COE538 Microprocessor Systems
Lab Project
Robot Guidance Challenge[1]

Peter Hiscocks
Department of Electrical and Computer Engineering
Ryerson University
phiscock@ee.ryerson.ca

Contents

1 Project Overview

This year, the COE538 Project is to program the *eebot* mobile robot with a navigation system that can find it's way through a maze, reverse, and back it's way out again. A possible maze is shown (not to scale) in figure 1.

1. The robot is started at the entry point and tracks down the guidance line.
2. The robot must navigate the **S** turns to demonstrate that the guidance algorithm is working correctly.
3. Whenever the *eebot* encounters a junction, it should make a decision which branch to take.
4. If the branch comes to a *dead end*, the robot will encounter a barrier that actuates the front bumper. It should then execute a 180^o turn, retrace the path, and take the other branch. It should also note that the branch taken was the incorrect one, and note the correct branch.
5. If the robot does not encounter a dead end on that path, it should remember that the branch it chose was the correct one.
6. This process continues until the robot reaches the maze *forward destination point*. The operator taps the rear bumper to indicate this to the robot.

[1] This lab project was adapted to be used with the HCS12 microcontroller by V. Geurkov.

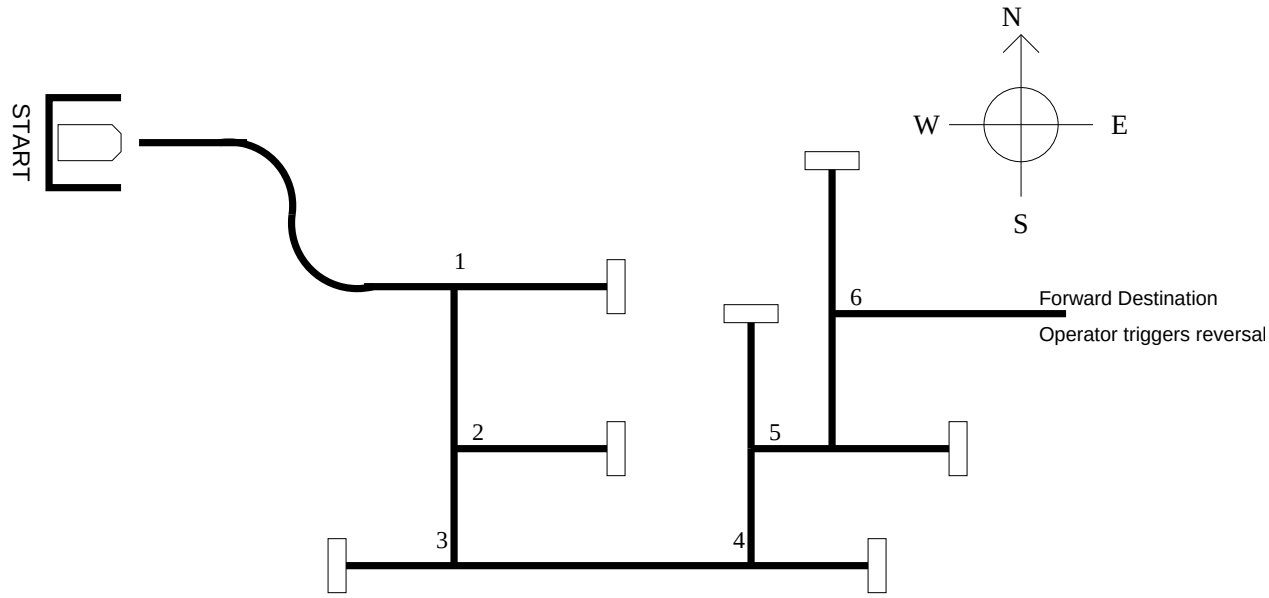

Figure 1: Robot Guidance Challenge

7. The robot should then retrace its path, back to the starting point, each time choosing the correct branch. In effect, the robot has *learned* or *solved* the maze.

