embeddedcontrollers@weebly.com 20 January 2014
© Gareth Scott

Stepper Motor Control v0.0.2

This small board controls a stepper motor's position, velocity and acceleration. But it also does something more. It runs a program that can make decisions based on your specific needs to make the motor respond intelligently. For example, suppose you want the motor to:

1. Move to a position 314 degrees from the current position.
2. Wait 27.1 seconds
3. Return to its original position

Here's the program to do that:

```
main () {
     returnVal = 0;
     startPosition = 0;
     // Store current position in the variable startPosition
     getPosition(&startPosition);
     // Move 314 degrees. Note 314 * 10000 = 3140000
     moveDegreeRelativeX10k(&returnVal, 3140000);
     // Sleep for 27100 milliseconds (27.1 sec)
     sleep(&returnVal, 27100);
     // Move back to original position stored in startPosition
     moveStepAbsolute(&returnVal, startPosition);
}
```

If you don't understand the above, we'll explain, don't panic. Try reading the above program knowing that anything after the double slashes '//' is a comment to us, not the controller. The controller disregards comments. For a detailed explanation, see the Examples section and if you need a gentle introduction to writing C programs there are many examples on the internet. But don't get distracted. You don't need to be an expert C programmer to get things working. Just remember that the compiler will translate those English-like statements such as *moveAbsolute* into code the controller can execute. The controller language provides an abstract layer on top of the native controller instruction set. All computer languages attempt to do this. This one is just directed at motor control.

Why

Turning a motor is hard. A lot harder than it used to be. There was a time when you could just connect a D-cell battery to the leads of a motor and get it spinning. That still works, but if you want precise control of your motor there's going to be a computer somewhere in the system. There are a lot of motor controllers out there, but we found there aren't a lot of controllers that allow you to concentrate on just telling the motor how to move. We hope that our controller allows you to do this and at a reasonable cost of both your time and money.

If you did have lots of spare time and expertise you could build your own motor controller and program it using a development environment like IAR or a GNU tool chain. That's actually what we did, but it does require a commitment to the problem at hand. And there are little issues like winding up in the hard fault handler immediately after the program starts and trying to figure out what happened. One time that happened to us because we switched microcontrollers and the new one had a different memory layout from the old one. The solution was modifying the linker configuration file. Not a big problem, but probably not what most people want to do.

On the other hand, the abstractions provided by this motor controller free you up from having to worry about stacks, heaps and memory layout and let you concentrate on moving the motor.

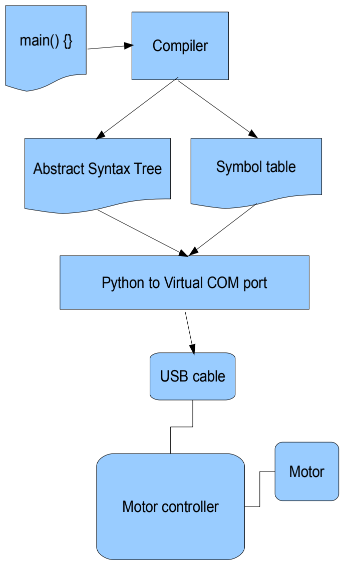
Having said that, all the code and board layout is open-source, so if you do want to modify it for your particular circumstances your welcome to do that. All we ask is that you keep the spirit of the GNU Copyleft agreement intact and give credit where it's due.

Overview

The development environment is currently Linux or Windows and require the following installations:

1. Python 2.6
2. PySerial (see: http://pyserial.sourceforge.net/pyserial_api.html#module-serial) Used to communicate over the virtual COM port with the controller.

Workflow:



Code downloaded to the processor is stored in its internal flash memory. When power is turned on, the program is restored from flash and execution will begin. A common idiom is to look for a key press before starting the main program. See example.

Language

The motor control language used here is very similar to C written by Kernighan and Ritchie. But it is a little different and, we hope, slightly easier to use. If you're familiar with C, the major differences are:

1) No include files like #include <stdio.h>. Everything in one source file.
2) Every variable is an integer and therefore no need to declare its type.
3) The return value is done through the first parameter and specified by the & symbol.
4) No library to link in. The source file is compiled to an intermediate form, downloaded to the controller and then run by a small interpreter.
5) No floating point values. Everything is an integer and since we're using an ARM chip those values can range from -2147483648 to 2147483647. If a stepper motor is moving at 1000 steps/sec, it will take about 24 days of continuous movement to overflow this number.
6) No post or pre-increment operator like: *++x;*

Intrinsic or built-in functions

Movement commands in Gray
Acceleration commands in Turquoise
Time commands in Yellow
Debug commands in Blue
Input/Output command in Magenta
Maintenance commands in Red – Caution

moveStepAbsolute(&returnVal, position);

