January 1, 2016 at 01:28

1. Introduction. This is the firmware portion of the propulsion system for our 2016 Champbot. It features separate thrust and steering, including piruett turning. Also an autonomous dive function has been added.

This will facilitate motion by taking "thrust" and "radius" pulse-width, or PWC, inputs from the Futaba-Kyosho RC receiver and converting them to the appropriate motor actions.

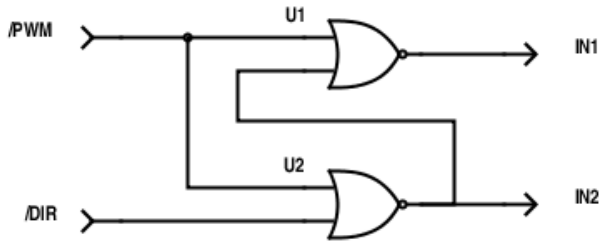
Also an autonomous dive function has been added.

Thrust is Channel 2, entering analog input A1, and Radius is channel 1, at A0. 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.

We are using the Wingxing DBH-01 (B/C) and the Inputs are unique on this. The PWM logic input goes to two different pins, depending on direction! The non-PWM pin must be held low. This is a big problem since PWM outputs have dedicated pins. Two AVR timers would be needed to control two motors; waistful.

The odd example in the DBH-01 datasheet has PWM on IN1 and LOW on IN2 for forward. For reverse, LOW on IN1 and PWM on IN2.

Rulling out multiple timers (four comparators), additional outputs, or a PLD, the best solution we could find was a adding glue logic. A single 74F02 was chosen; a quad NOR. Keeping this solution simple, one gate-



type and on one chip, required that the AVR outputs be inverted.
This one chip handles the logic for both motors. With this, the AVR outputs direction on one pin and PWM on the other. At the H-Bridge, the pin receiving PWM is selected based on motor direction. The remaining, non-PWM pin, is held low.

OC0A and OC0B is on pins 5 and 6 (D8 and D6) and are the PWM. A fail-safe relay output will be at pin 8.

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 μs including trim. 1500 μs is the pulse-width for stop. The levers cover $\pm$400 μs and the trim covers the last 100 μs.

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 through a dead-band. Stiction in the motor probably wouldn't allow it to move anyway, at this low duty-cycle. Both the PWM and safety relay will open. The motors will also be disabled when there are no input pulses; in this way champ wont run-off if the range is exceeded. This function is handled by the watchdog timer.

The radius control will also be the rotate control, if thrust is zero. Timer-Counter 0 is used for the PWM.

The ATmega328 has a 16 bit PWMs 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 solution 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.

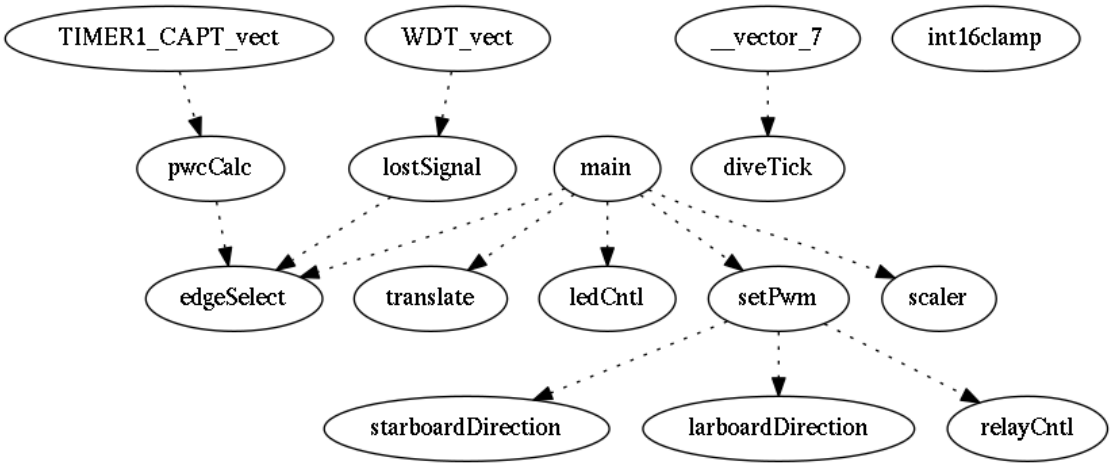
For the PWC measurement, this app note, AVR135, is helpful: www.atmel.com/images/doc8014.pdf

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".



