Chapter 15

High Speed Applications and Micro-Controllers

Features

Definitions
Questions Raised
Intelligent Designing versus Intelligent Processors
Real-Time Application

High Speed Applications and Micro-Controllers

Much of my career has involved the design of **high-speed** processing equipment. These designs have ranged from multiple web handlers (registration), to flying cutoff equipment, from *on-the-fly* web applications (*Master/Slave*), to product assembly equipment (indexing machinery), and designs including on-the-fly inspection stations (i.e., Lasers). The terms I encounter most often in the conceptual stage of design are *real-time*, *high-speed*, and *DSP*. Concerns such as ". . . will it be **fast** enough?", always seem to make their way into the conversation.

I have difficulty understanding why certain words and abbreviations impart such importance in the Motion Control industry without designers qualifying their use in given applications. Term usage of XT, AT, 286, 386, 486, Megabits, MegaFLOPS, and DSP among others, have had an enormous effect on the choice of host or control computer and programming language. I will never discount the need for a fast computer to handle extremely high data transfer rates. Neither will I discount the need for sending data to node ports at very high baud rates. What I will discount, however, is the folly of selecting a controlling device simply because it fits into today's world of acronyms!

The objective of this chapter is to demonstrate how systems can be analyzed and applied using simple concepts rather than complex formulas. The methods described can generally lead to less costly controls.

The terms—*real-time*, *high-speed*, and *DSP* will be defined first, in order to establish their meaning in **this** text. Understanding how they fit into the defined problem is a key to maintaining the smoothest possible transition from theory to meeting the *real-time* motion control requirement(s).

Definitions

Real Time – A time interval in which the controlling device must sense, acknowledge, and react to an event (count, trigger, etc.) with a required task (start, stop, latch, clamp, . . .). The relationship exists solely between the real time action of the **device under control**, and the real time reaction capability of the **controlling device**.

High-Speed – An arbitrary assignment denoting something that is occurring *faster* than our human ability to sense it.

DSP – An acronym for *D*igital *S*ignal *P*rocessor generally known for its *High-Speed* vector math capability and its fast multiple instruction cycle.

Although DSPs are exceptional tools when applied to vector processing, slower math processors and micro-controllers can also perform in a *high-speed/real-time* system if you have a good understanding of the required system and control processes.

Questions Raised

Real-Time Questions:

- Is this the time it takes a CPU to perform a program loop or process a line of code?
- Or, is it the time it takes a PLC to process a complete code sequence (scan)?
- Could it simply be the time it takes a CPU or program to capture and respond to an event occurring on a machine?
- Is the event to be captured random or repeatable?
- What happens if more than one event occurs at the same *time?* Is that possible?

In motion control, **real** events control the *real-time* window(s) of operation. Selecting a CPU and a programming language must depend solely on the true requirements of your system.

A trap to avoid is to select a control based on *familiarity* rather than *functionality*. Have you ever found yourself a situation where a customer's system required a *real-time* operating control (functionality as you saw it), but the customer wanted you to use a PLC (programmable logic controller) because they are used throughout the plant (familiarity as they saw it)? Did you argue the point? Did you do the job anyway using the PLC knowing it would not meet specification? Did it work, or fall short of the required operation? Did you price the job right? Did you make any money?

High-Speed Questions:

- What does this term really imply when used in conjunction with motion control systems? At just what speed is a system considered *high-speed*?
- On what criteria is the selection of the control for a high-speed system based?
- Why are some control methods better than others when used in high-speed applications? Will your budget allow you to afford all of the high-speed mechanisms required for the system to operate as specified (i.e., control, mechanics, sensors . . .)?

DSP Questions:

- Just how fast must the system processor be in order to maintain optimum control?
- Is the majority of the processing going to be *on-the-fly* calculations or extracted from tabled data?
- Would multi-processing be a better approach?
- Could the system functionality be described more efficiently?
- How does the update time relate to the overall system time constant (action/reaction) as the motion math update occurs in some finite period of time?
- At what point does the control update time period in relation to the system time constant become critical? Remember that system mechanical and electrical time constants control the systems ability to respond.

