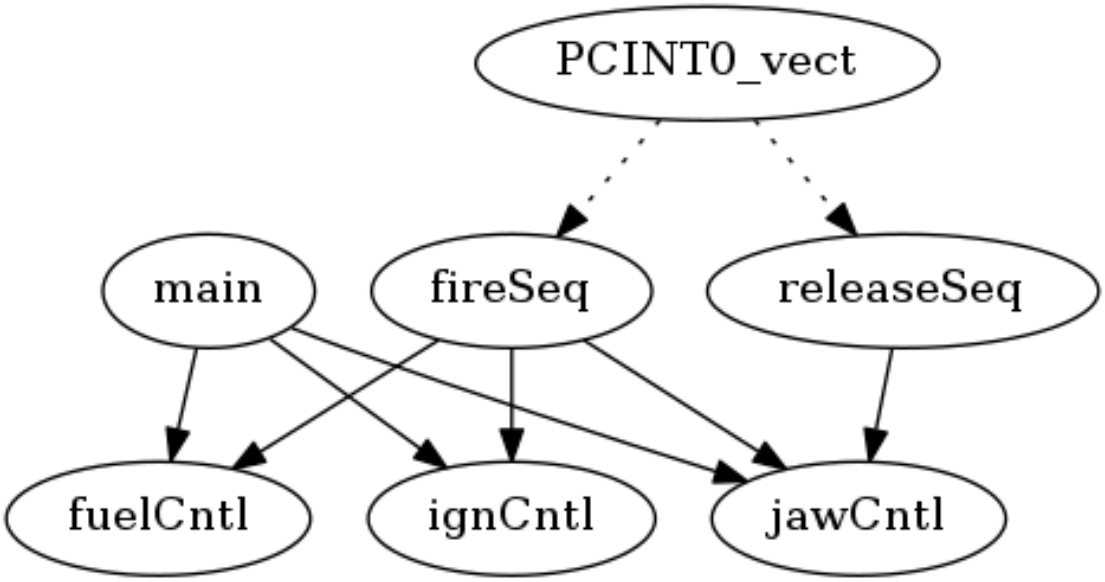
September 20, 2016 at 21:38

1. Introduction. This is the firmware portion of Jaw and Fire control.
This will facilitate two actions: opening the jaw to release the floating object and light the target on fire.
The jaw will close by return-spring so the action will to open it.
Fire is a sequence of opening the jaw, releasing the butane and firing the ignitor.



Extensive use was made of the datasheet, Atmel "Atmel ATtiny25, ATtiny45, ATtiny85 Datasheet" Rev. 2586QAVR08/2013 (Tue 06 Aug 2013 03:19:12 PM EDT) and "AVR130: Setup and Use the AVR Timers" Rev. 2505AAVR02/02.

⟨ Include 4 ⟩
⟨ Prototypes 5 ⟩
⟨ Global variables 6 ⟩

2. `"F_CPU"` is used to convey the Trinket clock rate.

```
#define F_CPU 8000000UL
```

3. Here are some Boolean definitions that are used.

```
#define ON 1
#define OFF 0
#define OPEN 1
#define CLOSE 0
#define SET 1
#define CLEAR 0
```

4. ⟨ Include 4 ⟩ ≡

```
#include <avr/io.h>      /* need some port access */
#include <util/delay.h>      /* need to delay */
#include <avr/interrupt.h>      /* have need of an interrupt */
#include <avr/sleep.h>      /* have need of sleep */
#include <avr/wdt.h>      /* have need of watchdog */
#include <stdlib.h>
#include <stdint.h>
```

This code is used in section 1.

5. ⟨ Prototypes 5 ⟩ ≡

```
void jawCntl(uint8_t state);      /* Jaw open and close */
void fuelCntl(uint8_t state);      /* Fuel on and off */
void ignCntl(uint8_t state);      /* on and off */
void releaseSeq(void);
void fireSeq(void);
```

This code is used in section 1.

6. My lone global variable is a function pointer. This lets me pass arguments to the actual interrupt handlers, should I need to. This pointer gets the appropriate function attached by one of the `"ISR()"` functions.

