Controlling the Beast

Interfacing Analog Controls to a Digital World

My workshop has been storing retired parts from my combat robot, Hexadecimator since it appeared on the original BattleBots™ TV show in the early 2000s. To see if these parts could be put to better use, I handed them over to my 14 year old grandson and his dad, and told them to build whatever they wanted. The parts I gave them consisted of motors, batteries, and a controller. (For a complete list, refer to **Table 1**.)

Later, I came across an old Microsoft Sidewinder gaming steering wheel and pedals tucked away in a cabinet. Once I had verified that deep down inside, ordinary potentiometers were used for steering and pedal position, I handed them over as well. After much cutting, welding, etc., they came up with an electric go-cart (**Figure 1**), Then, they asked *me* to wire it up.

The Control Interface

The motor controller was designed to be directly connected to a standard radio control (RC) receiver like you'd find in an RC car, plane, or boat. It has one input for throttle and one for steering. These inputs each accept a standard servo pulse stream, which is a pulse of 1-2 milliseconds, where a 1.5 ms pulse is neutral. Given these inputs for steering and throttle, the motor controller generates a pulse-width modulated (PWM) output to control the speed of each drive motor. To turn, the controller speeds up one motor while slowing down the other; this is sometimes called tank steering (as in the tracks on a battle tank). On the other side of the control interface are the pure analog signals from three potentiometers: one in the steering wheel; one on the accelerator pedal; and one on the brake pedal. To bridge this divide, I looked in my parts bin and plucked out an Atmel AVR microcontroller unit (MCU) — specifically, an ATtiny84. I chose it because:

Table 1. Retired components from Hexadecimator that were repurposed for the electric go-cart.

Quantity	Description	Supplier	Part #	Website
2	1 HP gear motor	NPC Robotics	NPC-T64	http://hobby.npcrobotics.com
2	Wheel & foam-filled tire	NPC Robotics	NPC-PT444	
2	Wheel adapter hub	NPC Robotics	NPC-PH448	
1	36V dual motor controller	Vantec	RDFR36E	www.vantec.com
4	36V 3.6 Ahr NiCd battery pack	Robotic Power	Custom	www.battlepack.com

By Tim Paterson

Post comments on this section and find any associated files and/or downloads at **www.servomagazine.com/index.php/magazine/article/october2015_Paterson**.

Resources

Atmel Studio
www.atmel.com/tools/ATMEL STUDIO.aspx
AVR Dragon
www.mouser.com
AVR IO Designer
www.forestmoon.com/Software
Test drive video
https://youtu.be/mDvxmWTZHqo

