ECE 375 Prelab 8

Remotely Operated Vehicle

Lab Time: Thursday 1000-1200

Eric Prather

Prelab

1. In this lab, you will be given a set of behaviors/actions that you need to have a proof-of-concept "toy" perform. Think of a toy you know of (or look around online for a toy) that is likely implemented using a microcontroller, and ***describe the behaviors it performs****. Here is an example behavior: "If you press button X on the toy, it takes action Y (or makes sound Z)".*

A cheap, simple electronic toy commonly distributed in the early 2000s was the Tamagotchi. It operated on a battery power-supply and was about the size of a palm. It has three input modalities: Buttons on its front face. It has one output modality, its LCD screen. More complex iterations of the toy had more complex parts, including infrared devices, games, different input and output paradigms, and etcetera. A simplified Tamagotchi functions as follows:

1. If a minute passes without input, the screen turns off.
2. If Input is received while the screen is off, the screen turns on.
3. Every second, the LCD displays one of several predefined hardcoded binary textures based on:
    a. A looping counter
    b. The state of five internal variables
4. Pressing combinations of input buttons change one of the internal variables and overrides the LCD output with various predefined binary textures.
5. Four internal variables gradually increase over time at varying rates.
6. A fifth internal variable updates every several hours automatically. It cannot be interfaced with via the front-facing input buttons.

2. For each behavior you described in the previous question, explain which microcontroller feature was likely used to implement that behavior, and give a brief code example indicating how that feature should be configured. Make your explanation as ATmega128-specific as possible (e.g., discuss which I/O registers would need to be configured, and if any interrupts will be used), and also mention if any additional mechanical and/or electronic devices are needed.

1. The screen-sleep feature was likely inputted with a clock. If this were implemented on the ATMega128, it would look something like:

   INIT:
   ```
   ldi mpr 0b00000010;  TCNT0 overflow "OCIE0"
   out TIMSK, mpr; Timer Interrupt Mask Register
   ldi mpr, 0b00000111 ; Prescaler bits (CS00,CS01,CS02 => clk_T0S/111), normal bit
                       ; (WGM01:0=0) must be set. Clear COM01:0. Clear FOC.
   out TCCR0, mpr      ; write to the correct register: TCCR0. Does both prescaling and mode
   ```
   INTERRUPT VECTOR CALL:
   ```
   ldi mpr, 0x00 ; false
   st SCREEN_TOGGLE_VALUE, mpr ; LCD driver on/off boolean
   ```
2. The screen-wakeup feature was likely done with buttons. If this were implemented on the ATMega128, it would look something like:

   INIT:
   ```
   ldi mpr, $00
   out DDRD, mpr
   ldi mpr, $FF
   out PORTD, mpr
   ldi mpr, ob00000111
   ```

```
sts EICRA, mpr
ldi mpr, (1<<INTF0) | (1<<INTF1) | (1<<INTF2)
out EIFR, mpr
ldi mpr, (1<<INT0) | (1<<INT1)
out EIMSK, mpr
```

INTERRUPT VECTOR CALLS:

```
; Blah blah blah conditionals and branching based on input vector triggered
ldi mpr, 0x01 ; pretend this  is true
st SCREEN_TOGGLE_VALUE, mpr ; LCD driver on/off boolean
```

3. The LCD animation feature was likely implemented with a clock, LCD driver, and set of hardcoded program memory values representing binary textures. Its usage of internal variables would involve data memory storage external to the space used by the LCD driver routines. If this were implemented on the ATMega128, it would look something like:

   INIT:

```
ldi mpr 0b00000010;  TCNT0 overflow "OCIE0"
out TIMSK, mpr; Timer Interrupt Mask Register
ldi mpr, 0b00000111 ; Prescaler bits (CS00,CS01,CS02 => clk_T0S/111), normal bit
                    ; (WGM01:0=0) must be set. Clear COM01:0. Clear FOC.
out TCCR0, mpr      ; write to the correct register: TCCR0. Does both prescaling and mode
```