⟨ Global variables 6 ⟩ ≡

```
void (*handleIrq)( ) = Λ;
int main(void)
    {
    ⟨ Initialize interrupts 28 ⟩
    ⟨ Initialize pin inputs 27 ⟩
    ⟨ Initialize pin outputs 24 ⟩
```

This code is used in section 1.

7. Of course, any interrupt function requires that bit “Global Interrupt Enable” is set; usually done through calling sei(). Doing this after the pin setup is the best time.

```
sei( );
```

8. Rather than burning loops, waiting for something to happen, the “sleep” mode is used. The specific type of sleep is ‘idle’. In idle, execution stops but timers continue. Interrupts are used to wake it.

⟨ Configure to wake upon interrupt 34 ⟩

9. This is the loop that does the work. It should spend most of its time in *sleep_mode*, comming out at each interrupt event.

```
for ( ; ; )
{
```

10. We don’t want anything cooking while we are asleep.

```
ignCntl(OFF);
fuelCntl(OFF);
jawCntl(CLOSE);
```

11. Now we wait in “idle” for any interrupt event.

```
sleep_mode( );
```

12. If execution arrives here, some interrupt has been detected.

```
if (handleIrq ≠ Λ)      /* not sure why it would be, but to be safe */
{
  handleIrq( );
  handleIrq = Λ;      /* reset so that the action cannot be repeated */
}     /* end if handleIrq */
}     /* end for */
return 0;      /* it's the right thing to do! */
}     /* end main() */
```

13. Interrupt Handling.

```
void releaseSeq( )
    {
```

14. This sequence will proceed only while the button is held.

```
jawCntl(OPEN);
while (¬(PINB & (1 ≪ PB3))) _delay_ms(10);
jawCntl(CLOSE); }
```

15.

```
void fireSeq( )
    {
    uint8_t firingState;
    enum firingStates {
      ready, opened, igniting, burning, cooling
    };
    firingState = ready;
```

16. This sequence will proceed only while the button is held. It can terminate after any state. `"_delay_ms()"` is a handy macro good for 2^{16} milliseconds of delay.

```
while (¬(PINB & (1 ≪ PB4))) {
```

17. The jaw opens here for fire.

```
if (firingState ≡ ready) {
  jawCntl(OPEN);
  firingState = opened;
  continue;
}
```

18. Ignitor is on.

```
if (firingState ≡ opened) {
  ignCntl(ON);
  firingState = igniting;
  continue;
}
```

19. Fuel opens.

```
if (firingState ≡ igniting) {
  fuelCntl(ON);
  firingState = burning;
  continue;
}
_delay_ms(10); }
```

20. Once the loop fails we set fuel and ignitor off and close the jaw.

```
ignCntl(OFF);
fuelCntl(OFF);
_delay_ms(5000);
jawCntl(CLOSE); }
```

21. The ISRs.

The ISRs are pretty skimpy as they mostly used to point *handleIrq*( ) to the correct function. The need for global variables is minimized.

22. This vector responds to the jaw input at pin PB3 or fire input at PB4. A simple debounce is included.

```
ISR(PCINT0_vect)
{
  const int8_t high = 32;
  const int8_t low = -high;
  int8_t dbp3 = 0;
  int8_t dbp4 = 0;
  while (abs(dbp3) < high) {
    if (¬(PINB & (1 ≪ PB3)) ∧ dbp3 > low) dbp3 --;
    else if ((PINB & (1 ≪ PB3)) ∧ dbp3 < high) dbp3 ++;
    _delay_ms(1);
  }
  while (abs(dbp4) < high) {
    if (¬(PINB & (1 ≪ PB4)) ∧ dbp4 > low) dbp4 --;
    else if ((PINB & (1 ≪ PB4)) ∧ dbp4 < high) dbp4 ++;
    _delay_ms(1);
  }
  if (dbp3 ≡ low) handleIrq = &releaseSeq;
  else if (dbp4 ≡ low) handleIrq = &fireSeq;
}
```

23. These are the supporting routines, procedures and configuration blocks.
Here is the block that sets-up the digital I/O pins.

