March 1, 2016 at 21:33

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.

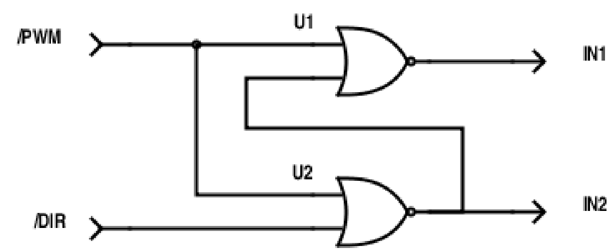
Also an autonomous dive function has been added.

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, i.e. 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 ± 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, section 16.6.3 is helpful.

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 `PWM`s.

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

This is esentialy a boat and so 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 11 ⟩
⟨ Global variables 12 ⟩

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

#define `F_CPU` 16000000_{UL}

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 AUTOMATIC 1
#define MANUAL 0
#define STOPPED 0
```

5. Here are some other definitions. It is critical that `MAX_DUTYCYCLE` is 98% or less.

```
#define CH2RISE 0 /* rising edge of RC's remote channel 2 */
#define CH2FALL 1 /* falling edge of RC's remote channel 2 */
#define CH1FALL 2 /* falling edge of RC's remote channel 1 */
#define MAX_DUTYCYCLE 98 /* 98% to support charge pump of bridge-driver */
#define OFF 0 /* the mode of being surfaced */
#define REMOTE 1 /* the mode of being surfaced */
#define DIVING 2 /* the mode of actively diving */
#define SUBMERGED 3 /* the mode of being submerged */
#define PIDSAMPCT 4 /* the PID sample count for derivatives */
```

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>
#include <assert.h>
```

This code is used in section 2.

7. This structure is for the PID or Direct Digital Control. *k_p* is the proportional coefficient. The larger it is, the bigger will be the effect of PID. *k_i* is the integral coefficient in resets per unit-time. *k_d* is the derivative coefficient. *m* is the output. Whatever is minimal energy is probably a good output to start with. *min* is the minimum allowed output. *max* is the maximum allowed output. *mode* can be manual or automatic;

⟨ Types 7 ⟩ ≡

```
typedef struct {
  int16_t k_p;   /* proportional action parameter */
  int16_t k_i;   /* integral action parameter in R/T */
  int16_t k_d;   /* derivative action parameter */
  int16_t t;   /* sampling period */
  int16_t setpoint;   /* setpoint */
  int16_t pPvN[PIDSAMPCT];   /* process value history */
  int16_t *pPvLast;   /* process value latest location */
  int16_t m;   /* latest output */
  int16_t mMin;   /* min output */
  int16_t mMax;   /* max output */
  int8_t mode;   /* 1 == automatic, 0 == manual */
} ddcParameters;
```

See also sections 8, 9, and 10.

This code is used in section 2.

8. Here is a structure type to keep track of the state of inputs, e.g. servo timing. Rise and Fall indicate the PWC edge times. *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 controlMode;
  uint16_t pressure;   /* pressure in ADC units */
  const uint16_t minIn;   /* input, minimum */
  const uint16_t maxIn;   /* input, maximum */
  ddcParameters *pPid_s;
} inputStruct;
```

9. 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 int8_t deadBand;   /* width of zero in terms of output units */
} transStruct;
```

10. This structure type keeps track of the state of dive and submerge.

```
⟨ Types 7 ⟩ +≡
  typedef struct {
    uint16_t diveTime;      /* 0.25 sec intervals allowed before it gets canceled */
    uint16_t submergeTime;  /* 0.25 sec intervals to remain at depth */
    int16_t starboardOut;   /* -255 to 255 */
    int16_t larboardOut;    /* -255 to 255 */
  } diveStruct;
```

11.
```
⟨ Prototypes 11 ⟩ ≡
  void relayCntl(int8_t state);
  void ledCntl(int8_t state);
  void larboardDirection(int8_t state);
  void starboardDirection(int8_t state);
  void pressureCalc(inputStruct *);
  void diveTick(inputStruct *);
  void pwcCalc(inputStruct *);
  void edgeSelect(inputStruct *);
  void translate(transStruct *);
  void setPwm(int16_t, int16_t);
  void lostSignal(inputStruct *);
  int16_t scaler(uint16_t input, uint16_t minIn, uint16_t maxIn, int16_t minOut, int16_t maxOut);
  int16_t int16clamp(int16_t value, int16_t min, int16_t max);
  void takDdcSetPid(ddcParameters *, int16_t p, int16_t i, int16_t d, int16_t t);
  int16_t takDdc(ddcParameters *);
```

This code is used in section 2.

