6.115 Final Project – Code Listing

Nada Amin
namin@mit.edu

May 2006

R31JP serial interface (minmon.asm)

I simply extended MINMON with commands F, B, L, K, H, S which instruct the robot to move around or send sensory information.

```
;    ***********************************************
;    *                                             *
;    *  MINMON - The Minimal 8051 Monitor Program  *
;    *                                             *
;    *  Portions of this program are courtesy of   *
;    *  Rigel Corporation, of Gainesville, Florida *
;    *                                             *
;    *  Modified for 6.115                         *
;    *  Massachusetts Institute of Technology      *
;    *  January, 2001  Steven B. Leeb              *
;    *                                             *
;    ***********************************************
stack   equ  2fh             ; bottom of stack
                             ; - stack starts at 30h -
errorf  equ   0              ; bit 0 is error status
;=============================================================
; 8032 hardware vectors
;=============================================================
    org 00h                  ; power up and reset vector
    ljmp start
    org 03h                  ; interrupt 0 vector
    ljmp start
    org 0bh                  ; timer 0 interrupt vector
    ljmp start
    org 13h                  ; interrupt 1 vector
    ljmp start
    org 1bh                  ; timer 1 interrupt vector
    ljmp start
    org 23h                  ; serial port interrupt vector
```

```
    ljmp start
    org 2bh                 ; 8052 extra interrupt vector
    ljmp start
;=========================================================
; begin main program
;=========================================================
    org     100h
start:
    clr     ea              ; disable interrupts
    lcall   init
    lcall   initadc
; initialize hardware
    lcall   print           ; print welcome message
    db 0ah, 0dh, "Welcome to 6.115!", 0ah, 0dh,"MINMON> ", 0h
monloop:
    mov     sp,#stack       ; reinitialize stack pointer
    lcall   initperiports   ; init or reinit the peri port chip
                            ; just in case
    clr     ea              ; disable all interrupts
    clr     errorf          ; clear the error flag
    lcall   print           ; print prompt
    db 0dh, 0ah,"*", 0h
    clr     ri              ; flush the serial input buffer
    lcall   getcmd          ; read the single-letter command
    mov     r2, a           ; put the command number in R2
    ljmp    nway            ; branch to a monitor routine
endloop:                    ; come here after command has finished
    sjmp monloop            ; loop forever in monitor loop
;=========================================================
; subroutine init
; this routine initializes the hardware
; set up serial port with a 11.0592 MHz crystal,
; use timer 1 for 9600 baud serial communications
;=========================================================
init:
    mov   tmod, #20h        ; set timer 1 for auto reload - mode 2
    mov   tcon, #41h        ; run counter 1 and set edge trig ints
    mov   th1,  #0fdh       ; set 9600 baud with xtal=11.059mhz
    mov   scon, #50h        ; set serial control reg for 8 bit data
                            ; and mode 1
    ret
;=========================================================
; monitor jump table
;=========================================================
jumtab:
    dw badcmd               ; command '@' 00
```

```
    dw badcmd               ; command 'a' 01
    dw gobackward           ; command 'b' 02 used
    dw badcmd               ; command 'c' 03
    dw downld               ; command 'd' 04 used
    dw badcmd               ; command 'e' 05
    dw goforward            ; command 'f' 06 used
    dw goaddr               ; command 'g' 07 used
    dw robothalt            ; command 'h' 08 used
    dw badcmd               ; command 'i' 09
    dw badcmd               ; command 'j' 0a
    dw goright              ; command 'k' 0b used
    dw goleft               ; command 'l' 0c used
    dw badcmd               ; command 'm' 0d
    dw badcmd               ; command 'n' 0e
    dw badcmd               ; command 'o' 0f
    dw badcmd               ; command 'p' 10
    dw badcmd               ; command 'q' 11
    dw readmem              ; command 'r' 12 used
    dw sensor               ; command 's' 13 used
    dw writab               ; command 't' 14 used
    dw badcmd               ; command 'u' 15
    dw badcmd               ; command 'v' 16
