ELE538 Microprocessor Systems
Lab Project
Robot Guidance Challenge

Peter Hiscocks
Department of Electrical and Computer Engineering
Ryerson Polytechnic University
phiscock@ee.ryerson.ca
August 25, 2002

Contents

1 Project Overview **1**

2 Rules of the Challenge Project **2**

3 Structure of the Program **3**

4 Hints on Developing a Large Software Project **4**

5 Subsystems **5**

List of Figures

1 Robot Guidance Challenge 2

1 Project Overview

The ELE538 Project is to equip the *eebot* mobile robot with a navigation system that can find and follow a line (the *track*) to a destination. A possible course with the various navigational challenges, is shown in figure 1.

1. The robot is started some distance from the track and must use *dead-reckoning* to drive in a straight line to find the track. To ensure that the robot is driving in a straight line, the revolutions for each wheel can be counted. Or you could trust to luck . . ..

2. The robot should detect the track, and then

3. make a precise right angle turn to line up with the track. Because the guider is located in front of the drive wheels, it will be necessary to advance the robot until its wheels straddle the track, and then execute a right turn. Alternatively, you could have the robot move forward and then spin in place to detect the new line.

4. These curves test the ability of the guider to track a serpentine path.

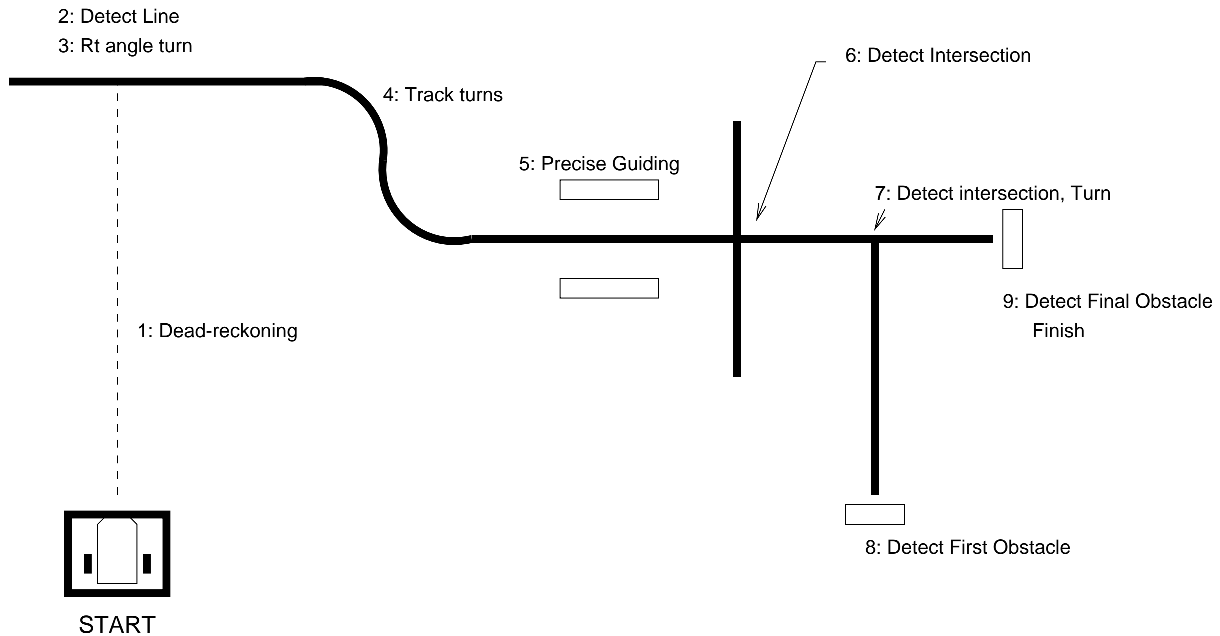

Figure 1: Robot Guidance Challenge

5. The robot must pass through a gap between two objects, precisely tracking the line. This section is designed to eliminate guidance methods that result in significant robot swinging over the guidance track.
6. The first intersection must be detected so that the robot knows where it is, but ignored for guidance purposes by continuing straight through the intersection.
7. At the second intersection, the robot must make a right angle turn and then pick up the new track.
8. The robot collides with the *first obstacle*, which trips the front bumper.
9. The robot turns through 108° and retraces its path back to the previous intersection. It makes a right turn and then collides with the *final obstacle*, which trips the front bumper again and finishes the challenge.

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 30
- The exact layout is subject to change, but will include features similar to those shown in figure 1. Students will be informed of any changes well in advance of the demo date.

- The guidance program must 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[1].

- 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.

3 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:

[1] 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.

- Equates (definitions) of microprocessor registers such as TCNTLO.
- Working storage area (the *data* segment), with definitions of variables (RAM locations), using FCB, ^FCC^,FCD and RMB 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 (RMB 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.

4 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[2] 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.

[2] 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.

With the `-c` option, the assembler `as11` can be directed to generate a *cross reference listing* of all variables used in the program. 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[3]. 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. 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.

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

  - 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.

5 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.