September 18, 2016 at 23:51

1. Introduction. This is the firmware portion of the propulsion and dive system for our 2016 Champbot. It features separate thrust and steering, including piruett turning.

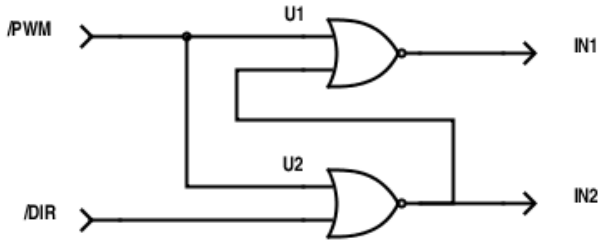
It facilitates lateral motion by taking "thrust" and "radius" pulse-width, or PWC, inputs from the Futaba-Kyosho RC receiver and converting them to the appropriate motor actions.

Thrust is receiver-channel 2, entering analog input ADC1 (PC1), and Radius is channel 1, at ADC0 (PC0). 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.

For PWM we are using Timer Counter 0. 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.

PWM is OC0B and OC0A (PD5 and PD6), located on Trinket pins 5 an 6. The A fail-safe relay output will be at pin 8 (PB0).

For 2016 an autonomous dive function has been added. As in 2015, dive is performed by full reverse thrust but, with this new feature, this thrust is modulated to maintain a specified depth, as determined by a pressure sensor in the electronics bay. The sensor signal connects to ADC2 (PC2) through a voltage divider. The divider scales the 5 volt range of the sensor to the 1.1 volt range of the ADC. By program, it will maintain this depth for 12 seconds, two seconds longer than required.

This timing is through a tick interrupt by Timer Counter 2.

Dive is initiated by a low signal on FTDI pin #4. A relay circuit across FTDI #4 and #1 (Gnd) will do this.

2. Implementation. The Flysky FS-IA6 receiver has six `PWC` channels. The pulse-width from the receiver is at 20 ms intervals but is adjustable. The pulses start simultaniously but end at the time corosponding to the controls. The on-time probably ranges from 1000–2000 μs including trim. 1500 μs is the pulse-width for center or stop. The levers may cover $\pm$400 μs and the trim may cover 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.

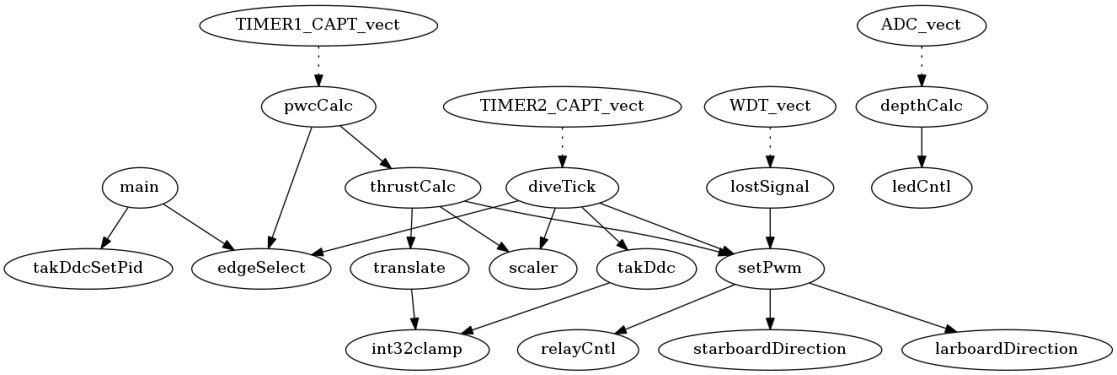
The best way to use this nice feature is to take the `PWC` signals into the `MUX`, through the comparator and into the Input Capture Unit.

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.

After each pulses captured from its respective channel, there's an 18 ms dead-time. That's over 250,000 clock cycles. This will provide ample time to sample pressure and do all of the 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 7 ⟩
⟨ Types 8 ⟩
⟨ Prototypes 14 ⟩
⟨ Global variables 15 ⟩

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

#define `F_CPU` 16000000_{UL}

4. Boolean definitions used everywhere.

```
#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. Other definitions. It is critical that `MAX_DUTYCYCLE` is 98% or less.

