Validation Results of Specifications for Motion Control Interoperability

Sandor Szabo, Fred Proctor

National Institute of Standards and Technology
Gaithersburg, MD 20899-0001

Abstract

The National Institute of Standards and Technology (NIST) is participating in the Department of Energy Technologies Enabling Agile Manufacturing (TEAM) program to establish interface standards for machine tool, robot, and coordinate measuring machine controllers. At NIST, the focus is to validate potential application programming interfaces (APIs) that make it possible to exchange machine controller components with a minimal impact on the rest of the system. This validation is taking place in the Enhanced Machine Controller (EMC) Consortium and is in cooperation with users and vendors of motion control equipment. An area of interest is motion control, including closed-loop control of individual axes and coordinated path planning. Initial tests of the motion control APIs are complete. The APIs were implemented on two commercial motion control boards that run on two different machine tools. The results for a baseline set of APIs look promising, but several issues were raised. These include resolving differing approaches in how motions are programmed and defining a standard measurement of performance for motion control.

This paper starts with a summary of the process used in developing a set of specifications for motion control interoperability. Next, the EMC architecture and its classification of motion control APIs into two classes, Servo Control and Trajectory Planning, are reviewed. Selected APIs are presented to explain the basic functionality and some of the major issues involved in porting the APIs to other motion controllers. The paper concludes with a summary of the main issues and ways to continue the standards process.

1. Introduction

What is motion control interoperability? In the “field-exchangeable” scenario, a service technician simply exchanges a board and runs an installation script. This scenario assumes that motion control functionality is implemented as a board that connects to the control system by way of a backplane. It assumes that no source code changes are made to the controller software (i.e., recompilation is not required). The scenario also assumes that existing machining part programs do not require modifications. Are all these assumptions realistic? Is this the direction where the industry is heading? In examining interoperability, the first step is to establish the minimal requirements and to propose possibly desirable features.

Once the requirements have been established, the next step is to define standards that enable interoperability. An important step toward interoperability is to develop Application Programmer Interfaces, or APIs. These are the software interfaces for a motion control library. The goal of the API is to isolate the specifics of a particular motion control board and to minimize its impact on the control system. The APIs allow integrators to select from a variety of motion control vendors, and allow motion control vendors to target their product for a variety of machines. This raises several questions, such as, “What are the minimal functional requirements of a motion control system? What is the process for selecting APIs? How are the APIs validated? How are the APIs extended?” The Enhanced Machine Controller (EMC) consortium [1] was formed to investigate these questions and to help guide possible future standards.

Focus on Machine Tool Controller

In this paper, the focus of interoperability is on machine tools. In many ways, the functionalities of a robot, an inspection machine, or a machine tool are very much related. But, they differ in their applications, terminology and programming methods, which makes them sufficiently distinct to treat them separately. For example, robots generally use a complete position and orientation representation of the end effector. This representation simplifies programming manipulation tasks in six dimensions. In machine tools, some axes control the tool, while others control positioning tables, spindles or tool changers. Thus, programs for machine tool operations use representations involving individual or groups of axes. Some progress has been made toward unifying servo control APIs because these cluster around control of a single motor. Creating a set of unified trajectory planning APIs is less obvious and often hinders discussion and consensus in the various communities. The approach used within the EMC consortium is to focus on APIs for machine tools with the understanding that robot applications will use existing machine tool APIs when applicable. A future goal is to determine if a more generalized set of APIs that satisfies all the communities can be distilled.

2. API Development Guideline

A number of guidelines may be used to develop a standardized set of APIs. The requirements, goals, and limitations will influence the specific guideline selected. In the EMC consortium, a primary goal is to devise a set of APIs that allows interoperability among motion control products. But how is the set of APIs derived? A systematic approach must be followed to ensure that results are obtained efficiently. The following guideline was used to produce a specific level of interoperability.

1. Establish a set of functional requirements.

2. Develop a set of APIs that satisfy the requirements.

3. Perform tests that validate whether the APIs satisfy the requirements.

4. Iterate steps 1, 2, and 3, until a satisfactory API specification is produced.

The next sections examine steps one through three.