12. My lone global variable is a function pointer. This lets me pass arguments to the actual interrupt handlers and acts a bit like a stack to store the next action. This pointer gets the appropriate function attached by the `ISR()` function.

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

```
⟨ Global variables 12 ⟩ ≡
  void (*handleIrq)(inputStruct *) = Λ;
  int main(void)
      {
```

This code is used in section 2.

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.

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 $\frac{4}{5}$ 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 counts 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.

```
const uint16_t minIn = 14970;    /* minimum normal value from receiver */
const uint16_t maxIn = 27530;    /* maximum normal value from receiver */
const int16_t minOut = INT16_MIN;    /* minimum value of thrust */
const int16_t maxOut = INT16_MAX;    /* maximum value of thrust */
```

14. Initially we will have the motors off and wait for the first rising edge from the remote. The PID parameters are instantiated and loaded with safe defaults. *takDdcSetPid*( ) is used to set the parameters.

```
inputStruct *pInput_s = &(inputStruct) {
.edge = CH2RISE,
.controlMode = OFF,
.pPid_s = &(ddcParameters) {
        .k_p = 1,
        .k_i = 1,
        .k_d = 1,
        .t = 1,
        .m = 0,
        .mMin = INT16_MIN,
        .mMax = INT16_MAX,
        .mode = AUTOMATIC
        }
};
```

15. This is the structure that holds output parameters.

```
transStruct *pTranslation_s = &(transStruct) {.deadBand = 10};
```

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

```
cli();
⟨Initialize the inputs and capture mode 58⟩⟨Initialize tick timer 60⟩⟨Initialize pin outputs 55⟩⟨Initialize
    watchdog timer 62⟩
```

17. Any interrupt function requires that bit "Global Interrupt Enable" is set.

```
sei();
```

18.

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

```
⟨Initialize the Timer Counter 0 for PWM 64⟩
```

19. 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. Another thing that will happen during sleep is an `ADC` conversion from the pressure sensor. Interrupts "Input Capture", "tick", "`ADC`" and "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 56 ⟩
ledCntl(`OFF`);

20. 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(*pInput_s*);

21. 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 (; ;) {

22. 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.

sleep_mode( );

23. If execution arrives here, some interrupt has woken it from sleep and some vector has possibly run. That 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-up for some other reason.

if ($handleIrq \neq \Lambda$) {
  handleIrq(*pInput_s*);
  $handleIrq = \Lambda$;
}

24. Here we scale the PWC durations and apply the "deadBand".

```
{
  int16_t outputCh1;
  int16_t outputCh2;
  if (pInput_s→controlMode ≠ OFF) {
    outputCh1 = scaler(pInput_s→ch1duration, minIn, maxIn, minOut, maxOut);
    outputCh2 = scaler(pInput_s→ch2duration, minIn, maxIn, minOut, maxOut);
  }
  else {
    outputCh1 = 0;
    outputCh2 = 0;
  }
  outputCh1 = (abs(outputCh1) > pTranslation_s→deadBand) ? outputCh1 : 0;
  outputCh2 = (abs(outputCh2) > pTranslation_s→deadBand) ? outputCh2 : 0;
  pTranslation_s→radius = outputCh1;
  pTranslation_s→thrust = outputCh2;
  pTranslation_s→track = 100;    /* represents unit-less prop-to-prop distance */
}
translate(pTranslation_s);
if (pInput_s→controlMode ≡ REMOTE) setPwm(pTranslation_s→larboardOut, pTranslation_s→starboardOut);
else setPwm(pTranslation_s→larboardOut, pTranslation_s→starboardOut);
```

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

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

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

27. 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;
}
```

28. Here is the ISR that fires at at about 64 Hz for the main dive tick. This is used for the dive-control loop.

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

29. Here is the ISR that fires after a successful ADC conversion. The ADC is used to determine depth from pressure.