Intelligent Designing versus Intelligent Processors

Usually, the majority of all high-speed applications share a dimension that allows them to be controlled by a slower speed processor—the dimension of **repeatability**. Knowing that the process will accurately repeat the operation sequence, gives you a wider range of choices for control selection.

Non-Repeatable Systems

This is the most difficult system design scenario to deal with. It's the one in which nothing ever happens the same way twice. What kind of system are we talking about? The most notable non-repeatable system type would be a robot navigating a corridor with obstacles placed randomly in its path. In this case there are three objectives:

(1) proceed without hitting any of the obstacles,
(2) maintain a reasonable velocity while doing it, and
(3) maintain *continuous path* motion versus *stop-and-go* type motion

The robot example is non-repeatable, but there are several things which you can do to ensure that this control method will work:

- One is to use sensors which can *reach out* further, allowing your system both a longer *look-ahead* time (to avoid the obstacles).
- Another is to make this avoidance motion appear **smooth**.
- A third thing is to multi-process. Using more than one processor, each to control smaller tasks, might allow for simpler programming steps and perhaps simpler debugging. Multi-processing might also allow better use of high powered DSP's in conjunction with slower speed micro-controllers for a more cost effective system solution.

Remember that here, unlike megabyte data handling operations (for instance, those used by AutoCad [1] or any other CAD/CAM software), a mechanical system generally operates slower, requiring only a dozen or so points of information per axis. Designing with this in mind will usually reduce the demand on the system's CPU.

Discussion

DSPs are not by themselves a cure-all for all *real-time* problems. Although technology has reduced the computer footprint while adding increased ability to perform more computations with higher throughput, the real test of a qualified design is simply to do as much work as possible with as little computer power as possible.

---

[1] A trademark of Autodesk

Your ability to avoid a catastrophe is based solely upon the thinking that you put into the project ***before it's built***. In the case of our robot, the ability to produce an action/reaction timing chart may not be practical when given the number of obstacle placement permutations. But you could create a timing chart of the worst-case scenario for the robot to physically move, in order to develop the look-ahead requirement. This way you can determine sensor requirements, system speeds, and other considerations to avoid collisions ***gracefully***.

In our robot example, your understanding of acceleration, velocity, and miscellaneous motion reaction times will definitely lead to better CPU, programming language, and the sensor selection necessary to keep the robot moving along a continuous motion path. In each area (CPU, hardware, and software), sensor and logic requirements for the system will emerge as your design progresses.

Selection of processor, hardware, language, and writing technique is directly related to the system timing requirement—the ***real-time*** requirement. If you put your design together without understanding exactly how each of these system functions interact with each other and the system mechanics, then the system control will be a design for disaster.

Repeatable Systems

Just what is a repeatable system, and how can you ensure that controller and system speed is a good fit? A repeatable system is cyclic. That is, it will accurately reproduce an action at specific time intervals controlled only by the product requirements, and the product requirements will determine the speed of the machine.

The simplest way to figure out whether a controller will *fit* the system requirement is to create a timing chart (Chapter 12). The timing chart shown in Figure 15.1 was used to develop the *real-time* operating requirements for a rotary brush-making machine, the type which is used on floor buffing units. The graph represents a 360-degree head cycle. The head cycle is then converted into a linear drilling motion which is used to drill hole patterns into a round block of wood at a maximum rate of 250 strokes per minute. While this action occurs at one station, a second station stuffs the drilled holes with bristles in a previously made disk.

The bottom-line purpose of the timing chart is to show all system actions versus elapsed time as the machine or system cycle progresses. Allowing for variations in system operation due to motor (motion) instabilities, switch and sensor propagation delays, etcetera, the chart can be adjusted to reflect smaller, or larger action windows to accommodate system component variations.

