-Testing speed controller

ETC 2006 Software Roadmap

- Build

Intro
This road map is an attempt to lay out all the software that needs to be written in order to get the 2006 ETC system operational. The Style Guide should be read first. The guide is important so that all our code conforms to a common style. A common style allows for better comprehension and portability between team members.

All the subsections in the Device Drivers section need to be coded. The applications in the Support Applications section need to be coded as well, but these are not critical to the functioning of the system. Don't worry about these until after the device drivers have been written, tested, and debugged.

The directional terms 'right' and 'left' assume the reader is sitting in the Baja car.

Style Guide
This style guide consists of three main sections: Layout, ISRs, and Structures. The Layout refers to where certain functions should be located within the code. The ISRs are the Interrupt Service Routines. In the ETC system, we follow the Linux convention of writing ISRs into Top Halves and Bottom Halves. See the ISRs section for more details.

The majority of information passing between functions is with the use of pointers to global structures. The Structures section explains why we do it this way and what the structures are that we use.

Layout
The layout of the code within the ETC AVR program can be broken into five distinct sections: Init, Capture, Control, Actuation, and Application. Within main() there is an infinite loop. Init is the only section which lies outside of this loop. The other four sections lie within the loop.

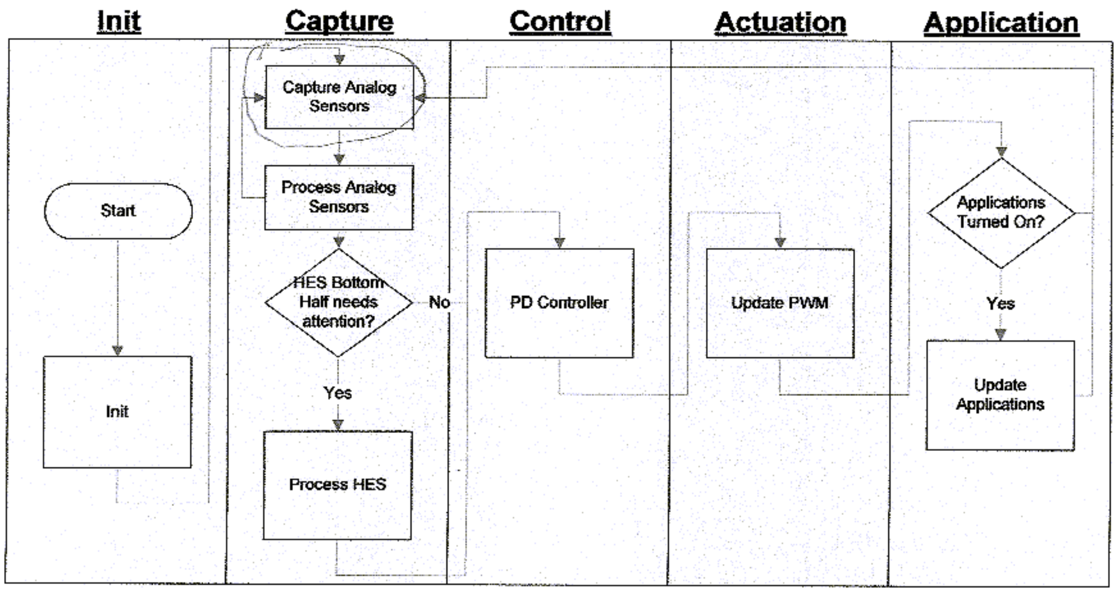

Figure 1 - Software Overview

Init
This section contains all of the initialization functions for enabling the various internal and external peripherals. This code is run only once (at boot up) before the infinite loop is entered.

The important things that are done in this section are:

- Initialize microcontroller peripherals by configuring the required registers
- Initialize all machine state variables into a known state
- Zero the actuators into a known state
- If the user has selected debug or sensor validation mode, the required initialization for these modes of operation should be performed.

Capture
This section is where the sensor data is captured. The ETC system has two types of sensors: digital-asynchronous sensors and analog-synchronous sensors. There are no digital-synchronous sensors or analog-asynchronous sensors on the car as of this writing.