```
ISR(ADC_vect)
{
  handleIrq = &pressureCalc;
}
```

30. 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 sets the controlMode to OFF.

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

31. 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 *pInput_s)
    {
```

32. 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.

```
switch (pInput_s→edge) {
case CH2RISE: pInput_s→ch2rise = ICR1;
  pInput_s→edge = CH2FALL;
  break;
case CH2FALL: pInput_s→ch2fall = ICR1;
  pInput_s→ch2duration = pInput_s→ch2fall − pInput_s→ch2rise;
  pInput_s→edge = CH1FALL;
  break;
case CH1FALL: pInput_s→ch1fall = ICR1;
  pInput_s→ch1duration = pInput_s→ch1fall − pInput_s→ch2fall;
  pInput_s→edge = CH2RISE;
  if (pInput_s→controlMode ≡ OFF) pInput_s→controlMode = REMOTE;
                    }
edgeSelect(pInput_s);
}
```

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

```
void lostSignal(inputStruct *pInput_s)
{
  pInput_s→controlMode = OFF;
  pInput_s→edge = CH2RISE;
  edgeSelect(pInput_s);
}
```

34. This procedure will count off ticks for a $\frac{1}{4}$ second event. Every tick it will setup ADC to get pressure sensor values during idle.

```
void diveTick(inputStruct *pInput_s)
{
  static uint8_t tickCount = 0;     /* we are here 64 times per second */
  if (pInput_s→edge ≡ CH2RISE)     /* while timing isn't too critical */
  {
    ADCSRA |= (1 ≪ ADEN);     /* Connect the MUX to the ADC and enable it */
    ADMUX = (ADMUX & #f0) | 2U;     /* Set MUX to channel 2 */
  }
  if (¬(++tickCount))     /* every 256 ticks */
  {
    if (pInput_s→controlMode ≥ DIVING) {     /* do the PID stuff here? */
      takDdc(pInput_s→pPid_s);
    }
    wdt_reset();     /* watchdog timer is reset */
  }
}
```

35. This procedure will filter ADC results for a pressure in terms of ADC units. First the comparator is reconnected to the MUX so that we miss as few RC events as possible. There is a moving average filter of size 32 or about $\frac{1}{2}$ second in size. That size is efficient since the division is a binary right shift of 5 places. Since the ADC is a mere 10 bits, and $2^{10} \times 32$ is only 2^{15}, the sum may safely be of size **uint16_t**.

```
void pressureCalc(inputStruct *pInput_s)
{
  static uint16_t buffStart[33];
  const uint16_t *buffEnd = buffStart + 33;
  static uint16_t *buffIndex = buffStart;
  static uint16_t sum;    /* range 0 to 32768 */
  ADCSRA &= ~(1 << ADEN);    /* reconnect the MUX to the comparator */
  *buffIndex = ADCL & ((uint16_t) ADCH) << 8;    /* drop in the ADC value */
  sum += *buffIndex;    /* include this new find in the sum */
  buffIndex = (buffIndex != buffEnd) ? buffIndex + 1 : buffStart;
  sum -= *buffIndex;    /* remove the oldest item from the sum */
  pInput_s->pressure = (sum >> 5);
}
```

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

```
void edgeSelect(inputStruct *pInput_s)
    {
    switch (pInput_s->edge) {
    case CH2RISE:    /* To wait for rising edge on servo-channel 2 */
      ADMUX = (ADMUX & #f0) | 1U;    /* Set to mux channel 1 */
      TCCR1B |= (1 << ICES1);    /* Rising edge (23.3.2) */
      break;
    case CH2FALL: ADMUX = (ADMUX & #f0) | 1U;    /* Set to mux channel 1 */
      TCCR1B &= ~(1 << ICES1);    /* Falling edge (23.3.2) */
      break;
    case CH1FALL: ADMUX = (ADMUX & #f0) | 0U;    /* Set to mux channel 0 */
      TCCR1B &= ~(1 << ICES1);    /* Falling edge (23.3.2) */
    }
```

37. 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) */
  }
```

38.

39. 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(uint16_t input, uint16_t minIn, uint16_t maxIn, int16_t minOut, int16_t maxOut)
    {
```

40. First, we can solve for the obvious cases. This can easily happen if the trim is shifted and the lever is at its limit.

```
  if (input > maxIn) return maxOut;
  if (input < minIn) return minOut;
```

41. 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 to take advantage of the high bits for precision.

```
const int32_t ampFact = 128L;
int32_t gain = (ampFact * (int32_t)(maxIn − minIn))/(int32_t)(maxOut − minOut);
int32_t offset = ((ampFact * (int32_t) minIn)/gain) − (int32_t) minOut;
return (ampFact * (int32_t) input/gain) − offset;
}
```

42. 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 the true 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. As before, the constant *ampFact* amplifies values for math so to take advantage of the high bits for precision. bits.

