OMAC API SET

Version 0.23

Working Document

OMAC API Work Group

October 12, 1999

TABLE OF CONTENTS

TABLE OF FIGURES

EXECUTIVE SUMMARY

Open modular architecture controller technology offers great potential for integration of process improvements and better satisfaction of application requirements. With an open architecture, controllers can be built from best value components from best in class services. The need for open-architecture controllers is high, but vendors are slow to respond. One reason for the delay in industry action is that no clear open-architecture solution has evolved. In an effort to promote open architecture control solutions, a workgroup within the Open Modular Architecture Controller (OMAC) users group is working on defining an OMAC Application Programming Interface (API). The goal of the OMAC API workgroup is to specify standard APIs for a set of open architecture controller components. This document contains background information, design methodology and actual API definitions.
As background, the following material will be presented:

- OMAC API definition of open architecture
- advantages and impediments to open architectures
- overview of the OMAC API reference model.

At a high level of conceptual design, the OMAC API reference model will be presented and includes the following items:

- OMAC API core modules
- application framework
- application design and examples.

The OMAC API reference model does not specify a reference architecture. Instead, modules can be freely connected. In lieu of a reference architecture, the document includes several reference examples.
At a detailed level of design, the OMAC API specification methodology will be presented and subscribes to the following principles:

- API programming abstraction is used
- Object Oriented techniques for encapsulation, inheritance, specialization and object interaction are applied
- Client/Server is the communication model
- Proxy Agents provide transparency of distributed communication
- Finite State Machine (FSM) is the behavior model
- Finite State Machine (FSM) are passed as data to then provide control
- Reusability of software components is achieved through foundation classes
- System objects are mirrored in human machine interface
- No specification of an infrastructure is attempted instead a commitment to a PLATFORM + OPERATING SYSTEM + COMPILER + LOADER + INFRASTRUCTURE SUITE is necessary for it to be possible to swap modules.

  .

1. BACKGROUND

Most Computer Numerical Control (CNC) motion and discrete control applications incur high cross-vendor integration costs and vendor-specific training. On the other hand, in a modular, standards-based, open-architecture controller modules can be added, replaced, reconfigured, or extended based on the functionality and performance required. Modifications to a module should provide equivalent or better functionality as well as offer different performance levels. Ideally, the module interfaces should be vendor-neutral, plug-compatible and platform independent.
However, it is important to note that openness alone does not achieve plug-and-play. One vendor's idea of openness need not be the same as another vendor's. Openness is but one step towards plug-and-play. In reality, plug-and-play openness is dependent on a standard. This leads to the following definition of an open architecture controller:

An open architecture control system is defined and qualified by its ability to satisfy the following requirements:

Open provides ability to piece together systems from components, provides ability to modify the way a controller performs certain actions, and provides ability to start small and upgrade as a system grows.

Modular refers to the ability of controls users and system integrators to purchase and replace controller modules without unduly affecting the rest of the controller, or requiring extended integration engineering effort.

Extensible refers to the ability of sophisticated users and third parties to incrementally add functionality to a module without completely replacing it.

Portable refers to the ease with which a module can run on different platforms.

Scalable allows different performance levels and size based on the platform selection. Scalability means that a controller may be implemented as easily and efficiently by systems integrators on a stand-alone PC, or as a distributed multi-processor system to meet specific application needs.

Maintainable supports robust plant floor operation (maximum uptime), expeditious repair (minimal downtime), and easy maintenance (extensive support from controller suppliers, small spare part inventory, integrated self-diagnostic and help functions.)

Economical allows the controller of manufacturing equipment and systems to achieve low life cycle cost.

Standard Interfaces allow the integration of off-the-shelf hardware and software components and a standard computing environment to build a controller. Standard interfaces are vital to plug-and-play.

Degree of openness can be evaluated by comparing a claim of openness against the above requirements. Herein, the concept of an open-architecture control system that supports openness, and the auxiliary requirements will be identified as **"open, openness or open architecture."**

1.1 ADVANTAGES OF OPEN ARCHITECTURE TECHNOLOGY

Based on specific instances of problems encountered by users of proprietary controllers, the following list of open-architecture requirements was generated. An open architecture should be able to do the following:

- provide a migration path from existing practices;
- allow an integrator/end user to add, replace, and reconfigure modules;

- provide the ability to modify spindle speed and feed rate according to some user-defined process control strategy;
- allow access to the real-time data at a predictable rate up to the servo loop rate;
- allow full 3-D spatial error correction using a user-defined correction strategy;
- decouple user interface software and control software and make control data available for presentation;
- provide capability to integrate controller with other intelligent devices;
- increase the ability for 3rd party software enhancements. Examples of 3rd party enhancements include:
  - replace a PID control law with a more sophisticated Fuzzy Logic control law
  - collect servo response data with a 3rd party tool, and set tuning parameters in the appropriate control law
  - add a force sensor, and modify the feed rate according to a user defined process model
  - perform high resolution straightness correction on any axis
  - replace the user interface with a 3rd party user interface that emulates a user interface familiar to your machine operators.

The initial validation strategy for the OMAC API would be to insure that this list of capabilities can be addressed.

1.2 IMPEDIMENTS TO OPEN ARCHITECTURE TECHNOLOGY

It is difficult to define a controller specification that is safe, cost-effective, and supports real-time performance.

A specification cannot be an island of technology. To be successful, a specification must satisfy legacy needs, factor in current practices, as well as anticipate evolving technologies. Attaining an open architecture specification that is flexible and isn't biased toward legacy or emerging technology can be hard.

Of great importance within the controls domain is the requirement for guaranteed, hard-real-time performance. Without this, safety is at risk. Safety is a major concern voiced within the controller industry that is especially concerned with the issues of liability and allocation of responsibility within an open architecture paradigm. Industry would have to adopt new practices for open architecture controllers. A greater responsibility would be placed on the integrator. Conformance testing would play a larger role. Conformance could require regression and boot-up testing and verification procedures to guarantee proper operation.

