# Chapter 14

# Language and Communications

### Features

*High-Level Languages and Real-Time Environments*
*Just How Fast Does It Have to Be?*
*Developing a Communications Method for a 32-Axis Profiling Machine*



## High-Level Languages and Real-Time Environments

Talk to any ten motion control engineers, and you are guaranteed to receive at least 20 entirely different solutions to your problem. This is not to say, however, that any of the solutions are wrong, since there can be hundreds of different solutions to any given problem. Whatever works, is right.

Engineers' points of view are based on such things as their industrial exposure, level of education, general thinking habits, project budgets, time constraints due to backlogged jobs, their track record, and lots of other things. From a customer's point of view, a particular designer's solution might be inadequate based on decisions of the customer's support personnel, their exposure, education level, additional required training, general thinking habits, budget, and so on.

Attempting to assemble the right motion control hardware into a functioning machine with several possible sets of restrictions[1] is difficult enough; but to base the motion computer selection primarily on its programming language rather than its functional ability, reliability, and cost is down right foolish! Considering a motion control board that is reliable, fits the application boundaries, allows for growth, and is cost effective should top the agenda in any decision making process.

If you're jumping to the conclusion that I advocate tradeoffs in software writing effort versus features or performance to insure peak system operation . . . I AM. After all, is not that what you are actually paid to do? Notice, however, I carefully stated: **effort** not **time**.

Writing in any high-level language can restrict system capability. The more user-friendly the high-level language is, the easier the program development effort becomes; and generally, the desired operation becomes more compromised. The bottom line is that a precomposed language, although reducing your programming effort, rarely allows you the precise control over the system that you are seeking. You can inadvertently be restricted to *playing games* around sequential operation rather than obtaining the simultaneous operation that you originally required. In one case, you might have to work out a complex communication protocol. Using a high-level language, you might have to ensure adequate data transfer to the peripheral rather than expend software writing effort implementing a more highly desirable direct bus operation. In other cases, you might have to employ specific I/O handling. A high-level language may not allow you to do a task that the hardware may otherwise allow.

And let's not forget *creeping feature-isms*, those *after-the-start-up-desires* that cause you to sing the *Rewrite Blues*. Of course there is the *let's-speed-it-up* requirement that makes every engineer cringe with frustration.

I have had opportunities to use a fair assortment of stand-alone and bus-based motion controllers; and, like any product, they each enjoy certain strengths and suffer certain weaknesses. The method of communicating to the motion control board (language) appears to be a major weakness. It appears as though high-level programming languages were developed to sell Product rather than to solve problems.

[1] Cycle time, product handling, serviceability, learning curve, budget, and schedule name only a few.

But isn't that the way it has to be? After all, no ONE software package can really solve every problem at hand.

Just how generic must a language or software package become before it loses its effectiveness? Let us assume for the moment that all available motion control boards on the market maintain the same hardware capability. The greatest weakness with any of these boards is now the responsibility that is given to the designer in software development. Software development required to do the task at hand can be thought of as a *base* from which the remaining software development effort will evolve. This base includes items within the computer such as: operator interface, data manipulation, table construction, and I/O handling.

Software development required to deal with a motion controller is a function of the value of the command set within a given controller and whether the controller is RS*xxx* based, Bus-based, or both. Two factors—command set and communication style— determine the effort that you will have to put forth in order to solidify the overall system operation.

Can the updating of system motion information be manipulated to operate in a more proficient manner, and make the presentation of the operation more meaningful to the user? Remember that long update times from keystroke entry to display tend to hamper operator efficiency (i.e., the control becomes the bottle-neck). Also, long RS*xxx* communication strings that have to be verified and then decoded by the motion device reduce the high speed *real-time* capability. In other applications, signals that must be detected during the motion might be missed if the motion device cannot multi-task extraneous data while maintaining precise control of the motion.