This procedure is intended for values from -255 to 255 or `INT16_MIN` to `INT16_MAX`.

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;
```

43. 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;
```

44. 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. It seems better to compromise speed rather than turning.

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);
```

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

```
piruett = -piruett;
trans_s->starboardOut = int16clamp(piruett, -max, max); }
}
```

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

```
void setPwm(int16_t larboardOut, int16_t starboardOut)
    {
    if (larboardOut >= 0) {
      larboardDirection(FORWARD);
      OCR0A = abs(larboardOut);
    }
    else {
      larboardDirection(REVERSE);
      OCR0A = abs(larboardOut);
    }
    if (starboardOut >= 0) {
      starboardDirection(FORWARD);
      OCR0B = abs(starboardOut);
    }
    else {
      starboardDirection(REVERSE);
      OCR0B = abs(starboardOut);
    }
```

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

```
if (larboardOut || starboardOut) relayCntl(CLOSED);
else relayCntl(OPEN);
}
```

48. 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);
}
```

49. 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);
}
```

50. 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);
}
```

51. This is the PID algorithm for the dive control. It is largely based on an algorithm from the book *Control and Dynamic Systems* by Yasundo Takahashi, et al. (1970). This is a nice, easy to compute iterative (velocity) algorithm.

Everything is integrated so the proportional starts as a derivative and the derivative starts as a second derivative. It's a unique form, since error is seen only through the integral.

Takahashi suggested a four point difference for the derivative, if the signal is noisy. Our signal may be very noisy so this feature has been included. Takahashi's four point difference was a bit involved, so to make this easy, I used numerical differentiation coefficients from the *CRC Standard Mathematical Tables, 27th Edition* (1985). The four point technique has also been extended to the proportional term. With all that it will have some inherent filtering. The coefficents in the array are arranged in order, to use on the oldest to latest sample.

A final difference from Takahashi's book form is that the integral is in terms of repeats per unit-time.

This function takes a structure pointer. That structure holds everything unique to the channel of control, including the process and output history.

The variable *offset* is set and used to move *pPvLast* to the destination of the next process sample. This location is the present location of the oldest sample. In mode `MANUAL` it just returns from here, but in `AUTOMATIC` the output is updated.

In updating the output, the derivitives are calculated, from the four last samples of the process variable, using the coefficients. This begins at the oldest sample, indicated by offest, and walks to the last.

Next, the error between process and setpoint is computed. We then integrate the process variable's derivative, the error and the process variable's second derivative. That results in a correction based on the process's proportional, the error's integral and process's derivative.

Finally, the running output is clamped to the limits, which could be the limits of the integer's type, or something smaller.

This function should be called whenever a fresh process variable has been written to *pPvLast*.

```
int16_t takDdc(ddcParameters *pPar_s)
{
  const int8_t derCoef[] = {2, -9, 18, -11};   /* these four coefficients are in sixths */
  const int8_t secDerCoef[] = {2, -5, 4, -1};   /* these four are in units */
  _Static_assert(sizeof (derCoef)/sizeof (derCoef[0]) == PIDSAMPCT, "PID sample mismatch");
  _Static_assert(sizeof (secDerCoef)/sizeof (secDerCoef[0]) == PIDSAMPCT, "PID sample mismatch");
  uint8_t offset = pPar_s->pPvLast - pPar_s->pPvN;   /* locate latest process variable */
  pPar_s->pPvLast = pPar_s->pPvN + (++offset) % PIDSAMPCT;   /* update the location for the next
      sample */   /* at this point offset points at the oldest sample */
  if (pPar_s->mode == AUTOMATIC) {
    int16_t dDer = 0, dSecDer = 0;
    for (int8_t coIdx = 0; coIdx < PIDSAMPCT; coIdx++) {
      dDer += derCoef[coIdx] * *(pPar_s->pPvN + offset % PIDSAMPCT);
      dSecDer += secDerCoef[coIdx] * *(pPar_s->pPvN + offset % PIDSAMPCT);
      offset++;
    }
    dDer /= 6;   /* since the derivative was in sixths we must divide by six */
    int16_t err = pPar_s->setpoint - *pPar_s->pPvLast;
    pPar_s->m += pPar_s->k_p * (dDer + pPar_s->k_i * err - pPar_s->k_d * dSecDer);
    pPar_s->m = int16clamp(pPar_s->m, pPar_s->mMin, pPar_s->mMax);
  }
  return pPar_s->m;
}
```