Once all of the required motion and interlocking scenarios are entered on the chart, the next step would be to fit the system with the proper CPU and language combination. The timing requirements for the CPU and language would be based on their ability to meet the requirements of the indicated event-time periods.

The processor speed, the operating language selection, the method of software writing, and the handling of the interrupt sequences will all be determined by the amount of time the system events have given you to do the required actions, in other words, the ***real-time*** window in which to do the required work.

So, it is the *real-time* term which controls the selection of CPU, software language, and software writing technique. The selection of CPU then becomes a matter of your ability to understand the real differences between the variety of available CPUs and your experience in writing efficient and effective control software for *high-speed* applications.

Remember, that certain languages are more **efficient** than others. If you write in only one language, and it happens to be Interpreted BASIC, for example, you might not find a **fast** enough or **cost effective** enough processor to do the job at hand.

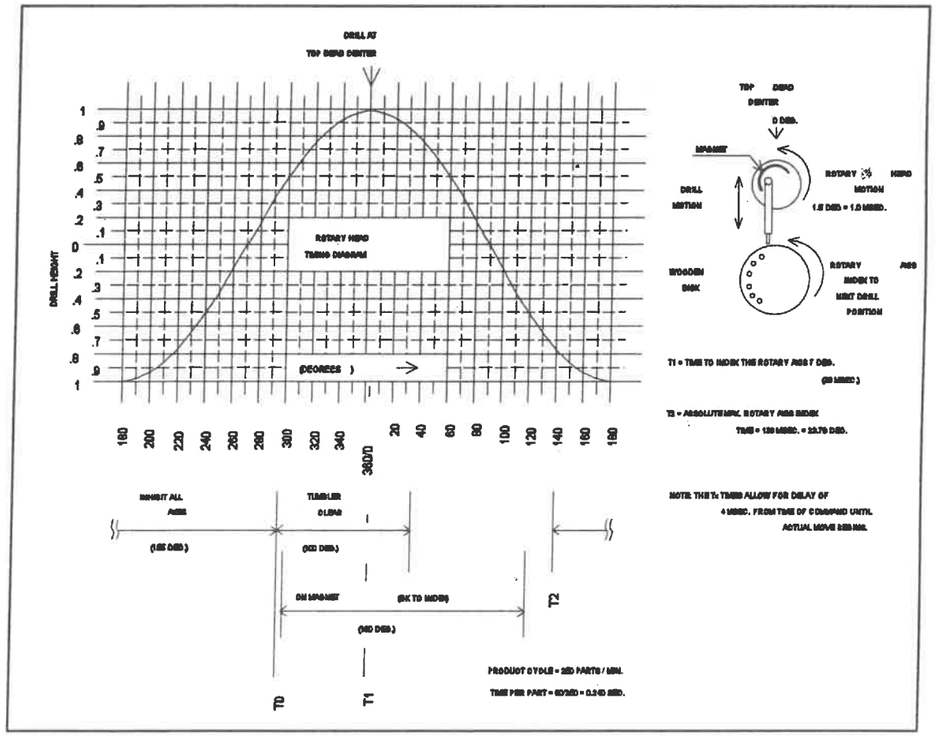

Figure 15.1 Timing chart for a rotary brush-making machine.

Example: A "High Speed," Real-Time Application

The following application example describes a system in which a 30-inch-long piece of material had to be applied to a continuous web process capable of traveling up to 7200 feet per minute. The customer specifications are listed in Section (I), and a complete description of my design method follows. This system was not only built, but met **all** of the specified design goals.

To add a bit of spice to this design application, you might find it an interesting challenge to use the section (I) specifications and develop your own approach. Compare it to the example. What did you do different? Remember , my solution is by no means the only solution.

Specifications . . .