    dw writmem              ; command 'w' 17 used
    dw badcmd               ; command 'x' 18
    dw badcmd               ; command 'y' 19
    dw badcmd               ; command 'z' 1a

;*****************************************************************
; monitor command routines
;*****************************************************************
;================================================================
; command readmem   'r'
; this routine reads & prints the hexadecimal content of the byte
; in external memory of which the location is the 4 hex digit
; address which follows the command
;================================================================
readmem:
    lcall getbyt            ; get address high byte
    lcall prthex
    mov dph, a              ; save in dph
    lcall getbyt            ; get address low byte
    lcall prthex
    mov dpl, a              ; save in dpl
    lcall crlf
    movx a, @dptr           ; read from ext mem
    lcall prthex            ; print byte to user
```

```
   ljmp   endloop          ; return
;=============================================================
; command writmem   'w'
; this routine writes a byte to a location in external memory
; the format is wXXXX=YY where XXXX is the 4 hex digit address
; of the location and YY is the hexadecimal content of the byte
;=============================================================
writmem:
   lcall getbyt            ; get address high byte
   lcall prthex
   mov dph, a              ; save in dph
   lcall getbyt            ; get address low byte
   lcall prthex
   mov dpl, a              ; save in dpl
   lcall getchr            ; skip the = character
   lcall sndchr
   lcall getbyt            ; get the byte content
   lcall prthex
   movx @dptr,a            ; write it to ext mem
   ljmp   endloop          ; return
;=============================================================
; command writab   't'
; this routine writes a table of bytes to external memory
; the table prompts for the fist byte to write at address XX00,
; where XX follows the command name
; entry stops after 256 bytes or if the user hits return without
; entering any data
;=============================================================
writab:
   lcall getbyt            ; get address high byte
   lcall prthex
   mov dph, a              ; save in dph
   mov dpl, #00h           ; LSB=00 a start
writabloop:
   lcall crlf
   mov a, dph              ; print MSB
   lcall prthex
   mov a, dpl              ; print LSB
   lcall prthex
   mov a, #20h             ; print space
   lcall sndchr
   lcall getbyt            ; get the byte content
   lcall prthex
   movx @dptr,a            ; write it to ext mem
   mov a, dpl              ; increment LSB
   inc a
```

```
   mov dpl, a
   jnz writabloop          ; loop again for next entry
   ljmp endloop            ; return
;=============================================================
; command goaddr  'g'
; this routine branches to the 4 hex digit address which follows
;=============================================================
goaddr:
   lcall getbyt            ; get address high byte
   mov   r7, a             ; save in R7
   lcall prthex
   lcall getbyt            ; get address low byte
   push  acc               ; push lsb of jump address
   lcall prthex
   lcall crlf
   mov   a, r7             ; recall address high byte
   push  acc               ; push msb of jump address
   ret                     ; do jump by doing a ret
;=============================================================
; command downld  'd'
; this command reads in an Intel hex file from the serial port
; and stores it in external memory.
;=============================================================
downld:
   lcall crlf
   mov   a, #'>'           ; acknowledge by a '>'
   lcall sndchr
dl:
   lcall getchr            ; read in ':'
   cjne  a, #':', dl
   lcall getbytx           ; get hex length byte
   jz    enddl             ; if length=0 then return
   mov   r0, a             ; save length in r0
   lcall getbytx           ; get msb of address
   setb  acc.7             ; make sure it is in RAM
   mov   dph, a            ; save in dph
   lcall getbytx           ; get lsb of address
   mov   dpl, a            ; save in dpl
   lcall getbytx           ; read in special purpose byte (ignore)
dloop:
   lcall getbytx           ; read in data byte
   movx  @dptr, a          ; save in ext mem
   inc   dptr              ; bump mem pointer
   djnz  r0, dloop         ; repeat for all data bytes in record
   lcall getbytx           ; read in checksum
   mov   a, #'.'