   A possible variation on this is that the robot first learns the maze. Then it is started again at the beginning and should navigate the maze without errors. However, reverse traversal of the maze should be doable and is much more exciting to watch.

2 Rules of the Challenge Project

- Students will be graded on the degree to which their program accomplishes the robot guidance task and the features of their guidance program.
- The project will be worth 15% of the final COE538 mark.
- The exact layout is subject to change at the last minute, but will include features similar to those shown in figure 1.
- Only **L** and **T** junctions are possible, so that there are only two choices at each intersection.
- Each branch leads either to an *obstactle block* that may be detected with the front bumper, or another intersection. (It is possible to design the maze so that the robot could go several intersections deep into a particular path before detecting an error. But we'll leave that complication for a future challenge.)
- Presolving the maze for the robot is not permitted. It must show learning behaviour by venturing down some incorrect branches and recovering.
- We recommend that the guidance program include
  - An *alive* indicator
  - Readout of battery voltage
  - Some indication of the current state of the robot

You are encouraged to include readouts for the sensor signals (either the raw data or the thresholded values indicated as *light* and *dark*). It may also be useful to show the state of the bumper switches.

- Demonstrations must be completed by the ending time of the scheduled lab in Week 13 of the term.
- Guidance methods are unrestricted. You may use *bang-bang* control, turning the motors on and off in response to various guider signals. Or you may use the line detector and variable motor speed to provide *proportional* control of heading.
- Robots are deemed to have *lost their way* when they collide with one of the boundaries of the challenge area.
- The robot hardware must not be modified in any way[2].
- Paper decorations may be mounted on the robot and held in place with *small* pieces of masking tape.
- A printed copy of the properly documented assembly language program `guidance.asm` must be submitted at the time of the demonstration. The program will also be submitted electronically at the demo, using the *submit538* instruction. Some proportion of the final mark will be allocated for the quality of the documentation.
- The robot is started by loading its program, placing it in the *Start* area, and touching one of the bumpers.
- Once it is started, no human assistance of the robot is permitted. It's on its own.
- Each demonstration is allowed two tries.
- If the demonstration fails and time permits, the student will be allowed to revise their program.
- At the discretion of the instructor, the instructor may require a demonstration on the bench that the robot has some chance of operating properly.
- Part marks are awarded in the case of robots that can exhibit some functionality that is relevant to this problem. (See section 4).

3 Solving the Maze

Here are some thoughts on solving the maze.

- The robot can detect an intersection because the guider pattern detectors will sense that the outrigger sensors B and/or D are activated in addition to the centre sensors A and C.
- When the robot is started, it may be initialized with the knowledge that it is facing East. It should be able to deduce a new heading based on it's current direction and the direction of the turn. For example, if it makes a right turn at intersection 1, it is now going South.
- Each time it detects an intersection, it should note that it has found a new intersection, giving it a number. The robot must must then make a decision which path to follow. During the exploration phase, this choice can be made randomly. The maze is designed so that there are never more than 2 possibilities at any intersection. The robot should note what direction it has chosen to pursue.
- The data structure to contain this information could be a numbered list in which the list entry number represents the intersection number and the data entered represents the direction chosen. There are 7 or so intersections and four possible directions: North, South, East and West. So a 7 byte list would hold the Maze Solution data.

[2] A possible exception to this rule may be required if we find that room lighting interferes with robot guidance. In that case, it may be necessary to provide skirts on the guider section. However, this is not expected to be a problem.

- If the robot next comes to a dead end, the front bumper will be actuated. In that case, the robot should execute a 180^o turn and retrace its path back to the previous intersection. It should then note the corrected decision at that intersection and follow the correct branch.

- If the robot next comes to an intersection, the previous decision was correct and the robot can continue.

  For example, suppose the robot enters the maze shown in figure 1 and encounters intersection # 1. It detects the branching path to the South and understands that it has a choice of continuing East or turning South. Randomly, it choses to continue East (which is incorrect).