To clarify a point, any single computer device is limited in its ability to function in its hardware environment or in *real-time* by the operating software. For adequate overall performance, you must maximize your computer's ability. For example, if you are using a software system that does not allow access to an existing RS232 port, you suffer a software, not a hardware, problem. This of course restricts the ability of the unit to access data. It may prove difficult to obtain peripheral information leading to software assumptions, or *bugs*. An example might be assuming an intact communications message without doing a checksum or CRC[2] operation.

When dealing with control design, you must consider the *real-time* system response. If you require *real-time* handling for successful operation, it might possibly be the first factor in the elimination of (or the selection of) certain control devices. For example, if you need to capture a 100-microsecond pulse during a machine cycle, it might be necessary to place this signal on an event driven input (interrupt). It is obvious that a 25-millisecond program scan could not be tolerated. Further, if the 100-microsecond pulse requires a response (or reaction) within 100 microseconds of being sensed, some form of interrupt-level programming is probably required. Would RS*xxx* work in this case? How about a BASIC language program?

Methods employed to insure capturing and responding to system signals require that the designer study the entire input/output operation in order to select the optimum device. Notice that I have considered only the software with respect to response time (scan rate) and not to actual software syntax. At this point, let's say the system design is within time constraints, and the software is not considered to be part of the problem. If you can trim the hardware down to a few brand-name selections, you can then decide if the embedded software will allow writing code in the most time-effective manner. If it

[2] Cyclic Redundancy Check



cannot, you must ask yourself, will the motion-control card allow writing the software in a different language, such as Assembly language, to help the motion controller maintain the degree of control the system requires?

To select the proper motion-control device, clearly you must consider system restrictions first. Then, after knowing all of the system *real-time* requirements, you can choose motion control devices fitting the system hardware requirements. At this point, software can be scrutinized for optimum handling.

For example, if the system requires high speed response, such as a film handler, a RS*xxx* may restrict the time it takes to set up and start the motion, and therefore, requires a bus-based control. If multiple axis control is required, a multitasking bus-based control with processors on each axis may be in order. This would provide true simultaneous system motion. If the motion is a distance from the host and/or a host is not part of the system, a stand-alone embedded motion control device with RS*xxx* may be necessary.

We then need to find out whether a stand-alone device can simply *slave* off of the operation, or if it should have its own processor to be able to maintain some or all operational control.

If you simply must write software, what language is best? Any high-level language is situation dependent, and even if it is written properly, it may still not handle the system adequately. Knowing this, you must not be afraid to write source code in the software language (or coding) required by the system (best suited for the job at hand). I don't mean to imply that you must *reinvent the wheel* each time you design a system; I want only to stimulate your thinking process. There are possible restrictions you may place on yourself and your system before the design effort if you allow the software to set the hardware design direction.

The first step is to remove the software language obstacle from the system design *equation*. Also, use of bus-based and RS*xxx* communication-based controllers do not apply to every application. It is your responsibility to determine the best *fit* for your system. Data handling from host to motion control card can be done at PC bus rates, over networks at rates of several megabits per second, or at slower rates over RS-232 lines. No matter what software language you are using, you can adapt a motion card. Remember, when you require a motion controller, do not let the software language restrict the choice of hardware for the application.

## Communications: Just How Fast Does It Have to Be?

I had an interesting conversation with a fellow who was about to embark on an exercise in communications. The problem he described was one of getting information from point A to other points (B, C, D . . . ) as quickly as possible. The machine was to use a series of stand-alone devices for point-to-point motion control and general I/O handling. As he saw it, information had to be routed around so fast, the entire system may as well have been one big **Bus** structure. However, what we discovered as a result of further discussion was that RS422, at 9600 Baud, was perfectly capable of handling the tasks at hand without degrading system performance.