2.1. Functional Requirements

An important early step in the API development process is to establish the functional requirements for a motion control system and to determine the desired level of interoperability. The requirements focus the designers' efforts, enabling them to establish concrete deliverables. Of equal importance, the requirements establish the metrics used to validate APIs.

The EMC team involved in the effort had a significant experience base in developing software and implementing machine tool control systems. This provided an intuitive feel for the type of functionalities needed. This intuition, along with the general scenario for a field-exchangeable capability, drove the selection of many of the functional requirements. An attempt was also made to categorize the requirements to better handle requirement changes or the desirability of certain features. Here is an example of some of the requirements that drove the API development:

Low level motion control

- achieves position and velocity control for each axis
- calibrates each axis
- specifies control parameters (e.g., PID gains and feedforward coefficients)
- specifies motor velocity and acceleration limits during motions
- automatically stops axes during unsafe operations (e.g., travel limits, velocity and acceleration limits, following errors)

Coordinated motion control

- plans and executes motions required the coordination of individual axes (e.g., straight lines, circular arcs)
- satisfies constraints for acceleration, deceleration, and blending between successive motions
- maintains a look-ahead queue of pending motions
- pauses, incrementally steps, and resumes motion

Advanced motion control

These requirements were identified for future consideration.

- modifies motions to compensate for cutter tool diameters
- plans and executes elliptical motions, splines, or other higher-order paths
- supports sensor-based motions (e.g., force control)

Installation and Set-up

These requirements address operational details that are not part of the API definitions, but are still significant to motion control interoperability.

- physical installation and configuration
- plug compatibility with connections to motor inputs and outputs

One important aspect that is not covered in these functional requirements is the ability of the controller to achieve path and machining accuracies. This has been omitted because achieving such accuracies partly depends on the machine tool characteristics. One may reference [2] for approaches in specifying and measuring accuracies in machine tools.

<u>2.2. API Development</u>

There are a number of ways to decompose a motion control library into APIs that give the controller flexibility to perform tasks. The task of machining a part is typically specified in numerical control (NC) programs, in languages such as EIA-274-D [3]. In general, there will be a high correlation between program instructions and motion control APIs. Nonetheless, there is latitude in selecting and developing a set of APIs that satisfy the functional requirements. Care must be taken to understand this latitude and not constrain the APIs to a particular programming language. With this in mind, some of the possible strategies considered were:

1. Adopt an existing motion control product for machine tool control. This strategy works well when a defacto standard exists. When such a standard does not exist, the drawback is the exclusion of a large segment of the community who do not use the adopted product.

2. Develop a generic set of APIs that motion controllers should provide and implement with in-house hardware and software (e.g., encoder input and analog output boards). The advantages for this approach are minimal bias in the API definitions. This places advantages or disadvantages on any one motion control vendor and a flexibility in implementing desirable functionalities. Disadvantages include the investment of engineering resources toward development and support, and the limited availability of the implemented hardware and software.

3. Select several commercial products and distill a common set of APIs. This approach minimizes the development and support investment, while reducing the nonuser portion of the community. The drawback is that the approach suffers from a breath-first strategy where each API must be designed and implemented for several products before a complete machine tool controller exists.

The final approach selected was a modified version of the third strategy. A baseline motion control board was selected, and a generic set of APIs that were implementable on the motion control board was developed. This enabled the rapid development of a testbed for validating APIs. A second commercial motion controller board was procured and the APIs were mapped to the board's environment. The main disadvantage to this strategy has been the temptation to rely on functions and behaviors of the baseline motion control board which could make the mapping to other implementations awkward.

In implementing the APIs, the EMC team performed the role of the motion control vendor: implementing a software interface (e.g., device driver) which provides conformance of the proprietary product to the generic API. In the future, it is expected that all vendors who decide to conform to the API specification, will develop software that maps the API instruction call to specific instructions supported by their product. In turn, the developer of complete machine tool controllers will make API instruction calls in their software. Some portion of the APIs will be executable software residing on the host. Depending on the vendor's implementation, the host software could range from a simple board communication mechanism to a complete motion control library implemented on the host. One can imagine the timing implications on the control system when considering the various implementation strategies.