```
#define MAX_DUTYCYCLE 98 /* 98% to support charge pump of bridge-driver */
```

6. Interrupt Controls to allow selective development.

```
#define WATCHDOG ON    /* reset and all */
#define ANALOG ON
#define TICK ON    /* TIMER2 */
#define CAPTURE ON    /* TIMER1 */
```

7. ⟨ Include 7 ⟩ ≡

```
#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.

8. 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 power 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;

```
#define PIDSAMPCT 4    /* The PID sample count for derivatives */
```

⟨ Types 8 ⟩ ≡

```
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 9, 10, 11, and 12.

This code is used in section 2.

9. pInput_s→edge may be any of these.

⟨ Types 8 ⟩ +≡

```
typedef enum {
  CH1RISE,      /* The rising edge of RC's remote channel 1 */
  CH1FALL,      /* The falling edge of RC's remote channel 1 */
  CH2RISE,      /* The rising edge of RC's remote channel 2 */
  CH2FALL,      /* The falling edge of RC's remote channel 2 */
  ALLOWPRESSURE     /* Period to allow pressure check */
} edges_t;
```

10. pInput_s→controlMode may be any of these.

```
⟨ Types 8 ⟩ +≡
  typedef enum {
    IDLE,      /* The mode of being surfaced */
    REMOTE,    /* The mode of being surfaced */
    DIVING,    /* The mode of actively diving */
    SUBMERGED  /* The mode of being submerged */
  } controlModes_t;
```

11. Here is the 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 8 ⟩ +≡
  typedef struct {
    uint16_t ch1rise;
    uint16_t ch1fall;
    uint16_t ch2rise;
    uint16_t ch2fall;
    uint16_t ch1duration;
    uint16_t ch2duration;
    uint8_t stopped;      /* boolean TRUE or FALSE */
    edges_t edge;
    controlModes_t controlMode;
    int16_t depth;      /* signed depth in cm */
    const uint16_t minIn;      /* input, minimum */
    const uint16_t maxIn;      /* input, maximum */
    ddcParameters *pPid_s;
  } inputStruct;
```

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

```
⟨ Types 8 ⟩ +≡
  typedef struct {
    int16_t thrust;      /* -255 to 255 */
    int16_t radius;      /* -255 to 255 */
    const int16_t track;
    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;
```

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

14. ⟨ Prototypes 14 ⟩ ≡

```
void relayCntl(int8_t state);
void ledCntl(int8_t state);
void larboardDirection(int8_t state);
void starboardDirection(int8_t state);
void depthCalc(inputStruct *);
void diveTick(inputStruct *);
void pwcCalc(inputStruct *);
void edgeSelect(inputStruct *);
void translate(transStruct *);
void setPwm(int16_t, int16_t);
void lostSignal(inputStruct *);
void thrustCalc(inputStruct *);
int32_t scaler(int32_t input, int32_t minIn, int32_t maxIn, int32_t minOut, int32_t maxOut);
int32_t int32clamp(int32_t value, int32_t min, int32_t max);
void takDdcSetPid(ddcParameters *, int16_tp, int16_ti, int16_td, int16_tt);
int16_t takDdc(ddcParameters *);
```

This code is used in section 2.

15. 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 15 ⟩ ≡

```
void (*handleIrq)(inputStruct *) = Λ;
int main(void)
    {
```

This code is used in section 2.

16. 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. The output, m, is ± is the range of uint8 and so positive. Note also that the maximum duty cycle is used as maximum. The setpoint of 100 cm is set here.

```
inputStruct *pInput_s = &(inputStruct) {
.edge = CH1RISE,
.controlMode = IDLE,
.pPid_s = &(ddcParameters) {
                .k_p = 1,
                .k_i = 1,
                .k_d = 1,
                .t = 1,
                .m = 0,
                .setpoint = 100,
                .mMin = −10000L,
                .mMax = 10000L,
                .mode = AUTOMATIC
                }
};
takDdcSetPid(pInput_s→pPid_s, 20, 2, 2, 1);
```

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

```
cli( );
⟨ Initialize watchdog timer 70 ⟩⟨ Initialize capture mode 66 ⟩⟨ Initialize pin inputs 63 ⟩⟨ Initialize pin
    outputs 62 ⟩⟨ Initialize tick timer 68 ⟩
```

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

```
sei( );
```

19.