  It then encounters the object block that terminates a branch and realizes that it has made a mistake. It must retrace its path by turning 180^o and driving West. When it reaches intersection #1 again, it knows that it has to take the other branch which is to the South. Because it is now facing West, this is a left turn. The robot would enter South as the corrected destination at intersection #1 in the Maze Solution list.

  So there must be a data structure that tells the robot *if we are going East, a right turn faces us South*. It would have 4×3 entries, as shown in the following table:

Current Direction	Right Turn	Straight Ahead	Left Turn
North	East	North	West
West	North	West	South
etc			

  Similarily, there should be another data structure that tells the robot *if we are going East and want to go South at the next intersection, that is a right turn.*

  (Alternatively, you might code into the program an understanding that directions sequence North, East, South, West, North ... as the robot rotates clockwise, ie, executes right turns.)

- This process of exploring the maze and documenting the correct decision at each intersection continues until the robot reaches the *Forward Destination* point. The operator triggers the rear bumper.

- The robot now retraces its path. Because it has recorded the correct decision at each intersection, it should be possible for it to retrace its path back to the entry point of the maze without error.

4 Development Strategy

This is not the kind of program that one sits down and writes from start to finish in one burst of activity. It is simply too complicated.

The problem must be decomposed into simpler sub-problems, and these sub-problems further decomposed, until each of the sub-problems is doable by a humanoid programmer. You can then write and test the code for this problem. Once the code is debugged, it has moved you systematically toward a final goal. This also provides achievements that will generate part marks.

Here are some possible sub-projects that contribute to the whole.

1. Get a small program going that actuates the *alive* indicator, shows the status of the guider, shows the status of the bumper switches.

2. Modify the program can so that the eebot motors can be started and stopped under bumper control.

3. Add in guider control, so the robot will track the line and navigate the first `S` turn. This will be worth part marks.

4. Add the 180^o rotation routine so that actuating the front bumper causes the robot to do an about-turn and retrace it's path. This will be worth part marks.

5. Add in intersection detection logic, so the robot can indicate when it has detected an intersection and the type of intersection. (You could put a message on the LCD each time an intersection is detected.)

6. Add in steering control logic so that the robot may be instructed to take the left, right or straight branch in an intersection. At this point you should be able to

   - follow the line and navigate the `S` turn.
   - detect an intersection and navigate one branch or the other.
   - detect an object block and execute a 180^o rotation.

   This is significant progress.

7. Create a data structure that the robot can read and use to determine the correct decision at each intersection. Load this up with the maze information and see if the robot can navigate the maze once it has been learned. This will be worth part marks.

8. Develop the artificial intelligence system that can learn from the maze and put the information in the navigation data structure. Develop it in such a way that it can be easily *bench-tested* with readouts and breakpoints.

   The first step is to construct the *algorithm* that will search the maze. This should be written out as pseudocode and then carefully checked against the maze layout to see if it acts correctly.

   The second step is to design the various data structures that support the program. Some of these structures are read-only, others will require writing information to them while the system is in operation. Carefully consider how to access the information in these data structures.

5 Structure of the Program

This is a complex software system. It will require interrupts, passing of parameters to subroutines and time-critical operations. The final program will be lengthy and include many subroutines. This is deliberate: one of the objectives of this course is to challenge you with a major software project that requires significant planning and design in addition to the production of assembly language code.

As you develop the design, you must have in mind the final structure of the program. It should include the following sections, preferably in the same order in your code listing as presented here:

- Equates (definitions) of microprocessor registers such as `TCNT`.
- Working storage area (the *data* segment), with definitions of variables (RAM locations), using `ds.b`, `ds.w`, `dc.b`, `dc.w` and `fcc` statements. This should have its own `org` directive.
- The *code* segment, with its own `org` directive. This is the computer program, starting with the initialization routines. In this section of the code, each initialization routine should be called with a `jsr` statement. Device interrupts and vectors should be initialized here.
- The last instruction in the initialization code is the `cli` statement, which turns on global interrupts.
- Following the initialization routines is the *main loop*. This starts with a label (*main* is a suitable choice) and ends with a `jmp main` instruction. The main loop includes subroutine calls to the various operational routines. For example, it would include one subroutine call to update the display and another to check the navigation state machine.

