EE Times Connecting the Global Electronics Community

designlines AUTOMOTIVE

Design How-To

Linear motor control without the math

Pramod Ranade, SPJ Embedded Technologies

4/1/2009 04:00 AM EDT

3 comments

To run a stepper motor smoothly, it's necessary to apply linear acceleration. However, conventional algorithms to generate linear acceleration require complicated mathematics, which makes it difficult to implement the algorithms in a field-programmable gate array (FPGA). My colleagues and I, instead,

have used a new algorithm that requires only addition and subtraction, yet produces smooth linear acceleration. This algorithm was first implemented as a C program and tested. Then it was implemented in an FPGA for use in a real application.

The algorithm had its genesis in a project we were doing that required a dedicated controller for a six-axis machine. Each axis was driven by a stepper motor, with speeds up to 100,000 steps per second (with microstepping). For smooth and noiseless movements, stepper motors need to be accelerated (and decelerated) linearly. In other words, the motor speed should increase or decrease linearly, until the desired speed is reached.

Apart from the six axes, the machine also needed many more I/O. So in any case, we needed a good microcontroller. We then tried to look for reference designs for generating linear speed profiles for stepper motors.

We came across a very good article-- "Generate stepper-motor speed profiles in real time" by David Austin (*Embedded Systems Programming,* January 2005, www.embedded.com/56800129). It suggested a new algorithm that can run on a simple (even 8 bit) microcontroller to generate linear speed profile for stepper motors. However, it probably could not be used to run six motors at once.

Moreover it would be difficult to use it even for one motor, when required speeds were as high as 100KHz--in other words, time gap between two consecutive steps could be as little as 10 microseconds. We concluded that the microcontroller alone would not be able to meet this requirement. So we decided to divide the tasks into two parts:

• A microcontroller to implement machine operation logic and interlocking. We chose Atmel AT91SAM9260.

• An FPGA to drive the six stepper motors. We chose Xilinx Spartan 3 (XC3S400).

Our hardware looked like **Figure 1**. The microcontroller firmware implements entire machine operation logic. Whenever necessary, the microcontroller tells the FPGA to move the motor(s). It basically tells the FPGA to move a motor *N* steps--where *N* maybe positive or negative, to indicate clockwise or counter-clockwise directions respectively. Apart from this *N*, the microcontroller also passes on some other parameters to the FPGA. Before getting into those details, let us first get familiar with the characteristics of stepper motors.

Our hardware design for running six stepper motors.