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

```
⟨ Initialize the Timer Counter 0 for PWM 72 ⟩
```

20. Rather than burning loops, waiting the balance 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 64 ⟩
```

21. 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);
```

22. 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, tick or watchdog timeout.

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

23. Now that a loop is started, the drive PWM has its values 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 to work with.

First, if we are still in this loop, ensure that the watchdog stays in interrupt mode.

```
#
if WATCHDOG WDTCSR |= (1 ≪ WDIE);
#
else WDTCSR &= ∼(1 ≪ WDIE);
#
endif sleep_mode ( );
```

24. 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 ≠ Λ) {
  handleIrq(pInput_s);
  handleIrq = Λ;
}
}     /* end for */
return 0;
}     /* end main() */
```

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

26. 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;
}
```

27. 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;
}
```

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

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

29. 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 IDLE.

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

30. 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)
    {
```

31. This is called by the input capture interrupt vector. If we are diving of submerged there is an early return so that the dive cannot be interrupted.

Counting always starts on the rising edge and stops on the falling. On the falling edges we can compute the durations using modulus subtraction.

Arrival at the last case establishes that there was a signal and sets mode to REMOTE.

```
if (pInput_s→controlMode ≥ DIVING) return;
switch (pInput_s→edge) {
case CH1RISE: pInput_s→ch1rise = ICR1;
  pInput_s→edge = CH1FALL;
  wdt_reset( );
  break;
case CH1FALL: pInput_s→ch1fall = ICR1;
  pInput_s→ch1duration = pInput_s→ch1fall − pInput_s→ch1rise;
  pInput_s→edge = CH2RISE;
  wdt_reset( );
  break;
case CH2RISE: pInput_s→ch2rise = ICR1;
  pInput_s→edge = CH2FALL;
  wdt_reset( );
  break;
case CH2FALL: pInput_s→ch2fall = ICR1;
  pInput_s→ch2duration = pInput_s→ch2fall − pInput_s→ch2rise;
  pInput_s→edge = ALLOWPRESSURE;
  if (pInput_s→controlMode ≡ IDLE) pInput_s→controlMode = REMOTE;
  wdt_reset( );
  break;
case ALLOWPRESSURE: pInput_s→edge = CH1RISE;
                }
edgeSelect(pInput_s);
thrustCalc(pInput_s);
}
```

32. This procedure connects pwc to pwm.

```
void thrustCalc(inputStruct *pInput_s)
    {
```

33. The Flysky FS-IA6 receiver channels start with a synchronous rising edge so we will start looking for that by setting *edge* to look for a rise on channel 1.

Center position of the controller should result in a count of about 23392, hard About $\frac{4}{5}$ of the range are the full swing of the stick, without trim. This is from about 25990 and 41850 ticks. This is when the FlySky is set at 50 Hz. It seems that the width is scaled by frequency.

. *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 = 25990U;      /* minimum normal value from receiver */
const uint16_t maxIn = 41850U;      /* maximum normal value from receiver */
const int16_t minOut = INT8_MIN;    /* minimum value of thrust */
const int16_t maxOut = INT8_MAX;    /* maximum value of thrust */
```

34. This is the structure that holds output parameters. Track represents the prop-to-prop distance. It should probably be adjusted to minimize turn radius. With the Flysky this value is pretty large at 520.

```
transStruct *pTranslation_s = &(transStruct) {.deadBand = 10, .track = 520
  /* Represents unit-less prop-to-prop distance */
};
```

35. Here we scale the `PWC` durations and apply the "deadBand".

