June 28, 2015 at 01:53

1. Introduction. This is the firmware portion of the propulsion system for our Champbot. It features separate thrust and steering as well as piruett turning.

This will facilitate motion by taking "thrust" and "turn" pulse-width inputs from the Futaba RC receiver by converting them to the appropriate motor actions. These are from Channel 2 at A1 and channel 1 at A0, respectivily. The action will be similar to driving an RC car or boat. By keeping it natural, it should be easier to navigate the course than with a skid-steer style control.

2. Implementation. Both pulse-width inputs will have some dead-band to allow for full stop.

The pulse-width from the receiver is at 20 ms intervals. The time ranges from 1000–2000 ms, including trim. 1500 ms is the width for stopped. The levers cover ±0.4 ms and the trim covers the balance.

Math for radius...I think this is right:

Where: t is track r is radius v is value f is factor

min r is 1 and 255 max r is 128

get this value for v [1] [128] [255]

r=factor*abs(127-v)

For non-zero r Inside=(2r-t)/2r Outside=(2r+t)/2r

Port motor pulse will be applied to ???, starboard will be at ???. The median time will be subtracted from them for a pair of signed values thrust and radius. The value will be scaled.

The thrust and radius will be translated to power to the port and starboard motors. When near median the motors will be disabled. The motors will also be disabled when there are no input pulses. Each motor need direction and power so that's 4 signals of output.

The radius control will also be the rotate control, if thrust is zero.

Adding the two signal of input, I need more I/O than the trinket has. So—I now have a $10 Pro Trinket with far more capability. It has an ATmega328.

The ATmega328 has a fancy 16 bit PWM with two comparators, Timer 1. This has an "Input Capture Unit" that may be used for PWC decoding. PWC being the type of signal from the RC receiver. That seems like as elegant a solutioni as I will find and it is recommended by Atmel to use it for this purpose.

One of the other timers will do more than fine for the two motors.

For the PWC measurement, this app note, AVR135, is helpful: www.atmel.com/images/doc8014.pdf

In the datasheet, this section is helpful: 16.6.3

The best way to use this nice feature is to take the PWC signals into the MUX, through the comparator and into the Input Capture Unit.

An interesting thing about this Futaba receiver is that the pulses are in series. The channel two's pulse is first, followed the channel one. In fact, channel two's fall is perfectly aligned with channel one's rise. This means that it will be possible to capture all of the pulses.

After the two pulses are captured, there's an 18 ms dead-time before the next round. That's over 250,000 clock cycles. This will provide ample time to do math and set the motor PWMs.

First pick the turn, set for a rising edge, wait, grab the time-stamp and set for falling edge, wait, grab the time-stamp, do modulus subtraction, switch the MUX, set for rising, reset the ICR, wait, grab the time-stamp, and do modulus sutraction for the second duration.

Extensive use was made of the datasheet, Atmel "Atmel-8271I-AVR- ATmega-Datasheet_10/2014".

⟨ Include 5 ⟩
⟨ Types 6 ⟩
⟨ Prototypes 9 ⟩
⟨ Global variables 10 ⟩

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

```
#define F_CPU 16000000UL
#define BAUD 9600
```

4. Here are some Boolean definitions that are used.

```
#define ON 1
#define OFF 0
#define SET 1
#define CLEAR 0
#define CH2RISE 0
#define CH2FALL 1
#define CH1FALL 2
```

5. ⟨ Include 5 ⟩ ≡

```
#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 2.

6. Here is a structure type to keep track of the state of output things, like motor settings.

⟨ Types 6 ⟩ ≡

```
typedef struct {
  uint8_t portOut;
  uint8_t starboardOut;
  int32_t thrust;
  int32_t turn;
  uint8_t failSafe;      /* safety relay */
} outputStruct;
```

See also sections 7 and 8.

This code is used in section 2.

7. Here is a structure type to keep track of the state of input things, like servo timing.

⟨ Types 6 ⟩ +≡

```
typedef struct {
  uint16_t ch2rise;
  uint16_t ch2fall;
  uint16_t ch1fall;
  uint16_t ch1duration;
  uint16_t ch2duration;
  uint8_t edge;
} inputStruct;
```

8. Here is a structure type to contain the scaling parameters for the scaler function.

⟨ Types 6 ⟩ +≡

```
typedef struct {
  uint16_t minIn;
  uint16_t maxIn;
  uint16_t minOut;
  uint16_t maxOut;
} scaleStruct;
```

9. ⟨ Prototypes 9 ⟩ ≡

```
void ledcntl(uint8_t state);      /* LED ON and LED OFF */
void pwcCalc(inputStruct *);
void edgeSelect(inputStruct *);
uint16_t scaler(scaleStruct *, uint16_t input);
```

This code is used in section 2.

10. My lone global variable may become a function pointer. This could let me pass arguments to the actual interrupt handlers. This pointer gets the appropriate function attached by the `"ISR()"` function.