I originaly wanted to use the word "Port" for the left-hand side, when facing the front. On a microcontroller that name is used for all of the ports so I chose the older word "larboard".

⟨ Include 6 ⟩
⟨ Types 7 ⟩
⟨ Prototypes 9 ⟩
⟨ Global variables 10 ⟩

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

```
#define F_CPU 16000000UL
```

4. Here are some Boolean definitions that are used.

```
#define ON 1
#define OFF 0
#define SET 1
#define CLEAR 0
#define TRUE 1
#define FALSE 0
#define FORWARD 1
#define REVERSE 0
#define CLOSED 1
#define OPEN 0
#define STOPPED 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 <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 2.

7. Here is a structure type to keep track of the state of remote-control input, e.g. servo timing. Rise and Fall indicate the PWC edges. `"edge"` is set to the edge type expected for the interrupt.

⟨ Types 7 ⟩ ≡

```
typedef struct {
  uint16_t ch2rise;
  uint16_t ch2fall;
  uint16_t ch1fall;
  uint16_t ch1duration;
  uint16_t ch2duration;
  uint8_t edge;
  uint8_t lostSignal;
  const uint16_t minIn; /* input, minimum */
  const uint16_t maxIn; /* input, maximum */
} inputStruct;
```

See also section 8.

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 larboardOut;       /* -255 to 255 */
    const int16_t minOut;      /* output, minimum */
    const int16_t maxOut;      /* output, maximum */
    const int8_t deadBand;     /* width of zero in terms of output units */
  } transStruct;
```

9. ⟨ Prototypes 9 ⟩ ≡

```
  void relayCntl(int8_t state);
  void ledCntl(int8_t state);
  void larboardDirection(int8_t state);
  void starboardDirection(int8_t state);
  void diveTick(inputStruct *);
  void pwcCalc(inputStruct *);
  void edgeSelect(inputStruct *);
  void translate(transStruct *);
  void setPwm(transStruct *);
  void lostSignal(inputStruct *);
  int16_t scaler(inputStruct *, transStruct *, uint16_t input);
  int16_t int16clamp(int16_t value, int16_t min, int16_t max);
```

This code is used in section 2.

10. 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.

This input structure is to contain all of the external inputs.

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

This code is used in section 2.

11. 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.

Center position of the controller results in a count of about 21250, hard larboard, or forward, with trim reports about 29100 and hard starboard, or reverse, 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.

`".minIn"` `".maxIn"` are the endpoints of the normal stick travel. The units are raw counnts as the Input Capture Register will use.

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.

Until we have collected the edges we will assume there is no signal.

```
  inputStruct input_s = { . edge = CH2RISE , . minIn = 14970 , . maxIn = 27530 , . lostSignal = TRUE } ;
```

12. This is the structure that holds output parameters. It's instantiated with the endpoint constants.

transStruct *translation_s* = { . *minOut* = −255 , . *maxOut* = 255 , . *deadBand* = 10 } ;

13. Here the interrupts are disabled so that configuring them doesn't set it off.

cli( );

⟨ Initialize the inputs and capture mode 50 ⟩⟨ Initialize pin outputs 47 ⟩⟨ Initialize watchdog timer 54 ⟩

14. Any interrupt function requires that bit "Global Interrupt Enable" is set; usually done through calling `"sei()"`.

sei( );

15.

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

⟨ Initialize the Timer Counter 0 for PWM 56 ⟩

16. 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, like the Input Capture Unit and PWM continue to operate. Interrupts, either *Input Capture* or *Watchdog*, 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, even if it is empty. This stumped me for a good while.