2.3. Validation Process

The validation process is a straightforward exercise once the requirements are established and the APIs are implemented. There are two inputs to the validation process. First is a validation metric which is a detailed breakdown of each requirement into some identifiable "yardstick" for measuring to assure that the requirement was satisfied. The common method to produce a validation metric is a detailed, itemized table that lists each functional requirement and a rating method. Table 1 contains a portion of an example validation metric.

Section	Requirement	Rating Method
1.0	Servo Level Operability.	
1.1	Jog each axis.	[pass|fail]
1.2	Stop jog on following error.	[pass|fail]
1.3	Stop on limit switch.	[pass|fail]

Table 1. Example of a validation metric.

The second input is a test plan that exercises the APIs to produce a rating score. The test plan describes the test set-up and lists the detailed steps of how the test is to be conducted. The plan contains references to the corresponding requirements being validated. The validation process is the execution of the test plan and a capture of the rating measurement. The two validation outputs are the results of the test and an analysis of the results. The results of the test are collected in a table similar to Table 1. The difference is a new column for the score. The analysis is important when a requirement is not met. The analysis should document the reason for failure and suggestions for meeting the requirement. In some cases, the suggestion may include recommendations for altering the requirements.

3. EMC Architecture

Figure 1 shows the EMC architecture which provides a broad overview of the major components in a machine tool controller. In this figure, boxes indicate the individual modules for which interfaces are intended to be defined and validated. These include Task Sequencing, Trajectory Generation, Servo Control, and Discrete Input/Output. The focus of motion control interoperability is to examine the interfaces with the Trajectory Generation and Servo Control modules.

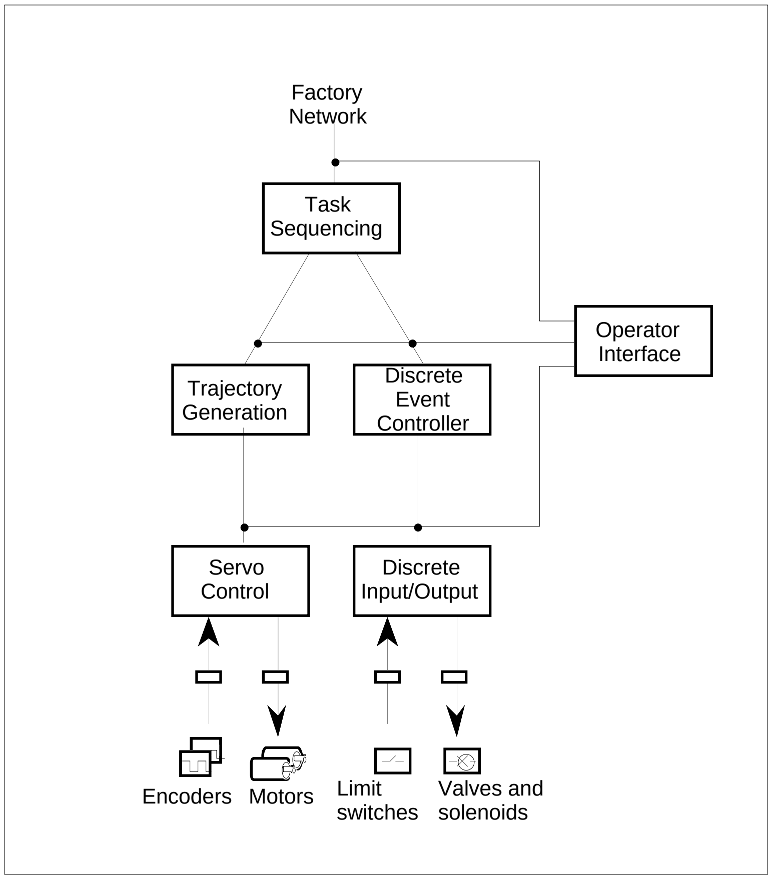

Figure 1. The EMC Architecture. Interfaces for Task Sequencing, Trajectory Generation, and Servo Control were targeted during the validation process.

Several interface specification mechanisms developed by the EMC team [4] pertain to the motion control APIs. First, the APIs are specified in the C++ programming language. The specification consists of messages going into each module and world model data provided by each module. Both the messages and world model data are implemented using C++ classes.