```
{
  int16_t outputCh1;
  int16_t outputCh2;
  if (pInput_s→controlMode ≠ IDLE) {
    outputCh1 = scaler(pInput_s→ch1duration, minIn, maxIn, minOut, maxOut);
    outputCh2 = scaler(pInput_s→ch2duration, minIn, maxIn, minOut, maxOut);
  }
  else {
    outputCh1 = 0;
    outputCh2 = 0;
  }     /* Apply deadband' */
  outputCh1 = (abs(outputCh1) > pTranslation_s→deadBand) ? outputCh1 : 0;
  outputCh2 = (abs(outputCh2) > pTranslation_s→deadBand) ? outputCh2 : 0;
  pTranslation_s→radius = outputCh1;
  pTranslation_s→thrust = outputCh2;
}
translate(pTranslation_s);
if (pInput_s→controlMode ≡ REMOTE) setPwm(pTranslation_s→larboardOut, pTranslation_s→starboardOut);
```

36. Here we check if dive is allowed and, if so, the dive signal is checked. If that signal is LOW, move to DIVING mode. The LED is used to indicate when both channels PWM's are zeros.

```
  if (pTranslation_s→larboardOut ∨ pTranslation_s→starboardOut) {
    pInput_s→stopped = FALSE;
#if 0
    ledCntl(OFF);
#endif
  }
  else {
    pInput_s→stopped = TRUE;
#if 0
    ledCntl(ON);
#endif
  }
  }
```

37. This procedure sets output to zero and resets edge in the event of a lost signal.

```
void lostSignal(inputStruct *pInput_s)
{
  if (pInput_s→controlMode ≤ REMOTE) pInput_s→controlMode = IDLE;
  if (pInput_s→controlMode ≡ IDLE) setPwm(0, 0);
  pInput_s→edge = ALLOWPRESSURE;
  wdt_reset( );
}
```

38. This procedure maintains various timers.

```
void diveTick(inputStruct *pInput_s)
    {
    const uint8_t oneSecond = 64;
    static uint8_t tickCount = 0;
    const uint16_t divingSeconds = 20 * oneSecond;
    static uint16_t divingCount = divingSeconds;
    const uint16_t submersedSeconds = 12 * oneSecond;
    static uint16_t submersedCount = submersedSeconds;
```

39. We are here 64 times per second through the tick interrupt. First, if a pressure reading is allowed, use edgeSelect to set up for a reading. It will only get the ADC value when ALLOWPRESSURE is true, happening after both pwc channels have been read.

```
  if (pInput_s→edge ≡ ALLOWPRESSURE) edgeSelect(pInput_s);
```

40. This will count off ticks for a $\frac{1}{4}$ second event. This is the Direct Digital Control crunch interval. Every interval the depth is collected and DDC is run so that it is always ready to take the reins.

```
  if (¬(tickCount += oneSecond/4)) {
    *pInput_s→pPid_s→pPvLast = pInput_s→depth;
    takDdc(pInput_s→pPid_s);
    if (pInput_s→controlMode ≥ DIVING) {
      int16_t output = scaler(−(pInput_s→pPid_s→m), pInput_s→pPid_s→mMin, pInput_s→pPid_s→mMax,
          −(MAX_DUTYCYCLE * UINT8_MAX)/100L, 0L);
      setPwm(output, output);
      if (*pInput_s→pPid_s→pPvLast ≥ 95L)      /* Being within 5 cm is fine */
        pInput_s→controlMode = SUBMERGED;
    }
    else      /* To ensure output is safe when it enters DIVING mode */
    {
      pInput_s→pPid_s→m = 0L;
    }
  }     /* Debounce the dive start */
  if (pInput_s→stopped ≡ TRUE) {
    const uint8_t debticks = oneSecond;
    static uint8_t debcount = debticks;
    if ((PIND & (1 ≪ PD0)) ∧ debcount < debticks) debcount++;
    if ((∼PIND & (1 ≪ PD0)) ∧ debcount > 0) debcount−−;
    if (¬debcount) pInput_s→controlMode = DIVING;
  }
```

41. These two timers limit the duration of these modes.