Analog Sensors
The analog-synchronous sensors consist of the following sensors:

- Pressure
- Temperature
- Current
- Steering

Each sensor is connected to the microcontrollers internal ADC via the ADC Multiplexer (MUX). Since only one channel can be attached to the ADC at a time, the microcontroller (uC) must cycle through each analog channel in a synchronous manner (one after the other).

After data has been sampled, it must be converted into a meaningful quantity – i.e. pressure sensor ADC data must be converted into a psi pressure reading. Since the uC ADC has an interrupt that signals when the conversion is done, sensor data can be processed while the ADC is capturing the next sensor signal – i.e. pressure sensor data can be converted from an ADC reading to meaningful data while the ADC is reading the steering wheel sensor. In this way the microcontroller can be used as efficiently (and as fast) as possible. The simultaneous capturing and processing of signals is represented in Figure 1 by the arrow going from the block labeled "Process Analog Sensors" to the block "Capture Analog Sensors".

Digital Sensors
The digital-asynchronous sensors consist of the following sensors:

- Hall Effect Sensors

The digital-asynchronous sensors consist of the two hall effect sensors (HES) used to monitor wheel speed and position. These two sensors are digital in nature and attach to two of the uC's interrupt channels allowing the sensors to operate in an asynchronous manner. The ISR top halves for these sensors debounce the sensors and record time data. The bottom half of the ISR computes the actual wheel speed and wheel speed ratio (WSR).

Since the updating of the HES is dependent on wheel speed, it is very likely that the uC may read in new ADC data before the HES data is updated. In this case, the uC will use the old HES data and move on to the Control section.

Control
In this section the captured data is fed into the PD controller algorithm. The algorithm is responsible for taking the meaningful sensor data as input and outputting a value for the PWM actuator in the Actuation section.

Actuation
The Actuation section is responsible for handling the output of the PD controller and using it to control the ETC braking motor. The actuation is performed by sending a PWM signal to the speed controller.

Application
The application section consists of the MUX control, sensor validation, and serial port programs. The MUX control allows the monitoring of each sensor on the oscilloscope via a single point of hardware. Sensor validation allows validation of each sensor at start-up. The serial port program allows a user to monitor system and sensor status via a laptop over the serial port.

Interrupt Service Routines
The interrupt handling scheme used in the ETC is similar to the one used in linux. An interrupt service routine is split into a top half and a bottom half.

Top Half
The top half is the actual interrupt vector where the microcontroller jumps to when the interrupt occurs. The top half should contain the smallest amount of code possible and should defer as much work as possible to the bottom half of the interrupt handler.

Bottom Half
The bottom half of the ISR is located within the infinite loop within main. Any processor intensive code should be located here. This makes sure that the processor can handle any interrupt requests without interrupting any other time critical code.

Structures
The ETC program makes use of global structures. Newbie uC programmers tend to edit global variables directly from subfunctions which can cause confusion in debugging or to pass several variables into a subfunction which causes unneeded copies of variables on the stack.

As an example, here is an example of what you ***should not*** do:

```
uint8_t capture_state(uint8_t variable1, uint8_t variable2, uint8_t
variable3, ...) {
   // Get state
   return (SUCCESS);
}

...

int main {
   uint8_t status = FAILURE;
   sVehicle vehicle_state;

   status = capture_state(captured_in_ticks, speed, heading);
   if (!status) { /* Freak out! */ }
}
```

This is the ***correct way*** to do it:

```
typedef struct vehicle_state_structure {
   uint32_t captured_in_ticks;
   uint16_t speed;
   int16_t heading;
} sVehicle;

uint8_t capture_state(sVehicle* pVehicle)
{
   // Get state
   return (SUCCESS);
}

...
```

```
int main {
   uint8_t status = FAILURE;
   sVehicle vehicle_state;

   status = capture_state(&vehicle_state);
   if (!status) { /* Freak out! */ }
}
```

In the first example, each variable must be passed in (8-bits each). In the second example, only a single 16-bit address is passed. This allows faster context switching from function to sub-function and eliminates useless copies of variables onto the stack.

In addition, the use of structures is good coding practice. It's what professionals do. If you ever take a peak at the Linux kernel source, you'll see that they use nothing but structure manipulation just as above.