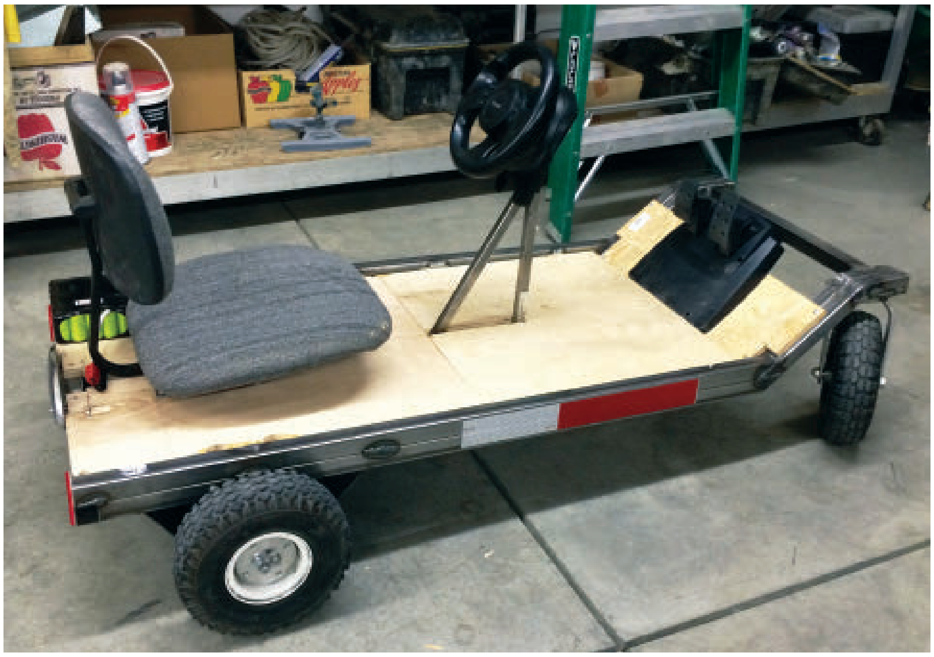


Figure 1. It's mechanically complete, but note the complete lack of wires!

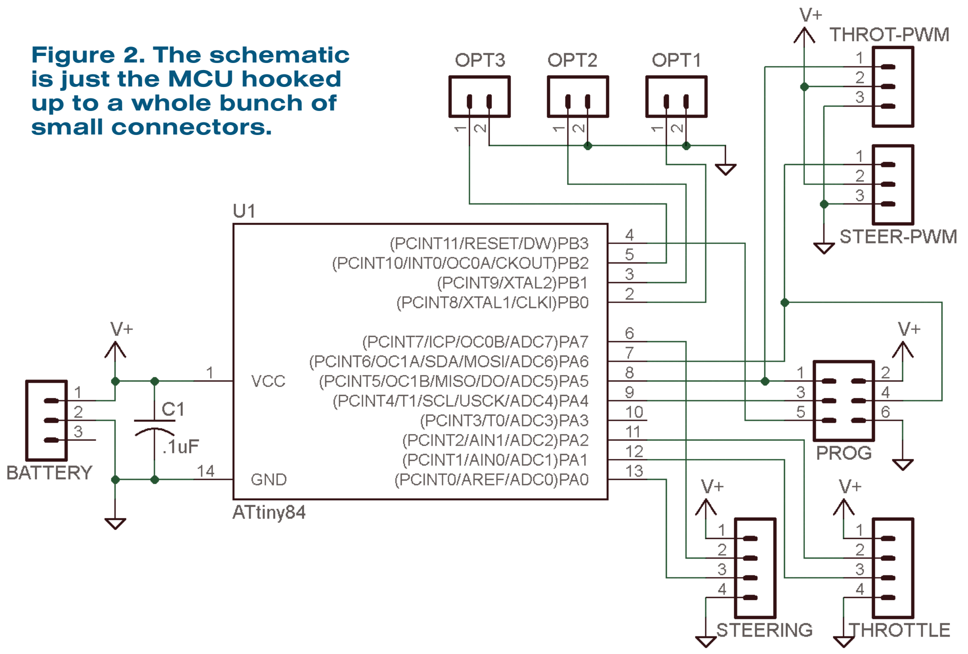
- In a 14-pin DIP, it's a nice size with enough pins to do the job.
- It has hardware PWM — perfect for generating the servo pulse stream.
- It has plenty of analog inputs.

No other components are needed to create this interface, except for a few filter capacitors and some connectors. The simple schematic is shown in **Figure 2**.

I gathered the MCU and the connectors together onto a spare perfboard as shown in **Figure 3**. **Figure 4** shows the hand-wiring I did with wire-wrap wire to hook the connectors up to their appropriate pins on the MCU.

The only part I bought for this project was a battery holder. It retains three AAA batteries to run the MCU at a nominal 4.5V, although it should continue to work down to 2.7V. Conveniently, the holder has its own power switch.

Figure 2. The schematic is just the MCU hooked up to a whole bunch of small connectors.



Going "Raw"

At the time I wrote this, *Nuts & Volts* magazine (**www.nutsvolts.com**) had just started a series called *Beyond the Arduino* by Andrew Retallack. It's a tutorial on working with a "raw" AVR MCU — with no help from the Arduino "ecosystem." I always work directly with the MCU, and this control interface for the go-cart is no different. If the following description of how I did it seems to skip any steps, *Beyond the Arduino* might fill in some of the blanks.