```
  divingCount = (pInput_s→controlMode ≡ DIVING) ? divingCount − 1 : divingSeconds;
  if (divingCount ≡ 0) pInput_s→controlMode = IDLE;
  submersedCount = (pInput_s→controlMode ≡ SUBMERGED) ? submersedCount − 1 : submersedSeconds;
  if (submersedCount ≡ 0) pInput_s→controlMode = IDLE;
#if 0
  if (pInput_s→controlMode ≥ DIVING) ledCntl(ON);
  else ledCntl(OFF);
#endif
  }
```

42. This procedure will filter ADC results for a pressure in terms of ADC units and convert that to signed integer depth in centimeters. 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*.

Next, this must be in terms of depth. The sensor was tested and has a 0.49 V output at the surface and that output increases by 12 mV per centimeter of depth.

By measurement of the sensor's output, 1000 units of ADC is 4.14 Volts or 0.00414 mV/unit. Offset is then $\frac{0.49}{0.00414}$ for 118 units. That means 118 ADC units at the surface.

Gain is $\frac{0.00414}{0.012}$ for 0.345. Since 0.345 is a floating point, and we don't want slow, massive floating-point libraries, we can multiply this by 2^5, and round it, for a gain of 11. Depth will then be in a large integer of $\frac{1}{32}$ cm. Then we will just need to right-shift that big number by 5 places (thus dividing by 32) to get to integer centimeters.

Conversion to depth is signed since the offset is fixed and the sensor may drift a bit. The danger in an unsigned is that when surfaced, the offset subtraction may result in an instant extreme, apparent depth.

```
void depthCalc(inputStruct *pInput_s)
    {
    const int16_t offset = 118;      /* units of ADC offset from zero depth */
    const int16_t gain = 11;      /* units of gain in 1/32 cm per ADC unit */
    static uint16_t buffStart[1 << 4] = {0};
    const uint16_t *buffEnd = buffStart + (1 << 4) - 1;
    static uint16_t *buffIndex = buffStart;
    static uint16_t sum = 0;      /* Accommodates size up to 1¡¡6 only */
```

43. Now that we have our sample, pass control back to the input capture interrupt.

```
  ADCSRA &= ~(1 << ADEN);      /* Reconnect the MUX to the comparator */
  ADMUX = (ADMUX & ~0xf0) | 1U;      /* Set to mux channel 1 */
  sum -= *buffIndex;      /* Remove the oldest item from the sum */
  *buffIndex = (uint16_t)ADCW;      /* Read the whole 16 bit word with ADCW */
  sum += *buffIndex;      /* Include this new item in the sum */
  buffIndex = (buffIndex != buffEnd) ? buffIndex + 1 : buffStart;
  pInput_s->depth = (int16_t)(((sum >> 5) - offset) * gain)/32;
#if 1      /* test one meter */
  if (pInput_s->depth > 100) ledCntl(OFF);
  else ledCntl(ON);
#endif
  }
```

44. The procedure edgeSelect configures the "Input Capture" unit to capture on the expected edge type from the remote control's proportional signal.

```
void edgeSelect(inputStruct *pInput_s)
    {
    switch (pInput_s→edge) {
    case ALLOWPRESSURE: ADCSRA |= (1 ≪ ADEN);
        /* Connect the MUX to the ADC and enable it */
      ADMUX = (ADMUX & ~f0) | 2U;      /* Set MUX to channel 2 */
      return;      /* no need to stick around */
    case CH1RISE:      /* To wait for rising edge on rx-channel 1 */
      ADMUX = (ADMUX & ~f0) | 0U;      /* Set to mux channel 0 */
      TCCR1B |= (1 ≪ ICES1);      /* Rising edge (23.3.2) */
      break;
    case CH1FALL: ADMUX = (ADMUX & ~f0) | 0U;      /* Set to mux channel 0 */
      TCCR1B &= ∼(1 ≪ ICES1);      /* Falling edge (23.3.2) */
      break;
    case CH2RISE:      /* To wait for rising edge on rx-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 */
      ADCSRA &= ∼(1 ≪ ADEN);      /* Reconnect the MUX to the comparator */
      TCCR1B &= ∼(1 ≪ ICES1);      /* Falling edge (23.3.2) */
    }
```

45. Since the edge has now 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) */
}
```

46. 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*. This is used to translate the stick position of the remote into terms that we can use.

```
int32_t scaler(int32_t input, int32_t minIn, int32_t maxIn, int32_t minOut, int32_t maxOut)
    {
```

47. 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;
```

48. 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 values. OK, maybe I could use the preprocessor but compiler optimization probably makes this pretty good.

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 * (maxIn - minIn))/(maxOut - minOut);
int32_t offset = ((ampFact * minIn)/gain) - minOut;
return (ampFact * input/gain) - offset;
}
```

49. 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.