  For a program of this size, the initialization code and main loop should easily fit onto one page. It simplifies debugging to minimize the size of these code sections because it is easy to determine where breakpoints should be inserted and the precise sequence of code events.

- The rest of the code containes the system subroutines. Each subroutine should be visibly set off by some visual device such as extra space or a row of characters. Each subroutine must have a header that includes a title, description of the routine function, what is passed into the routine, and what it returns. The organization of the routines should be such that similar routines are grouped together and the simplest routines are last in the listing. The assembler should determine the address of each subroutine: do not use `org` statements in this section. If there are definitions that naturally go with a particular subroutine and no other routine, then you may wish to include the `equ` equate statements with the subroutine. Otherwise, put the equates at the top of the program.

  Variables that are used by subroutines should be grouped with all the other variables (`ds.b` statements etc) at the top of the listing. Why not put them with the subroutine? Because, ultimately, the code may be move to EEPROM or some other form of non-volatile memory and the read-write registers must be moved to or stay in some area of RAM. So the code and the working RAM areas should be kept separate and distinct.

  If you stick to the organization suggested here, this can be done quite easily because there is one `org` statement for the RAM working storage area and one `org` statement for the code. Moving the code is then primarily a matter of changing its `org` statement.

6 Hints on Developing a Large Software Project

- **Beware of Name Conflicts**. Because this is a big program and we do not have a linking loader[3] available to us, *every label is global in scope and must be distinct*. Consequently, you will need to be very careful to use labels that are different from each other. So a label name like `loop` is very risky - it's likely to be used somewhere else in the program.

  A good approach to this is to preface each label name of this type with a some characters that are unique to the subroutine. For example, in a display routine you would use the label `disp_exit` rather than `exit`.

  The CodeWarrior generates a *cross reference listing* of all variables used in the program by typinf <ls> in the command line. This is useful in determining what variable names you have used.

- **Back up files**. At the end of each session, or after important changes and bug fixes, be extremely careful that you have backed up your files and have printed listings of your work. It is unlikely that the network will lose your work – files are backed up extensively – but it is possible that you may inadvertently in the heat of battle delete some important file.

  You should have one directory called something like *development* where you do development work. This is where code is written, modified and tested. A second directory called *tested code* (or something similar) should contain the most recent working version of the software[4]. It is important that you not modify files in the *tested code* directory. If you can't get something to work and you feel that the harware is bad, for example, you should be able to return to a previous version of the program, one that was working in the past, and test that.

- **Number Versions**. It is good practice to give each version of the program its own revision number. It is quite frightening to have two versions of the project, both with the same name, and not know which one is the most recent. If the code changes, it gets a new revision number. This also tells you how many versions you constructed, which can be entertaining.

- **Instrument your code**. The program *instrumentation* consists of a routine (many of which you have already written) that displays the behaviour of the program to help with debugging and to reassure the operator that the program is functioning correctly.

  For example, if you want to check that a particular routine is being called at the correct time, you could write one character to the LCD on entry to the routine and a second character on leaving the subroutine.

[3] A *linking loader* allows each subroutine to be assembled separately and the variable names are private to that subroutine. Name conflicts are then much less likely.

[4] If you know RCS (*Revision Control System*) or SCCS (*Source Code Control System*) you may wish to use them on this project.

Or you could pulse the *alive* indicator. If the *alive* LED flashes or stays on continuously, the subroutine is being exercised.

