‘Assgn1_2020_Simulator.exe’, a separate Cygwin executable which you run at the same time as your control program. This program loads the environment map from “map.txt”, simulates the movement of the robot, tracks its location and heading, and provides touch and distance sensor data.

- A Code::Blocks project ‘Assgn1_2020_Control’ which contains the following controller related files:

    - A header file ‘robotControl.h’ containing some macros and declarations.

    - A source file ‘robotControlInterface.c’ which provides a set of subroutines that interface with the robot simulation program and obviate the need for you to understand some of the finer details of the interface at this point in time. A summary of the function of each subroutine is provided below.

    - A source file ‘robotControl.c’ which provides a template for a solution and makes use of ‘robotControl.h’ and ‘robotControlInterface.c’. The example source code in ‘robotControl.c’ is for Part A (Manual Control) and so should be run with a map that specifies manual mode; it attempts to move the robot forward on each press of ‘f’ or ‘F’ and prints the currently known map on each press of ‘m’ or ‘M’. You should modify the source code in ‘robotControl.c’ to create your state machine control to satisfy the requirements of this specification. You can change the name of the file if you wish.

It is compulsory to use the robot simulator executable and the Code::Blocks project provided. It is possible to code the whole solution only by updating ‘robotControl.c’, although you may wish to add further source/header files for increased modularity.

Once you have compiled your controller project and wish to test it, the two programs (i.e. Assgn1_2020_Simulator.exe and your controller program executable) need to be running at the same time in separate windows. To facilitate this:

- Place the two executables in the same directory so that they can share resources correctly when running
- Place a map file “map.txt” in the same directory. The map file you use should be appropriate to the type of control (manual or autonomous) that you are trying to test.
- Start the simulator first by double clicking on Assgn1_2020_Simulator.exe. It is important that the simulator is started first for correct operation.

- Once the simulator is running, start the controller by double clicking on its executable.
- Only the controller window will accept keyboard input, so you must ultimately select it as the active window and keep the focus on it.

VERY IMPORTANT – DO NOT click on any other windows while the programs are running. Keep the focus on the controller window, otherwise keyboard input will not be accepted! You should see text output in both the controller and simulator windows.

Functions of subroutines contained in robotControlInterface.c

The first two subroutines control the connection to the robot simulator program.

void robotOpen();

Call this routine once at the top of your program to initialize the link to the robot simulator and configure user input.

void robotClose();

Call this routine once at the end of your program to close the link to the robot simulator.

The next few subroutines relate to management of the map that the controller builds.

void initializeMapContents(int mapContents[][MAXIMUM_MAP_SIZE]);

Call this routine with a 2D array representing a map to initializes all (valid) locations in the controller map to UNKNOWN_WHETHER_OCCUPIED_OR_NOT_OCCUPIED (-1) except (0,0) which is always UNOCCUPIED (0).

int setMapContents(int mapContents[][MAXIMUM_MAP_SIZE], int i, int j, int value);

Call this routine with a 2D array representing a map to set the occupancy status of a specific co-ordinate (i,j) in the map. The occupancy status is represented by the value object and can be one of OCCUPIED (1), NOT_OCCUPIED (0) or UNKNOWN_WHETHER_OCCUPIED_OR_NOT_OCCUPIED (-1)

The function will return one of the following:
SET_MAP_CONTENTS_SUCCESS (0)
SET_MAP_CONTENTS_INVALID_LOCATION (-1)
SET_MAP_CONTENTS_INVALID_VALUE (-2)

void printMapContents(int mapContents[][MAXIMUM_MAP_SIZE], int robot_i, int robot_j);

Call this routine to print the 2D array representing the controller map as currently known using '1' for OCCUPIED, '0' for NOT_OCCUPIED and 'X' for UNKNOWN_WHETHER_OCCUPIED_OR_NOT_OCCUPIED. The routine also prints the current robot location as ‘R’ using the last two arguments as robot co-ordinates (i.j).

The next subroutine allows printing of status information to the screen:

void printScreenLine(double simTime, int i, int j, int robotHeading, int tn, int te, int ts, int tw, int distanceSensorHeading, int distanceSensorValue, char* stateName);

Call this routine to print status information (simulation time, robot location, robot heading, touch sensor values, distance sensor heading, distance sensor value and current state name) to the controller window.

The next two subroutines set the robot motors for various tasks:

int setRobotTrackMotors(int motorLeftTrack, int motorRightTrack, int robotHeading);

Call this routine with MOTOR_OFF (0), FORWARD (1) or BACKWARD (-1) as an argument to independently set the speed of each of the left and right track motors for moving and turning, and one of NORTH (0), EAST (1), SOUTH (2) or WEST (3) to signal the current robot heading as tracked by the controller.

The function will return one of the following:
SET_ROBOT_TRACK_MOTORS_SUCCESS (0)
SET_ROBOT_TRACK_MOTORS_ILLEGAL_MOTOR_CONFIG (-1)
SET_ROBOT_TRACK_MOTORS_NOT_READY_TO ACCEPT_INSTRUCTION (-2)
SET_ROBOT_TRACK_MOTORS_UNABLE_TO_MOVE_IN_SPECIFIED_DIRECTION (-3)

int setRobotDistanceSensorMotor(int motorDistanceSensor);

Call this routine with MOTOR_OFF (0), CLOCKWISE (1) or ANTICLOCKWISE (-1) as an argument to turn the distance sensor motor.