The first thing needed to go raw is the software development environment. Atmel Studio is a complete IDE (Integrated Development Environment) for the AVR, and can be downloaded for free from the Atmel website. It includes a smart editor, C and C++ compilers that generate very efficient code, a sophisticated debugger, and device programming software.

The other thing needed to go direct to the MCU is not free — the device programmer. I use the Atmel AVR Dragon available from Mouser Electronics for $54. It is much more than just a programmer, allowing you to use the debugger in Atmel Studio to set breakpoints, single step through the program, examine and change variable values, etc. — all while running in-circuit with live I/O.

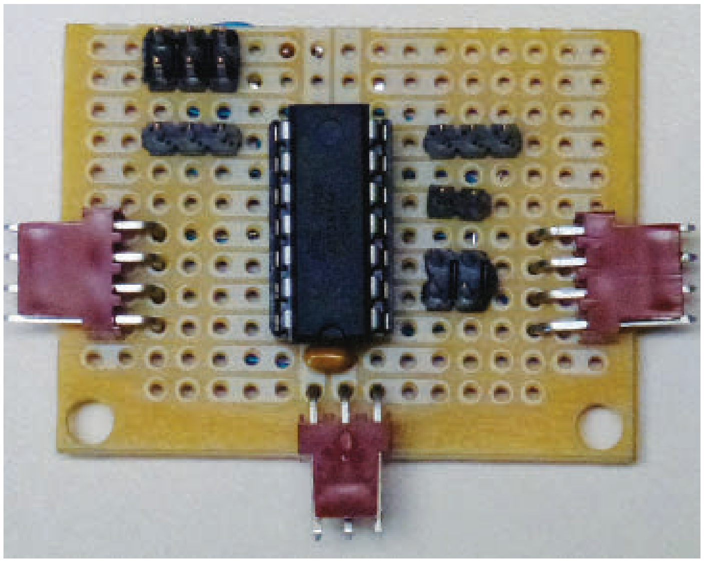


Figure 3. Top view of wired controller.

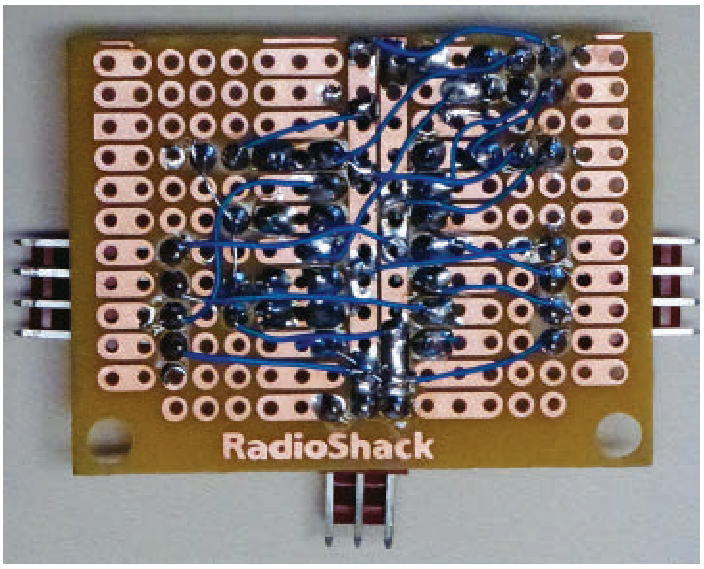


Figure 4. Bottom view of wired controller. I guess I won't be getting any more of these perfboards at RadioShack.

I could not have gotten this project working without this debugging ability. I was in the garage with the go-cart drive wheels held off the ground with blocks, my laptop running Atmel Studio connected to the AVR Dragon which, in turn, was plugged into the control interface.