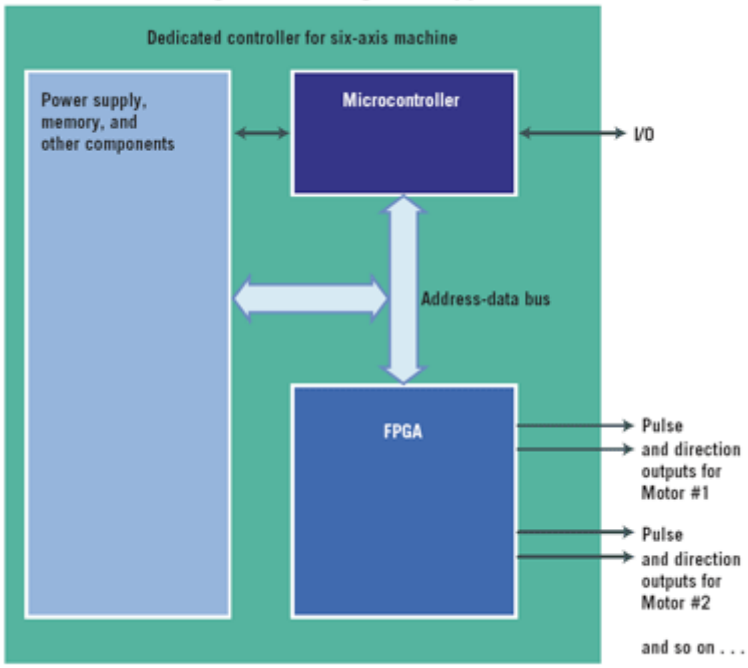

Figure 1

View the full-size image

Like all other motors, stepper motors cannot withstand and respond to infinite acceleration. For example, a motor specification might say "maximum speed = 5,000 steps/sec." However, if you suddenly apply 5-KHz pulse train to the motor, it may not immediately start running at that speed--perhaps, it may not run at all. Usually, the manufacturer also specifies a "maximum starting speed"--say 1,000 steps/sec. It means, even if you suddenly apply a 1-KHz pulse train, the motor will (almost) instantaneously start running at that speed. So, how do you run the motor at full speed of 5,000 steps/sec? Obviously, you should first apply 1-KHz pulse train and then slowly increase the frequency up to 5 KHz. But exactly how "slowly"? Again, that is usually specified by manufacturer as "maximum acceleration--for example, 25,000 steps/sec^2. It means, if you try to accelerate faster than that, the motor may miss a few steps or make jerky and noisy movements.

Thus our problem can be specified as: we need to move a motor *N* steps, within the specified time; while doing so, we must not attempt jump-start the motor at speeds higher than "maximum starting speed"; must not apply acceleration higher than "maximum acceleration"; and must never exceed "maximum speed." So these are input parameters for FPGA:

• **Desired position:** number of steps to move (*N*).

• **Available time:** (in seconds) We must reach the desired position within this much time.

• **Start speed:** it is the speed (in steps/sec) at which the stepper motor can "start" running. In other words, the stepper motor can go from 0 to this speed in no time. This parameter is governed by mechanical design of the stepper motor. Usually this will be specified by the motor manufacturer.

• **Speed limit:** it is the maximum speed (steps/sec), the motor can run at safely. This will be specified by motor manufacturer too.

• **Acceleration limit:** it is the maximum acceleration (steps/sec^2), the motor can accept. If we apply acceleration higher than this limit, the motor cannot run smoothly and it may even slip. This parameter will be also specified by motor manufacturer.

Based on this, we're required to generate either triangular or trapezoidal speed profile--as shown in **Figures 2** and **3**.

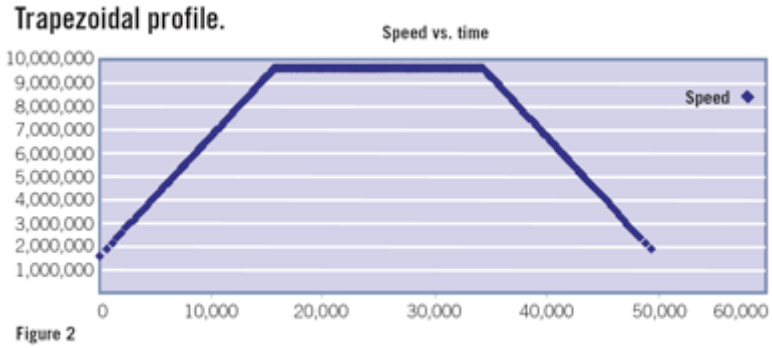

Figure 2

View the full-size image

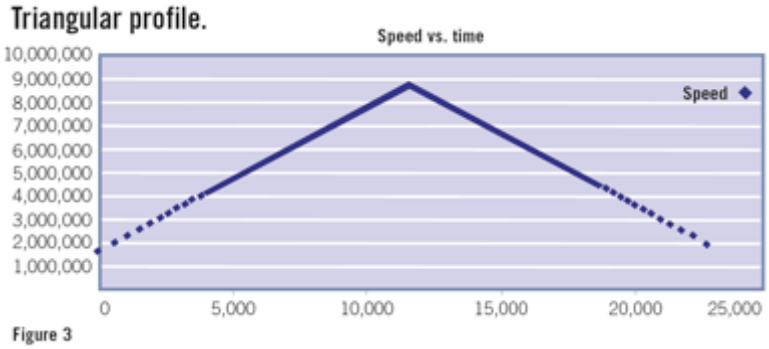

Figure 3

View the full-size image

In either case, note that constant (linear) acceleration is used, so the speed vs. time chart only includes straight line segments. Since the acceleration is not changing and since it's always less than the limit, the motor moves smoothly. The problem is how to achieve this linear acceleration.

