Smart Products Lab 3

Tyler Morrison

February 4, 2019

Section 1.

a) Algorithms for manipulating, reading and writing to registers.

- `int pinMode(int pin_number, PinModes p_mode) override:`
  1. use `getPtr` to get the memory register needed to write the pin mode.
  2. use `readRegBits` to get the current register state.
  3. cast the pin mode integer to the associated (binary) register state.
  4. bit shift the desired register state over to the correct registers based on the pin number.
  5. bit shift a mask of zeros to the target registers.
  6. get the new state of the register by ANDing the mask and the old pin state and ORing that with the desired register changes.
  7. use `setRegBits` to write the new desired state to the target register.
- `int digitalRead(int pin_number) override:`
  1. use `getPtr` to get the memory register needed to write the pin mode.
  2. use `readRegBits` to get the current register state.
  3. bit shift the single-bit mask over to the target bit in the register.
  4. AND the mask with the current regsiter state and shift the pin to the front of the register.
  5. convert the register binary to an `int` and return it.
- `int digitalWrite(int pin_number, DigitalOut out_value) override:`
  1. use `getPtr` to get the memory register needed to write the pin mode.
  2. use `readRegBits` to get the current register state.
  3. bit shift a 1 over to the target bit in the register.
  4. bit shift a single bit mask to the target bit in the register.
  5. AND the mask with the current register state and OR that with the desired desired output bit.
  6. use `setRegBits` to write the new desired state to the target register.

b) Efficiency considerations I don't know that I did much to improve efficiency other than write good code. In the future, I guess one could load in all the pin settings in a cache during the class constructor so as to avoid having to figure out which register to use everytime one reads or writes to a pin.

c) Extra functions

- `uint32_t readRegBits(void*):` read the bits in a register from the pointer.
- `int setRegBits(void*, uint32_t):` write the bits in a register given by the pointer.
- `GPIOregisters getPinModeReg(int):` return the GPIOregisters enum object that corresponds to the register for setting the pin mode of the provided pin number.
- `GPIOregisters getPinReadReg(int):` return the GPIOregisters enum object that corresponds to the register for reading the pin state of the provided pin number.
- `GPIOregisters getPinHighReg(int):` return the GPIOregisters enum object that corresponds to the register for setting the pin state high for the provided pin number.
- `GPIOregisters getPinLowReg(int):` return the GPIOregisters enum object that corresponds to the register for setting the pin state low for the provided pin number.
- `int readPinMode(int):` read the pinmode of a pin number
- `PinSettings updatePinSettings(int):` update only the pin settings of a pin that can change, i.e: the state and mode.
- `void showPins():` print all the pins to the screen.
- `void showPins(const std::vector<int> &):` print the selected pins to the screen.

d) Bitwise operators Bitshifts were used to shift desired outputs and masks over to the correct bits in the register. Bit-wise AND's and OR's were used with bitmasks to change single bits in the register without affecting bits other than the targets.

e) Error checking The main error checking I did was check for and throw errors for pin numbers that were outside of the allowed range.

Section 2.

a) What information would have helped me better understand the lab? I am still a little bit confused on the behavior of my Pi when using the memory mapping to change GPIO pin states. It doesn't always seem to work as I expect, as sometimes changing one pin state changes others as well. I have scoured my code for a but but I can't seem to find any.

I'd like to know a little bit more about memory mapping but I can look that up on my own too.

b) Outputs