By having this setup, I'm free to choose any MCU from the huge lineup of AVRs that Atmel has to offer. These range from eight to 40 pins in a DIP, and even more (and less) pins in SMD. The AVR I used in this project only cost about $1.50, so after multiple projects it can even save money over the Arduino route.

Getting Started, a.k.a., Initialization

To get started on the firmware, I ran the New Project wizard in Atmel Studio. This set things up for the MCU I had selected and created the first source code file. As with any C/C++ program, the program starts with a function called *main()*, and there it was — staring at me, empty. I always get to this point and say to myself, "Oh, crap!"

Look at the datasheet of any AVR and you'll see a long list of hardware features that give it tremendous flexibility. When it comes time to set up these features for a specific application, however, there can be quite a learning curve. I used to pour over the datasheet, and often I could find application notes from Atmel for specific hardware functions. Now, I have a tool that makes initialization much easier.

Called AVR IO Designer, it is a free tool written by my good friend, Scott Ferguson. I met Ferg when he headed the team that made the first version of Visual Basic, and I was assigned a small role in the project. (After VB1 was finished, he moved on to other things at Microsoft, while I continued to work on VB through VB6.) Later, we worked together on combat robots (he was on the Hexadecimator team), and as we experimented with using an AVR to control robots, he came up with AVR IO Designer to make it easier.

The tool comes with a 23 page manual, but I'll describe some of the highlights of how I used it for this project. **Figure 5** shows a screenshot of the starting page of Designer where basic parameters are set. The critical items are the choice of AVR, the clock frequency it will run at, and the file names used for the generated code. For file names, I have personally established a convention where I use the project name with "Init" appended; thus, the *AnalogPwm* project has initialization files named *AnalogPwmInit*.

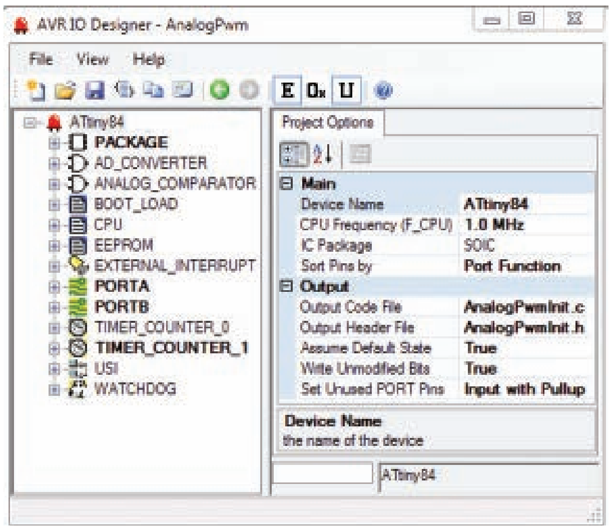
The main initialization task needed for the control interface is to set up the PWM system for the servo pulse streams: two channels with (initially) a neutral pulse width of 1.5 ms at a frequency around 60 Hz. The ATtiny84 has one 16-bit timer than can do the job, and its setup with Designer is shown in the screenshot in **Figure 6**. Note that every value in boldface means it is different from its default value, but that doesn't mean I had to figure out how to set it.

When I chose a counter frequency of 60 Hz, for example, Designer automatically selected the closest value it could achieve, and calculated the counter period field. It also automatically set the value of the input capture register, which is what actually determines the frequency. (I collapsed the Input Capture section for the screenshot, so the value doesn't show up in the figure.)

Likewise, for each of the output registers, I set the time to 1.5 ms and Designer calculated the register value needed to achieve this at the given clock speed. I also gave each output register a name which I can then use in my program when I assign new values to change the pulse width.

I used Designer to set up the I/O port pins, as well. This mainly consists of giving a pin a name, setting the direction (input or output), and optionally adding an internal pull-up for inputs. Again, the name I choose can be used in the program to reference that specific I/O pin.