Part of the problem is solved by microprocessor: first, we need to decide which of the two speed profiles can be used. Whenever possible, we would like to

generate triangular profile. To check whether this is possible:

1. We can begin at **start_speed**.

2. We can accelerate up to **speed_limit**.

3. Then we decelerate back to **start_speed** and stop.

So if we use triangular speed profile, at best we can achieve:

average_speed = (speed_limit + start_speed) / 2 ;

So the distance covered in **available_ time** would be:

distance_covered = available_time *average_speed ;

4. If this **distance_covered** is >= **desired_position**, we can use triangular profile.

5. If not, we may try to use trapezoidal profile.

Due to space limitations, this article will describe the case of triangular profile only. This is how the microprocessor proceeds with needed computation:

required_average_speed = desired_position / available_time ;

From this value, we can calculate the top speed that we need to achieve:

required_max_speed = (required_average_speed * 2) -start_speed ;

Of course, if (required_max_ speed > speed_limit), it means we cannot achieve the desired results. Let us proceed further assuming that required_max_speed is **<=** speed_ limit.

Now we know starting speed and top speed. Also, from the shape of the profile (shown in **Figure 2**) we know that we need to achieve top speed in exactly half of available time. So we can calculate required acceleration:

required_acceleration = (required_max_speed - start_ speed) / (available_time / 2) ;

such as:

required_acceleration = (required_max_speed - start_speed) * 2 / available_time ;

Of course, if (**required_acceleration** is > **acceleration_limit**), it means we cannot achieve the desired results. Let us proceed further assuming that **required_acceleration** is <= **acceleration_limit**.

So now we have this modified set of input parameters:

• **Starting speed** = **start_speed** (steps/sec)

• **Top speed** = **required_max_speed** (steps/sec)

• **Acceleration** = **required_acceleration** (steps/sec^2)

• **Deceleration** = **-required_acceleration** (steps/sec^2) (value same as acceleration but negative)

• **Number of steps to move** = **desired_position**

Calculations up to this point (including the divisions) can easily be done by the microprocessor within a few microseconds. Then these values can be passed on to FPGA when we need to start running the motor. After this point, the FPGA is expected to take over the job of running motor, thereby freeing the microcontroller to do other things.

The role of FPGA in this process begins here, where we must generate a pulse train for the stepper motor. While accelerating, the time period between two consecutive pulses must

reduce; while decelerating, it must increase.

After generating first pulse, we need to calculate time period until next pulse, wait until that calculated period elapses, and then generate the next pulse. This will go on until the desired position is achieved--in other words, the desired number of pulses have been generated.

We require speed to increase (or decrease) linearly, so the time period--which is inversely proportional to speed--will decrease (or increase) *nonlinearly*.

Now the FPGA must continuously calculate the time period between two consecutive pulses, and when the time period elapses, it should generate next pulse.

The very first pulse will be generated immediately when microprocessor tells FPGA to start motion. Then, the second pulse will be generated after a period of T1, the third pulse will be generated after T2, and so on, roughly as shown in **Figure 4**.

The first pulse is generated when microprocessor tells FPGA to start motion (opening T1). The second pulse is generated after a period of T1, the third pulse after T2, and so on.