A further hindrance is the fact that modules are not "self-contained." Defining an infrastructure within which the modules can operate is necessary and quite difficult. An **infrastructure** is defined as the services that tie the modules together and allow modules to use platform services. The infrastructure is intended to hide specific hardware and platform dependence; however, this is often difficult to achieve.

Containing the scope of the specification is also difficult. Openness goes beyond run-time APIs. There can be "other" APIs, including configuration, integration, and initialization. As an example, consider the simple use of a math library API. Even there, specification of the math library implementation must be done to select either a floating point processor or software emulation.

Finally, group and industry dynamics can be a problem. From a workgroup perspective, getting people to agree can be a challenge because there are difficult trade-offs in modularization, scope, life cycle benefits, costs, time to market, and complexity. It is recognized that industry will find it difficult to adopt the OMAC paradigm, due to entrenchment in the legacy of prior implementations, the “comfort zone” of past practice and culture, the investment hurdle to effect change, and the shortage of skilled resources. Proper acculturation, training and education of people and an orderly introduction, demonstration, deployment, and scale-up will be needed to realize the potential benefits. From an industry perspective, many companies do not perceive any direct benefit from an open architecture. Overcoming the workgroup inertia and industry skepticism by promoting and demonstrating the benefits of open architecture remains a fundamental key to open architecture acceptance.

2 REFERENCE MODEL

The OMAC API requirements were derived from the OMAC or “Open Modular Architecture Controller” requirements document [OMA94]. The OMAC document describes the problem with the current state of controller technology and prescribes open modular architectures as a solution to these problems. OMAC defines an open architecture environment to include Platform, Infrastructure, and Modules.
In the interest of flexibility, scalability, and reusability, OMAC API does not specify a fixed architecture. Instead, OMAC API assumes a reference model described by this abstraction hierarchy:

- Foundation Classes
- Modules
- Architectural Design
- Detailed Design Framework

The **Foundation Classes** are derived from decomposing a generic controller into classes. These classes define the controller class hierarchy. Foundation classes are then grouped into **Modules** that become plug-and-play components. A controller is generated by selecting from different implementations of OMAC Modules containing **object** implementations of the foundation classes. A system design is divided into two phases. The first phase is **Architectural Design** and deals with system decomposition into OMAC Modules. The second phase is called **Detailed Design** and is responsible for detailing individual object API, that is, the object attributes and methods. In this case, the design uses the OMAC API or extends the API to suit the application.

2.1 FOUNDATION CLASSES

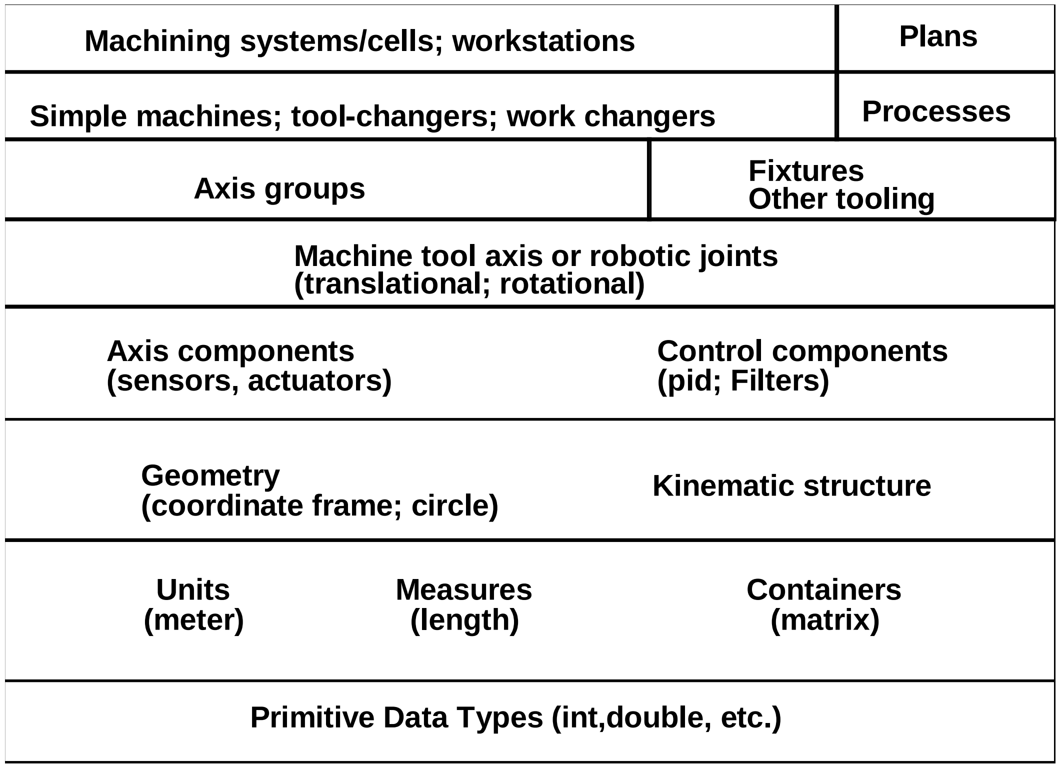

Machining systems/cells; workstations
Plans
Simple machines; tool-changers; work changers
Processes
Axis groups
Fixtures
Other tooling
Machine tool axis or robotic joints
(translational; rotational)
Axis components
(sensors, actuators)
Control components
(pid; Filters)
Geometry
(coordinate frame; circle)
Kinematic structure
Units
(meter)
Measures
(length)
Containers
(matrix)
Primitive Data Types (int,double, etc.)