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;
```

50. 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*. From testing it seems like this track value is fine.

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

51. 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 = int32clamp(speed − rotation, −max, max);
  trans_s→starboardOut = int32clamp(speed + rotation, −max, max);
}
else      /* piruett mode */
{ lock = (abs(piruett) > pirLockLevel) ? ON : OFF;
trans_s→larboardOut = int32clamp(piruett, −max, max);
```

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

```
piruett = −piruett;
trans_s→starboardOut = int32clamp(piruett, −max, max); }
}
```

53. This procedure sets the signal to the H-Bridge. Forward is positive an negative is reverse. For the PWM we load the value into the unsigned registers. There is no duty-cycle limit here.

```
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);
    }
```

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

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

55. 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);
}
```

56. Here is a simple procedure to flip the Relay Closed or Open from pin D8.

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

57. 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);
}
```

58. 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);
}
```

59. A simple 32 bit clamp function.

```
int32_t int32clamp(int32_t value, int32_t min, int32_t max)
{
  return (value > max) ? max : (value < min) ? min : value;
}
```

60. This is the DDC or Direct Digital Control 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.

$$\Delta m_N = K_p(c_{N-1} - c_N) + K_i(r_N - c_N) + K_d(2c_{N-1} - c_{N-2} - c_N)$$

where

$$K_i = T/T_1,\ K_d = T_d/T.$$

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 computed from the distance between pointers. *offset*, with some modulus arithmatic, is used to move *pPvLast* to the destination of the next process sample. This new location is also the present location of the oldest sample—that will be the first sample used for the derivatives. In mode `MANUAL` it just returns from here, but in `AUTOMATIC` the output is updated.

The derivitives are then 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 latest.

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.

A safe value should be written to m before starting control. This function should be called after 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");
  int32_t total;
  uint8_t offset = pPar_s→pPvLast - pPar_s→pPvN;
  pPar_s→pPvLast = pPar_s→pPvN + (++offset % PIDSAMPCT);
    /* at this point offset points at the oldest sample */
  if (pPar_s→mode ≡ AUTOMATIC) {
    int16_t dDer = 0, dSecDer = 0;
    for (int8_t coefIdx = 0; coefIdx < PIDSAMPCT; coefIdx++) {
      dDer += derCoef[coefIdx] * pPar_s→pPvN[offset % PIDSAMPCT];
      dSecDer += secDerCoef[coefIdx] * pPar_s→pPvN[offset % PIDSAMPCT];
      offset++;
    }      /* Since the derivative was in sixths we must divide by six */
    dDer /= 6;
    int16_t err = pPar_s→setpoint - *pPar_s→pPvLast;
```

```
    total = pPar_s→k_p * (dDer + pPar_s→k_i * err - pPar_s→k_d * dSecDer);
    pPar_s→m = int32clamp((pPar_s→m + total), pPar_s→mMin, pPar_s→mMax);
  }
  return pPar_s→m;
}
```

61. Control Initialization. Takahashi Direct 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;
}
```

62. ⟨ Initialize pin outputs 62 ⟩ ≡

```
/* 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 */
DDRB |= ((1 ≪ DDD3) | (1 ≪ DDD4));      /* Data direction to output (sec 14.3.3) */
  /* set the dive input port direction; This is pin #8 */
DDRB |= (1 ≪ DDD0);
```

This code is used in section 17.

63. ⟨ Initialize pin inputs 63 ⟩ ≡

```
/* set the dive input port direction; This is FTDI pin #4 */
DDRB &= ∼(1 ≪ DDD0);      /* set the dive input pull-up */
PORTD |= (1 ≪ PORTD0);
```

This code is used in section 17.

64. ⟨ Configure to idle on sleep 64 ⟩ ≡