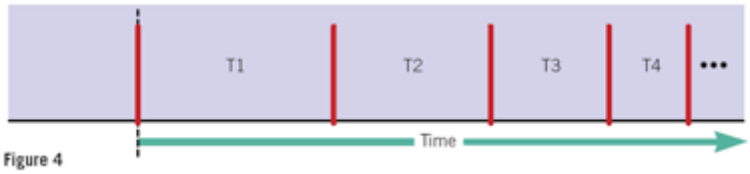

Figure 4

View the full-size image

Thus T1, T2, T3...T*n* represent the time period between *n*th and (*n*+1)th pulse.

We know the starting speed, so we can calculate T1 as:

T1 = 1 / start_speed ;

because at *t* = 0, speed = **start_speed**

Then we also know the value of acceleration. So we can calculate value of speed at *t* = T1:

speed_at_T1 = start_speed + (required_acceleration * T1) ;

Thus now we know speed at *t* = T1, in other words, at the time of generating the second pulse. From this, we can calculate T2 as:

T2 = 1 / speed_at_T1 ;

In other words:

T2 = 1 / (start_speed + (required_acceleration * T1)) ;

By extending the same logic further, we can calculate T2, T3, and so on, or T*n* for any value of *n*. But this calculation is fairly complicated and includes division. Implementing division in an FPGA is not easy and would require too many gates. In brief, this "common sense" algorithm is:

1. Calculate time period to next pulse.

2. Wait until that much time period elapses.

3. Generate next pulse.

4. Go back to Step 1 and repeat until desired number of pulses is over.

New algorithm?
The difficulty now lies in the "time-period calculation"--because there we need to solve nonlinear equations--necessitating the creation of a new algorithm that takes a different approach, as follows.

1. Check the current time.

2. Multiply it by speed, to get "expected current position" (because speed * time = distance).

3. If the "actual current position" is less than expected, then move ahead (until we reach the expected position). In case of stepper motor, we can only move integer number of steps. So we can "move ahead" only when difference between expected and actual position is greater than or equal to one step.

4. Repeat this until the final "desired position" is achieved.

Let us illustrate it. Say speed is 0.2 steps/sec. In other words, one step every five seconds (I have purposely chosen this ridiculously small value of speed, because it makes the illustration easier). Let us start from $t = 0$ and keep calculating the "speed-time product," every one second. After every one second, let us also check whether we need to generate a pulse now. **Table 1** shows the results.

The results of calculating the "speed-time product" every one second and checking whether we need to generate a pulse.

A	B	C=A*B	D	E=C-D	F
Current time (seconds)	Speed (steps / second)	Speed-time product (= distance = "expected position")	Current position	Difference between "expected position" and "current position"	Generate a pulse when difference is >= 1
1	0.2	0.2	0.0	0.2	No pulse
2	0.2	0.4	0.0	0.4	No pulse
3	0.2	0.6	0.0	0.6	No pulse
4	0.2	0.8	0.0	0.8	No pulse
5	0.2	1.0	0.0	1.0	Pulse
6	0.2	1.2	1.0	0.2	No pulse
7	0.2	1.4	1.0	0.4	No pulse
8	0.2	1.6	1.0	0.6	No pulse
9	0.2	1.8	1.0	0.8	No pulse
10	0.2	2.0	1.0	1.0	Pulse
11	0.2	2.2	2.0	0.2	No pulse
12	0.2	2.4	2.0	0.4	No pulse
13	0.2	2.6	2.0	0.6	No pulse
14	0.2	2.8	2.0	0.8	No pulse
15	0.2	3.0	2.0	1.0	Pulse
16	0.2	3.2	3.0	0.2	No pulse
17	0.2	3.4	3.0	0.4	No pulse
18	0.2	3.6	3.0	0.6	No pulse
19	0.2	3.8	3.0	0.8	No pulse
20	0.2	4.0	3.0	1.0	Pulse
21	0.2	4.2	4.0	0.2	No pulse
22	0.2	4.4	4.0	0.4	No pulse
23	0.2	4.6	4.0	0.6	No pulse
24	0.2	4.8	4.0	0.8	No pulse
25	0.2	5.0	4.0	1.0	Pulse
26	0.2	5.2	5.0	0.2	No pulse
27	0.2	5.4	5.0	0.4	No pulse
and so on...					