Figure 1: Controller Class Hierarchy

The decomposition of a generic controller into classes spans many levels of abstraction and has elements for motion control and discrete logic necessary to coordinate and sequence operations. Figure 1 portrays the class hierarchy derived from a controller decomposition. At the lower levels, the Foundation Classes are the building blocks that may be found in multiple modules. For example, the class definition of a Geometry "position" would be found in most modules. Moving up the hierarchy, the Foundation Classes broaden their scope to define device abstractions for such motion components as sensors, actuators, and PID control laws. As the scope broadens however, not all software objects have physical equivalents. Objects such as axis groups are only logical entities. Axis groups hold the knowledge about the axes whose motion is to be coordinated and how that coordination is to be performed. Services of the appropriate axis group are invoked by user-supplied plans.

Within Foundation classes, OMAC API define base classes and add to the base classes using the Object Oriented concept of inheritance to define derived classes. OMAC API also uses inheritance to maintain levels of complexity. Level 1 constitutes base functionality seen in current practice. Level 2 constitutes functionality expected of advanced practices. Higher levels constitute advanced capability seen in emerging technology, but unnecessary for simple applications.

2.2 MODULES

OMAC API defines a **module** to have the following characteristics:

- significant piece of software used in composing controller
- grouping of similar classes
- well-defined API
- well-defined states and state transitions

- replaceable by any piece of software that implements the API, states, and state transitions.

Using the OMAC Specification [OMA94] as a baseline, Figure 2 diagrams the OMAC API Modules including a brief description of a module's general functional requirements. The Modules have the following general responsibilities:

Axis modules are responsible for servo control of axis motion, transforming incoming motion setpoints into setpoints for the corresponding actuators.

Axis Group modules are responsible for coordinating the motions of individual axes, transforming an incoming motion segment specification into a sequence of equi-time-spaced setpoints for the coordinated axes.

OMAC Base Class provides a uniform API base class for an OMAC module. The OMAC base class defines a state model and methods for start-up and shutdown. The OMAC Base Class defines a uniform name and type declaration and provides an error-logging interface. The OMAC Base Class maintains a global directory service for name lookup and reference binding.

Capability is an object to which the Task Coordinator delegates for specific modes of operation. Capability corresponds to the traditional CNC modes (AUTO, MANUAL, MDI, etc.) At the Capability Level, there is no coordination between Capabilities. A Capability is a Control Plan Unit (see Control Plan module) with the distinction being that a Capability is Control Plan Unit associated with a Task Coordinator module.

Control Law components are responsible for servo control loop calculations to reach specified setpoints.

Control Plan consists of a series of related **Control Plan Units (CPU)** and forms the basis of control and data flow within the system. A Control Plan Unit is a base class that contains finite state logic. A **Motion Segment** is a derived class of Control Plan Unit for motion control. **Discrete Logic Unit** is a derived class of Control Plan Unit for discrete logic control. **Capability** is a derived class of Control Plan Unit used within a Task Coordinator and because it is such a significant piece of software, it is also considered an OMAC API module.