Moves the motor to a position relative to its 0 position. The parameter *position* is in motor steps.

moveStepRelative(&returnVal, position);

Moves the motor to a position relative to its current position.

moveDegreeAbsoluteX10k(&returnVal, degreeX10k);

Moves the motor to a position relative to its 0 position. The parameter *degreeX10k* is in degrees multiplied by 10000.

moveDegreeRelativeX10k(&returnVal, degreeX10k);

Moves the motor to a position relative to its current position.

setVelocityX10k(RPMx10k);

Rather than move the motor a certain number of steps or degrees, this function will accelerate the motor to the specified velocity and hold it there indefinitely. Another *setVelocityX10k()* command can be issued while the motor is servicing an initial *setVelocityX10k()* call. Also, calling with a parameter of 0 will bring the motor to a controlled stop.

getPosition(&returnVal);

Sleep the specified number of milliseconds.

setPosition(setVal);

Sleep the specified number of milliseconds.

sleep(milliseconds);

Sleep the specified number of milliseconds.

sleepUntil(milliseconds);

This is similar the sleep() function above, but has the added ability to take into account the time the motor has spent moving. In other words, timing can be set to start from when the motor starts moving. Example to follow ****

getMillisecondCount(&returnVal);

Return the number of elapsed milliseconds since the timer was reset.

setMillisecondCount(setVal);

Set the millisecond timer.

setRPMx10k(minRPMx10k, maxRPMx10k);

Set the minimum and maximum RPM value (x10k). For example, if the mimimum speed is 0 RPM and the maximum speed is 200 revolutions-per-minute, multiple that by 10000 and make the call *RPMx10k(0, 2000000);*
Multiplying by 10000 allows you floating-point precision using an integer. If you wanted an RPM of 2.7128, the value sent to this function would be 27128

setAccelMicroSec(setVal);

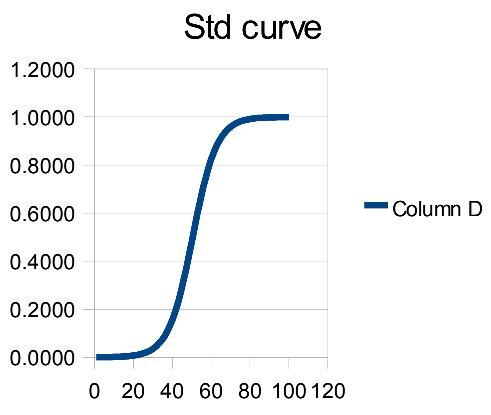
The number of microseconds to in the accelerate or decelerate phase of the motor movement. Maximum value is 4000000 us (4 sec) and the minimum value is 1000 us.

setAccelSharpness(minSharpness, maxSharpness);

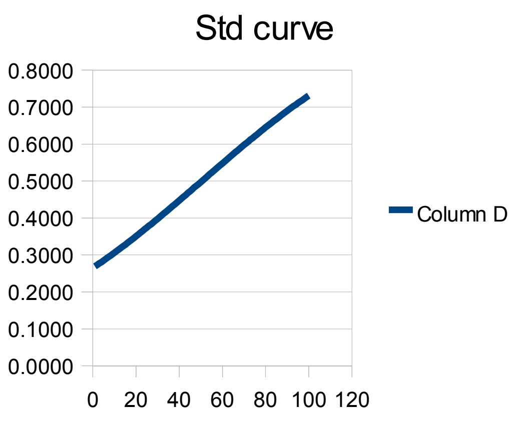
Specify the sharpness of the acceleration curve. These qualitative values can take on a value from 1 to 8 in integral steps. A value of 8 yields a smooth S-curve, while a value of 1 results in a linear acceleration curve. The curve itself is a piecewise linear approximation to the function:

$$v(t) = 1/1 + e^{-at}$$

Where v = velocity, v(t) = velocity at time t and a = sharpness



sharpness = 8



sharpness = 1

waitForIdle(&returnVal, milliSec);

Wait until the motor has stopped running. The *milliSec* parameter tells this function how long to wait before returning. A value of 0 causes it to return immediately. A *returnVal = 1* means the motor is idle, *returnVal = 0* means the motor is still running.

getInput(&returnVal);

There are 5 digital input lines associated with the controller *in0*, *in1*, *in2*, *in3* and *in4*. The return value represents the voltage on these lines when the decimal value is converted to binary. Example: if all inputs are at 0 volts with respect to the system ground, *returnVal* = 0. If *in0* = 1 and *in1* = 1, then *returnVal* = 3 which in binary is 0011. The right most 1 value represents *in0* and the 1 to the left of that is *in1*.

setOutput(setVal);

