July 5, 2015 at 00:40

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 "radius" pulse-width inputs from the Futaba RC receiver by converting them to the appropriate motor actions. These are from Channel 2 at analog input 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. The Futaba receiver has two PWC channels. The pulse-width from the receiver is at 20 ms intervals. The on-time ranges from 1000–2000 ms, including trim. 1500 ms is the pulse-width for stopped. The levers cover ±0.4 ms and the trim covers the balance.

Both pulse-width inputs will need some dead-band to allow for full stop.

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 signals of input, I need more I/O than the original trinket has. So—I now have moved to a $10 Pro Trinket with far more capability. It has an ATmega328, just like and Arduino.

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.

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.

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

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.

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

⟨ Include 6 ⟩
⟨ Types 7 ⟩
⟨ Prototypes 10 ⟩
⟨ Global variables 11 ⟩

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
```

5. Here are some other definitions.

```
#define CH2RISE 0
#define CH2FALL 1
#define CH1FALL 2
#define MAX_DUTYCYCLE 98      /* 98% to support charge pump of bridge-driver */
```

6. ⟨ Include 6 ⟩ ≡

```
#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.

7. Here is a structure type to keep track of the state of remote-control input, e.g. servo timing.

⟨ Types 7 ⟩ ≡

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

See also sections 8 and 9.

This code is used in section 2.

8. Here is a structure type to keep track of the state of translation items.

⟨ Types 7 ⟩ +≡

```
typedef struct {
  int16_t thrust;      /* -255 to 255 */
  int16_t radius;      /* -255 to 255 */
  int16_t track;      /* 1 to 255 */
  int16_t starboardOut;      /* -255 to 255 */
  int16_t portOut;      /* -255 to 255 */
} transStruct;
```

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

⟨ Types 7 ⟩ +≡

```
typedef struct {
  int16_t minIn;      /* input, minimum */
  int16_t maxIn;      /* input, maximum */
  int16_t minOut;      /* output, minimum */
  int16_t maxOut;      /* output, maximum */
  int8_t deadBand;      /* width of zero in terms of output units */
} scaleStruct;
```

10. ⟨ Prototypes 10 ⟩ ≡

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

This code is used in section 2.

11. 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 the `"ISR()"` function.

```
⟨ Global variables 11 ⟩ ≡
  void(*handleIrq)(inputStruct *) = Λ;
```

This code is used in section 2.

12. Here is *main*( ).

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

13. 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 } ;
```

14. Center position of the controller results in a count of 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 `"inputScale_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 and seem good.

```
scaleStruct inputScale_s = { . minIn = 14970 ,      /* ticks for hard right or down */
. maxIn = 27530 ,      /* ticks for hard left or up */
. minOut = −255 , . maxOut = 255 , . deadBand = 5 } ;
transStruct translation_s;
⟨ Initialize the inputs and capture mode 33 ⟩⟨ Initialize pin outputs 30 ⟩
```

15. 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 */
}
```

16.

The PWM is used to control port and starboard motors through OC0A (D5) and OC0B (D6), respectivly.

```
⟨ Initialize the Timer Counter 0 for PWM 35 ⟩
```

17. 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 up.

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 31 ⟩
ledcntl(OFF);
edgeSelect(&input_s);
```

18. 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 ( ; ; ) {
```

19. Now that a loop is started, the PWM is given an initial value and we wait in `"idle"` for the edge on the channel selected. Each sucessive loop will finish in the same way.

```
setPwm(&translation_s);
sleep_mode( );      /* idle */
```

20. 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 */
translation_s.radius = scaler(&inputScale_s, input_s.ch1duration);
translation_s.thrust = scaler(&inputScale_s, input_s.ch2duration);
translation_s.track = 100;      /* represent unitless prop-prop distance */
translate(&translation_s);
```

21. Some temporary test code here.

```
if (translation_s.portOut ≥ 127)  ledcntl(ON);
else  ledcntl(OFF);
}     /* end for */
return 0;      /* it's the right thing to do! */
}     /* end main() */
```

22. Here are the ISRs.

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

23. 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){
```

24. 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);
}
```

25.
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) */
    }
```

26. 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) */
}
```

27. 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);
}
```

28.

29. Supporting routines, functions, procedures and configuration blocks.

30. ⟨ Initialize pin outputs 30 ⟩ ≡

```
{   /* set the led port direction; This is pin #17 */
  DDRB |= (1 << DDB5);    /* 14.4.9 DDRD The Port D Data Direction Register */
  DDRD |= ((1 << DDD5) | (1 << DDD6));    /* Data direction to output (sec 14.3.3) */
}
```

This code is used in section 14.