Supplementing the message specification is a model of data transfer, the Neutral Manufacturing Language (NML) [5]. This model provides for "mailboxes" of data with one or more readers and writers. Each module is modeled as a cyclic process, which reads its input command from its supervisor, reads the status of its subordinates (or sensors), and computes and sends outputs to its subordinates (or actuators).

The interface specification is divided into two parts: commands that each module will perform and status that each module will maintain. The syntax for a command is:

nml_*level*_*command*

where nml signifies a command in the NML, *level* refers to the level of control, servo or trajectory, and *command* is the action to perform, home or jog.

Two types of APIs are required to interface to the world model. One is for updates, and the other is for queries. The syntax for the update and the query, respectively, are:

*level*_wm_*data* and *level*_wm_set_*data* where *level* associates the data to a level of control and *data* is the value to set or to return. Several examples are illustrated below.

4. Motion Control APIs

The APIs that were first developed were designed to provide a basic level of motion control. These APIs were implemented using the baseline commercial motion control board. The APIs were ported to a second motion control board after the initial test was completed.

Units and coordinate systems are implicit in the APIs. Units are always millimeters for linear measurements, degrees for angular measurements, and seconds for time. The origin of the coordinate system for the APIs is the home or calibration position. That is, commanding axes to their zero position will result in their moving to the home position.

The following sections show some of APIs and some factors that affected the API development.

4.1. Servo APIs

The servo APIs are used to control a single axis. Position control is proportional-derivative-integral (PID) error compensation with velocity- and acceleration feedforward terms. Motions for an axis follow trapezoidal or s-curve velocity profiles. The parameters for an axis are contained in an initialization file. An example specification for an axis is shown in Table 2. The following sections contain selected examples of the C++ language APIs.

[X_AXIS]	Tag
P=200000	Proportional gain
I=10000	Integral gain
D=1280	Derivative gain
VF=0	Velocity feed forward
AF=0	Acceleration feed forward
COUNTS_PER_MM =1578.4	Encoder conversion factor
JOG_SENSE =1	Toggle between 1 and -1 to reverse commanded versus actual motions during jog.
HOME_SENSE =1	Toggle between 1 and -1 to reverse commanded versus actual motions during home.

Table 2. Initialization file specification for a single axis.

Jogging

```
extern int nml_servo_set_gains(int axis, NML_SERVO_GAINS gains);
extern int nml_servo_jog(int axis, double speed);
extern int nml_servo_jog_stop(int axis);
extern int nml_servo_incr_jog(int axis, double speed, double incr);
extern int nml_servo_abs_jog(int axis, double speed, double abspos);
```

Prior to jogging an axis, the axis gains are set using `nml_servo_set_gains`. Jogging an axis consists of accelerating the axis to the commanded `speed` (in millimeters per second) and maintaining the speed until a stop is received, a specified position is reached or a limit switch is activated. The direction of the jog is based on the sign of `speed`. The `nml_servo_jog` function will continue until a `nml_servo_jog_stop` is received. The incremental version of jog will stop once the axis has moved the prescribed increment. In the absolute version, the goal of the jog is specified as a position that is relative to the origin of the machine work volume.

Each board supports the various jog functions. The motion API concept of a coordinate system and direction may differ from the motion control board concept. This is resolved by setting the JOG_SENSE parameter in the initialization file (see table 2). If JOG_SENSE is negative, a commanded positive jog results in a negative encoder motion (decreasing encoder count). Setting the proper encoder direction is handled within the jog APIs.

Homing

```
extern int nml_servo_home(int axis);
extern const NML_JOINT_FLAG servo_wm_home();
extern const NML_JOINT servo_wm_homepos();
```

The nml_servo_home commands the motion controller to home a specific axis. The axis is driven until a home switch is activated. The encoders are captured at the precise instant the home switch is activated. The function servo_wm_home returns a flag indicating when the home search is complete and the function servo_wm_homepos returns the home location. The home position is used to define the origin of the machine work volume.

Implementing basic homing APIs were straightforward for both boards. Some of the factors addressed included determining the proper direction to travel in search of the home switch and handling the case where the axis is powered on the wrong side of the home switch. Setting the proper direction for a home is performed in the API based on the HOME_SENSE flag. If the flag is positive, the home switch is in the positive direction (increasing encoder counts). Determining the proper action when the machine comes up on the wrong side of the home switch is handled within the task sequencing logic that supervises the motion control system. Any strategy can be implemented, but all must use the motion APIs for moving and homing the axes.