In this particular case, he had one thing going against him before the project even began. That one thing was his assumption that information should be processed in ***bulk format***! Bulk data transfer is the typical approach used in the PLC world where it is important to process **all** data before decisions can be made. It is also fairly common in the PC world where experience with multiple axis mechanical systems is limited but inevitably encountered. Although, in the field of motion control, information can generally be parsed out still allowing more than adequate system operation.

### Think about it . . .

Is it really possible to run a mechanical system faster than the ability of some form of electronic control to keep up? Wouldn't it be possible to invoke partial data transfers to stay ahead of the operation while still satisfying all of the handling requirements?

In any motion control system, it is important to set priorities before dealing with the transfer of data. Items such as machine speed (parts per unit time), type of axis coordination required and I/O handling will all play a part in how the data can be parsed and delivered to the remote devices.

What is usually thought of and put in place is a system in which one device is assigned to be a *Master* and the others as *Slaves* to the process. None of the stand-alone *Slaves* maintains the control to make real world decisions based on immediate situations. Limited exceptions may include emergency stop or limit switch activation. This means that the lone *Master* must handle the vast majority of the data transfers; and therefore, the communication rate will need to be increased by orders of magnitude.

Does this mean that by parsing the data, each unit must be a *Master?* Possibly, but what it really implies is that the designer must do something that has probably been bypassed, such as develop a machine timing diagram or a software flow chart. If the machine can be simplified into subsections (which could possibly be broken down again into simpler sections), or if the software can be developed before the hardware takes shape, you can gain several things:

- The power of the stand-alone processor can be used to aid system performance, rather than simply act as a *dumb* follower.
- The amount of transfer data can be reduced for each communication requirement, and therefore, so can the communications rate.
- The ability to debug the control is simpler due to the smaller machine sections involved.



A better method for interlocking the machine sections can be designed in, leading to methods that can usually offset software development. The *creeping feature-isms* that always crop up can be put in place faster and with less difficulty.

You do not need to be an expert in the field of communications to realize that by properly organizing your thoughts about the machine operation via the use of timing diagrams and software flow charts you can better structure your communication requirements. The ability to enhance the communication method in the following example does not necessarily fit every situation, but it fits many. The key is figuring out if it fits **your** situation.

The following example shows how a 32-axis machine communication scheme was derived by understanding exactly how the machine worked. An operator interface software flow chart was developed **prior** to a the machine timing diagram. This was to ensure that none of the required operator tasks would be left out of the timing diagram. I also felt comfortable in knowing that the timing diagram would be developed with both the machine operation and the operator in mind. The machine control that eventually evolved used stand-alone devices for the axis motion. A PC host was used for the operator interface as well as general machine handling. This method used 32-axis stand-alone processors, and eight second stage processors to interface between the PC and the axis controllers in groups of four. One PC host processor was used to interface to the operator and communicate with the second stage processors, as well as talk with an office PC for general inventory control.

The final design used only three separate software packages. This was only one more than would have been required had the control been setup as a *Master* with multiple *Slaves*. However, the design provided *Slaves* with the capability of making decisions rather than just acting as simple dumb remote terminals.

## Developing a Communications Method for a 32-Axis Profiling Machine

First, let's outline the project. The machine, in design, actually consists of two machine units, each with eight axes of motion control on opposite sides of the product travel (see Figure 14.1). Both machines are capable of cutting any required pattern simultaneously along two of the product edges (see Figure 14.2). After the completion of the cutting operation on machine #1, the product is transferred to machine #2 that is set at 90 degrees relative to the first unit (Figure 14.1). Therefore, machine #2 can do similar cutting operations to the alternate two product edges.