Designer is not entirely complete. I also needed to initialize the analog-to-digital converter (ADC), but it doesn't have a nice GUI interface with dropdown lists to set the options of the ADC. You can view the individual registers as

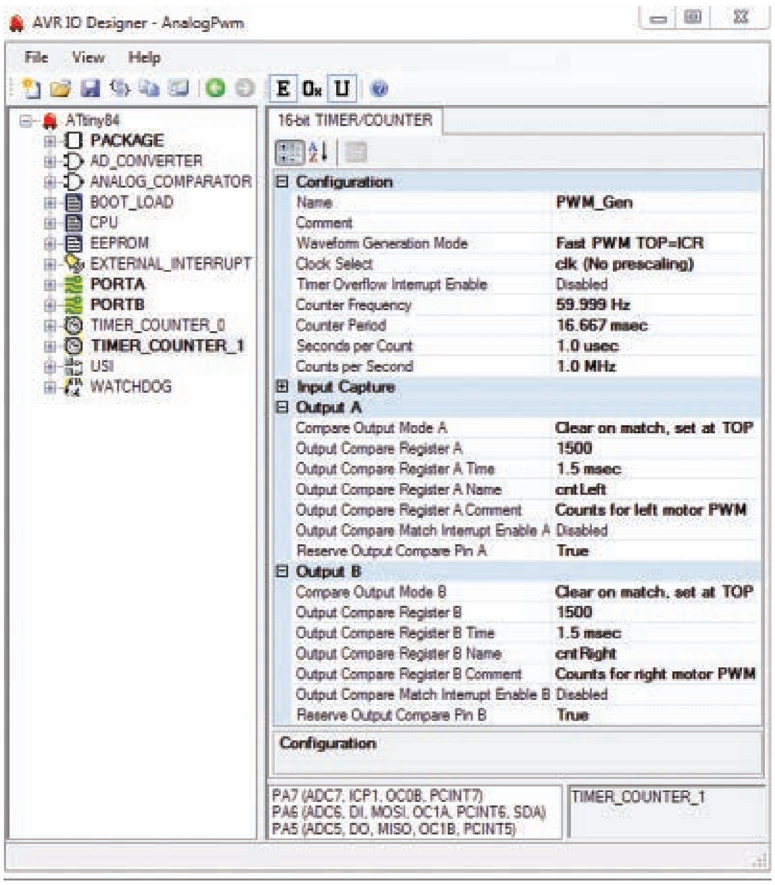


Main	
Device Name	ATtiny84
CPU Frequency (F_CPU)	1.0 MHz
IC Package	SOIC
Sort Pins by	Port Function
Output	
Output Code File	AnalogPwmInit.c
Output Header File	AnalogPwmInit.h
Assume Default State	True
Write Unmodified Bits	True
Set Unused PORT Pins	Input with Pullup

Device Name
the name of the device

ATtiny84

Figure 5. Screenshot showing initial setup for the controller in AVR IO Designer.



Configuration	
Name	PWM_Gen
Comment	
Waveform Generation Mode	Fast PWM TOP=ICR
Clock Select	clk (No prescaling)
Timer Overflow Interrupt Enable	Disabled
Counter Frequency	59.999 Hz
Counter Period	16.667 msec
Seconds per Count	1.0 usec
Counts per Second	1.0 MHz
Input Capture	
Output A	
Compare Output Mode A	Clear on match, set at TOP
Output Compare Register A	1500
Output Compare Register A Time	1.5 msec
Output Compare Register A Name	cntLeft
Output Compare Register A Comment	Counts for left motor PWM
Output Compare Match Interrupt Enable A	Disabled
Reserve Output Compare Pin A	True
Output B	
Compare Output Mode B	Clear on match, set at TOP
Output Compare Register B	1500
Output Compare Register B Time	1.5 msec
Output Compare Register B Name	cntRight
Output Compare Register B Comment	Counts for right motor PWM
Output Compare Match Interrupt Enable B	Disabled
Reserve Output Compare Pin B	True