4.2. Trajectory APIs

The trajectory APIs are used for coordinated control of multiples axes so the tool will follow a desired path. Basic paths include loosely coordinated end-point moves, straight lines and circles. Loosely coordinated moves (rapid feeds or traverses) result in unspecified paths during the motion, but all axes are constrained so that they end up at the final position at the same time. When machining a part on a machine tool, it is desirable for the axes to be tightly coupled. These trajectories are called feeds.

During a machining operation, it is also desirable to blend motions together. This enables the tool to more efficiently remove metal. A queue is needed so the motion controller will properly blend consecutive trajectories. Several functions must be implemented to support the queue. Queue management becomes more complex when the operator decides to single step through machining instructions after they have been queued. Because one machining instruction (a line in an RS-274 part program) may cause more than one instruction to be sent to a motion control board, a mechanism must be installed to enable correlation between queue depth and the part program instruction. This was implemented by requiring that all motion APIs return the number of instructions they send to the queue. The task sequencer then maintains the correlation between the current queue depth and the part program instruction. A select portion of the APIs and insights into the implementation on different boards are shown below.

traverse motions

```
extern int nml_traj_set_traverse_rate(double rate);
extern int nml_traj_straight_traverse(NML_ROTPOSE &pos);
extern const int traj_wm_inpos();
```

The traverse APIs provide for fast motions within the machine work volume. The nml_traj_set_traverse_rate function specifies the speed for future traverses entered later into the queue. The rate parameter is defined as some percentage of the maximum axis speed. The nml_traj_straight_traverse function enters a command to perform a straight line motion into the queue. The NML_ROTPOSE type parameter consists of the position coordinates of the goal and the orientation coordinates of the goal rotational axes. The traj_wm_inpos function returns a nonzero value if the current trajectory is complete.

Both boards support traverse motions. The APIs perform the kinematics to transform the goal into encoder counts and transforms rate commands into speed commands in encoder counts.

feed motions

```
extern int nml_traj_set_feed_rate(double rate);
extern int nml_traj_set_feed_override(double override);
extern int nml_traj_set_origin(NML_ROTPOSE &origin);
extern int nml_traj_use_tool_length_offset(double length);
extern int nml_traj_straight_feed(NML_ROTPOSE &pos);
extern int nml_traj_arc_feed(double f, double s, double r, double aep, CANON_PLANE
plane);
```

The feed APIs provide for more precise control of machining operations. Similar to the traverse function, the feed rate can be specified for all subsequent traverses. The feed rate can also be overridden by the operator by using the nml_traj_set_feed_override function with an override percentage. Specifying the origin using the nml_traj_set_origin function allows a programmer to write a part program using coordinates relative to a reference frame attached to the part. Specifying a tool length offset using the nml_traj_use_tool_length_offset function offsets the trajectory along the tool axis. This enables a programmer to specify part cutting operations independent of the precise tool length.

Both boards have a similar straight line trajectory generator that is called using the nml_traj_straight_feed function. One board expects all goals in coordinates relative to the current position. This was accomplished by a simple transformation in the host portion of the API. One board expects a BEGIN command at the beginning of a sequence of motions and an END command at the end. The END is important because the trajectory generator causes an instant deceleration if the queue is empty without the END. New APIs were added to specify begin and end of sequences of motions.

Both boards support circle trajectories in planes that are parallel to the machine axes with slightly different parameters. The nml_traj_arc_feed function provides a general syntax for arc, circular and helical trajectories. The plane parameter specifies one of the three planes that are parallel to the machine axes, CANON_PLANE_XY, CANON_PLANE_YZ, CANON_PLANE_XZ. The f and s parameter are coordinates of the center of the arc. The coordinates follow the plane specification. For example, f and s are the X and Z coordinates if the plane argument is CANON_PLANE_XZ. The r parameter is the radius of the arc. The aep is the axis end point that defines the distance of translation out of the plane which is completed along with the arc. This parameter is used to specify helical motions. The helical motion, claimed to be supported by both boards, was not implemented in the first phase of API development.