- Application wheel design as shown in Figure 15.2
- Two single axis slave motors and a master encoder.
- Operates up to 7200 feet per minute (82 mph).
- Target position accuracy: 1 wheel revolution (3 ft.).
- Applicator roller must touch the applicator wheel within the gap or non-material portion of the wheel.
- Operation to proceed at 1 hour intervals.
- Each *Slave* to velocity ratio from 255:1 to 1/65536:1.
- Velocity match within 0.1% *master* to *slave*.
- System must signal a cutter that material is at the cut position.
- Applicator roller is operated by an **air** jump.
- System cost must be kept as low as possible (as small a motor/amplifier package as possible).
- The customer can write in BASIC language only.

Define the problem . . .

There are three critical parts to the given specification:

- The application jump roller is controlled by an air operator.
- The motor needs to be *small* possibly reducing the system acceleration capability.
- The point to apply the material is known only to the host computer, and it can be altered at any time by an operator.

Simplify the Problem . . .

The key to any solution is to define the problem in as few words as possible. In this example the following three problem definitions can be made:

- **Match velocity**
- **Apply material**
- **Signal Cutter**

Notice that the complexity of the problem as presented in Section I above has been redefined in simplified terms. Each problem can now be addressed separately to arrive at the final solution.

Solution . . .

At 7200 feet per minute (approximately 82 miles per hour), it didn't take much thought to realize that the host BASIC language would not keep up with the *real-time* requirement of the actual operation. Therefore, the objective was to handle all *real-time* requirements in Assembly code, while operator and other general machine handling functions were done in the BASIC. Since cost was an issue, it was advantageous to use a low cost CPU such as the Intel 8032 for the entire job. Besides the low cost of the 8032 it could be purchased with the Intel BASIC-52[2] language embedded in the processor. Because of this, support hardware and software could be reduced, creating a very cost effective system.

Note that I selected an 8052 without concerning myself about the software writing technique. I could do this because of my experience with the 8032 and other processors. However, with no prior experience with a particular type of processor, the method I would normally use to check out a CPU, is to first determine the average instruction cycle time. Next, I would list all functions which must be handled in each of the *real-time* operation windows (a system timing chart would be helpful at this point). These operations would include, but would not be restricted to emergency stop, keyboard, serial communications, I/O monitoring, higher priority interrupts, and others.

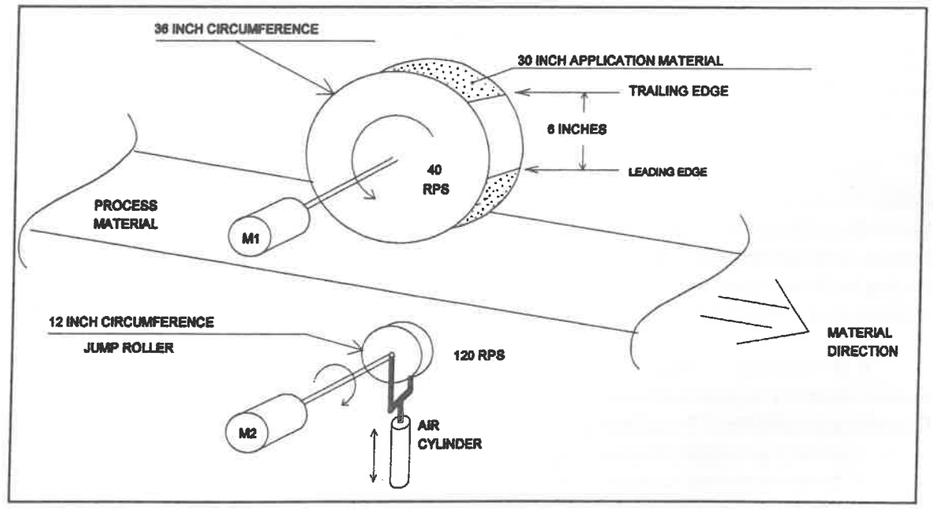

Figure 15.2 High speed applicator.

[2] BASIC-52 is a registered trademark of Intel. Micromint Inc. offers 805x and 803x microcontrollers with mask programmed BASIC-52.