```

```
   lcall sndchr            ; handshake '.'
   sjmp  dl                ; read in next record
enddl:
   lcall getbytx           ; read in remainder of the
   lcall getbytx           ; termination record
   lcall getbytx
   lcall getbytx
   mov   a, #'.'
   lcall sndchr            ; handshake '.'
   ljmp  endloop           ; return
getbytx:
   lcall getbyt
   jb    errorf, gb_err
   ret
gb_err:
   ljmp badpar

;*****************************************************************
; monitor support routines
;*****************************************************************
badcmd:
   lcall print
   db 0dh, 0ah," bad command ", 0h
   ljmp endloop
badpar:
   lcall print
   db 0dh, 0ah," bad parameter ", 0h
   ljmp endloop
;================================================================
; subroutine getbyt
; this routine reads in an 2 digit ascii hex number from the
; serial port. the result is returned in the acc.
;================================================================
getbyt:
   lcall getchr            ; get msb ascii chr
   lcall ascbin            ; conv it to binary
   swap  a                 ; move to most sig half of acc
   mov   b, a              ; save in b
   lcall getchr            ; get lsb ascii chr
   lcall ascbin            ; conv it to binary
   orl   a, b              ; combine two halves
   ret
;================================================================
; subroutine getcmd
; this routine gets the command line.  currently only a
; single-letter command is read - all command line parameters
```

```
; must be parsed by the individual routines.
;
;=============================================================
getcmd:
    lcall getchr            ; get the single-letter command
    clr   acc.5             ; make upper case
    lcall sndchr            ; echo command
    clr   C                 ; clear the carry flag
    subb  a, #'@'           ; convert to command number
    jnc   cmdok1            ; letter command must be above '@'
    lcall badpar
cmdok1:
    push  acc               ; save command number
    subb  a, #1Bh           ; command number must be 1Ah or less
    jc    cmdok2
    lcall badpar            ; no need to pop acc since badpar
                            ; initializes the system
cmdok2:
    pop   acc               ; recall command number
    ret
;=============================================================
; subroutine nway
; this routine branches (jumps) to the appropriate monitor
; routine. the routine number is in r2
;=============================================================
nway:
    mov   dptr, #jumtab     ;point dptr at beginning of jump table
    mov   a, r2             ;load acc with monitor routine number
    rl    a                 ;multiply by two.
    inc   a                 ;load first vector onto stack
    movc  a, @a+dptr        ;          "          "
    push  acc               ;          "          "
    mov   a, r2             ;load acc with monitor routine number
    rl    a                 ;multiply by two
    movc  a, @a+dptr        ;load second vector onto stack
    push  acc               ;          "          "
    ret                     ;jump to start of monitor routine


;*************************************************************
; general purpose routines
;*************************************************************
;=============================================================
; subroutine sndchr
; this routine takes the chr in the acc and sends it out the
; serial port.
```

```
;=============================================================
sndchr:
   clr  scon.1              ; clear the tx  buffer full flag.
   mov  sbuf,a              ; put chr in sbuf
txloop:
   jnb  scon.1, txloop      ; wait till chr is sent
   ret
;=============================================================
; subroutine getchr
; this routine reads in a chr from the serial port and saves it
; in the accumulator.
;=============================================================
getchr:
   jnb  ri, getchr          ; wait till character received
   mov  a,  sbuf            ; get character
   anl  a,  #7fh            ; mask off 8th bit
   clr  ri                  ; clear serial status bit
   ret
;=============================================================
; subroutine print
; print takes the string immediately following the call and
; sends it out the serial port.  the string must be terminated
; with a null. this routine will ret to the instruction
; immediately following the string.
;=============================================================
print:
   pop   dph                ; put return address in dptr
   pop   dpl
prtstr:
   clr   a                  ; set offset = 0
   movc  a,  @a+dptr        ; get chr from code memory
   cjne  a,  #0h, mchrok    ; if termination chr, then return
   sjmp  prtdone
mchrok:
   lcall sndchr             ; send character
   inc   dptr               ; point at next character
   sjmp  prtstr             ; loop till end of string
prtdone:
   mov   a,  #1h            ; point to instruction after string
   jmp   @a+dptr            ; return
;=============================================================
; subroutine crlf
; crlf sends a carriage return line feed out the serial port
;=============================================================
crlf:
   mov   a,  #0ah           ; print lf
```

```
   lcall sndchr
cret:
   mov   a, #0dh          ; print cr
   lcall sndchr
   ret
;=============================================================
; subroutine prthex
; this routine takes the contents of the acc and prints it out
; as a 2 digit ascii hex number.