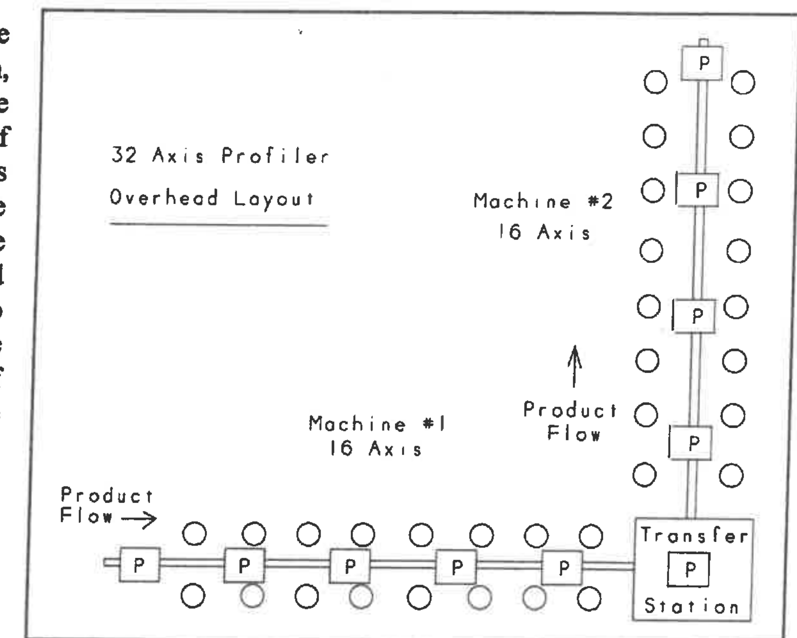


Figure 14.1 A 32-axis Profiler.

Not all of the motions are for cutting requirements. Some of the stations are cutters; some are sanders, buffers, . . . allowing a reasonably large variety of work to be performed on the products to be produced.

The machine must be capable of operating at speeds up to 80 feet-per-minute with variations no greater than ±1/32 (0.031) inch.

The concept of this machine, is to produce a family of parts, which will form perhaps a kitchen cabinet set or some other type of furniture when all grouped together.

The product material that enters machine #1 can be of random lengths and must be measured before being worked on, but after they are captured by the machine. Because products are of various lengths, the host computer must precalculate the move profiles required for given length ranges and then must form the move programs necessary for given product lengths. Finally, the control must allow program editing, even on the current program without stopping the line.

To control thirty-two axes of independent motion, each requiring point to point linear and dual-axis circular interpolation moves on parts of varying size moving perpendicular to the tools all being coordinated to a chain drive over which they have no control, seems at first glance to be an impossible task. At a second glance, it appears as if an extremely fast communication method from the *Master* to the *Slave* axis controllers is imperative. However, if an effective method of measuring parts, producing profiles and parsing data could be developed, the communication aspect of the project might just become a minor element of the overall design scope.

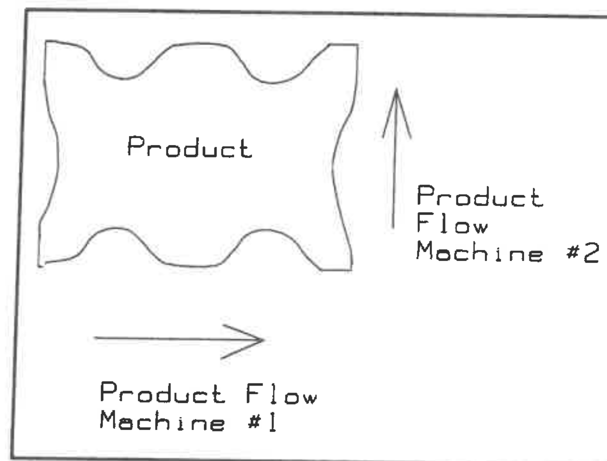


Figure 14.2 Diagram showing Product flow.

In this particular design, it was necessary to set down some ground rules regarding how the operator and the machine were to perform. As with any control, there were to be certain things required of it to make it a viable piece of equipment. These minimal functions were . . .