```
{
  SMCR &= ∼((1 ≪ SM2) | (1 ≪ SM1) | (1 ≪ SM0));
}
```

This code is used in section 20.

65. Configuration. 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 comming from the receiver. Default is `ICR` on channel 0 but by setting the MUX to channel `ADC2` and clearing `ADEN`, an ADC conversion will occur on the next idle. Conversion will take about 191 μs and will complete with an interrupt.

66. ⟨ Initialize capture mode 66 ⟩ ≡

```
{    /* 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 */
#
if ANALOG ADCSRA |= (1 << ADIE);    /* ADC to interrupt on completion */
# endif    /* 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 */
#
if CAPTURE TIMSK1 |= (1 << ICIE1);    /* Enable input capture interrupt (sec 16.11.8) */
# endif    /* 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) | (1 << REFS1);    /* Set ADC to use internal 1.1 V reference */
}
```

This code is used in section 17.

67. For a timer tick at each $\frac{1}{4}$ second. We will use timer counter 2, our last timer. This 8 bit timer only has a 10 bit prescaler so it will need to be divided a lot. The prescaler is set to it's maximum of 1024 for 15625 Hz from the 16 Mhz clock. The timer is set to `CTC` mode so that the time loop is trimmable. In software, we want to divide by a power of two so we can use a simple compare and, here in this timer, no counter resets.

We ultimatly need a divisor of $\frac{15625}{4}$ or 3906. A divisor of 256 is possible in the counter. 3906.2/256 is very near a power of 2, specificaly 16. 3906.2 is 244.14, and, according to the datasheet, it interupts one count past that. The time is trimmed to make 16 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 $\frac{16 \times 16^6}{256 \times 1024 \times (1+243)} \approx 0.25 seconds$. The interrupt is enabled `TIMSK2` for output compare register A. With all that we will have interrupt `TIMER2 COMPA` fire every 15.616 ms.

68. ⟨ Initialize tick timer 68 ⟩ ≡

```
{
  TCCR2B |= (1 ≪ CS22) | (1 ≪ CS21) | (1 ≪ CS20);      /* maximum prescale (see 18.11.2) */
  TCCR2A |= (1 ≪ WGM21);      /* CTC mode (see 18.11.1) */
  OCR2A = 243U;
#if TICK
  TIMSK2 |= (1 ≪ OCIE2A);      /* Interrupt on a compare match */
#endif
}
```

This code is used in section 17.

69. See section 11.8 in the datasheet for details on the Watchdog Timer. This is in the "Interrupt Mode" through WDIE. When controlled remotly or in an autonomous dive this should not time-out.

70. ⟨ Initialize watchdog timer 70 ⟩ ≡

```
{
  wdt_reset( );
  MCUSR &= ∼(1 ≪ WDRF);
  WDTCSR |= (1 ≪ WDCE) | (1 ≪ WDE);      /* This combo unlocks it */
  WDTCSR = (1 ≪ WDE) | (1 ≪ WDP2) | (1 ≪ WDP0);
  WDTCSR |= (1 ≪ WDIE);      /* Set the interrupt on */
    /* Reset is after about 0.5 seconds (see 11.9.2) */
}
```

This code is used in section 17.

71. PWM setup isn't too scary. Timer Counter 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.

72. ⟨ Initialize the Timer Counter 0 for PWM 72 ⟩ ≡

```
{    /* 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 19.

⟨ Configure to idle on sleep 64 ⟩ Used in section 20.
⟨ Global variables 15 ⟩ Used in section 2.
⟨ Include 7 ⟩ Used in section 2.
⟨ Initialize capture mode 66 ⟩ Used in section 17.
⟨ Initialize pin inputs 63 ⟩ Used in section 17.
⟨ Initialize pin outputs 62 ⟩ Used in section 17.
⟨ Initialize the Timer Counter 0 for PWM 72 ⟩ Used in section 19.
⟨ Initialize tick timer 68 ⟩ Used in section 17.
⟨ Initialize watchdog timer 70 ⟩ Used in section 17.
⟨ Prototypes 14 ⟩ Used in section 2.
⟨ Types 8, 9, 10, 11, 12 ⟩ Used in section 2.