;=============================================================
prthex:
   push acc
   lcall binasc           ; convert acc to ascii
   lcall sndchr           ; print first ascii hex digit
   mov   a, r2            ; get second ascii hex digit
   lcall sndchr           ; print it
   pop acc
   ret
;=============================================================
; subroutine binasc
; binasc takes the contents of the accumulator and converts it
; into two ascii hex numbers.  the result is returned in the
; accumulator and r2.
;=============================================================
binasc:
   mov   r2, a            ; save in r2
   anl   a, #0fh          ; convert least sig digit.
   add   a, #0f6h         ; adjust it
   jnc   noadj1           ; if a-f then readjust
   add   a, #07h
noadj1:
   add   a, #3ah          ; make ascii
   xch   a, r2            ; put result in reg 2
   swap  a                ; convert most sig digit
   anl   a, #0fh          ; look at least sig half of acc
   add   a, #0f6h         ; adjust it
   jnc   noadj2           ; if a-f then re-adjust
   add   a, #07h
noadj2:
   add   a, #3ah          ; make ascii
   ret

;=============================================================
; subroutine ascbin
; this routine takes the ascii character passed to it in the
; acc and converts it to a 4 bit binary number which is returned
```

```
; in the acc.
;=============================================================
ascbin:
   clr    errorf
   add    a, #0d0h          ; if chr < 30 then error
   jnc    notnum
   clr    c                 ; check if chr is 0-9
   add    a, #0f6h          ; adjust it
   jc     hextry            ; jmp if chr not 0-9
   add    a, #0ah           ; if it is then adjust it
   ret
hextry:
   clr    acc.5             ; convert to upper
   clr    c                 ; check if chr is a-f
   add    a, #0f9h          ; adjust it
   jnc    notnum            ; if not a-f then error
   clr    c                 ; see if char is 46 or less.
   add    a, #0fah          ; adjust acc
   jc     notnum            ; if carry then not hex
   anl    a, #0fh           ; clear unused bits
   ret
notnum:
   setb   errorf            ; if not a valid digit
   ljmp   endloop


;=============================================================
; mon_return is not a subroutine.
; it simply jumps to address 0 which resets the system and
; invokes the monitor program.
; A jump or a call to mon_return has the same effect since
; the monitor initializes the stack.
;=============================================================
mon_return:
   ljmp   0

;=============================================================
; initperiports
; initializes the 8255 (on address FE01xh) for writing on all
; ports
;=============================================================
initperiports:
   mov dptr, #0FE13h        ; write control word
   mov a, #80h
   movx @dptr, a
   ret
```

```
;=============================================================
; initadc
; initializes the A to D converter (on address FE0xh)
;=============================================================
initadc:
   mov dptr, #0FE00h        ; simply write and read from AD
   mov a, #0                ; to initialize interrupt position (P3.2)
   movx @dptr, a
   mov dptr, #0FE00h
   movx a, @dptr
   ret

;=============================================================
; command goforward  'f'
; instruct the robot to go forward
; motors are connected to port B
; PB0 = L+
; PB1 = L-
; PB2 = R+
; PB3 = R-
;=============================================================
goforward:
   mov dptr, #0FE11h
   mov a, #05h
   movx @dptr, a
   ljmp  endloop            ; return

;=============================================================
; command gobackward  'b'
; instruct the robot to go backward
;=============================================================
gobackward:
   mov dptr, #0FE11h
   mov a, #0Ah
   movx @dptr, a
   ljmp  endloop            ; return