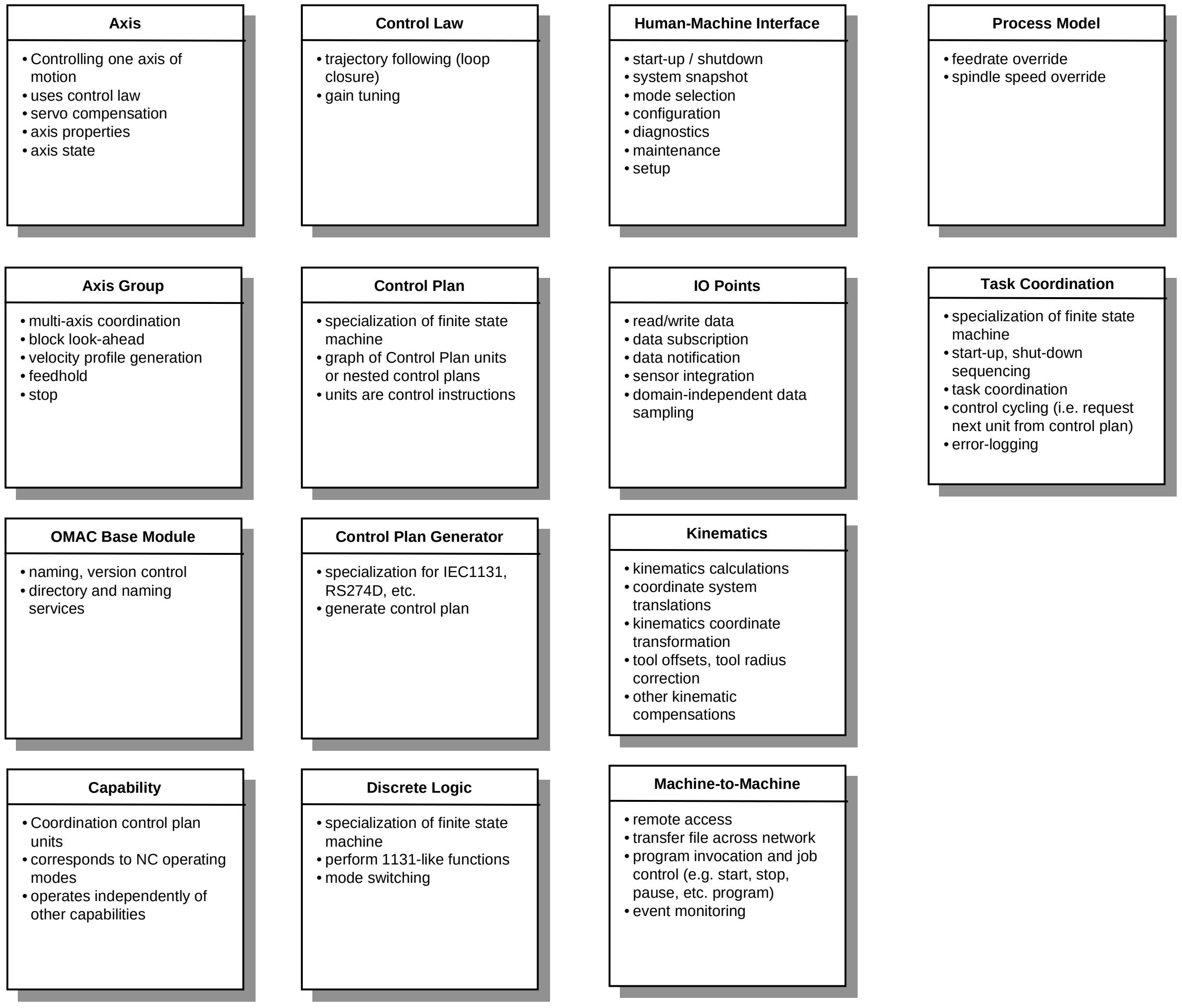

Axis
• Controlling one axis of motion
• uses control law
• servo compensation
• axis properties
• axis state
Control Law
• trajectory following (loop closure)
• gain tuning
Human-Machine Interface
• start-up / shutdown
• system snapshot
• mode selection
• configuration
• diagnostics
• maintenance
• setup
Process Model
• feedrate override
• spindle speed override
Axis Group
• multi-axis coordination
• block look-ahead
• velocity profile generation
• feedhold
• stop
Control Plan
• specialization of finite state machine
• graph of Control Plan units or nested control plans
• units are control instructions
IO Points
• read/write data
• data subscription
• data notification
• sensor integration
• domain-independent data sampling
Task Coordination
• specialization of finite state machine
• start-up, shut-down sequencing
• task coordination
• control cycling (i.e. request next unit from control plan)
• error-logging
OMAC Base Module
• naming, version control
• directory and naming services
Control Plan Generator
• specialization for IEC1131, RS274D, etc.
• generate control plan
Kinematics
• kinematics calculations
• coordinate system translations
• kinematics coordinate transformation
• tool offsets, tool radius correction
• other kinematic compensations
Capability
• Coordination control plan units
• corresponds to NC operating modes
• operates independently of other capabilities
Discrete Logic
• specialization of finite state machine
• perform 1131-like functions
• mode switching
Machine-to-Machine
• remote access
• transfer file across network
• program invocation and job control (e.g. start, stop, pause, etc. program)
• event monitoring

Figure 2: OMAC Modules

Control Plan Generator modules are responsible for translating application programs into Control Plans. As examples, programs written in the RS274D [RS274] and IEC 1131-3 [IEC93] languages can be translated into Control Plans.

Discrete Logic modules are responsible for implementing discrete control logic or rules that can be characterized by a Boolean function from input and internal state variables to output and internal state variables. More than one discrete logic module is permitted, but not necessary. Multiple discrete logic modules is similar to having many PLC's networked together within the same computing platform.

Human Machine Interface (or HMI) modules are responsible for human interaction with a controller including presenting data, handling commands, and monitoring events. Defining a presentation style (e.g., GUI look and feel, or pendant keyboard) is not part of OMAC API effort.

I/O Points are responsible for the reading of input devices and writing of output devices through a generic read/write interface. The goal is to provide an abstraction for the device driver. Logically related IO may be clustered within a Discrete Logic module.

Kinematics Models modules are responsible for geometrical properties of motion. Computing forward and inverse kinematics, mapping and translating between different coordinate systems, applying geometric correction and tool offsets, and resolving redundant kinematic solutions are examples of kinematic model functionality.

Machine-to-Machine modules are responsible for connecting and communicating to controllers across different domains (address spaces). An example of this functionality is the communication from a Shop Floor controller to an individual machine controller on the floor.

Process Model is a module that contains dynamic data models to be integrated with the control system. Process control modules (not detailed by this specification) produce adjustments or corrections to nominal rates and path geometry. Feedrate override and thermal compensation are examples of process model functionality. The process model is crucial to the concept of extensible open systems.

Task Coordinator modules are responsible for sequencing operations and coordinating the various motion, sensing, and event-driven control processes. The task coordinator can be considered the highest level Finite State Machine in the controller.

Some clarifying observations about modules include:

- Interchangeable modules may differ in their performance levels.
- Modules may provide more functionality (added value) than required in the specification. **Specialization** of a module interfaces is the mechanism to achieve additional functionality.
- A controller may have more than one instance of a module.
- Modules can be explicitly control-related (e.g., Axis) or be inheritance-related encapsulating common functionality (e.g., OMAC Base Class.)
- Modules do not need to run as separate threads (or intelligent agents.) Systems can be built from a single thread of execution.
- Modules can contain multiple threads of execution.
- Modules may be used to build other components. For example, a discrete mechanism, such as a tool changer component, can be built using OMAC modules.
- Multiple instances of a module are required to handle different configurations. For example, assume a system with 3 axes `x, y, z` and a `spindle`. Three Axis Group objects would be created at configuration time, `ag1, ag2, ag3`, with the following configuration:

```
ag1: x, y, z
ag2: spindle
ag3: x, y, z, spindle
```

For most machining where the motion control and the spindle are loosely related, references to `ag1` and `ag2` would be used. However to do a Rigid Tap requiring tight synchronization of the spindle and motion, a reference to `ag3` would be used.

2.3 ARCHITECTURAL DESIGN

Since there is no explicit OMAC reference architecture, composing a system architecture from OMAC modules is left to the developer. This offers much flexibility, but without guidance, can be confusing. This section will give some application architecture examples for clarification. This section starts with a simple application and then develops a series of examples to illustrate the stages of development one might encounter when building an application architecture. The examples highlight the static relationship between OMAC modules (as opposed to the data flow.) However, an underlying assumption is directives flow from top to bottom.

2.3.1 OPERATOR CONTROL OF A SET OF IO POINTS EXAMPLE

The simplest case is an operator controlling several IO points. The OMAC API model allows the connection of a Human Machine Interface (HMI) object to several IO points. Figure 3 shows the simple connection between HMI and IO points. Within the diagram, an arrow indicates a **reference** from one object to another.