Instrumentation is enormously helpful in identifying situations like *The program was in state 3 and when it entered the motor control routine it crashed*. This immediately narrows down the problem area. Without instrumentation, one has situations like, *It was working and then it stopped*, which is not much information at all.

- **Test early, test often**. There is no way a project of this magnitude will work properly the first time it is run. It will required debugging at every level from the simplest subroutine up to the main loop. A good strategy for a program of this size is to *design top down, code bottom up*. That is,
  - Design the overall architecture of the program (main loop, interrupts, subroutines) so that the big picture is in mind.
  - Start with the low-level routines that interface with the hardware and test them thoroughly.
  - Once sufficient low-level routines are constructed and tested, place them into the overall structure in such a way that a simple, basic program can operate. In the worst case, this can be demonstrated to the lab supervisor for part marks. In the best case, it will show that the basic design concept is correct. For example, you might start with a robot program that runs the wheel motors until the bumper detects a collision.
  - Add one feature, at any level of this working program. For example, you might add the subroutine to read the guider at the bottom level, and then modify the main loop to read the guider values and display them. Carefully test this modification.
  - Continue this process of *incremental development and test* until the project meets requirements.

7 Subsystems

To accomplish the project task, the *eebot* will require a number of subsystems integrated into a complete program. These include:

- **Navigation Manager** Software routines that manage the navigation of the robot, for example, during search mode for the guide track.
- **Guidance Routine** Routine that interprets guider readings and commands the motors appropriately when following the guide track.
- **Sensor Scanner** Reads the photodetector sensors and thresholds the readings to determine when a track is in view.
- **Motor Speed Control** Controls the relative speeds of the motors to steer the robot and execute turns.
- **Rotation Counters** Used when executing accurate turning manoevres.
- **Bumper Detectors** Detect when the robot has collided with an object.

8 Hints

As a result of the student blood, sweat, tears, successes and failures in doing this project, we offer up the following hints. These are hints that students provided in their final report on the project.

- *I should have started earlier. I underestimated the difficulty of doing this project and ran out of time.* This project cannot be done in a week or two. The most successful programs were the ones that started before the Grand Panic in the lab, and the student had lots of time to debug the inevitable problems.

- *Read the guider frequently.* This is a negative feedback system, and delays introduce instability into such a system. Consequently, the microprocessor should check the guider frequently, which turns out to be several times per second. A number of students had built a software delay into their LCD display routine, and this distracted the microprocessor from the guidance task so that it missed certain important guider events.

  With careful design of the software, it should be possible to update the display occasionally while checking the guider frequently.

- *Beware of the 'galloping state' problem.* Each time the processor makes a transition from one state to another, the state machine must verify that it has left the previous state completely. The state machine must check for any intermediate state before proceeding to the next major state. Otherwise, you will get a situation where the state machine apparently jumps from state 4 to state 6 without properly detecting state 5. It actually goes through state 5, but does so too quickly for the conditions to be detected. This bug is much easier to debug if the machine reports its current state on the LCD display.

  For example, this might be a problem when the microprocessor detects an intersection. The first time it detects the intersection, the micro notes it as intersection #1. But then the micro immediately detects the same intersection again (remember, the micro is executing tens of thousands of instructions per second) and thinks it has found another intersection, #2.

  The program must search for an intersection (that's #1) **and then the end of the intersection #1** before starting to look for the next intersection, which is #2. A state machine approach to this is very helpful.

- *Learn to use breakpoints.* The software breakpoint is the most powerful debugging tool that you have in assembly language. It can be used to determine what the machine was doing when things went wrong, or if a particular branch of code is being executed at all. Surprisingly many students did not really understand how to use breakpoints and so wasted many hours in trial-and-error attempts to fix bugs. *This program is too big and complicated for trial-and-error to be effective in finding problems.* You must be systematic, and you will need to use breakpoints.