;=============================================================
; command goleft  'l'
; instruct the robot to go left
;=============================================================
goleft:
   mov dptr, #0FE11h
   mov a, #06h
   movx @dptr, a
```

```
   ljmp   endloop            ; return

;=============================================================
; command goright   'k'
; instruct the robot to go right
;=============================================================
goright:
   mov dptr, #0FE11h
   mov a, #09h
   movx @dptr, a
   ljmp   endloop            ; return


;=============================================================
; command robothalt   'h'
; instruct the robot to halt
;=============================================================
robothalt:
   mov dptr, #0FE11h
   mov a, #00h
   movx @dptr, a
   ljmp   endloop            ; return

;=============================================================
; command sensor   's'
; returns a hexadecimal reading of the sensor x value,
; where x is a digit after the s command
; x=0 (front)
; x=1 (back)
; x=2 (left)
; x=3 (right)
; Note that the 74HC4d67 (16-channel analog (de)multiplexer)
; is connected as follows:
; PA0 = S0
; PA1 = S1
; PA2 = S2
; PA3 = S3
; PA4 = !E
; Z goes to A/D input
;=============================================================
sensor:
   lcall getchr              ; get ascii chr
   lcall sndchr              ; send it back
   lcall ascbin              ; conv it to binary
   mov dptr, #0FE10h         ; configure demultiplexer for
   movx @dptr, a             ; this sensor
```

```
    lcall readadc           ; read digital value
    lcall prthex            ; print it
    ljmp  endloop           ; return

;=========================================================
; readadc
; Reads the value of the A/D and stores it in the accumulator.
;=========================================================
readadc:
    mov dptr, #0FE00h     ; Start conversion
    mov a, #0
    movx @dptr, a
    wait:                 ; Wait for conversion to complete.
        jb P3.2, wait
    mov dptr, #0FE00h     ; Read digital value.
    movx a, @dptr
    ret

; end of MINMON
```

Python library serial interface (librobot.py)

librobot.py provides the basic functionality to command the robot from Python. It uses pySerial, a Python module to access the serial port, freely available at http://pyserial.sourceforge.net.

```
"""Python Library for 6.115 Final Project
May 2006
Nada Amin (namin@mit.edu)

This library interfaces with the R31JP using the serial port and
commands the robot to move around or to send sensor information.
"""
import serial
import sys

# constants for both directions and sensors
BACK = 'b'
FRONT = 'f'
LEFT = 'l'
RIGHT = 'r'
STOP = 's'

# list of all possible constants for direction commands
GO_LST = [FRONT, LEFT, RIGHT, BACK, STOP]
```

```
# table that maps a constant to a direction command
GO_TABLE = {FRONT : 'F', BACK : 'B', LEFT : 'L', RIGHT : 'K', STOP : 'H'}

# table that maps a direction to constant to its opposite
REVERSE_DIRS = {FRONT : BACK, BACK : FRONT, LEFT : RIGHT, RIGHT : LEFT, STOP : STOP}

# list of all possible constants for sensory position
SENSOR_LST = [FRONT, LEFT, RIGHT, BACK]
# table that maps a constant to a sensory position
SENSOR_TABLE = {FRONT : '0', BACK : '1', LEFT: '2', RIGHT : '3'}

PRETTY_SENSORS = {FRONT : 'front', LEFT: 'left', RIGHT : 'right', BACK: 'back'}
PRETTY_DIRS = {FRONT: 'forward', LEFT: 'left', RIGHT: 'right', BACK: 'backward'}

NUM_SENSORS = len(SENSOR_LST)

class Robot:
    """The Robot class defines the basic library interface to the
    R31JP-controlled robot."""
    def __init__(self):
        """Initializes the serial port (will propage the errors from
        the serial library)."""
        self.tty = serial.Serial(0)

    def _ttyread(self):
        """Waits for a message on the serial port and returns it.
        Note that this procedure is not robust, because it might wait
        forever if no message comes.