31. ⟨ Configure to idle on sleep 31 ⟩ ≡

```
{
  SMCR &= ~((1 << SM2) | (1 << SM1) | (1 << SM0));
}
```

This code is used in section 17.

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

33. ⟨ Initialize the inputs and capture mode 33 ⟩ ≡

```
{   /* 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 14.

34. PWM setup isn't too scary. Timer Count 0 is configured for "Phase Correct" PWM which, according to the datasheet, is preferred for motor control. OC0A (port) and OC0B (starboard) are set to clear on a match which creates a non-inverting PWM. The prescaler is set to clk/8 and with a 16 MHz clock the f is about 3922 Hz.

35. ⟨ Initialize the Timer Counter 0 for PWM 35 ⟩ ≡

```
{   /* 15.9.1 TCCR0A Timer/Counter Control Register A */
  TCCR0A |= (1 << WGM00);    /* Phase correct mode of PWM (table 15-9) */
  TCCR0A |= (1 << COM0A1);    /* Clear on Comparator A match (table 15-4) */
  TCCR0A |= (1 << COM0B1);    /* Clear 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) */
}
```

This code is used in section 16.

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

```
uint16_t scaler(scaleStruct *inputScale_s, uint16_t input){
```

37. 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 > inputScale_s→maxIn) return inputScale_s→maxOut;
else if (input < inputScale_s→minIn) return inputScale_s→minOut;
```

38. If it's not that simple, then compute the gain and offset and 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, also the code is all in one place.

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

```
const int32_t ampFact = 128L;      /* factor for precision */
int32_t gain  =  (ampFact * (int32_t)(inputScale_s→maxIn -
    inputScale_s→minIn))/(int32_t)(inputScale_s→maxOut - inputScale_s→minOut);
int32_t offset = ((ampFact * (int32_t)inputScale_s→minIn)/gain) - (int32_t)inputScale_s→minOut;
return (ampFact * (int32_t)input/gain) - offset; }
```

39. We need a way to translate `"thrust"` and `"radius"` in order to carve a `"turn"`. This procedure should do this but it's not going to be perfect as drag and slippage make thrust increase progressivly more than speed. It should steer OK as long as the speed is constant and small changes in speed should not be too disruptive.

This procedure is intended for values from -255 to 255.

```
void translate(transStruct *trans_s){ int16_t speed;
    int16_t rotation;
    int16_t difference;
    const int16_t max = (MAX_DUTYCYCLE * UINT8_MAX)/100;
    const int16_t ampFact = 128;      /* factor for precision */
    speed = trans_s→thrust;      /* cheating a bit here */
```

40. Here we convert desired radius to thrust-difference by scaling to speed. Then that difference is converted to rotation by scaling it with `"track"`. The radius sensitivity is adjusted by changing the value of `"track"`.

```
difference = (speed * ((ampFact * trans_s→radius)/UINT8_MAX))/ampFact;
rotation = (trans_s→track * ((ampFact * difference)/UINT8_MAX))/ampFact;
```

41. Any rotation involves one motor turning faster than the other. At some point, faster is not possible and so the requiered clipping is here.

"max" is set at to support the limit of the bridge-driver's charge-pump.

```
if ((speed − rotation) ≥ max) trans_s→portOut = max;
else if ((speed − rotation) ≤ −max) trans_s→portOut = −max;
else trans_s→portOut = speed − rotation;
if ((speed + rotation) ≥ max) trans_s→starboardOut = max;
else if ((speed + rotation) ≤ −max) trans_s→starboardOut = −max;
else trans_s→starboardOut = speed + rotation;
} void setPwm(transStruct *trans_s)
{
  OCR0A = (uint8_t)(trans_s→portOut > 0 ? trans_s→portOut : −trans_s→portOut);
  OCR0B = (uint8_t)(trans_s→starboardOut > 0 ? trans_s→starboardOut : −trans_s→starboardOut);
}
```

⟨ Configure to idle on sleep 31 ⟩ Used in section 17.
⟨ Global variables 11 ⟩ Used in section 2.
⟨ Include 6 ⟩ Used in section 2.
⟨ Initialize pin outputs 30 ⟩ Used in section 14.
⟨ Initialize the Timer Counter 0 for PWM 35 ⟩ Used in section 16.
⟨ Initialize the inputs and capture mode 33 ⟩ Used in section 14.
⟨ Prototypes 10 ⟩ Used in section 2.
⟨ Types 7, 8, 9 ⟩ Used in section 2.