It's your system; make your own list, but don't forget to include a safety shutdown, and other critical interlocking functions.

Next, I would sketch out a preliminary software flowchart showing only the critical *real-time* components of the operation. With this I could then develop a simple but effective software listing. In order to maintain the fastest possible code cycle time when writing, I avoided the use of calls or subroutines as much as possible. This practice results in lengthier source code, but it substantially reduces code run-time. By using a debugger or software simulator to benchmark the process, I can roughly estimate if code execution time has a chance to work in the actual system. I would then add up to 40% more code execution time to allow for code add-ins missed in the conception phase.

With the process selection made, I worked out which sections of the machine operation were to be handled in what language—BASIC-52 or 8032 Assembler. Solving each of the three problems listed above separately allows me to better implement the 8032 interrupt handling program and the main BASIC software.

Match Velocity

At the top system velocity of 7200 feet per minute, one DAC step (in a linear voltage system with a 12-bit DAC) produces a stability of ±0.0488%, well within the required 0.1% stability specification.

Discussion

Any motion controller gain structure will modulate the DAC to accommodate system following error. This will make the system motion appear steady even though the DAC signal (not the trajectory generator) is oscillating. I suspect that in over 99% of all systems, the DAC output does modulate. However, for extremely high stability requirements, each part of the controlling system (motor, motor amplifier, etc.) must be able to hold the given system stability requirement by itself.

Each step of the *Slave* trajectory generator DAC output, from zero to top velocity, should represent a change in velocity within the required system stability range, as should the system resolution, the stability of the motor amplifier, the stability of the motor, and the sensitivity of the motor amplifier package, since they are each a major factor in controlling system stability. If we assume that the system-loading or friction does not change while in motion, then ideally, the *Slave* controller will be able to stabilize on or within a specific DAC output step and not modulate for the rest of the move.

It is necessary to ensure that the **velocity match update time** will be fast enough to hold the motion controller trajectory generator velocity within the required velocity match window of 0.1%. Controlling the stability of the *actual* application wheel and jump roller velocities is not the direct task of the trajectory generator. It is the task of the gain structure used by the motion controller and the ability of the servo package to respond to the DAC step variations the controller will produce.

The general rule is to set the *Slave velocity match* update time to a value equal to or less than the time it takes for the actual motion to wind up outside the specified stability window with respect to the *actual Master* velocity. This is done as follows:

- **Calculate the master system mechanical and electrical time constants:**

$$Electrical\ TC = \frac{L}{R} \qquad Mechanical\ TC = \frac{(J_{SYS})(\Omega_{MOTOR})}{(Ke)(Kt)}$$

Where: L = Motor Inductance in Henries
R = Motor Resistance in Ohms
J_{SYS} = System Inertia in KgMet²
Ω_{MOTOR} = Motor Resistance in Ohms
Ke = Motor Voltage constant in V/krpm
Kt = Motor Torque constant in NM/Amp

- **Determine the required time for the slave trajectory generator to respond to a 0.1% shift in velocity:**

The *Master* system could accelerate or decelerate in 30 seconds. Assuming linear *Master* response at the low end of its time constant curve, a 0.1% velocity change in the *Master* could occur in:

(30) (0.001) = 30 milliseconds

- **Select an appropriate update time to maintain the system within the required stability of 0.1%:**

From the previous discussion, an adequate update time for velocity match calculations will be something less than 30 milliseconds. Using the infamous *x10 Rule*, I set the *Slave* update time and the *Slave* acceleration time to 0.003 seconds and 3 seconds respectively.