The function will return one of the following:
SET_ROBOT_DISTANCE_SENSOR_MOTOR_SUCCESS (0)
SET_ROBOT_DISTANCE_SENSOR_MOTOR_ILLEGAL_MOTOR_CONFIG (-1)
SET_ROBOT_DISTANCE_SENSOR_MOTOR_NOT_READY_TO_ACCEPT_INSTRUCTION (-2)

The next few subroutines get information from the robot simulator.

double getSimTime(void);

Call this routine to get a double representing the current time from the simulator in seconds since the simulation began; note this does not necessarily reflect real time.

int getMapNumberOfRows();

Call this routine to get an int representing the number of rows in the map.

int getMapNumberOfColumns();

Call this routine to get an int representing the number of columns in the map.

int getOperationMode();

Call this routine to get an int representing the control mode:
MANUAL_CONTROL (1)
AUTOMATIC_CONTROL (2)

int getTouchSensorValueNorth(), int getTouchSensorValueEast(), int getTouchSensorValueSouth(), int getTouchSensorValueWest();

Call one of these routine to get an int representing the current touch sensor value for NORTH, EAST, SOUTH or WEST respectively.

NOT_OCCUPIED (0)
OCCUPIED (1)

int getDistanceSensorValue();

Call this routine to get an int representing the current distance sensor value (0, 1, 2, NO_OBJECT_IN_RANGE = -1) for the current distance sensor heading.

int isSimulatorReadyForNextInstruction();

Call this routine to get an int representing whether the simulator is ready for the next instruction

OFF (0) - not ready
ON (1) - ready

char getKey(void);

Call this routine to get the most recent key press by the user which has not already been handled, (the function then resets the key to NO_KEY to prevent handling the same key press more than once)

The function returns the most recent key press by the user which has not already been handled as a char, otherwise NO_KEY (0).

int isRobotSimulationQuitFlagOn();

Call this routine to determine whether the simulator is quitting after receiving a 'q' or ‘Q’ key press.

The function will return one of the following:

OFF (0) - quit flag off
ON (1) - quit flag on

The next subroutine causes the control program to sleep.

void sleepMilliseconds(long ms);

Call this routine to put the calling thread to sleep for a certain number of ms specified in the argument.

Important note:

Programs submitted must be the work of each individual student. Exchange of ideas on conceptual issues between students is expected, but students must ***write their own programs*** and not share code.

Assessment 1 Rubric:

The rubric is presented on the following page. In addition to the rubric, there are also penalties for not complying with this specification in certain areas as follows. These are at the discretion of the examiner.

Issue	**Penalty**
The packaging of submitted files does not meet the specified requirements (see page 1).	Up to 30 marks
When compiling the source files, there are unresolved compiler warnings (e.g. variables being used uninitialized) which can cause the program to fail to work correctly on one machine while working correctly on another.	Up to 30 marks
The provided source files (and in particular the interface subroutines defined in robotControlInterface.c) have not been used – note: you do not necessarily need to use all subroutines in robotControlInterface.c	Up to 50 marks
The controller reads the map file directly instead of building the map based upon sensor values reported by the simulator.	Up to 50 marks

ELE3307 Assignment 1 marking criteria

Each attribute below relates to a key element of the assignment, marked on a scale of 0 to 100%. Zero marks will be awarded for no attempt.

Attribute	**Poor**	**Adequate**	**Good**	**Excellent**	**Mark**
Part A functionality - user input	The program completely fails to respond to user input as specified (including the cases of compile time or runtime errors)	The program correctly responds to some user inputs per the specification	The program correctly responds to most user inputs per the specification	The program correctly responds to all user inputs per the specification	/ 50
Part A functionality - Location/heading tracking and map building	The program does not attempt to track the location/heading of the robot and build the map, or it does so incorrectly for the most part	The program attempts to track the location/heading of the robot and build the map, but there is a major bug	The program for the most part tracks the location/heading of the robot correctly and builds the map correctly, but there are some subtle bugs	The program always tracks the location/heading of the robot correctly and builds the map correctly	/ 20
State Diagram(s)	Most details are incorrect or do not represent actual program operation	Fairly well drawn state diagram, could have been abstracted better, some details (e.g. states, transitions, labels) are incorrect or missing	Well drawn state diagram, a small number of details (e.g. states, transitions, labels) are incorrect or missing	Complete and correct state diagrams, completely matches actual program operation	/ 40
Other Documentation (e.g. test plan/results)	Documentation is poorly written or does not explain design/implementation/testing	Documentation is limited or explanation of design/implementation/testing is incomplete	Documentation is adequately written and explains design/implementation/testing.	Documentation is well written and explains design/implementation/testing clearly.	/ 20
Program Implementation	The program is implemented with limited use of accepted programming techniques/comments.	The program is implemented with some use of accepted programming techniques/comments.	The program is implemented with adequate use of accepted programming techniques/comments.	The program is implemented with good use of accepted programming techniques/comments.	/ 20

Part B functionality	The robot always fails to complete its mission successfully with non trivial map files	The robot sometimes fails to complete its mission successfully depending on the map file	The robot usually succeeds in completing its mission successfully, but there are some subtle bugs	The robot always succeeds in completing its mission successfully including printing the correct final map	/ 25
Part B performance	There is little or no optimisation to reduce the mission time, or the optimisation is incorrect	There is limited optimisation to reduce the mission time, or the optimisation is waypoint file specific	The optimisation to reduce the mission time performs well and is comparable to the sample solution	Excellent level of optimisation, superior to the sample solution for most map files	/ 25

Marker's Comments:

TOTAL MARK	/ 200