- Internal Self-Test and diagnostics.
- General machine configuration (i.e., number and resolution of axes).
- Program Editing, and Tool information.
- File handling –(e.g., Load/Save)
- Manual data input functions (MDI) – (e.g., Jog, I/O)
- Program Run – (e.g., Auto/ Single Step)

With the rules set in place, I developed a preliminary software flow chart. As the software direction evolved, I began thinking of methods to handle the parsing, and transfer of the required information.

One item that immediately came to mind was that the measurement of the parts entering the machines could be a task of either the host or any one of the *Slave* units. Then the problem would be



that the collected information would have to be passed around to all of the axis controllers via communications. But what if it could be configured so that all of the axis controllers measured the parts independently? This would eliminate one information packet from the communication scheme.

Expanding upon this thought, the host software could then concentrate on operator interface, program profile development, general I/O handling, axis (*Slave*) monitoring, and other types of system commands such as Start, Stop, and Jog. The actual running operation of the machine could then be handled by the *Slave* units themselves.

Because the host was to control all of the part profile development, the programs required for each part length to be processed could simply be downloaded to the specific *Slaves* prior to the actual machine motion (but after pressing the <RUN> button). As the products entered the machine, they would be measured simultaneously by each *Slave*, and the measurements would be held in each of the *Slave* axes motion controllers (FIFO operation) for use as the product entered that particular work station.

Since there could be up to 20 different lengths involved in the production of a family of products, there would be 20 times 32 axes of program data to download. If each part program contained 200 bytes of data (a conservative estimate), there would be a total of 128,000 bytes (32 x 20 x 200) to download.

What would be a reasonable operator wait time for this download procedure to complete before he saw any action on the machine? One second . . . two . . . ten? I elected to not exceed five seconds since I knew that the code formulation I would use would be considerably less than 200 bytes per program. Therefore, the transfer time would decrease as well. Transferring 128,000 bytes of data in five seconds results in 25,600 bytes per second or 256 Kbaud (Kbps), still a little fast.

Knowing that the product could travel at a maximum of 80 feet per minute and that there is a minimum of two feet between work stations, it would take 1.5 seconds for the product to reach the first station and 1.5 seconds for each station after that. The most critical station is the first.

All of the first station move profiles had be loaded into both the left and the right hand first station axes within 1.5 seconds of the start signal (2 Ft / (80 Fpm / 60 Sec)). What I elected to do was send the required data station by station until all 32 stations were complete. To do this, it would be required to send 20 programs x 200 bytes x 2 stations or 8000 bytes of data in 1.3 seconds maximum. This amounts to 8000 bytes x 10 bits per byte / 1.3 seconds or 61.538 Kbaud (Kbps). The computer system that was selected, had a data transfer rate of 62.5 Kbps (1.28 seconds). This was set up on an RS485 two-wire communications line. Machine motion START could now occur immediately without hesitation, which improved the machine efficiency (no delays) and enhanced the operational package as viewed by the operator.

The moment the <RUN> button was pressed, the machine would come "alive" and begin moving the product down the line. The host would also begin sending the part program profiles to each axis motion controller at 1.4 second intervals. Therefore, successive station programs would be loaded before the product parts actually reaching any of the upcoming stations. The result of this scenario is that it would take the part approximately 24 seconds to pass completely through the machine but only 22.4 seconds for all of the axis software to be loaded.

This was the approach used, since it satisfied both the "high speed" communication requirement and the instantaneous appearance of machine operation. Also, once all of the family of programs were downloaded, no further high speed communications were required, since the remaining machine operation could be handled entirely by the *Slave* units.

Not all system communication problems lend themselves to this interpretation, but it is **rare** to find a machine that cannot be subject to data parsing in such a way that adequate operation and reasonable baud rates will not coexist.

It is important to grasp the idea of the use of software flow charts and timing diagrams to help in the development of machine to software coordination. Being able to separate the operational control into a distributed processor-based system, rather than relying on *Master/Slave* concepts, can only usually save you time and money.



**Notes:**