⟨ Configure to idle on sleep 48 ⟩

ledCntl(`OFF`);

17. Since `"edge"` is already set, calling *edgeSelect*( ) will get it ready for the first rising edge of channel 2. Subsequent calls to *edgeSelect* rotates it to the next edge type.

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 or watchdog timeout.

for (; ;) {

19. Now that a loop is started, the PWM is value and we wait in `"idle"` for the edge on the channel selected. Each sucessive loop will finish in the same way. After three passes `"translation_s"` will have good values.

setPwm(&*translation_s*);
sleep_mode( );

20. If execution arrives here, some interrupt has woken it from sleep and some vector has possibly run. The possibility is first checked. The pointer `"handleIrq"` will be assigned the value of the responsible function and then executed. After that the IRQ is nulled so as to avoid repeating the action, should it wake for some other reason.

```
if (handleIrq ≠ Λ) {
  handleIrq(&input_s);
  handleIrq = Λ;
}
translation_s.radius = scaler(&input_s, &translation_s, input_s.ch1duration);
translation_s.thrust = scaler(&input_s, &translation_s, input_s.ch2duration);
translation_s.track = 100;      /* represents unit-less prop-to-prop distance */
translate(&translation_s);
```

21. The LED is used to indicate both channels PWM's are zeros.

```
if (translation_s.larboardOut ∨ translation_s.starboardOut) ledCntl(OFF);
else ledCntl(ON);
}     /* end for */
return 0;
}     /* end main() */
```

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

23. Here is the ISR that fires at each captured edge. Escentialy it grabs and processes the *Input Capture* data.

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

24. Here is the ISR that fires at at about 32 Hz for the main dive tick. This is used for the PI controll loop.

```
ISR(TIMER2_COMPA_vect)
{
  handleIrq = &diveTick;
}
```

25. When the watchdog timer expires, this vector is called. This is what happens if the remote's transmitter signal is not received. It calls a variant of *pwcCalc* that only flips the *lostSignal* flag.

```
ISR(WDT_vect)
{
  handleIrq = &lostSignal;
}
```

26. 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 1500 μs 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 2^{16} counts of the 16 bit register.

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

27. 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.

Arrival at the last case establishes that there was a signal, clears the flag and resets the watchdog timer.

```
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;
  input_s→lostSignal = FALSE;
  wdt_reset( );      /* watchdog timer is reset at each edge capture */
                }
edgeSelect(input_s);
}
```

28. This procedure sets output to zero in the event of a lost signal.

```
void lostSignal(inputStruct *input_s)
{
  input_s→lostSignal = TRUE;
  input_s→edge = CH2RISE;
  edgeSelect(input_s);
}
```

29. This procedure will count off ticks for a 0.25 second event. Since capture events are guaranteed to have a ¿1 ms period, or 16000 clock cycles, there should be no lost PWC edges.

```
void diveTick(inputStruct *input_s)
{
  static uint8_t tickCount = 0;
  if (¬(++tickCount & ∼#80))      /* every 128 ticks */
  { }
}
```

30.
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:     /* To 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) */
    }
```

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

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

32.

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

```
int16_t scaler(inputStruct *input_s, transStruct *trans_s, uint16_t input)
    {
    uint16_t solution;
```

34. First, we can solve for the obvious cases. One is where there is no signal; when "lostSignal" is "TRUE". The other is where the input exceeds the range. This can easily happen if the trim is shifted.

```
  if (input_s→lostSignal ≡ TRUE) return 0;
  if (input > input_s→maxIn) return trans_s→maxOut;
  if (input < input_s→minIn) return trans_s→minOut;
```

35. If it's not that simple, then compute the gain and offset and then continue in the usual way. This is not really an efficient 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 value.

The constant "ampFact" amplifies values for math so I can take advantage of the extra bits for precision. Dead-band is applied when it returns.