```
⟨ Global variables 10 ⟩ ≡
  void (*handleIrq)(inputStruct *) = Λ;
```

This code is used in section 2.

11. Here is *main*( ).

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

12. The Futaba receiver leads with channel two, rising edge, so we will start looking for that by setting `"edge"` to look for a rise on channel 2.

```
inputStruct input_s = { . edge = CH2RISE } ;
outputStruct output_s;
```

13. Center reports about 21250, hard left, or up, with trim reports about 29100 and hard right, or down, with trim reports about 13400.

About 4/5ths of that range are the full swing of the stick, without trim. This is from about 14970 and 27530 ticks.

This `"scale_s"` structure holds the parameters used in the scaler function. The `"In"` numbers are raw from the Input Capture Register.

At some point a calibration feature could be added which could populate these but the numbers here were from trial and error.

```
scaleStruct scale_s = { . minIn = 14970 ,      /* ticks for hard right or down */
. maxIn = 27530 ,      /* ticks for hard left or up */
. minOut = 1 , . maxOut = 255 } ;
⟨ Initialize the inputs and capture mode 30 ⟩⟨ Initialize pin outputs 27 ⟩
```

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

```
sei( );
{     /* for test purposes */
  DDRD &= ~(1 << DDD3);      /* Clear the PD3 pin */      /* PD3 (PCINT0 pin) is now an input */
  PORTD |= (1 << PORTD3);      /* turn On the Pull-up */
    /* PD3 is now an input with pull-up enabled */
  EICRA |= (1 << ISC10);      /* set INT1 to trigger on ANY logic change */
  EIMSK |= (1 << INT1);      /* Turns on INT1 */
}
```

15. Rather than burning loops, waiting the ballance of 18 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.

It's important to note that an ISR procedure must be defined to allow the program to step past the sleep statement.

```
⟨ Configure to idle on sleep 28 ⟩
ledcntl(OFF);
edgeSelect(&input_s);
```

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

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

17. Now that a loop is started, we wait in "idle" for the edge on the channel selected.

```
sleep_mode( );      /* idle */
```

18. If execution arrives here, some interrupt has woken it from sleep and some vector has possibly run. The pointer handleIrq will be assigned the value of the responsible function.

```
if (handleIrq ≠ Λ)      /* in case it woke for some other reason */
{
  handleIrq(&input_s);
  handleIrq = Λ;      /* reset so that the action cannot be repeated */
}     /* end if handleirq */
output_s.turn = scaler(&scale_s, input_s.ch1duration);
output_s.thrust = scaler(&scale_s, input_s.ch2duration);
if (output_s.turn ≥ 255) ledcntl(ON);
else ledcntl(OFF);
}     /* end for */
return 0;      /* it's the right thing to do! */
}     /* end main() */
```

19. Here are the ISRs.

```
ISR(INT1_vect)
{
}
ISR(TIMER1_CAPT_vect)
{
  handleIrq = &pwcCalc;
}
```

20. This procedure computes the durations from the PWC signal edge capture values from the Input Capture Unit. With the levers centered the durations should be about 1.5 ms so at 16 Mhz the count should be near 24000. The range should be 17600 to 30400 for 12800 counts, well within the range of the 64 kib of the 16 bit register..

```
void pwcCalc(inputStruct *input_s){
```

21. On the falling edges we can compute the durations using modulus subtraction and then set the edge index for the next edge. Channel 2 leads so that rise is first.

```
switch (input_s→edge) {
case CH2RISE: input_s→ch2rise = ICR1;
  input_s→edge = CH2FALL;
  break;
case CH2FALL: input_s→ch2fall = ICR1;
  input_s→ch2duration = input_s→ch2fall − input_s→ch2rise;
  input_s→edge = CH1FALL;
  break;
case CH1FALL: input_s→ch1fall = ICR1;
  input_s→ch1duration = input_s→ch1fall − input_s→ch2fall;
  input_s→edge = CH2RISE;
}
edgeSelect(input_s);
}
```

22.
The procedure edgeSelect configures the Input Capture unit to capture on the expected edge type.

```
void edgeSelect(inputStruct *input_s){
    switch (input_s→edge) {
    case CH2RISE:     /* wait for rising edge on servo channel 2 */
      ADMUX |= (1 ≪ MUX0);     /* Set to mux channel 1 */
      TCCR1B |= (1 ≪ ICES1);     /* Rising edge (23.3.2) */
      break;
    case CH2FALL: ADMUX |= (1 ≪ MUX0);     /* Set to mux channel 1 */
      TCCR1B &= ∼(1 ≪ ICES1);     /* Falling edge (23.3.2) */
      break;
    case CH1FALL: ADMUX &= ∼(1 ≪ MUX0);     /* Set to mux channel 0 */
      TCCR1B &= ∼(1 ≪ ICES1);     /* Falling edge (23.3.2) */
    }
```

23. Since the edge has been changed, the Input Capture Flag should be cleared. It's odd but clearing it involves writing a one to it.