        """
        while self.tty.inWaiting()==0:
            pass
        return self.tty.read(self.tty.inWaiting())

    def go(self, x):
        """Commands robot to go in the given direction (or stop)."""
        self.tty.write(GO_TABLE[x])
        out = self._ttyread()
        return out == GO_TABLE[x]+'\r\n*'

    def go_forward(self):
        """Commands the robot to go forward."""
        return self.go(FRONT)

    def go_backward(self):
        """Commands the robot to go backward."""
        return self.go(BACK)
```

```
    def go_left(self):
        """Commands the robot to go left."""
        return self.go(LEFT)

    def go_right(self):
        """Commands the robot to go right."""
        return self.go(RIGHT)

    def stop(self):
        """Commands the robot to stop."""
        return self.go(STOP)

    def sensor(self, x):
        """Returns the sensory value for the given sensor position."""
        self.tty.write('s'+SENSOR_TABLE[x])
        out = self._ttyread()
        val = out[-5:-3]
        return int('0x'+val, 16)

    def obstacle(self, x):
        """Returns whether there is an obstacle at the given sensor
        position."""
        return self.sensor(x) >= 80

    def obstacle_front(self):
        """Returns whether there is an obstacle on the front."""
        return self.obstacle(FRONT)

    def obstacle_back(self):
        """Returns whether there is an obstacle on the back."""
        return self.obstacle(BACK)

    def obstacle_left(self):
        """Returns whether there is an obstacle on the left."""
        return self.obstacle(LEFT)

    def obstacle_right(self):
        """Returns whether there is an obstacle on the right."""
        return self.obstacle(RIGHT)

    def go_until_obstacle(self, x, txt=''):
        """Goes in the given direction, until the sensor in that
        direction detects an obstacle, and stops. txt is an optional
        message to be printed at every round.
        """
```

```
        self.go(x)
        while not self.obstacle(x):
            sys.stdout.write(txt)
            sys.stdout.flush()
        self.stop()

    def go_forward_until_obstacle(self, txt=''):
        """Moves forward until there is an obstacle on the front, then
        stops."""
        self.go_until_obstacle(FRONT, txt)

    def go_backward_until_obstacle(self, txt=''):
        """Moves backward until there is an obstacle on the back, then
        stops."""
        self.go_until_obstacle(BACK, txt)

    def go_left_until_obstacle(self, txt=''):
        """Moves left until there is an obstacle on the left, then
        stops."""
        self.go_until_obstacle(LEFT, txt)

    def go_right_until_obstacle(self, txt=''):
        """Moves right until there is an obstacle on the right, then
        stops."""
        self.go_until_obstacle(RIGHT, txt)
```

Sample programs (`demorobot.py`)

`demorobot.py` showcases a few sample programs. In particular, the `DemoRobot` class extends the library interface to record and replay or undo sequences of moves, while also optionally avoiding obstacles.

```
"""Sample programs that use the librobot library.
May 2006
Nada Amin (namin@mit.edu)
"""
import librobot
import sys
import time

def ex1():
    """Simply go forward and stop when an obstacle is detected in
    front."""
    print ex1.func_doc
    robot = librobot.Robot()
```

```
    if robot.obstacle_front():
        print 'obstacle front'
    else:
        print 'no obstacle front'
        print 'moving forward'
        sys.stdout.flush()
        robot.go_forward()
        while not robot.obstacle_front():
            print '.',
            sys.stdout.flush()
        print
        print 'obstacle front'
        robot.stop()
        print 'stopping'
    print 'bye bye'
    robot.tty.close()

def ex2():
    """Avoids obstacle. Only stops when surrounded by obstacles.
    Looks and moves in the order of librobot.SENSOR_LST.