Analog Sensor Structure
Each analog sensor has the following structure:

```
typedef struct analog_sensor_structure {
   uint8_t ADC_channel;
   uint16_t ADC_value;
   int16_t actual_value;
   uin8_t multiplier;
} analog_sensor;      ← uint_8 flag          #define new-data 1
```

`ADC_channel` is the ADC channel to which the sensor is attached. `ADC_value` is the last value read by the ADC from the sensor. `actual_value` is the meaningful value which is computed from the `ADC_value` – i.e. the pressure in psi for the pressure sensor.

`multiplier` is coefficient multiplier for `actual_value`. For instance, let's we compute an `actual_value` of 3.56 degrees from the `ADC_value` for the steering sensor. Since floating point numbers are incredibly inefficient in an 8-bit microcontroller and we can't represent fractions with an integer, we store 356 in `actual_value` and 2 in `multiplier` and we know that the actual answer is `actual_value` / 10^`multiplier` = 3.56.

Digital Sensor Structure
Each asynchronous digital sensor has the following structure:

```
typedef struct async_sensor_structure {
   uint32_t old_clock;
   uint32_t delta_clock;
   uint8_t wheel_speed;
   uin8_t multiplier;
} async_sensor;
```

`old_clock` contains the value of the global `clock40` variable the last time the top half of the sensors ISR was serviced. `delta_clock` contains the time difference between `old_clock` and the current value of the `clock40` variable. `wheel_speed` is the computed wheel speed in miles per hour. `multiplier` is the multiplier coefficient as explained in the section 'Analog Sensor Structure'.

Car Structure
The car has it's own state structure as well:

```
typedef struct car_structure {
      uint8_t car_speed;
      uint16_t WSR;
      uint16_t DWSR;
} baja;
```

This structure is far from complete. It will be updated as needed.

Device Drivers
The ETC is composed of many individual devices (sensors, actuators, and peripherals) which each require their own device drivers. Every device driver has an initialization component and a processing component. The initialization component is run once in the Init section of software and the processing component runs within the infinite loop.

Sensor device drivers are responsible for converting ADC data into a meaningful values, actuator device drivers are responsible for manipulating the actuators, and peripheral device drivers do miscellaneous tasks.

Sensor Devices

Pressure Sensor
Init
- The right pressure sensor is attached to Port F, pin 6 (PF6), which is also ADC channel 6 (ADC6).
- The left pressure sensor is attached to Port F, pin 7 (PF7), which is also ADC channel 7 (ADC7).

Processing
The voltage range of the ETC pressure sensors is from .5V to 4.5V with a frequency response of 1Khz. The pressure sensor data is stored in the following global structure variables:

```
analog_sensor right_pressure, left_pressure;
```

The conversion from ADC values to pressure (in psi) is borrowed from the 04 ETC teams code:

```
uint8_t compute_pressure(analog_sensor *this_sensor) {
      uint16_t ADC_value = this_sensor->ADC_value;
      if(ADC_value <= 110)
                  this_sensor->actual-value = 0;
            else if(ADC_value > 538)
                  this_sensor->actual_value = 539;
            else
                  this_sensor->actual_value =
                  (((((ADC_value  / 2) * 122) / 100) * 2) - 125);
}
```

Hall Effect Sensors
Init

- The right HES sensor is attached to Port E, pin 4 (PE4) which is also External Interrupt channel 4 (INT4).
- The left HES sensor is attached to Port E, pin 5 (PE5) which is also External Interrupt channel 5 (INT5).

Each interrupt channel should be configured for rising edge.

Processing
The HES sensor data is stored in the following global structure variables:

```
async_sensor right_HES, left_HES;
```

Processing of the interrupt caused by a HES takes place in both the top and bottom halves of the interrupt ISRs. The top half is responsible for debouncing the signal – in order to eliminate spurious readings from noise. And calculate the change in time between readings, which is stored in the delta_clock variable. Here is some pseudo code for the top half:

```
/*
 *Interrupt Handler for the HES sensor attached to right wheel.
 */
INTERRUPT(INT4) {
      //Move the value to a local variable for easy manipulation.
      uint32_t new_clock = (clock40.low * 65536) + TCNT1;

      //If the pin is zero, then this interrupt was triggered by noise.
      if(!debounce(&PINE, 4))
            return;

      //Compute the clock cycles between readings.
      if new_clock < right_HES.old_clock
            //131072 = 2^17
            right_HES.delta_clock = (131072 + new_clock) - old_clock;
      else
            right_HES.delta_clock = new_clock - old_clock;

      //Update old_clock value for next ISR.
      right_HES.old_clock = new_clock;

      //Set the bottom half flag so that we service the bottom half.
      bottom_half_ISR_flag = 1;
}
```