- *Test the Hardware* If some particular aspect of the robot is not doing what it should, the first thing to check is that the hardware is working properly. Yeah, it's a pain if the hardware is broken, and it's not your fault. But identifying a hardware fault is a part of this discipline, so you should be able to do it. Not to fix the hardware, but to identify the problem and report it to the lab supervisor.

  - For example, we found that some hardware problems were traced to the bench power being disconnected at the floor outlet. Disconnect the ribbon cable from the robot and if all the LEDs go out, the power was leaking through the logic signals rather than coming through the power adaptor.
  - When testing the robot, ensure that the adaptor is plugged into the correct socket in the robot – the one at the bow (front).
  - It is possible to turn either motor ON and make it run in either direction, directly from the monitor program.
  - It is possible to read the A/D converter with a few instructions to the monitor. This allows checking the guider hardware.

- The question of the difference between `equ`, `dc.b` and `ds.b` is often not well understood, so here is a clarification:

  - `equ` (equate) is used to define something (as in `foo equ 24`) that is then stored in (becomes part of) the instruction with a statement like `ldaa #foo`. There are other ways to use `equ` as well, such as `bar equ foo+10`, which would make `bar` equal to 34.
  - `dc.b` (define constant byte) is used to define something and allocate storage space for it. For example, you could create a lookup table with a series of `dc.b` statements. eg:`lookup dc.b 01 02 03 04`. `dc.b` statements create data that is part of the completed program. In this case, you could use indexed addressing into this data structure to retrieve a data value.

- `ds.b` (define storage byte) simply reserves an area of memory, so the next instruction is assigned a location some distance further on. For example, for the code sequence:

```
      org  6000
foo   rmb  $10
      ldaa foo
      ldaa foo+2
```

  The program would begin at $6000. The value of `foo` would be $6000. There would be $10 (16) locations reserved, starting at $6000 and ending at $600F. Their contents are uninitialize, ie, could be anything. The `ldaa` statement would be assembled at address $6010. The statement `ldaa foo` would load the *contents* of location `foo` into the accumulator A. The statement `ldaa foo+2` would load the contents of location 6002 into the accumulator A. So if you load something from one of these $10 registers, you can't predict what will be in the accumulator. So you would probably store something into these locations before loading it.

- *The guider was a problem.* In the original design, the guider signals were marginal because (a) the light from one LED would leak into another photoresistor (the *crosstalk* problem), and (b) the photoresistor detectors were not close enough to the road surface. Most students overcame the difficulties and got the guider to work successfully, but it consumed more time and effort than it should have.

  But then, a completely new guider has been constructed and installed (at great expense and effort, by the way), and the operation is much more reliable. In the new design, the signal delta between light and dark levels is typically more than one volt. The operation is improved by having one LED light at a time, under control of the microprocessor, so there is no crosstalk. The LEDs are now mounted on the top of the guider board, which makes it possible to move the guider closer to the road surface. This improves the signal from the photoresistors.

  These hardware changes require completely new software to operate the guider. By the time you are working on the project, you should have a working guider routine that you developed or was provided by the lab instructor.

9 Documentation and Submission

When you have finished your project, whether it works completely or not, you must submit

- The source (.asm) file, properly commented, and formatted. The source file is submitted electronically using the usual method.
- A two page summary of your results on paper, which is given to your lab instructor. The two page summary should include:
  - A description of your completed project (ie, the main blocks of code) and what it was able to accomplish
  - A list of the main problems you encountered and how you dealt with them. For example, you might have had difficulty getting the software to work because you needed to use a different method of organization. I am particularly interested in hearing an insights, with respect to managing a major software project, that you may have acquired while doing this project. For example, would you approach another project the same way, or would you do it differently? What worked well and did not work? What areas were particularly difficult to construct or debug?

  The report does not need to be fancy. However, it must include your name, student number, and lab section number at the top of the document. The pages must be numbered and stapled together.

  This report and your software documentation will be graded and together they will count for 5% of the 538 lab mark.