```
  const int32_t ampFact = 128L;
  int32_t gain = (ampFact * (int32_t)(input_s→maxIn − input_s→minIn))/(int32_t)(trans_s→maxOut −
      trans_s→minOut);
  int32_t offset = ((ampFact * (int32_t)input_s→minIn)/gain) − (int32_t)trans_s→minOut;
  solution = (ampFact * (int32_t)input/gain) − offset;
  return (abs(solution) > trans_s→deadBand) ? solution : 0;
  }
```

36. 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. Since *speed* is not known, we will use *thrust*. It should steer OK as long as the speed is constant and small changes in speed should not be too disruptive. The sign of larboardOut and starboardOut indicates direction. The constant "ampFact" amplifies values for math so I can take advantage of the extra bits for precision. bits.

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

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

```
void translate(transStruct *trans_s)
    {
    int16_t speed = trans_s→thrust;      /* we are assuming it's close */
    int16_t rotation;
    int16_t difference;
    int16_t piruett;
    static int8_t lock = OFF;
    const int8_t PirLockLevel = 15;
    const int16_t max = (MAX_DUTYCYCLE * UINT8_MAX)/100;
    const int16_t ampFact = 128;
```

37. 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;
piruett = trans_s→radius;
```

38. Any rotation involves one motor turning faster than the other. At some point, faster is not possible and so the leading motor's thrust is clipped.

If there is no thrust then it is in piruett mode and spins CW or CCW. While thrust is present, piruett mode is locked out. Piruett mode has a lock function too, to keep it from hopping into directly into thrust mode while it is spinning around. This is partly for noise immunity and partly to help avoid collisions.

```
if (trans_s→thrust ≠ STOPPED ∧ lock ≡ OFF) {
  trans_s→larboardOut = int16clamp(speed − rotation, −max, max);
  trans_s→starboardOut = int16clamp(speed + rotation, −max, max);
}
else      /* piruett mode */
{ lock = (abs(piruett) > PirLockLevel) ? ON : OFF;
trans_s→larboardOut = int16clamp(piruett, −max, max);
```

39. For starboard, piruett is reversed, making it rotate counter to larboard.

```
piruett = −piruett;
trans_s→starboardOut = int16clamp(piruett, −max, max); }
}
```

40. This procedure sets the signal to the H-Bridge. For the PWM we load the value into the unsigned registers.

```
void setPwm(transStruct *trans_s)
    {
    if (trans_s→larboardOut ≥ 0) {
      larboardDirection(FORWARD);
      OCR0A = abs(trans_s→larboardOut);
    }
    else {
      larboardDirection(REVERSE);
      OCR0A = abs(trans_s→larboardOut);
    }
    if (trans_s→starboardOut ≥ 0) {
      starboardDirection(FORWARD);
      OCR0B = abs(trans_s→starboardOut);
    }
    else {
      starboardDirection(REVERSE);
      OCR0B = abs(trans_s→starboardOut);
    }
```

41. We must see if the fail-safe relay needs to be closed.

```
  if (trans_s→larboardOut ∨ trans_s→starboardOut) relayCntl(CLOSED);
  else relayCntl(OPEN);
  }
```

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

```
void ledCntl(int8_t state)
{
  PORTB = state ? PORTB | (1 ≪ PORTB5) : PORTB & ∼(1 ≪ PORTB5);
}
```

43. Here is a simple procedure to flip the Relay Closed or Open from pin #8.

```
void relayCntl(int8_t state)
{
  PORTB = state ? PORTB | (1 ≪ PORTB0) : PORTB & ∼(1 ≪ PORTB0);
}
```

44. Here is a simple procedure to set thrust direction on the larboard motor.

```
void larboardDirection(int8_t state)
{
  if (state) PORTD &= ∼(1 ≪ PORTD3);
  else PORTD |= (1 ≪ PORTD3);
}
```

45. Here is a simple procedure to set thrust direction on the starboard motor.

```
void starboardDirection(int8_t state)
{
  if (state) PORTD &= ~(1 << PORTD4);
  else PORTD |= (1 << PORTD4);
}
```

46. A simple 16 bit clamp function.

```
int16_t int16clamp(int16_t value, int16_t min, int16_t max)
{
  return (value > max) ? max : (value < min) ? min : value;
}
```