The bottom half of the ISR is responsible for computing the actual wheel speed velocity and WSR. Of course, the WSR can not be computed until the bottom half of the right and left HES sensors have had a chance to execute. Here is some pseudo code for the bottom half of the ISR:

```
uint8_t Compute_Wheel_Speed(async_sensor *this_sensor) {

#define ROTOR_LENGTH 2          //Length in inches between rotor notches.
#define LOC_MULT 2              //Local Multiplier. Same as multiplier.
#define VEL_CONV_CONST 5682     //Velocity Conversion Constant between
                                //inches/millisec to miles/hr * LOC_MULT.

      uint16_t velocity;

      this_sensor->multiplier = LOC_MULT;

      //# of milliseconds passed
      velocity = (this_sensor->delta_clock)/16000;

      //Now velocity is it terms of inches/millisecond.
      velocity = ROTOR_LENGTH / velocity;

      //Now velocity is in terms of mi/hr * multiplier.
      this_sensor->wheel_speed = velocity*VEL_CONV_CONST;

}


inline uint8_t compute_WSR(void) {
      Vehicle_state.WSR = right_HES.wheel_speed / left_HES.wheel_speed;
}
```

Thermal Sensor
Init

- The temperature sensor is attached to Port F, pin 1 (PF1), which is also ADC channel 1 (ADC1).

Processing
The voltage range of the thermal sensors is from 0V to 2.5V. The temperature sensor data is stored in the following global structure variables:

```
analog_sensor temp_sensor;
```

The conversion from ADC values to temperature is unknown as of this writing. The datasheet provides a complex equation but the conversion needs to be determined through testing.

Current Sensors
Init

- The motor current sensor is attached to Port F, pin 2 (PF2), which is also ADC channel 2 (ADC2).

Processing
The voltage range of the current sensor is 0 to 2.5 volts. The current sensor data is stored in the following global structure:

```
analog_sensor cur_sensor;
```

The equation for converting from ADC values to current is illustrated in the Electrical System Roadmap.

Steering Sensor
Init

- The steering sensor is attached to Port F, pin 0 (PF0), which is also ADC channel 0 (ADC0).

Processing
The voltage range of the steering sensor is 0 to 5 volts. The steering sensor data is stored in the following global structure:

```
analog_sensor steer_sensor;
```

The steering sensor driver algorithm must be verified through testing, but here is some pseudo code based of the '04 Baja teams code which converts steering ADC directly to DWSR:

```
float wsr_des_lu[255];                          // global wsr desired array

/**
 * Build a wheel speed ratio look up table in the form of an array
 * This function should be run in the INIT section.
 */
void declare_WSR(void)
{
        // Build WSR desired look-up table
        for(int i = 0; i <= 255; i++)
        {
                if(i >= 0 && i < 35)     {wsr_des_lu[i] = 1.55;}       // 1
                else if(i >= 35 && i < 46) {wsr_des_lu[i] = 1.49;}     // 2
                else if(i >= 46 && i < 60) {wsr_des_lu[i] = 1.39;}     // 3
                else if(i >= 60 && i < 73) {wsr_des_lu[i] = 1.30;}     // 4
                else if(i >= 73 && i < 87) {wsr_des_lu[i] = 1.20;}     // 5
                else if(i >= 87 && i < 105)    {wsr_des_lu[i] = 1.04;} // 6
                else if(i >= 105 && i < 122)  {wsr_des_lu[i] = 0.96;} // 7
                else if(i >= 122 && i < 139)  {wsr_des_lu[i] = 0.89;} // 8
                else if(i >= 139 && i < 152)  {wsr_des_lu[i] = 0.83;} // 9
                else if(i >= 152 && i < 177)  {wsr_des_lu[i] = 0.77;} // 10
                else if(i >= 177 && i < 184)  {wsr_des_lu[i] = 0.72;} // 11
                else if(i >= 184 && i < 256)  {wsr_des_lu[i] = 0.67;} // 12
        }
}


float convert_steering_ADC_to_DWSR (analog_sensor *this_sensor) {

        return wsr_des_lu[this_sensor.ADC_value >> 2];
}
```

Actuator Devices
Speed Controller
Init

- The motor driver (RET713) is attached to Port E, pin 3 (PE3), which is also Timer/Counter3 Output Compare Pin A (OC3A).