Next, I needed to check the system resolution to ensure first, the ability of each *slave* trajectory generator to properly follow the *Master* encoder, and second, the ability to generate an adequate number of error counts in order to maintain smooth *actual* motion.

$$\frac{7200\ \frac{ft}{min}}{\frac{60\ sec}{3\ \frac{ft}{rev}}} = 40\ \frac{rev}{sec}$$

$$\left(4000\ \frac{cnts}{rev}\right)(40\ rps)(3ms\ \ update\ \ time) = 480\ \frac{cnts}{update}$$

Each accumulated count from the *master* encoder would represent a 0.208% (1/480) change in speed. To meet the 0.1% speed match specification, it would be necessary to increase the system *Master* resolution. We redefined the *Master* encoder to be a 10000 Cnt/Rev unit yielding a per count speed change of 0.08%, thus meeting the specification. The controller position counters would accumulate over 100 counts in each motion control update period (256

microseconds). Proper setup of the gain structure in each of the *Slaves* would ensure smooth motion and proper response to *Master* encoder variations.

I redefined the velocity match update time to be 4.096 milliseconds accommodating the 256 microsecond motion controller update period (simply dividing by 16 would get the *Master* count value per *Slave* servo update period). It also gave the processor more time to accumulate velocity counts allowing calculation of the *Slave* velocity with a higher degree of precision.

Note that the velocity match update time period should not be made excessively fast with respect to the *Master* system actual time constants. This is so because extremely fast update periods can, in some situations, create following instabilities which would require lower gain settings, defeating what we are trying to do, which is control a high speed operation with tight motion stability.

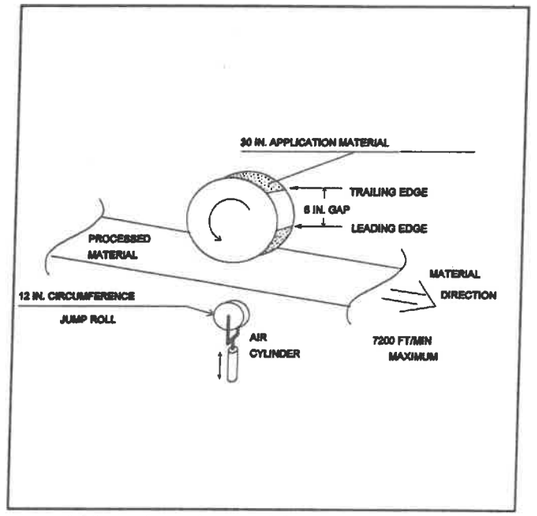

Figure 15.3 Applying the material.

I determined that the velocity match update should be handled in Assembly code.

Apply Material

Figuring out the method to apply the material to the product stock took a little longer to develop. The jump roller had to contact the material wheel in the gap between the leading and trailing edges of the material to be considered a successful operation (see Figure 15.3). Since the jump was being operated by an air cylinder, the repeatability of the jump action had to be tested. I placed sensors on the jump mechanism and used a storage oscilloscope to test for the time period between the signal given to *jump*, and the actual completion of the jump action. I found the jump time to be 100 ± 0.5 milliseconds, yielding a ± 0.72 inch reliability window at 7200 feet per minute. Since the initial window was 6 inches, the new window was now (6 – 1.44) or 4.56 inches.

Next, from the time an application signal was given, I had to figure out how the material could be applied within one revolution of the application wheel, using an 8032 low-speed processor.

Figure 15.4 illustrates the method I used. When the host CPU initiated an application, it would signal the material applicator CPU to start the velocity match cycle 10 seconds prior to the actual application signal. This would allow the application wheel and the jump roller motor velocities to reach and stabilize at the *Master* velocity prior to the point of actual application.