5. Validation Tests and Results

The purpose of the validation tests is to determine to what degree interoperability has been achieved. The tests were conducted on two milling machines. The first machine is a three axis vertical tabletop mill, where the majority of in-house development and testing takes place. The second machine is a Kearney & Trecker 800 four-axis horizontal mill located at the General Motors Powertrain facility in Pontiac, Michigan. This mill is in daily use, and shop personnel provide valuable feedback based on real operations. Each mill is controlled by the EMC controller described above. The tests centered on determining the capabilities and limitations of the APIs, and to obtain results for future improvements. The validation test is divided into three parts.

5.1. Installation and set-up

To simplify the wiring of each motion control board to the machine tools, standard wiring connectors were built which allowed fast connector-based interchanging. This wiring- and signal-level specification was not part of the API effort, although it expedited the interoperability studies. A single axis of the interface is shown in Table 3.

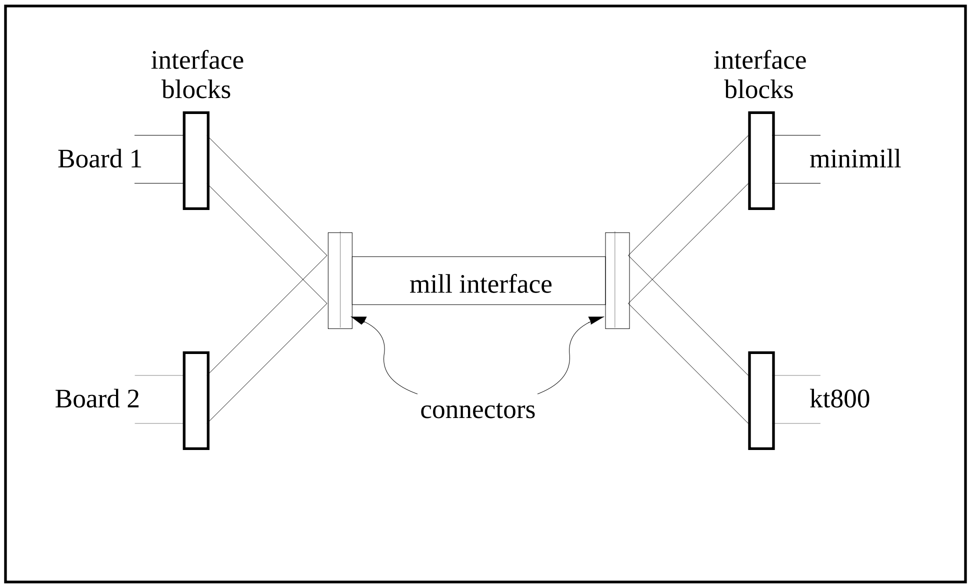

Figure 2. Physical interconnect scheme between motion control card and either mill.

Each mill provides an interface block that maps the mill wiring to the standard mill interface. This allows the controller to be connected to either mill simply by cable connection. On the controller side, each motion control board also provides an interface block that maps the standard mill interface to the motion control channels. This allows a motion control board to be easily exchanged.

X amplifier - ref
X amplifier +ref
X amplifier enable
X home switch
X +lim switch
X -lim switch
X encoder A+
X encoder B+
X encoder I+
X encoder A-
X encoder B-
X encoder I-
encoder gnd
encoder +5V

Table 3. Standard motor interface for a single axis.

The electronic interface between the motion controllers and the motors themselves were not the subject of the API effort. There exists established de facto standards (e.g., ±10 volt velocity inputs) or formal standards (e.g., the SERCOS digital drive interface specification) that apply.

The board configuration process differed substantially from both hardware and software perspectives between each board. No attempt was made to define standard bus interfaces or internal addressing schemes. In both cases, the vendor's configuration tools were used to set-up the boards before testing and API development.

5.2. Servo level APIs

The servo level APIs were initially tested using manual-mode procedures for calibrating (homing) each axis, measuring positioning accuracy, and performing single-axis velocity jogging and position moves. This procedure validates the servo-level APIs that are accessible to external processes, such as the operator interface.