    """
    print ex2.func_doc
    robot = librobot.Robot()
    # for printing reasonable and non-redundant information
    # prev: index of the sensor without an obstacle on the previous round
    prev = -1
    # nl: whether we need a newline for pretty printing
    # (this could happen after we print a dot and then comes some text)
    nl = False
    while True:
        # stop_now: keeps track of whether there are obstacles all around
        stop_now = True
        # go through all sensors and check for obstacles
        for i in range(0, librobot.NUM_SENSORS):
            x = librobot.SENSOR_LST[i]
            if robot.obstacle(x):
                # obstacle... try the next sensor
                if prev<=i:
                    if nl:
                        print
                        nl = False
                    print 'obstacle', librobot.PRETTY_SENSORS[x]
            else:
                # no obstacle... this is where we're heading
                if prev!=i:
                    if nl:
```

```
                    print
                    nl = False
                print 'no obstacle', librobot.PRETTY_SENSORS[x]
                print 'going', librobot.PRETTY_DIRS[x]
                robot.go(x)
                prev = i
            else:
                # same as previous round
                print '.',
                nl = True
            stop_now = False
            break
        # there are obstacles in all directions, so stop
        if stop_now:
            print 'Obstacles all around.'
            print 'Just stop.'
            robot.stop()
            break
        sys.stdout.flush()
    robot.tty.close()

def ex3():
    """Goes in each sensor direction until obstacle, in the order of
    librobot.SENSOR_LST.  Then stops."""
    print ex3.func_doc
    sys.stdout.flush()
    robot = librobot.Robot()
    for i in range(0, librobot.NUM_SENSORS):
        x = librobot.SENSOR_LST[i]
        if robot.obstacle(x):
            print 'obstacle', librobot.PRETTY_SENSORS[x]
            sys.stdout.flush()
            continue
        else:
            print 'going', librobot.PRETTY_DIRS[x]
            sys.stdout.flush()
            robot.go(x)
            while not robot.obstacle(x):
                print '.',
                sys.stdout.flush()
            robot.stop()
            print
            print 'obstacle', librobot.PRETTY_SENSORS[x]
            sys.stdout.flush()
    print 'Stop.'
    print 'bye bye'
```

```
    robot.tty.close()

def ex3b():
    """Goes in each sensor direction until obstacle, in the order of
    librobot.SENSOR_LST. Then stops."""
    print ex3b.func_doc
    sys.stdout.flush()
    robot = librobot.Robot()
    for i in range(0, librobot.NUM_SENSORS):
        x = librobot.SENSOR_LST[i]
        if robot.obstacle(x):
            print 'obstacle', librobot.PRETTY_SENSORS[x]
            sys.stdout.flush()
            continue
        else:
            print 'going', librobot.PRETTY_DIRS[x]
            sys.stdout.flush()
            robot.go_until_obstacle(x, '. ')
            print
            print 'obstacle', librobot.PRETTY_SENSORS[x]
            sys.stdout.flush()
    print 'Stop.'
    print 'bye bye'
    robot.tty.close()

def ex4():
    """Follows the obstacle. If there is more than one obstacle,
    stops."""
    print ex4.func_doc
    sys.stdout.flush()
    robot = librobot.Robot()
    # prev: tracks the index of the obstacle in the previous round
    prev = -1
    # nl: tracks whether we need a newline for pretty printing
    nl = False
    while True:
        obstacles = map(lambda x: robot.obstacle(x), librobot.SENSOR_LST)
        # if there's more than one obstacle, quit
        true_obstacles = filter(lambda x: x, obstacles)
        if len(true_obstacles)>1:
            robot.stop()
            if nl:
                print
                nl = False
            print 'More than one obstacle.'
            print 'Bye bye!'
```

```
            break
        # update move according to where the obstacle occurred
        # note that i==NUM_SENSORS means that no obstacle occurred
        for i in range(0, librobot.NUM_SENSORS+1):
            if i==librobot.NUM_SENSORS or obstacles[i]:
                if prev==i:
                    # same as previous round
                    print '.',
                    nl = True
                else:
                    # different from previous round
                    if nl:
                        print
                        nl = False
                    if i<librobot.NUM_SENSORS:
                        x = librobot.SENSOR_LST[i]
                        print 'going', librobot.PRETTY_DIRS[x]
                        robot.go(x)
                    else:
                        print 'stopping'
                        robot.stop()
                    prev = i
                sys.stdout.flush()
                break

class DemoRobot(librobot.Robot):
    """The DemoRobot class extends librobot.Robot to perform
    sophisticated actions, requiring memory of the past."""