   INTERRUPT VECTOR CALL:

```
ldi YL, LOW(LCD_DISPLAY_AREA)
ldi YH, HIGH(LCD_DISPLAY_AREA)
; Do conditional logic based on internal variables to know which address to get the
binary texture from.
ldi ZL, LOW(TARGET_TEXTURE)
ldi ZH, HIGH(TARGET_TEXTURE)
ldi counter, 255
TEXTURE_DISPLAY_LOOP:
ld mpr, Z+
st Y+, mpr ; LCD driver on/off boolean
dec counter
cpi counter, 0x00
brne TEXTURE_DISPLAY_LOOP
```

4. The input feature which changes variables and outputs to the LCD likely used tri-state buffer input pins. If these input modalities were implemented on the ATMega128, they would look something like:

```
ldi mpr, $00
out DDRD, mpr
ldi mpr, $FF
out PORTD, mpr
ldi mpr, ob00000111
sts EICRA, mpr
ldi mpr, (1<<INTF0) | (1<<INTF1) | (1<<INTF2)
out EIFR, mpr
ldi mpr, (1<<INT0) | (1<<INT1)
out EIMSK, mpr
```

   INTERRUPT VECTOR CALLS:

```
; Blah blah blah conditionals and branching based on input vector triggered
ldi mpr, EQU_SET_PREDEFINED_VALUE ; depends on variable to change, current value of
variable, input set, etc.
st INTERNAL_VARIABLE_N, mpr ; Wherever the pertinent internal variable is stored in
memory
```

5. The automatic incrementing of variables over various amounts of time were likely handled with a clock. If this were implemented on the ATMega128, it would look something like:

   INIT:

```
ldi mpr 0b00000010;  TCNT0 overflow "OCIE0"
out TIMSK, mpr; Timer Interrupt Mask Register
ldi mpr, 0b00000111 ; Prescaler bits (CS00,CS01,CS02 => clk_T0S/111), normal bit
```

```
                        ; (WGM01:0=0) must be set. Clear COM01:0. Clear FOC.
out TCCR0, mpr          ; write to the correct register: TCCR0. Does both prescaling and mode
```

INTERRUPT VECTOR CALL:

```
push mpr
in mpr, sreg
push mpr
push zl
push zh
ldi ZL, LOW(VARIABLE_TO_INCREMENT)
ldi ZH, HIGH(VARIABLE_TO_INCREMENT)
ld mpr, Z
inc mpr
st Z, mpr ; LCD driver on/off boolean
pop zh
pop zl
pop mpr
out sreg, mpr
pop mpr
```

6. Same as above

3. Each ATmega128 USART module has two flags used to indicate its current transmitter state: the Data Register Empty (UDRE) flag and Transmit Complete (TXC) flag. What is the difference between these two flags, and which one always gets set first as the transmitter runs? You will probably need to read about the Data Transmission process in the datasheet (including looking at any relevant USART diagrams) to answer this question.

The data register empty flag (UDRE) is set when the USART Data Register (UDRn) register is ready to be written to. It triggers interrupt vector $003E (p60). The Transmit complete flag (TXC) is set when the final significant value in the Transmit Shift Register is shifted out. It triggers interrupt vector $0040 (p60).

Data moves from the UDRn to the transmit shift register. It is possible for the transmission frame to be empty, complete, or in process while there is new data waiting in UDRn. Pages 178-129 describe this relationship.

Notably, in standard USART procedures, UDRE is typically set before TXC. This is because that first the data is moved from UDRE to the shift register, clearing UDRn, and *then* TXC slowly shifts out the data inside of it. More information on these flags can be found on page 188 of the datasheet.

4. Each ATmega128 USART module has one flag used to indicate its current receiver state (not including the error flags). For USART1 specifically, what is the name of this flag, and what is the interrupt vector address for the interrupt associated with this flag? This time, you will probably need to read about Data Reception in the datasheet to answer this question.

Note: For identically behaving flags, for some reason, the "n" in the pin name provided in the datasheet must be substituted with the number of the register the flag is being applied to, despite the fact that all "n" registers have the exact same evaluation.

The answer to this prelab question can be pulled specifically from HW3. More precisely:

The **name** of the flag is "USART Receive Complete" (RXC1) – see page 188 of the datasheet. It is in USART Control and Status Register A (UCSR1A). The **interrupt vector** associated with this flag is $003C – see page 59 of the datasheet.