The rationale for such a simple example is to show that the OMAC API is not monolithic, and a small system together can be put together. With this ability, OMAC systems can start small and be pieced together.

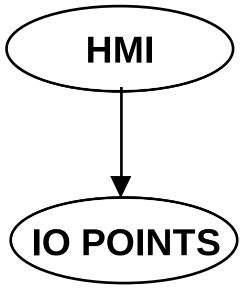


Figure 3: Operator Control of a Set of IO Points

2.3.2 ONE AXIS BOOTSTRAP

After establishing an HMI and IO connection, the natural progression in building a CNC machine tool controller is to add an axis of motion under manual control. This scenario is typical in offline assembly and testing of an axis that may eventually be assembled in a multi-axis CNC machine tool. Jogging and Homing are the primary functionality used. At this point, there is no coordination with any other motion, mechanism, or state in the NC machine tool. During this stage of the assembly of a machine tool, it is also helpful to perform the calibration, tuning, or health monitoring tests.

The Axis Module coordinates IO points. Assume that the IO points will consist of a PWM motor drive, an amplifier enable control, an amplifier fault status signal, an A-QUAD-B encoder with marker pulse and switches for home and axis limits. Figure 4 shows a one-axis system that uses two Control Laws, one for PID control of Position, and another to do PID control of velocity. The Axis will output accelerations to the actuator and read encoder values through IO points referenced by the Axis module. For operator control of the axis, an HMI module mirrors exists for the Axis module as well as mirrors for each Control Law module. The mirrors provide a snapshot of control system objects and use proxy agents for communication.

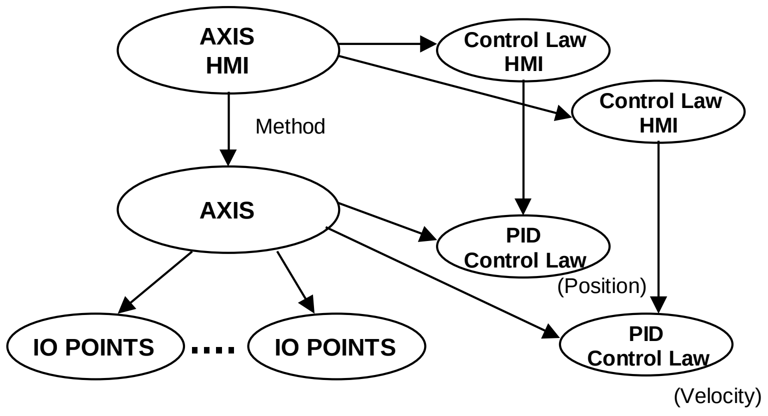


Figure 4: Simple, Single Axis, Jog/Home Only System

2.3.3 PROGRAMMABLE LOGIC EXAMPLE

Consider a case of work-handling equipment that provides peripheral functions for a CNC machine tool. The equipment includes two hydraulically actuated, two-position on-off mechanisms, named, Loader and Unloader. Let their sensing, actuation, and control be under a Discrete Logic module, named `LUNL` whose sequence of operations was originally specified in some manner conforming to IEC 1131-3, and subsequently translated into a Control Plan Unit, named `CPlunl`.

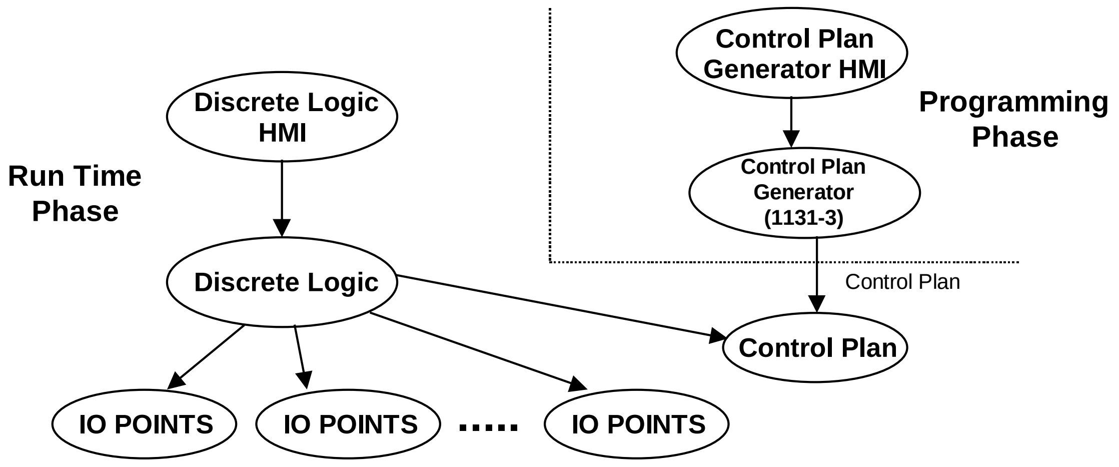


Figure 5: Loader/Unloader Discrete Logic Control

Figure 5 illustrates the relationship of different OMAC modules within this LUNL application. Within the block diagram, two phases, Programming Phase and Run Time Phase, are shown. However, other phases are to be considered including a Configuration Phase and an Initialization Phase. The following steps sketch the different phases of system development.

I. In the Programming phase,

a. Develop IEC 1131-3 code that performs logical mapping of IO functionality

b. Generate a number of Control Plan Units (CPU), possibly one associated with each state.

c. Group Control Plan Units to become a LUNL Control Plan (i.e., `CPlunl`)

II. At configuration phase,

a. Perform physical mapping of IO functionality

b. Load Control Plan into the Discrete Logic Module

III. At initialization phase,

a. Resolve external object and module references

b. Register events

IV. At runtime phase,

a. Clients (e.g., HMI or IO Points) generate events

b. The LUNL Discrete Logic Module executes each ControlPlanUnit at an assigned scan rate. A ControlPlanUnit executes as a Finite State Machine (FSM).

