September 1, 2015 at 22:59

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.

place-holder code below ===========================

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 ⟩
⟨ Types 5 ⟩
⟨ Prototypes 6 ⟩
⟨ Global variables 7 ⟩
```

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 <stdlib.h>
#include <stdint.h>
```

This code is used in section 1.

5. Here is a structure to keep track of the state of things.

⟨ Types 5 ⟩ ≡

```
  typedef struct {
    uint8_t count;
  } statestruct;
```

This code is used in section 1.

6. ⟨ Prototypes 6 ⟩ ≡

```
  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 fireseq(statestruct *);
  void releaseseq(statestruct *);
```

This code is used in section 1.

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

```
⟨ Global variables 7 ⟩ ≡
  void (*handleirq)(statestruct *) = Λ;
```

This code is used in section 1.

8. Here is *main*( ).

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

9. The prescaler is set to clk/16484 by ⟨ Initialize the timer 27 ⟩. With `"F_CPU"` at 8 MHz, the math goes: $\lfloor \frac{0.5 seconds \times (8 \times 10^6}{16384} \rfloor = 244$. Then, the remainder is $256 - 244 = 12$, thus leaving 244 counts or about 500 ms until time-out, unless it's reset.

```
  statestruct s_state; ⟨ Initialize pin inputs 23 ⟩⟨ Initialize pin outputs 22 ⟩
```

10.

```
  jawcntl(CLOSE);    /* PB0 */
  fuelcntl(OFF);    /* PB1 */
  igncntl(OFF);    /* PB2 */
```

11. Here the timer is setup.

```
  ⟨ Initialize the timer 27 ⟩
```

12. Of course, any interrupt function requires that bit "Global Interrupt Enable" is set; usually done through calling sei().

```
  sei( );
```

13. Rather than burning loops, waiting 16 ms 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 28 ⟩
```

14. 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 ( ; ; ) {
```

15. Now we wait in "idle" for any interrupt event.

```
  sleep_mode( );
```

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

```
  if (handleirq ≠ Λ)    /* not sure why it would be, but to be safe */
  {
    handleirq(&s_state);    /* process the irq through it's function */
    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() */
```

17. Interrupt Handling.

```
void releaseseq(statestruct *s_now)
{ }
```

18.

```
void fireseq(statestruct *s_now)
{ }
```

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

This is the vector for the main timer. When this overflows it generally means the ASE has been off for as long as it took `TCINT1` to overflow from it's start at `NOWAVETIME`.

```
    /* Timer ISR */
ISR(TIMER1_OVF_vect)
{
  handleirq = Λ;
}
```

20. This vector responds to the jaw input at pin PB4 or PB4.

```
    /* Clear Button ISR */
ISR(PCINT0_vect)
{
  handleirq = &releaseseq;
}
```

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

22. ⟨ Initialize pin outputs 22 ⟩ ≡

```
{     /* 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 9.

23. ⟨ Initialize pin inputs 23 ⟩ ≡

```
{     /* set the jaw input pull-up */
  PORTB |= (1 << PORTB3);     /* set the fire input pull-up */
  PORTB |= (1 << PORTB4);     /* enable change interrupt for jaw input */
  PCMSK |= (1 << PCINT3);     /* enable change interrupt for fire input */
  PCMSK |= (1 << PCINT4);     /* General interrupt Mask register for clear-button */
  GIMSK |= (1 << PCIE);
}
```

This code is used in section 9.

24. Here is a simple function to operate the jaw.

```
void jawcntl(uint8_t state)
{
  PORTB = state ? PORTB | (1 << PORTB0) : PORTB & ~(1 << PORTB0);
}
```

25. Here is a simple function to operate the fuel.

```
void fuelcntl(uint8_t state)
{
  PORTB = state ? PORTB | (1 << PORTB1) : PORTB & ~(1 << PORTB1);
}
```

26. Here is a simple function to operate the ignition.

```
void igncntl(uint8_t state)
{
  PORTB = state ? PORTB | (1 << PORTB2) : PORTB & ~(1 << PORTB2);
}
```

27. A very long prescale of 16384 counts is set by setting certain bits in TCCR1.
⟨ Initialize the timer 27 ⟩ ≡

```
{
  TCCR1 = ((1 << CS10) | (1 << CS11) | (1 << CS12) | (1 << CS13));     /* Prescale */
  TIMSK |= (1 << TOIE1);     /* Timer 1 f_overflow interrupt enable */
}
```

This code is cited in section 9.

This code is used in section 11.

28. 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 28 ⟩ ≡

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

This code is used in section 13.