52. Takahashi Discrete Digital Control PID and Period initialization. Call this once to set parameters, or when they are changed.

```
void takDdcSetPid(ddcParameters *pPar_s, int16_t p, int16_t i, int16_t d, int16_t t)
{
  pPar_s->t = t;
  pPar_s->k_p = (int16_t) p;
  pPar_s->k_i = (int16_t) i/pPar_s->t;
  pPar_s->k_d = (int16_t) d/pPar_s->t;    /* set the process value pointer to the first position */
  pPar_s->pPvLast = pPar_s->pPvN;
}
```

53. 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);
}
```

54. 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;
}
```

55. ⟨ Initialize pin outputs 55 ⟩ ≡

```
    /* set the led port direction; This is pin #17 */
DDRB |= (1 << DDB5);    /* set the relay port direction; This is pin #8 */
DDRB |= (1 << DDB0);    /* 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 16.

56. ⟨ Configure to idle on sleep 56 ⟩ ≡

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

This code is used in section 19.

57. This section configures the analog section for both analog and input capture through the `MUX`. Since the `MUX` is used `AIN1` and `AIN0` may still be used for digital data. Default is `ICR` on channel 0 but by setting the MUX to channel 2 and clearing `ADEN`, an ADC conversion will occour on the next idle. Conversion will take about 191 μs and will complete with an interrupt.

58. ⟨ Initialize the inputs and capture mode 58 ⟩ ≡

```
{   /* ADCSRA ADC Control and Status Register A */
  ADCSRA &= ~(1 << ADEN);   /* Conn the MUX to (-) input of comparator (sec 23.2) */
  ADCSRA &= ~((1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0));   /* prescaler to 128 */
  ADCSRA &= ~(1 << ADIE);   /* ADC to interrupt on completion */
    /* 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 << ADC2D) | (1 << ADC1D) | (1 << ADC0D));   /* Disable din (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 = (ADMUX & #f0) | 0U;   /* Set to mux channel 0 */
  ADMUX &= ~(1 << REFS0);   /* Set ADC to use VREF */
}
```

This code is used in section 16.

59. For a timer tick at each $\frac{1}{4}$ 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—a lot. The prescaler is set to it's 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 so we can use a simple compare, and no resets. A divisor of 256 looks perfect since it is a small as we can go and still fit the ticks in the small 8 bit timer. The time is trimmed to make 256 passes close to 0.25 seconds by loading compare register, `OCR2A`, with 243. The interval, with the software divisor, is $f = \frac{f_{CPU}}{divisor \times prescale \times (1 + register_{compare})}$ or $\frac{16 \times 10^6}{256 \times 1024 \times (1+243)} \approx 0.25 seconds$. 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 0 and 128. The test could look a bit like `"!(++tickCount␣\&␣~|divisor|U)"` except at 256; but we are at 256 so `"!(++tickCount)"` will do.

60. ⟨ Initialize tick timer 60 ⟩ ≡

```
{
  TCCR2B |= (1 << CS22) | (1 << CS21) | (1 << CS20);   /* maximum prescale (see 18.11.2) */
  TCCR2A |= (1 << WGM21);   /* CTC mode (see 18.11.1) */
  OCR2A = 243U;   /* Do I need to make this clearer? */
  TIMSK2 |= (1 << OCIE2A);   /* Interrupt on a compare match */
}
```

This code is used in section 16.

61. See section 11.8 in the datasheet for details on the Watchdog Timer. This is in the "Interrupt Mode". When controlled remotley 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.

62. ⟨ Initialize watchdog timer 62 ⟩ ≡

```
{
  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 16.

63. 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.

64. ⟨ Initialize the Timer Counter 0 for PWM 64 ⟩ ≡

```
{    /* 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 18.