2.3.4 DRILLING MOTION CONTROL EXAMPLE

An example describing programmed NC for one-axis drilling will be developed. A typical one-axis drilling workstation would perform holeworking operations, e.g., drilling with a spindle drill-head, boring a precision bore, counter-boring the bored hole, or probing the (axial) location of the counterbored shoulder.

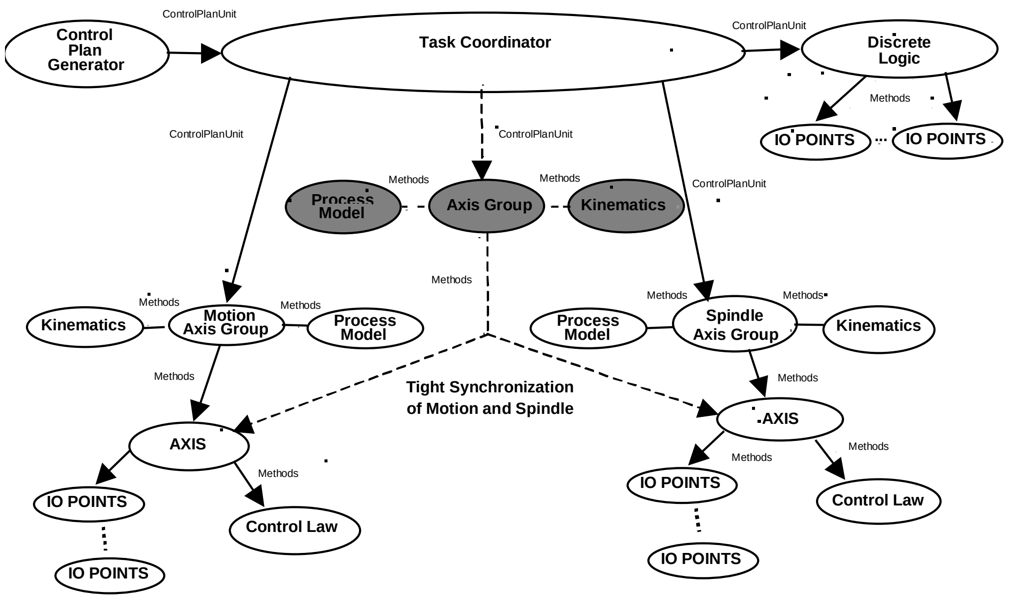


Figure 6: Drilling Example

Figure 6 illustrates the module and component relationships for a drilling application. Z motion requires an Axis module for servoing and an AxisGroup module for Cartesian motion. Spindle control requires another Axis module to interface to drive components assumed to provide a facility for setting spindle speed and direction and to start and stop spindle rotation. The Spindle requires an Axis Group for rate and override control. A third Axis Group is necessary for synchronized control of both the Motion Axis and the Spindle Axis (shown as shaded with dashed line connections). Generally, the Spindle Axis will not need a Control Law, however, when it is synchronized with motion it will require servoed control.

In the diagram, a Task Coordinator exists to provide program control. A ControlPlanGenerator module translates a part program into ControlPlanUnits. The primary command communication between modules is reflected in the diagrams by showing the keyword `Method` or `ControlPlanUnits` (which uses a method to pass it) next to an arrow. A Discrete Logic Module, typical of the previous example, exists as an equivalent for part loading and unloading, as well as machine state (e.g., temperature, estop). To improve predictability and reduce variation, a Process Model module will exist to integrate sensing and control to prevent tool breakage by monitoring spindle torques and thrust forces. A simple Kinematics module exists to model the workspace and handle different tool offsets and part placements.