Like *getInput()* above, this function allows you to set the values of the 5 output lines, *out0, out1, out2, out3* and *out4*.
Note that the *setVal* parameter does not have an & because it does not return a value.
Example: if you want to set *out1 = 1*, *out2 = 0*, *out3 = 0* and *out4 = 1*, then the binary digits would look like this: 10010. We didn't specify *out0* so we're just setting it to 0 (the right most bit). The number 10010 in binary is 18 decimal, the value to put in *setVal*.
One note is that *out0* is controlled internally by a heart-beat task that blinks on and off once per second. You can turn this output off, but the internal code will turn it back on again.
The output lines are connected to LED's. If you want to see the value of an input line use something like this:

```
inputValue = 0;
getInput(&inputValue);
setOutput(inputValue);
```

getADC(&returnVal, adcIndex);

The 5 analog-to-digital converters are names *adc0* through *adc4*. Select which one you want to use in *adcIndex*. The *returnVal* will contain the 12-bit (0-4095) value of the voltage on that pin.

getTemp(&returnVal);

Returns internal temperature of processor in degrees Celsius. This can be used to indirectly monitor the temperature of your process knowing that the processor temperature is usually proportional to points nearby.

printNumber(setVal);

Used for debugging. The number in *setVal* will be sent out the USB port which is configured as a CDC (Communications Device Class) virtual COM port. The number is enclosed in angle-brackets like this: <12>

reset();

Equivalent to pressing the reset button on the board.

`warningFlash();`

Save current program and symbol table in flash memory. When the controller starts up again it will start running this saved program.

`warningBootloader();`

The prefix is 'warning' because you have to be set up with the tools to download updated firmware to the controller. In other words, once you issue this command, the device will be waiting for new firmware and won't be running your program. You can
Once this command is issued, you'll need two pieces of software, 1) dfuprog.exe which is used to download the firmware through the USB port and 2) the firmware, which you can download from our web site at embeddedcontrollers.weebly.com
You'll also need Device Firmware Update drivers for your Windows computer. These are provided by Texas Instruments and can be found in the *windows_drivers* directory.
If you do get stuck with a dead board send it back to us and we'll reflash it for you. Please provide a box with postage to send it back.

Hardware and specifications

The processor is a 32-bit ARM chip from Texas Instruments. Specifically, this M4 device has a built-in floating point macro cell. The TM4C1233D5PM runs at 50MHz and uses FreeRTOS v7.5.2 to manage the interpreter.

The stepper motor is driven by an Allegro 4984 processor and drives a 2-phase bipolar stepper motor using step and direction inputs.

The interpreter executes about 40000 statements per sec.

Examples

1. Synchronizing control of 2 motors

2) Simple sweep-second hand. Motor turns 360/60 = 6 degrees per second.

```
main () {
     while (1) {
          // The motor will take a certain number of milliseconds
          //  to move so this second-hand won't be very accurate.
          moveDegreeRelativeX10k(60000);  // 6 degrees x 10000
          sleep(1000);
     }
}
```

3) More accurate sweep-second hand

```
main () {
     returnVal = 0;
     setPosition(0);
     while (1) {
          // loop forever
          sixtySeconds();
          // 6. Move to original position.
          moveStepAbsolute(&returnVal, 0);
     }
}

sixtySeconds () {
     secondCount = 0;
     while (secondCount < 59) {
          // 2. Set timer
          setMillisecondCount(0);
          // 3. Move 6 degrees
          moveDegreeRelativeX10k(60000);  // 6 degrees x 10000
          // 4. Wait for 1000 ms (1 sec) to elapse
          ms = 0;    // short for milleseconds
          while (ms < 1000) {
               getMillisecondCount(&ms);
          }
          // 5. Increment number of seconds
          secondCount = secondCount + 1;
     }
}
```

3. Home

4. Dynamic home

5. Move motor 3.1 inches

6. Wait for switch press to continue processing

7. Calculate time between 2 points in the program

Contact

If you think you've found a bug or are having a problem you can't get around please contact us. We're at embeddedcontrollers@yahoo.com.
The thing we ask is that you've spent sufficient time trying to figure out the problem (at least 20 minutes), and that you include enough detail for us to understand and, hopefully duplicate the problem.

You can find the source for this controller at: https://github.com/GarethS/

Please contact us with suggestions, frustrations and constructive criticisms. If you have a special requirement and don't have the resources to implement it, we do consulting at reasonable rates.

We've tried hard to provide something that we think is unique in the motor control market place, but this is our first pass at getting something out. Please be patient with us. We'll do our best to help you and your motor.

Problems, bugs, requests

1. Homing not accurate. This is a known bug to be addressed later.

2. Allow different threads of execution.

3. Allow interrupt routine programming