5.3. Trajectory APIs

The trajectory APIs were tested using part programs written in RS-274. Various part programs were written to test specific motion APIs and to test how well the motions blend. A basic program commands nml_straight_feed motions in planes parallel to the axes. The program complexity is increased by adding motions requiring diagonal moves and finally circles. A single program that machines a circle, diamond and square on a single block tests all the basic trajectory APIs. The circle/diamond/square part was cut using the second board. It became evident that some modifications to the APIs and the task sequencer were needed.

The first modification arose from the need to control when commands are executed. Some commands should execute immediately, for example, overriding the current feed rate. Other commands are placed in the queue for execution after previously queued commands. This type of problem appeared when turning on the spindle which requires an open loop analog output from a channel. On the second board, the command was not queued for execution after the tool was moved to a desired position. The command was executed immediately, even before the axes started moving. There was no viable means to cause this to be executed later without deviating from the design to queue commands for blending. The task sequencer was modified to detect this type of transition and to insert a nml_stop_squence call. Deciding whether the motion controller requires a stop can be handled using a PAUSE_IO_TRANSITION flag which is set at initialization time based on parameters in an initialization file. In the future, the board vendor can remove the requirement by clearing the flag and the controller will be able to respond accordingly.

Another need for modification resulted when attempting to transition from an nml_straight_feed to an nml_arc_feed without a stop. The second board distinguishes between in-plane (two axis) and out-of-plane (three axis) straight line motions. To avoid using individual commands for two and three axis moves, it was decided that the nml_straight_feed would always use three axis motions. But the board requires a stop when transitioning between three axis straight lines and arcs. The task sequencer was modified to detect this type of transition and to insert a nml_stop_squence call that waits for the end of a sequence of queued commands and stops the machine. Forcing a stop should not be the standard mode of operation because the stop slows down the machining process and can add unwanted machining artifacts to the surface. For boards operating without this requirement, it can be handled in a manner that is similar to that described above by defining another initialization flag called PAUSE_ARC_LINES. In the future, the ability to handle all types of transitions would be desirable.

6. Conclusions and Issues

A basic level of interoperability for motion control boards was defined, implemented and successfully validated. The results indicate that standardizing on a procedure for installation and set-up with APIs for basic servo and trajectory level motion control is feasible. API interoperability was achieved by linking with the appropriate board specific motion control library and reading an initialization file with standard parameters that are set according to the vendor's requirements. This approach serves well. A more detailed model is needed for parameterizing a machine model. Once a complete model exists, a controller that adapts to the capabilities of all motion controllers that support the model can be built.

More advanced trajectory algorithms are needed. This raises the issue of how to incorporated their functionality. Because a direct link between a trajectory API and a command supported by the RS274 interpreter usually exists, it implies taking advantage of any new functionality requires modifying the part program. But, how much effort is required to add the new functionality? Adding a new function to the EMC controller requires modifying and recompiling the interpreter, the task sequencer and the motion control library. In practice, the motion control vendor would provide the API without the need for recompilation, just linking. That leaves the interpreter and the task sequencer as potentially requiring modification before adding new functionalities. One approach is to provide an RS274 instruction to pass a string, where the string contains the name and arguments of a vendor supplied API. Such an approach is now being explored. This may allow motion control vendors to provide advanced trajectory APIs without recompilation of control code. Any modifications would be limited to part programs, something that must take place in any case. In the long term, as these more advanced trajectory become widely used, they could become part of a higher level API set.

7. References

1. Proctor, F., Michaloski, J., Shackleford, W., Szabo, S., “Validation of Standard Interfaces for Machine Control,” Intelligent Automation and Soft Computing: Trends In Research, Development, and Applications, Vol 2, TSI Press, Albuquerque, NM, 1996.

2. “Methods for Performance Evaluation of Computer Numerically Controlled Machining Centers,” ANSI/ASME B5.54-1991, Version 1.0, 1991.

3. Electronic Industries Association, “Interchangeable Program Formats for Numerically Controlled Machines,” EIA-274-D, 1979.

4. Shackleford, W., and Proctor, F. M., “The Real-time Control System Library,” Internet Location: http://isd.cme.nist.gov/~shackle/rcslib/.

5. Martin Marietta, “Next Generation Controller (NGC) Specifications for an Open System Architecture Standard (SOSAS) Revision 2.0,” National Center for Manufacturing Sciences, August 1994.