2.4 DETAIL DESIGN FRAMEWORK

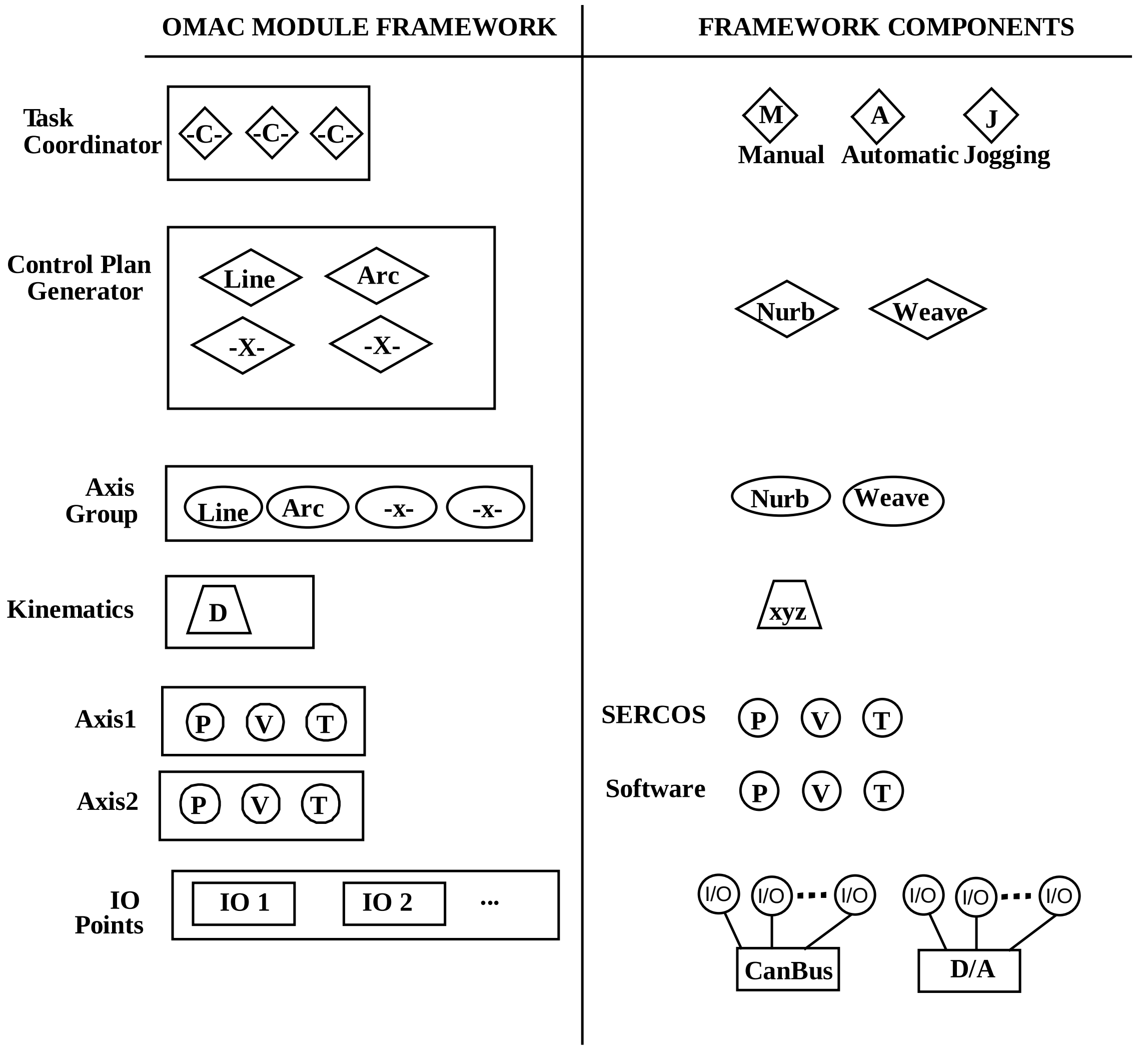


Figure 7: Design Framework

The **Detailed Design** is responsible for detailing individual object API, that is, the object attributes and methods. At this phase, one determines which objects are available, the extent of object capabilities, and whether the objects need to be bought or built. This phase corresponds to putting a system together with the OMAC API **Framework**. Frameworks are object-oriented technology consisting of sets of prefabricated software and building blocks that are extensible and can be integrated to execute well-defined sets of computing behavior. Frameworks are not simply collections of classes. Rather, frameworks come with rich functionality and strong "pre-wired" interconnections between the object classes.

This contrasts with the procedural approach where there is difficulty extending and specializing functionality; difficulty in factoring out common functionality; difficulty in reusing functionality that results in duplication of effort; and difficulty in maintaining the non-encapsulated functionality.

With frameworks, application developers do not have to start over each time. Instead, frameworks are built from a collection of objects, so both the design and the code of a framework may be reused.

In the OMAC API Framework the prefabricated building blocks are the implementations of 1) OMAC modules and 2) framework components (e.g., ControlPlanUnits). As a simple example, Figure 7 illustrates a Detailed Design for assembling a controller application. An application developer buys modules and components as commercial off-the-shelf (COTS) technology. Then, the application developer configures the modules and "puts the pieces together" by linking the purchased COTS ".o" object files.

Modules are configured based on their references to other objects. For the Axis modules in the example, references are needed for position (P), velocity (V) or torque (T) Control Law modules. These references could be to objects in software, hardware or some combination of hardware and software. For software P control, a Control Law object from the Software set is selected. For hardware P control, a Control Law object from the SERCOS[IEC95] set is selected. The applications developer is also responsible for mapping the logical IO points onto physical devices (e.g., D/A or CanBus).

Modules are also configured based on the selection of Control Plan Units (CPU) that define module responsibilities. Within the example, there is a Task Coordinator module that has containers for inserting Capability CPU (in the figure represented by a -C- framed by a diamond). The Capabilities include Manual, Automatic or Jogging. The application developer is free to put one or more of these Capabilities into the Task Coordinator or develop a unique Capability. For Control Plan Generator and Axis Group, the application developer is already provided Line and Arc CPU but can plug in NURB or Weave CPU.

Using the OMAC API Framework, application development involves three groups:

Users define the behavior requirements and the available resources. Resources include such items as hardware, control and manufacturing devices, and computing platforms. For behavior, the user defines the performance and functionality expected of the controller. Performance includes such characteristics as speed or accuracy. Functionality defines the controller capability such as the ability to handle planar part features versus complex part features.

System Integrators select modules and framework components to match the application performance and functional requirements. The system integrator configures the modules to match the application specification. The system integrator uses an integration architecture to connect modules and verify system operation. The system integrator also checks compliance of modules to validate the user-specification of performance and timing requirements.

Control Component Vendors provide module and framework component products and support. For control vendors to conform to an open architecture specification, they would be required to conform to several specifications including the following:

- customer specifications
- module class specification
- system service specification

The system service describes the platform and infrastructure support (such as communication mechanisms) and the resources (disks, extra memory, among others) available. Computer boards have a device profile that includes CPU type, CPU characteristics and the CPU performance characteristics. Included within the profile is the operating system support for the CPU. A spec sheet or computing profile [SOS94] is required to describe the system service specification that would include such areas as platform capability, control devices, and support software.

3 SPECIFICATION METHODOLOGY

The primary goal of the OMAC API workgroup is to define standard API for the Modules. This section will refine the concept of "API" and describe the OMAC API specification methodology. The API specification methodology applies the following principles:

- Stay at API level of specification. Use IDL or MIDL to define interfaces.
  - Use Object Oriented technology.
  - Use general Client Server communication model, but use state-graph to model state behavior.
  - Use Proxy Agents to hide distributed communication.
  - Do not specify an infrastructure.
  - Finite State Machine (FSM) is model for data and control.
  - Mirror system objects in human machine interface.

The following sections will discuss these principles.