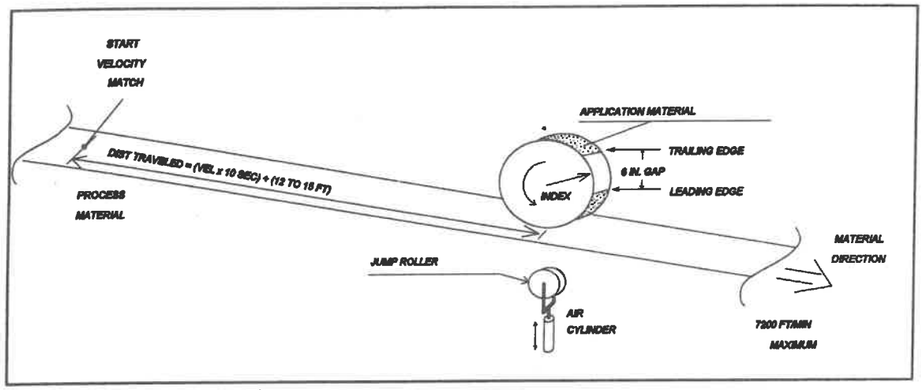

Figure 15.4 Distance versus Time.

For position control of the 6 inch gap, the leading edge of the material would be loaded onto the application wheel at a point three inches behind the motor encoder index marker. This meant that without any significant calculations the location of the material leading edge position would be known at any time by simply adding three inches to the absolute index marker position.

The next requirement was to set up a precise physical leading edge location of the material to be applied,—from the moment the jump signal was issued. At 7200 feet per minute. the main line would travel 12 feet in 100 milliseconds. Since the application wheel was 3 feet in circumference, the application wheel would make 4 revolutions in the time it would take the jump action to complete once the jump signal was given. Therefore, when running at 7200 feet per minute, and once the 8032 received the signal to apply the material, it only had to get the **next** index marker position. At that moment it would simply signal the application roller to jump. The material would then be applied at a physical location exactly 12 feet from the point at which the processor issued jump signal.

But what about lower speeds?

I accommodated **all** speeds by fixing the application leading edge position 15 feet (5 index markers) from the moment the "apply" signal was issued (refer to Figure 15.5). By doing this, I could construct a table of distance traveled in 100 milliseconds as a function of line velocity. I used this to figure out how far in advance of the 15 foot application point it would be necessary to activate the jump roller. In this way, the host CPU would "know" that the material would be applied at a position 12 to 15 feet out from the current point on the line at the moment of issuing the apply signal. This was true not only at high speeds, but at any speed, thus meeting the 1 revolution (3 feet) specification.

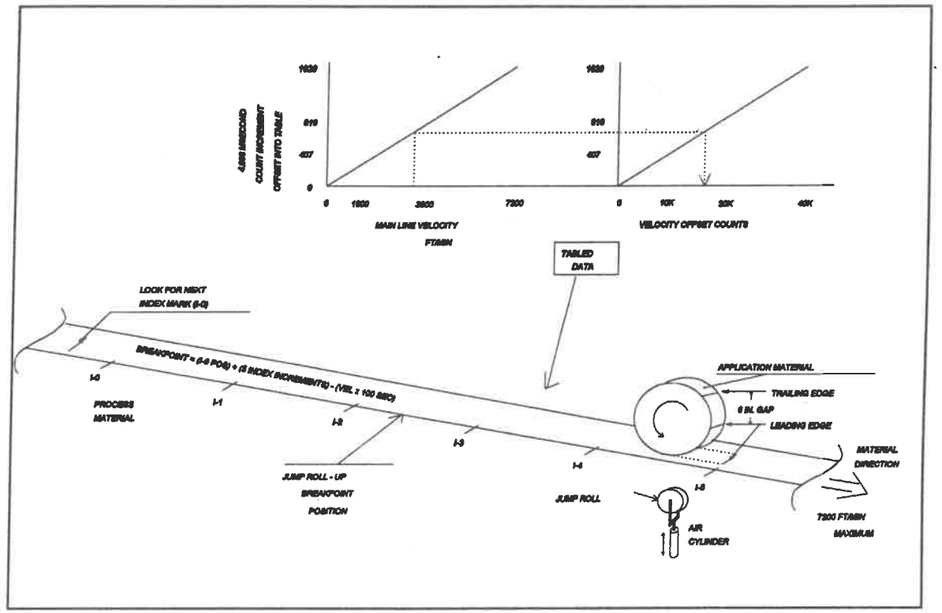

Figure 15.5 Material application considerations

I built a velocity table to allow the CPU to offset into the table directly from the count acquired in the 4.096 millisecond update period. Knowing that the resolution of the system was 10000 counts per 3 feet of motion, the maximum count which would be registered in the 4.096 millisecond update counter would be:

$$MxCnt = \left(\frac{7200 \ \frac{ft}{min}}{\frac{3 \ \frac{ft}{rev}}{60 \ sec}} \right) \left(10000 \ \frac{cnts}{rev} \right) (0.004096 \ sec) = 1638.4 \ counts$$

Therefore, each count collected in an update period would be worth 1/1638.4 of the distance that the main line would have traveled in the time it takes the jump roller to complete its action—100 milliseconds. That meant that the *gap* window (distance from the point of applied *jump* signal) would be a value equal to 0.088 inches per count increment (12 feet / 1638.4). Since a counter reading can only be guaranteed within ±1 count, the effective *gap* would reduce from 4.56 to 4.384 inches. Allowing for worst case software scenarios (about to fire the jump when distracted by an interrupt call), I allowed another 0.5 milliseconds of distance deviation (0.72 inches) to reduce the ever shrinking gap from 4.384 inches to 3.664 inches.

Since the jump firing point was a calculated distance from the index marker absolute position (which was set up as a fixed mechanical position referenced to gap), I could be confident that the jump roller would always make contact in the material wheel gap. No special *gap* position calculations would be necessary since the gap was mechanically controlled with respect to the material wheel Index marker.

Signal the Cutter

The final operation was to signal the cutter unit that the leading edge of the applied material was at the proper location. This was easy, since I knew the absolute position of the material leading edge upon jump roller release; and, therefore, I knew where the leading edge of the material would be applied. It was simply a matter of adding the distance from the leading edge of the applied material to the cutter to the absolute position of the index mark previously obtained (refer to Figure 15.6). This position would then be set in as a breakpoint in the motion controllers CPU which in-turn would be used to signal the cutter that the applied material was at the proper location.

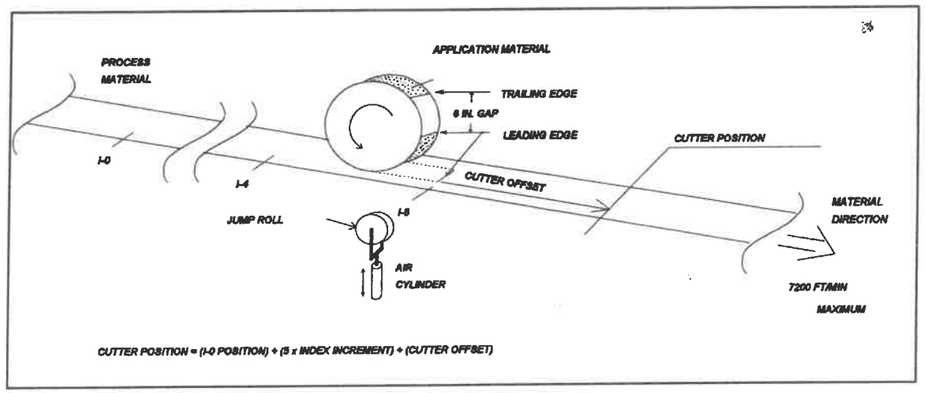

Figure 15.6 Positioning the cutter.

Discussion

Regardless of processor speed, I probably would have done nothing different to do this task. The objective in good design is to accommodate the variables; control the system; and overall, keep the software effort to a minimum to simplify debugging and maintenance. In this application, the *real-time* system concerns were met with an 8032 Assembly-code software package. Would a DSP have been more effective? It probably would have saved code (multiple byte manipulation versus single byte), but at an increased system hardware cost. Since the customer will manufacture several hundred of these systems per year, the DSP would definitely not have been the best choice for the job.