

BB3

Tak Auyeung

July 31, 2003

Bugs and potential problems...

- 300 Ω value swapped with 100k Ω at top of board
- VDD and signal (top two pins) swapped at servo connector
- 10k Ω resistor between VCC and RESET missing
- 100k Ω pull-up resistor may be too high, try 10k or 20k
- bend pin 8 of the voltage regulator out side to avoid unnecessary resets (due to voltage dip)

Features:

- two servo connectors
- two bottom sensor connectors (reflective type)
- three forward/side sensor connectors (ranging type)
- LCD port
- two LEDs and two push buttons
- four DIP switch positions

Minimum `#include` directives (at the top)

```
#include <avr/io.h>
```

```
#include <avr/ina90.h>
```

A simple program:

```
int main(void)
{
}
```

Execution of a program

- always starts with the `main` subroutine
- when `main` finishes, the board starts all over again

Controlling input/output registers

- use `input` to read an I/O register
- use `output` to write to an I/O register
- or simply use assignment (=) statements with the predefined names

I/O pins

- each pin is set to high impedance input for safety
- all I/O pins can serve as general purpose binary I/O pins
- some I/O pins can also specialize for ADC, PWM and etc.
- 8 pins are grouped to a “port”
- many ports (A, B, C and D)

Direction of I/O pins:

- controlled by DDR registers
- each register controls 8 pins (of the same port)
- pin direction is “bitmapped” (one bit for each pin)
- bit 0 for pin 0 of a port
- a value of 0 for input (default)
- a value of 1 for output

Simple example to set direction:

- pins 0, 1, 4, 7 of port A be input, the other four as output
- bits 2, 3, 5 and 6 should have values of 1
- statement to do this:

```
DDRA = (1 << 2) | (1 << 3) | (1 << 5) | (1 << 6);
```

- a simpler statement (if you know hexadecimal and binary math):

```
DDRA = 0x6c;
```

To change just a few pins:

- previous approach is only useful for setting direction for all 8 pins of a port
- to change just one or a few pins, you must use bit-wise operators
- example: to turn just pin 2 and pin 5 to output without affecting other pins (setting bits):

```
DDRA |= (1 << 2) | (1 << 5);
```

- example: to turn just pin 3 and pin 7 to input without affecting other pins (clearing bits):

```
DDRA &= ~((1 << 3) | (1 << 7));
```

Controlling output pins:

- first configure a pin as input
- use the PORT? I/O registers
- also bitmapped
- a value of 0 means low (0V), a value of 1 means high (5V)

Controlling the “pull-up” of input pins:

- first configure a pin as input
- to use pull-up is the same as making an output pin high (bit value of 1 in PORT?)
- not to use pull-up is the same as making an output pin low (bit value of 0 in PORT?)

Reading the state of a pin

- works for input *and* output pins
- read I/O register PIN?
- bitmapped, a value of 0 means low, a value of 1 means high
- example: to do `blah()` if bit 2 of port C is high:

```
if (PINC & (1 << 2)) blah();
```
- most switches sink a pulled-up pin, so look for 0 when a switch is closed!

But why bother? Subroutines are already defined!

Push buttons:

- use `initPushButtons()`; to initialize
- `isButton1Pushed()` returns non-zero if and only if button 1 (the one to the left) is pushed
- `isButton2Pushed()` returns non-zero if and only if button 2 (the one to the right) is pushed

LEDs:

- use `initLEDs()`; to initialize
- use `swLED1(0)`; to turn off LED1 (the left one)
- use `swLED1(1)`; to turn on LED1
- use `swLED2(0)`; to turn off LED2 (the right one)
- use `swLED2(1)`; to turn on LED2

DIP switch:

- use `initDIPSwitch()`; to initialize
- use `readDIP()` to read the switches
- bitmapped, position 1 is bit 0
- active-high polarity, if a switch is “ON”, the corresponding bit has a value of 1

Calibration constants:

- includes zero positions of servo motors
- includes boundary value of bottom sensors
- includes inside area value of bottom sensors
- use `readConstants()`; to read constants from EEPROM
- use `writeConstants()`; to write constants to EEPROM

Servo control signal:

- use `initTimer1ForPWM()`; to initialize
- initialize *after* `readConstants()`
- use `leftmotor(offset)`; to control the left motor
- use `rightmotoroffset`; to control the right motor
- `offset` in units of $0.5\mu\text{s}$
- an `offset` of 1000 means 0.5ms
- positive `offset` means forward for both motors

Timing support:

- initialized by `initTimer1ForPWM()`; no need to initialize separately
- use `waitMilliseconds(ms)` to wait for `ms` milliseconds
- only accurate to units of 20ms
- specify a positive `ms` value of less than 65536

ADC support:

- even the bottom sensors can be read by the ADC!
- fire-and-forget set up
- initialize by `initADC()`;
- five logical channels:
 - `ADC_LEFTBOTTOM`
 - `ADC_RIGHTBOTTOM`
 - `ADC_CENTERFRONT`
 - `ADC_LEFTFRONT`
 - `ADC_RIGHTFRONT`

ADC support (continued):

- in units of 0.0025V
- call `getAdc(chan)` to read a channel
- for example, to get the value of the front center sensor:
`getAdc(ADC_CENTERFRONT)`
- lowest value is 0
- highest value is 1023

ADC support (continued):

- continuous scanning background operation
- when `adcCounter` changes value, there is a new scan
- makes no sense to call `getAdc` continuously unless `adcCounter` changes

Jump starting with the demo program

- change only `main` and add new functions
- change the rest only if you know what you are doing!

Modes

- OFF-ON-OFF-OFF: servo calibration for left motor
- ON-ON-OFF-OFF: servo calibration for right motor
- OFF-OFF-ON-OFF: bottom sensor calibration
- ON-OFF-ON-OFF: “don’t fall” running mode

Servo calibration:

- set DIP switches
- cycle power
- press button 1 to decrease “zero” pulse length
- press button 2 to increase “zero” pulse length
- if no button is pressed for 1 second, the current value is saved, and LED2 is lit

Bottom sensor calibration

- set DIP switches
- cycle power
- place robot over the “edge” or boundary so the bottom sensors are on top of the edge
- press button 1, LED1 should be lit shortly
- place robot over the “inside area” so the sensors sense the inside area
- press button 2, LED2 should be lit shortly
- done!

Don't fall code

- mostly contained in `dontfall()`
- `interpretBottomSensors()` returns a value indicating which sensor(s), if any, is inside the boundary (read the comments for more details)
- `isInside` returns non-zero if and only if the value of a sensor is closer to the value of inside area

Suggestions

- find offsets to control the servos at intermediate speeds (for better motion control)
- utilize a front facing range-finding sensor to hunt for a target
- extend the calibration subroutines to calibrate the range finding sensor
- be careful that range finding sensors has a minimum distance of 10cm!