Processing
To drive the RET713 a square wave with a period of 20mS must be generated. Within the period of the wave, a high time of 2mS is full forward, 1ms is full reverse, and 1.5ms if stop.

Although driving the 16-bit timer at 16MHz will cause it to overflow in 4ms, a scheme similar to the clock40 variable should be used so that the clock can be driven at 16MHz. This will allow for maximum resolution in speed control of the motor.

Peripheral Devices
Timer/Counter1

Init
The Timer/Counter1 is a peripheral internal to the ATMega128. It should be configured for free running mode and clocked directly from the 16Mhz uC clock frequency.

Processing
The 16-bit register TCNT1 comprises the lower 16-bits of the clock40 variable:

8-bit	16-bit	16-bit
clock40.high	clock40.low	TCNT1

As Timer/Counter1 rolls over, it updates the clock40 variable. At 40 effective bits, the clock40 variable will roll over after 19 hours at a clock frequency of 16 MHz. Here is some pseudo code for use in the Timer/Counter1 rollover interrupt:

```
struct clock_struct {
   uint8_t high;
   uint16_t low;
} clock40;


TCNT Rollover ISR {

      clock40.low++;

      if( clock40.low == 0 )
            clock40.high++;
}
```

ADC MUX Controller
This section will describe the algorithm for multiplexing the on-chip ADC between the different analog sensors. It requires an init algorithm that can be borrowed from old code, but it will also require a function for setting up capture of a new channel and an ISR for processing.

The scheme is to set up capture, return and process the previous sensor, then when processing is done, check a flag set by the ISR. If the flag is set, then capture the next sensor and continue processing.

LCD Controller
For now all the algorithms for controlling the on-board LCD are well known. However, if we are to put an LCD display in the cockpit, then a new hardware/software scheme will be necessary. This is not critical to the ETC system though, so this section will be developed at a later date.

Support Algorithms
Debounce
Debounce is an algorithm for ensuring the logic state of a pin. This should be used on all external interrupts to ensure that the interrupt was not triggered by noise. Port and Pin numbers are input to the function and it returns the debounced signal on that pin. If the signal is a logic one it returns 1 and if it's a logic zero it returns 0.

```
/************************************************************************
 * debounce - Debounce a switch
 * ----------------------------
 *  This function debounces a switch. It takes the address to a port
 *  and a pin number and returns the debounced input.
 ***********************************************************************/
uint8_t debounce(uint8_t *port, uint8_t pin)    {

      uint8_t p1 = 0xFF;        //Dummy variable used for debouncing

      //Loop until only one '1' is left to be shifted out
      while((p1 != 0x01) && (p1 != 0x80))       {
            if(*port & pin)
                  p1 = p1<<1;         //If the Switch read as a logical
                                      //1, shift left
            else
                  p1 = p1>>1;         //If the Switch read as a logical
                                      //0, shift right
      }

      //Return debounced results
      if(p1 == 0x80)
            return 1;
      else
            return 0;
}
```

Support Applications

Serial Port

The serial port is used in debugging and testing. A terminal program will be run on a laptop to monitor various system states in real time. The microcontroller is responsible for sending the system states to the laptop over the serial port at certain time intervals. This application still needs to be designed.

Sensor Validation

The concept of sensor validation centers on the belief that you can't trust any of your sensors until you validate them. You always assume that at boot your sensors have been damaged until they are validated.

While we won't validate each sensor at each boot up, a validation scheme needs to be designed for each sensor. Once this is designed, it will allow the microcontroller to test and validate that each sensor is working correctly while in the field.

Analog MUX Control

The MUX Controller we will work on last since it's not an essential peripheral. It's purpose is to allow us to connect the oscilloscope to one point on the board and be able to monitor all the analog sensors from that one point.