Configuration

PA7 (ADC7, ICP1, OC0B, PCINT7)
PA6 (ADC6, DI, MOSI, OC1A, PCINT6, SDA)
PA5 (ADC5, DO, MISO, OC1B, PCINT5)

TIMER_COUNTER_1

values or on a bit-by-bit basis, and for each bit the pane in the lower right has a pretty extensive explanation of what it does. I could have clicked on the bits I wanted to set, but (for no particular reason) I put the values in the code myself. Once I had entered everything I wanted into Designer, I clicked on the Emit Code button on the tool bar and it generated *AnalogPwmInit.h* and *AnalogPwmInit.c*. The header (ih) file is all about defining names to use in the program, while the code (ic) file has the initialization code. I edited the code file Atmel Studio had created for me by adding this line near the top:

```
#include "AnalogPwmInit.h"
```

I then added a similar *include* for the code file **inside** the function *main()*, so it looked like this:

```
int main(void)
{
// Include initialization code generated by AVR
// I/O Designer
#include "AnalogPwmInit.c"

     while(1)
     {
          //TODO:: Please write your application
          //code
     }
}
```

For all the help Designer gave me in getting the initialization right, there were only seven lines of actual code — each an assignment of a constant to an MCU register. There are also lots of comments in the code file (available at the article link), listing all the inputs I entered into Designer.

Calibration

It was clear from operating the steering wheel and pedals that they did not move over the full 300° range of the potentiometers. I had to expect random start and stop points, so I needed to build a calibration procedure into the firmware. The calibration procedure kicks in using some of the jumpers I put in the design. I chose this sequence:

1. Apply jumper 1 with everything at the neutral position. That is, the steering wheel is centered and both pedals are released. At the moment the jumper makes contact, the firmware takes an analog measurement for each input which establishes the neutral point. (No doubt this happens many times as the contacts bounce.)
2. Apply jumper 2 with both pedals pressed all the way and the steering wheel hard left. (This ended up taking two people, because I needed both hands to get the jumper installed.) Another set of analog measurements is taken.
3. Remove jumper 1 with the steering wheel hard right. The firmware takes another measurement for steering, but the pedals are already done.

For the pedals, using the calibration results is straightforward. When operating, the neutral position value is subtracted from each reading. The range — maximum minus neutral — is used to create a scale factor.

For steering, I wanted to be sure center was neutral. To achieve this, the firmware computes the range from center to left and from center to right. The *smaller* of these is then chosen, ensuring that full range can be reached in either direction. The neutral and range are used in exactly the same way as they are on the pedals.

Once the calibration procedure is complete, the result is stored in EEPROM in the MCU. The EEPROM is treated as a single struct. The entire contents are copied to RAM at

Programming in C++ vs. C

When starting a project in Atmel Studio, you have the choice of using C or C++ as the programming language. Since C++ is a superset of C for all practical purposes, I always choose C++. I see no reason to cripple the set of features available to me.

Sometimes the argument is made that C++ can be inefficient and/or bloat the code — especially on an MCU. It's true that some features can, but there are also features that don't cost anything. I'll throw out some specifics that may sound like gibberish: On an MCU, I don't think I would ever use virtual functions or the new operator for heap allocation; on the other hand, I am a big fan of inheritance (also called derived classes) and templates.

In this particular project, I utilized just one feature not available in C. Typically, you think of a struct as a container whose members are data. In C++, a struct can also have members that are functions. (To which you might respond: Huh? What would that mean?)

A member function makes sense when the purpose of the function is to perform an operation using the members plus any passed-in arguments. In this case, the value of an analog input is passed to the *Scale()* member function. Using the members that contain the neutral value and the scale factor (set up in the calibration step), the function returns the properly scaled (-500 to +500) result for the analog input.