47. ⟨ Initialize pin outputs 47 ⟩ ≡

```
DDRB |= (1 << DDB5);      /* set the led port direction; This is pin #17 */
DDRB |= (1 << DDB0);      /* set the relay port direction; This is pin #8 */
DDRD |= (1 << DDD5);      /* 14.4.9 DDRD The Port D Data Direction Register */
  /* larboard and starboard pwm outputs */
DDRD |= ((1 << DDD5) | (1 << DDD6));      /* Data direction to output (sec 14.3.3) */
  /* larboard and starboard direction outputs */
DDRD |= ((1 << DDD3) | (1 << DDD4));      /* Data direction to output (sec 14.3.3) */
```

This code is used in section 13.

48. ⟨ Configure to idle on sleep 48 ⟩ ≡

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

This code is used in section 16.

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

50. ⟨ Initialize the inputs and capture mode 50 ⟩ ≡

```
{     /* 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.

51. For a timer tick at each 1/4 second. We will use timer counter 2, our last timer. It only has an 8 bit prescaler so it will be too fast and will need to be divided. The prescaler is set to the maximum of 1024. The timer is set to CTC mode so that the time loop is trimable. That will be pretty fast so we need more division in software. We want to divide by a power of two for efficiency and so 0x80 (0d128) is as small as we can go. This is also half a uint8_t so the math works out very well. The time is trimmed to make 128 passes about 0.25 seconds by loading compare register, OCR2A, with 0xf3 (0d243). The interrupt is enabled TIMSK2 for output compare register A. With all that we will have interrupt TIMER2 COMPA fire every 31 ms. For the software division we will increment an uint8_t in the handler on each pass and do something at both 0x00 and 0x80 (0d128). The test would be !(tickCount 0x80).

52. ⟨ Initialize tick timer 52 ⟩ ≡

```
{
  TCCR2B |= (1 << CS22) | (1 << CS21) | (1 << CS20);      /* maximum prescale (see 18.11.2) */
  TCCR2A |= (1 << WGM21);      /* CTC mode (see 18.11.1) */
  OCR2A = #f3;      /* Do I need to make this clearer? */
  TIMSK2 |= (1 << OCIE2A);      /* Interrupt on a compare match */
}
```

53. See section 11.8 in the datasheet for details on the Watchdog Timer. This is in the "Interrupt Mode". When controlled remotlely or in an autonomous dive this should not time-out. It needs to be long enough to allow for the 0.25 ms autonomous dive loop.

54. ⟨ Initialize watchdog timer 54 ⟩ ≡

```
{
  WDTCSR |= (1 << WDCE) | (1 << WDE);
  WDTCSR = (1 << WDIE) | (1 << WDP2) | (1 << WDP0);      /* reset after about 0.5 seconds (see 11.9.2) */
}
```

This code is used in section 13.

55. 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 used for PWM. The prescaler is set to clk/8 and with a 16 MHz clock the f is about 3922 Hz. We are using `"Set"` on comparator match to invert the PWM, suiting the glue-logic which drives the H-Bridge.

56. ⟨ Initialize the Timer Counter 0 for PWM 56 ⟩ ≡

```
{    /* 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 << COM0B1);      /* Set/Clear on Comparator B match (table 15-7) */
  TCCR0A |= (1 << COM0A0);      /* Set on Comparator A match (table 15-4) */
  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) */
}
```

This code is used in section 15.

⟨ Configure to idle on sleep 48 ⟩ Used in section 16.
⟨ Global variables 10 ⟩ Used in section 2.
⟨ Include 6 ⟩ Used in section 2.
⟨ Initialize pin outputs 47 ⟩ Used in section 13.
⟨ Initialize the Timer Counter 0 for PWM 56 ⟩ Used in section 15.
⟨ Initialize the inputs and capture mode 50 ⟩ Used in section 13.
⟨ Initialize tick timer 52 ⟩
⟨ Initialize watchdog timer 54 ⟩ Used in section 13.
⟨ Prototypes 9 ⟩ Used in section 2.
⟨ Types 7, 8 ⟩ Used in section 2.