```
TIFR1 |= (1 ≪ ICF1);     /* (per 16.6.3) */
}
```

24. Here is a simple procedure to flip the LED on or off.

```
void ledcntl(uint8_t state)
{
  PORTB = state ? PORTB | (1 ≪ PORTB5) : PORTB & ∼(1 ≪ PORTB5);
}
```

25.

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

27. ⟨ Initialize pin outputs 27 ⟩ ≡
```
{   /* set the led port direction; This is pin #17 */
  DDRB |= (1 << DDB5);
}
```
This code is used in section 13.

28. ⟨ Configure to idle on sleep 28 ⟩ ≡
```
{
  SMCR &= ~((1 << SM2) | (1 << SM1) | (1 << SM0));
}
```
This code is used in section 15.

29. To enable this interrupt, set the ACIE bit of register ACSR.

30. ⟨ Initialize the inputs and capture mode 30 ⟩ ≡
```
{   /* ADCSRA  ADC Control and Status Register A */
  ADCSRA &= ~(1 << ADEN);     /* Conn the MUX to (-) input of comparator (sec 23.2) */
    /* 23.3.1 ADCSRB  ADC Control and Status Register B */
  ADCSRB |= (1 << ACME);      /* Conn the MUX to (-) input of comparator (sec 23.2) */
    /* 24.9.5 DIDR0  Digital Input Disable Register 0 */
  DIDR0 |= ((1 << AIN1D) | (1 << AIN0D));     /* Disable digital inputs (sec 24.9.5) */
    /* 23.3.2 ACSR  Analog Comparator Control and Status Register */
  ACSR |= (1 << ACBG);      /* Connect + input to the band-gap ref (sec 23.3.2) */
  ACSR |= (1 << ACIC);      /* Enable input capture mode (sec 23.3.2) */
  ACSR |= (1 << ACIS1);     /* Set for both rising and falling edge (sec 23.3.2) */
    /* 16.11.8 TIMSK1  Timer/Counter1 Interrupt Mask Register */
  TIMSK1 |= (1 << ICIE1);     /* Enable input capture interrupt (sec 16.11.8) */
    /* 16.11.2 TCCR1B  Timer/Counter1 Control Register B */
  TCCR1B |= (1 << ICNC1);     /* Enable input capture noise canceling (sec 16.11.2) */
  TCCR1B |= (1 << CS10);      /* No Prescale. Just count the main clock (sec 16.11.2) */
    /* 24.9.1 ADMUX  ADC Multiplexer Selection Register */
  ADMUX &= ~((1 << MUX2) | (1 << MUX1) | (1 << MUX0));     /* Set to mux channel 0 */
}
```
This code is used in section 13.

31. The scaler function takes an input, as in times from the Input Capture Register and returns a value scaled by the parameters in structure `"scale_s"`.

uint16_t scaler(**scaleStruct** **scale_s*, *uint16_t input*){

32. First, we can solve for the obvious cases in which the input exceeds the range. This can easily happen if the trim is shifted.

if (*input* > *scale_s→maxIn*) **return** *scale_s→maxOut*;
else if (*input* < *scale_s→minIn*) **return** *scale_s→minOut*;

33. If it's not that simple, then compute the gain and offset then continue in the usual way. This is not really an effecient method, recomputing gain and offset every time but we are not in a rush and it makes it easier since, if something changes, I don't have to manualy compute and enter these values and the code is all in one place.

The constant 100 amplifies it so I can take advantage of the extra bits for precision.

```
int32_t gain = (100L * (int32_t)(scale_s→maxIn - scale_s→minIn))/(int32_t)(scale_s→maxOut -
    scale_s→minOut);
int32_t offset = ((100L * (int32_t)scale_s→minIn)/gain) - (int32_t)scale_s→minOut;
return (100L * (int32_t)input/gain) - offset; }
```

⟨ Configure to idle on sleep 28 ⟩ Used in section 15.
⟨ Global variables 10 ⟩ Used in section 2.
⟨ Include 5 ⟩ Used in section 2.
⟨ Initialize pin outputs 27 ⟩ Used in section 13.
⟨ Initialize the inputs and capture mode 30 ⟩ Used in section 13.
⟨ Prototypes 9 ⟩ Used in section 2.
⟨ Types 6, 7, 8 ⟩ Used in section 2.