Certainly, this is not difficult to do in C using a global function. The function would be passed a pointer to one of the three instances of struct *PwmScale* that contain the calibration values; something like this:

```
int Scale(PwmScale *pwm, int reading);
...
steering = Scale(&pwmSteering, reading);
```

When the function is declared inside the struct, the syntax of the function call would look like this:

```
steering = pwmSteering.Scale(reading);
```

The code generated by the compiler for each of these is exactly the same! The call to the *Scale()* function is passed a pointer to the struct either way.

Inside the C version of *Scale()*, there might be a line something like this:

```
reading -= pwm->neutral;
```

In the C++ version of the function, data members of the struct can be directly referenced. The equivalent code would go like this:

```
reading -= neutral;
```

Once again, the compiler generates exactly the same code for both. The explicit passing of the pointer and dereferencing it in C are all done implicitly in C++.

So, what's the benefit? Primarily as an organizational tool. The idea is called encapsulation, where the code and the data it manipulates live in the same container. It also gives member functions the same freedom from name conflicts as data members — you can have a *Scale()* function in another struct that does something different. The code is streamlined by removing the pointer dereference, which can make it both easier to read and to write.

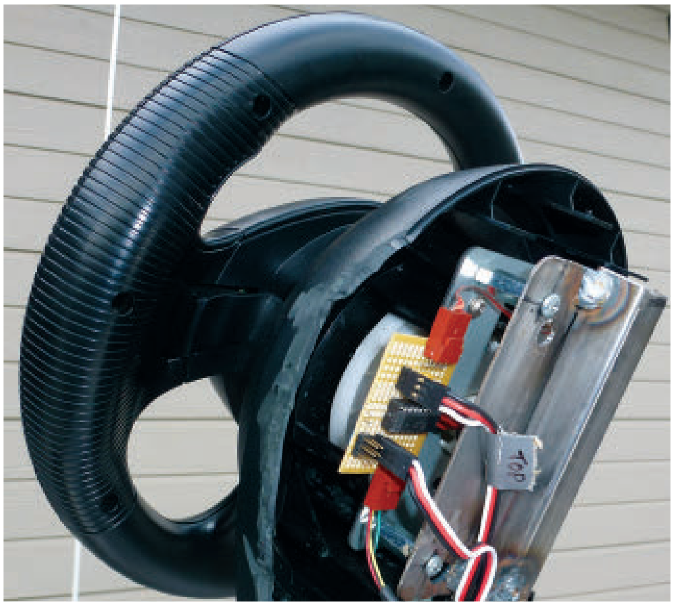


Figure 7. Controller mounted to the back of the steering wheel.

startup, and the entire block is written to EEPROM when calibration is finished.

The code for calibration is about twice the size as the code for operation! It was also responsible for most of the bugs — even some problems that seemed like operation issues sometimes turned out to be a calibration problem.

The Math of Scaling

During operation, the *main* loop will continuously read the analog inputs, then translate and scale them to values appropriate for the servo pulse stream. "Translate" just means shifting the zero point by subtracting the neutral value. Scaling adjusts the range so it goes from zero to 500 (a value I gave the name *PWM_PULSE_RANGE*), representing the ± 500 µS off nominal servo pulse width.

The pedals are assigned the functions *forward throttle* and *reverse throttle* — there are no brakes! Each is scaled to the 0-500 range and are never negative. The combined throttle setting just comes from subtracting reverse from forward, giving a -500 to +500 range. You'd expect only one pedal to be depressed at a time, but if not, the result still makes sense.

The steering wheel is scaled directly to the -500 to +500 range. The negative values come naturally once the neutral (center) value is subtracted from the reading.

Since the ADC output is an integer in the range of 0-1023, the scale factor for the pedals is almost certainly less than one (depending on how much of the analog range is used), while for steering it could be greater or less than one. To make scaling fast and efficient in the never-ending *main* loop, I wanted to avoid floating-point calculations.