Table 1

View the full-size image

Clearly, we are generating one pulse every five seconds--so we are correctly running at the desired speed of 0.2steps/sec.

In the above example, we have assumed that speed remains same throughout--at 0.2 steps/second. However, even if speed is changing, the same rule holds good--in other words, generate a pulse when speed-time product exceeds the current position by at least one (step).

We can now write this new algorithm more elaborately, as follows:

1. Initially, **speed_time_product** = 0, **current_position** = 0 ;

2. Repeat the following loop periodically:

   - A. Calculate difference between **speed_time_product** (speed * time) and **current_position**. As long as **current_position** remains same, this difference will keep increasing; because time is increasing.
   - B. If the difference is greater than or equal to one, then:
     - a. Generate a pulse.
     - b. This pulse will move the motor by one step. It means, we "move ahead", so increment **current_position** by one.

3. Repeat the loop until desired number of pulses are generated.

In this algorithm, we need to calculate **speed_time_product**. But we don't even need multiplication to do this. We know initial speed. We also know that at *t* = 0, **speed_time_product** = 0. Every one second, we can calculate new value of **speed_time_product** just by an addition, like this:

speed_time_product = speed_time_product + speed ;

Of course, this is possible, because "second" is always used as unit of measurement of time. In other words, it's possible because:

• We repeat this calculation every *one second*.

• Speed is measured in steps *per second*.

In reality, repeating this calculation every one second would not be useful, because we would be restricted to speeds below 1 Hz. We may have to repeat this loop much more frequently--say once every millisecond (or even faster). Then the formula for **speed_time_product** calculation would change to:

speed_time_product = speed_time_product + (speed /1,000) ;

assuming we repeat the loop every 1ms. The "division by 1,000" can be avoided if we measure speed as steps/ms (instead of steps/sec). Thus, if speed is measured in steps/ms, *and* if calculation is repeated every 1 ms, the formula:

speed_time_product = speed_time_product + speed ;

still holds good. This way, it's still possible to avoid division.

However, measuring speed in steps/ms will give rise to use of fractional numbers. We can avoid it by multiplying all values by 1,000 (or 1,000,000 or some such factor).

In our case, we actually repeat the loop every 1 µs (microsecond)--which is not a big deal for the FPGA since we supplied it a 20-MHz clock. We needed this, because we wanted to achieve speeds of 100 KHz--in other words, one hundred thousand steps per second.

A program in C repeats the loop every 1 µs, so values of speed, time, and so forth, have been scaled appropriately.

Title-7

Multiply not divide
The conventional algorithm calculates "time period" (between consecutive steps) repeatedly, so requires division. The new algorithm calculates "speed-time product" repeatedly, so requires multiplication instead of division. Further, since we already have a loop, this multiplication can be easily implemented as cumulative addition.

This algorithm is implemented in a C program--step.c, which is posted on Embedded.com. I compiled it with Microsoft C, but since it doesn't use any non-ANSI constructs, theoretically it can be compiled with any ANSI C compiler--hopefully including gcc.

VHDL code for FPGA was later written based on this C program.

This program (step.c) produces results in the form of a text file. I copied contents of this text file into a spread-sheet and produced *X-Y* chart from time and speed columns. These same charts are actually reproduced here as **Figures 2** and **3**.

Pramod Ranade *is CTO at SPJ Embedded Technologies Pvt. Ltd. The company offers consulting services for embedded systems design and development. Pramod can be reached at pramod.ranade@spjsystems.com.*

More Related Links

Is Foxconn Ready for Cars?

Taiwan Eyes Automotive Market

Apple's In, But DMV Is Telling Us More

MIPI Goes Beyond Mobile, Camera

Renesas ‘Opens’ Autonomy for Cars