24. ⟨ Initialize pin outputs 24 ⟩ ≡

```
{    /* 14.4.9 DDRD  The Port D Data Direction Register */    /* set the jaw port direction */
  DDRB |= (1 << DDB0);    /* set the fuel port direction */
  DDRB |= (1 << DDB1);    /* set the ignition port direction */
  DDRB |= (1 << DDB2);
}
```

This code is used in section 6.

25. Timer Counter 0 is configured for "Phase Correct" PWM which, according to the datasheet, is preferred for motor control. OC0A and OC0B are set to clear on a match which creates a non-inverting PWM.

26. ⟨ Initialize Timer 26 ⟩ ≡

```
{    /* 15.9.1 TCCR0A  Timer/Counter Control Register A */
  TCCR0A |= (1 << WGM00);    /* Phase correct, mode 1 of PWM (table 15-9) */
  TCCR0A |= (1 << COM0A1);    /* Set/Clear on Comparator A match (table 15-4) */
  TCCR0A &= ~(1 << COM0A0);    /* Set on Comparator A match (table 15-4) */
  TCCR0A |= (1 << COM0B1);    /* Set/Clear on Comparator B match (table 15-7) */
  TCCR0A &= ~(1 << COM0B0);    /* Set on Comparator B match (table 15-7) */
    /* 15.9.2 TCCR0B  Timer/Counter Control Register B */
  TCCR0B |= (1 << CS01);    /* Prescaler set to clk/8 (table 15-9) */
}
```

27. ⟨ Initialize pin inputs 27 ⟩ ≡

```
{    /* set the jaw input pull-up */
  PORTB |= (1 << PORTB3);    /* set the fire input pull-up */
  PORTB |= (1 << PORTB4);
}
```

This code is used in section 6.

28. ⟨ Initialize interrupts 28 ⟩ ≡

```
{    /* enable change interrupt for jaw input */
  PCMSK |= (1 << PCINT3);    /* enable change interrupt for fire input */
  PCMSK |= (1 << PCINT4);    /* General interrupt Mask register */
  GIMSK |= (1 << PCIE);
}
```

This code is used in section 6.

29. Here is a simple procedure to operate the jaw.

```
void jawCntl(uint8_t state)
{
  if (state) {
    OCR0A = #ff;
    _delay_ms(200);
    OCR0A = #ff ≫ 1;
  }
  else {
    OCR0A = #00;
  }
}
```

30. Here is a simple procedure to operate the fuel.

```
void fuelCntl(uint8_t state)
{
  if (state) {
    OCR0B = #ff;
    _delay_ms(200);
    OCR0B = #ff ≫ 1;
  }
  else {
    OCR0A = #00;
  }
}
```

31. Here is a simple procedure to operate the ignition.

```
void ignCntl(uint8_t state)
{
  PORTB = state ? PORTB | (1 ≪ PORTB2) : PORTB & ∼(1 ≪ PORTB2);
}
```

32. See section the datasheet for details on the Watchdog Timer. We are not using it right now.

33. ⟨ Initialize watchdog timer 33 ⟩ ≡

```
{
  WDTCR |= (1 ≪ WDCE) | (1 ≪ WDE);
  WDTCR = (1 ≪ WDIE) | (1 ≪ WDP2);      /* reset after about 0.25 seconds */
}
```

34. Setting these bits configure sleep_mode() to go to “idle”. Idle allows the counters and comparator to continue during sleep.

⟨ Configure to wake upon interrupt 34 ⟩ ≡

```
{
  MCUCR &= ∼(1 ≪ SM1);
  MCUCR &= ∼(1 ≪ SM0);
}
```

This code is used in section 8.

⟨ Configure to wake upon interrupt 34 ⟩ Used in section 8.
⟨ Global variables 6 ⟩ Used in section 1.
⟨ Include 4 ⟩ Used in section 1.
⟨ Initialize Timer 26 ⟩
⟨ Initialize interrupts 28 ⟩ Used in section 6.
⟨ Initialize pin inputs 27 ⟩ Used in section 6.
⟨ Initialize pin outputs 24 ⟩ Used in section 6.
⟨ Initialize watchdog timer 33 ⟩
⟨ Prototypes 5 ⟩ Used in section 1.