The alternative is to use fixed-point binary numbers. In my case, I have a 16-bit scale factor of which 10 bits are to the right of the binary point, and six bits to the left. Or, another way to look at it is that the scale factor is a count of 1/1024 units (1024 = 2^10). If I were computing a floating-point scale factor, it would go like this:

```
scale = 500.0 / range;
```

To compute the fixed-point scale factor, it looks something like this:

```
scale = 500 * 1024L / range;
```

The "L" on the constant ensures it is calculated with 32-bit (not 16-bit) operations. The integer division above truncates the result, but I'd like it to be rounded. The traditional rounding method of adding a half can't be applied to integer arithmetic. Instead, we add a half by adding half the divisor before performing the division:

```
scale = (500 * 1024L + range / 2) / range;
```

This scale factor is calculated once for each input during calibration. The *main* loop then uses it for each reading, something like this:

```
value = (reading - neutral) * (long)scale /
1024;
```

The multiply operation must be done with 32-bit precision, so that's achieved by the cast to type long. The compiler is smart enough to recognize division by 1024 as 2^10 and actually performs a 10-bit right shift.

A few more things need to happen with this result, like making sure it's boxed into the -500 to +500 range we need. The project source code shows these steps.

Mix It Up

Given steering and throttle inputs, the Vantec motor controller produces left and right tank drive outputs — a process called *mixing*. The Vantec mixing algorithm is very simple:

1. As throttle increases, the drive to both wheels increases.
2. As steering increasingly deviates from neutral, the drive to one wheel increases while the other decreases. "Decrease" can even mean reverse.

This always made perfect sense to me and worked fine in the BattleBots arena. Even with zero throttle, the steering control would cause the wheels to be driven — one forward and one backward. Every victory in the BattleBox was celebrated by firing the flipper while spinning in place.

It's Crazy, Man

Now, imagine you're in your car and you've just started the engine. Without stepping on the accelerator, you give the wheel a turn — and the car starts spinning in place! This is not the behavior any driver would expect. Check out the YouTube video listed in the Resources — it was the first test drive, using the Vantec's built-in mixing.

From a safety standpoint, this behavior was clearly unacceptable. Just bumping the steering wheel without touching the throttle could cause sudden violent motion and hurt somebody. The mixing algorithm had to change, which meant taking it out of the Vantec and putting it in my firmware. The Vantec has a whole bunch of configuration jumpers, so setting it up to treat the inputs as left drive and right drive was no problem.

I had to come up with my own mixing algorithm, and here's what I settled on: As steering increases, one wheel slows down (but never reverses). This is done in proportion; for example, if the steering wheel was turned 50% to the right, then the right wheel would be given 50% of the drive of the left wheel. (At a 100% turn, the wheel would be stopped.) Since the steering can only reduce drive, when the throttle is at zero, the steering has no effect. No more spinning in place.

Another attribute of this mixing algorithm is that for a given constant steering input, the go-cart will follow the same arc regardless of throttle (if the wheels don't slip). It is also more linear than the original algorithm, which had reduced steering effect at higher speed because it couldn't increase the speed of one wheel any further.

Here is sample code that implements this mixing algorithm. It assumes steering and throttle are already scaled to the -500 to +500 range:

```
adjust = throttle * (long)steering / 500;
throttle += 1500;   // neutral pulse width
// choose which wheel to slow down
if (steering > 0)
{
     right = throttle - adjust;
     left = throttle;
}
else
{
     left = throttle + adjust;
     right = throttle;
}
// Set the PWM registers
OCR1A = left;
OCR1B = right;
```

Now, That's the Ticket!

Once I burned this algorithm into the AVR's program Flash, all the drivers agreed: The handling of the electric go-cart was now safe and predictable. My work was done.

You can take a look at the source code if you're interested in more of the specifics. If you do, I suggest you start with the ReadMe.txt file that gives an overview. **SV**

Tim Paterson is the original author of DOS — an operating system used very widely on PCs in the 1980s and '90s. In his retirement, he spends most of his time making electronic gizmos to make life easier, usually built around an AVR (four OSH Park orders so far this year!).