    def __init__(self):
        librobot.Robot.__init__(self)
        self.seq = None

    def _do_move(self, x, prev):
        """(Re-factored) Performs move x and prints appropriate
        progress information."""
        if prev!=x:
            # different from previous round
            print
            if x!=librobot.STOP:
                print 'going', librobot.PRETTY_DIRS[x]
                self.go(x)
            else:
                print 'stopping'
                self.stop()
        print '.',
```

```
    def _do_move_if_no_obstacle(self, x, prev):
        """(Re-factored) Performs move x if there is no obstacle and
        prints appropriate progress information."""
        if prev!=x:
            # different from previous round
            print
            if x!=librobot.STOP:
                print 'going', librobot.PRETTY_DIRS[x]
            else:
                print 'stopping'
        if x!=librobot.STOP and self.obstacle(x):
            self.stop()
            print 'o',
        else:
            self.go(x)
            print '.',

    def record_moves(self):
        """Follows the obstacle.  Stops if there is more than one
        obstacle.  Records the sequence of moves thus generated."""
        # the record of moves
        self.seq = []
        # prev: tracks the move of the previous round
        prev = None
        while True:
            obstacles = map(lambda x: self.obstacle(x), librobot.SENSOR_LST)
            # if there's more than one obstacle, quit
            true_obstacles = filter(lambda x: x, obstacles)
            if len(true_obstacles)>1:
                self.stop()
                print
                print 'More than one obstacle.'
                print 'Bye bye!'
                break
            # update move according to where the obstacle occurred
            # note that i==NUM_SENSORS means that no obstacle occurred
            for i in range(0, librobot.NUM_SENSORS+1):
                if i==librobot.NUM_SENSORS or obstacles[i]:
                    if i==librobot.NUM_SENSORS:
                        x = librobot.STOP
                    else:
                        x = librobot.SENSOR_LST[i]
                    self.seq.append(x)
                    self._do_move(x, prev)
                    prev = x
```

```
                    sys.stdout.flush()
                    break

    def type_moves(self):
        """Type out the moves on the screen using f(orward),
        b(ackward), l(eft), r(ight), s(top). Any other key quits. The
        moves are recorded as a sequence for playback."""
        self.seq = []
        prev = None
        tstart = None
        while True:
            try:
                x = raw_input("f(orward), b(ackward), r(ight), l(eft), s(top)")
            except KeyboardInterrupt:
                break
            if prev!=None:
                tend = time.time()
                tdur = int(round((tend - tstart)*10))
                self.seq += [prev]*tdur
            if x in librobot.GO_LST:
                self.go(x)
                tstart = time.time()
                prev = x
            else:
                break
        self.stop()
        print 'Bye bye!'

    def playback_moves(self, skipStops=False, avoidObstacles=False, curseq=None):
        """Replays the curseq or the last recorded sequence of moves."""
        if curseq==None and self.seq==None:
            print 'No sequence has been recorded.'
            return
        if curseq==None:
            curseq = self.seq
        # prev: tracks the move of the previous round
        prev = None
        for x in curseq:
            if not skipStops or x!=librobot.STOP:
                # sleep a bit to simulate the time of reading the sensors, etc.
                time.sleep(0.1)
                # update move according to the sequence of moves
                if avoidObstacles:
                    self._do_move_if_no_obstacle(x, prev)
                else:
                    self._do_move(x, prev)
```

```
                prev = x
                sys.stdout.flush()
        self.stop()
        print
        print 'Done.'
        print 'Bye bye!'

    def undo_moves(self, skipStops=False, avoidObstacles=False):
        """Undo moves of last recorded sequence, by performing both a
        time and direction reversal."""
        if self.seq==None:
            print 'No sequence has been recorded.'
            return
        rev = map(lambda x: librobot.REVERSE_DIRS[x], self.seq)
        rev.reverse()
        self.playback_moves(skipStops, avoidObstacles, rev)
```