Hello Collin/Syntax Error/6077!

There are a couple main things that we've done to improve our teleop code's responsiveness. The first has to do with button checking and stateless button operations, and the second concerns significant, er, *improvements* to the JoystickDriver file.

First, if you'd like to view our source code, all of it is available on Google Code (project ftc-team-4278). This will link you to the file browser, but if you use Subversion, feel free to checkout a copy and look through it! It might help to see specifically what I'm talking about in the files.

The first improvement to teleop comes from the simple idea that code run fewer times runs faster. Cut down on the number of iterations and checks, and you simultaneously cut down on the number of cycles used for dead processing. Our teleop code is laid out in a pretty simple structure:

```
task main() {
    unlockArmMotors(); clearEncoders();
    waitForStart();

    while(true) {
        getJoystickSettings(joystick);
        checkJoystickButtons();

        setRightMotors(powscl(JOY_Y1)-powscl(JOY_X1)/1.1);
        setLeftMotors(powscl(JOY_Y1)+powscl(JOY_X1)/1.1);
    }
}
```

We do two things: check to see if the buttons have changed, and set the motor powers. The core principle here is actually the same as I imagine most robots' code is: do things with the buttons, then do things with the joysticks (or vice versa). The optimization comes from the `checkJoystickButtons()` function; it only runs code if the buttons have changed. Here's how that function works:

```
#define JOY_BTN joystick.joy1_Buttons //from teleoputils.h

short btn = JOY_BTN;
void checkJoystickButtons() {
    if(btn == JOY_BTN) return;
    for(short i = 11; i >= 0; i--) {
        if((btn>>i) ^ (JOY_BTN>>i)) {
            invokeButton(i, ((btn & (1 << i)) == 0));
            btn ^= 1<<i;
        }
```

```
        }
    }
```

The first line is the most important. `if(btn == JOY_BTN) return`; i.e. if the button state has not changed, do nothing. This is important, as checking the state of each button is (comparatively) expensive. The rest of this is just a pile of binary math, which results in the function `invokeButton()` being called with the arguments `int button, bool pressed` to indicate which button has changed to which state.

If you're really interested, here's what's going on in a bit more detail (I'll assume you're familiar with XOR):

In essence, the `if` statement is using the XOR operator to check if the previous state (`btn`) is different than the current state (`JOY_BTN`). The statement `btn>>i` shifts the button value right by the button number; in essence, it makes the i^{th} button the first digit of the binary `btn`. It does the same thing on `JOY_BTN`. If the first digits differ, the XOR operator will make the first digit of the output 1. The if statement will check this digit for true/false value. And, if this value is 1 (i.e. differing values), then it calls `invokeButton`. The next line simply sets that value to 0 in `btn` so it isn't called again.

The second set of optimizations actually occurred within the JoystickDriver itself. It turns out that (unsurprisingly) the current JoystickDriver is actually relatively inefficient. *(A side note: We've optimized our JoystickDriver.c for use with one controller only - you probably don't want to use our driver if you want two controllers)* Our philosophy with this driver has been simple: **if you keep removing things, and it still works, then good. You're improving it.** So, I'm going to point out a couple things which can be safely trimmed, but I challenge you to see how much you can trim without breaking it. I guarantee it will be at least twice as fast when you're done with it. We got ours down to 126 lines… :]

I guess the main thing here is the `displayDiagnostics` task, which, it turns out, actually uses a significant number of cycles. You know that debug info that shows up on your screen with voltages? Turns out, that actually comes from the `JoystickDriver` - and it isn't actually all that useful. You only really need to see it once, and don't need it running constantly in the background. You can simply delete the entire task. As well as the `disableDiagnosticsDisplay` function. And pretty much everything concerning diagnostic output.

I guess my main advice with this file is: most of it isn't needed. Make sure you understand what you're removing, but it's pretty much safe to remove most of it.

Hopefully this helps! If you have any questions, feel free to contact me personally. I'd be happy to help!

Noah Sutton-Smolin
E: noahsutsmo@gmail.com
S: althorn.estvan
C: 858-342-8926