3.1 API SPECIFICATION

API stands for Application Programming Interface, and refers to the programming front-end to a conceptual black box. The API consists of a list of signatures per black box. A **signature** specifies the front-end with a function name, calling sequence, and return parameter. For example, "`double cos(x)`" specifies a cosine signature. The API is concerned with the signature, not the implementation. For the cosine, implementation could be it table-lookup or Taylor series. However, the API does specify performance, which in turn, affects the implementation. For the cosine API, performance may dictate speed over accuracy so that computing a cosine should be as fast and not necessarily as accurate as possible.

A **standard** API is helpful because programming complexity is reduced when one alternative exists as opposed to several. For example, the cosine signature is generally accepted as `cos(x)`, not `cosine(x)`. This is a small but significant standardization. At a programmatic level, the importance of a standard API can be seen within the Next Generation Inspection Project (NGIS) at NIST[NGI]. The NGIS project has integrated three commercial sensors and one generic sensor into the Coordinate Measuring Machine controller. Each sensor had a different "front-end" - one had a Dynamically Linked Library (.DLL) interface, one had a memory mapped interface, one had a combination port and memory mapping. None of the sensors had the same API. Yet, all of the sensors were "open."

APIs can be defined in any number of programming languages. This creates a problem when defining a standard API since the controller industry uses a variety of languages and platforms. OMAC API chose IDL, (Interface Definition Language) [COR91] or MIDL (Microsoft IDL) [MIDL] , as its specification language since it solves this problem. IDL is a technology-independent syntax for describing interfaces. In IDL, interfaces have attributes (data) and operation signatures (methods). IDL supports most object-oriented concepts including inheritance. IDL translates to object-oriented (such as C++ and JAVA) as well as non-object-oriented languages (such as C). IDL specifications are compiled into header files and stub programs for direct use by application developers. The mapping from IDL to any programming language could potentially be supported, with mappings to C, C++, and JAVA available.

To clarify the problem of unifying the specification, consider the mapping of the OMAC API IDL onto three different validation testbeds. Figure 8 illustrates mapping IDL to the different implementation strategies. For ICON, the standard API in IDL has to be mapped into JAVA. At the University of Michigan, they are using the ROSE CASE tool to design their controller. ROSE accepts C++ header through a reverse engineering process. At the NIST testbed, the IDL will be translated into C++

headers and use the Enhanced Machine Controller and its infrastructure[PM93]. For these three implementations, only the IDL specification can be mapped into all the languages needed to support the applications.

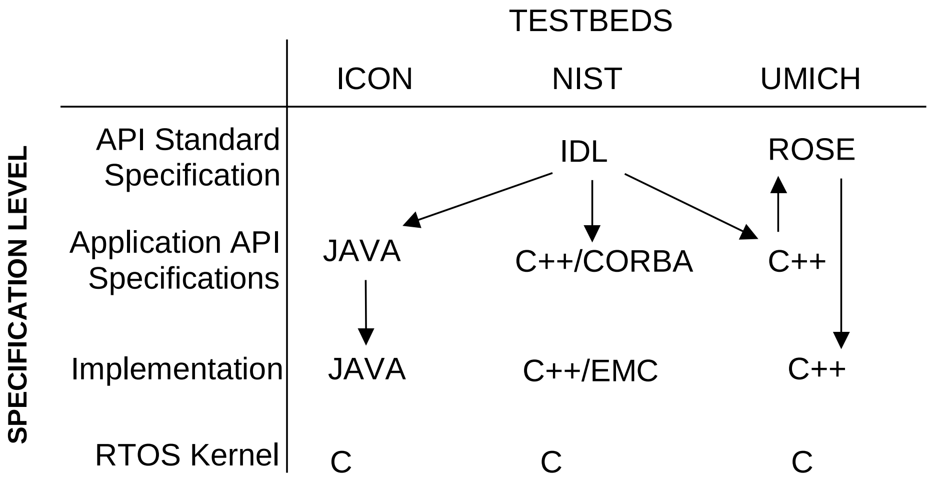


Figure 8: Specification Language Mapping

3.2 OBJECT ORIENTED TECHNOLOGY

OMAC API uses an object-oriented (OO) approach to specify the modules' API with class definitions. The following terms will define key object-oriented concepts. A **class** is defined as an abstract description of the data and behavior of a collection of similar objects. Classes **aggregate** data and methods. Class definitions offer **encapsulation** hiding details of a classes implementation. An **object** is defined as an instantiation of a class. For example, `SERCOS-Driven` Axis describes an instance of an Axis class in the running machine controller. A three-axis mill would have three instantiations of that class - the three objects implementing that class. An **object-oriented program** is considered a collection of objects interacting through a set of published APIs. A by-product of the object-oriented approach is **data abstraction**, which is an effective technique for extending a type to meet programmer needs.

3.2.1 INHERITANCE

Inheritance is useful for developing data abstraction. OO classes can inherit the data and methods of another class through class derivation. The original class is known as the **base** or **supertype class** and the class derivation is known as a **derived** or **subtype class**. The derived class can add to or customize the features of the class to produce either a **specialization** or an **augmentation** of the base class type, or simply to reuse the implementation of the base class. To achieve a object-oriented framework strategy[Le95], all OMAC API class signatures (methods) are considered "virtual functions." Virtual functions allow derived classes to redefine an inherited base class method.
To illustrate inheritance, consider the case of a simplified Axis module acting as a server. Assume that the Axis API only allows the functionality to set a variable x. The following sketches a base and a derived Axis class definition.

```
class Axis
{
  virtual void setX(float x);
private:
  double myx;
}

application()
{
  Axis ax1;
  ax1.setX(10.0);
}
```

To extend the base server class, a class **myAxis** is derived to add an offset to its X value before each set. This could also be achieved on the server side if so desired.

```
class myAxis : public Axis
{
        virtual void setX(float x){ x= x + offset; Axis::setX(x); }
  private:
        double myx;
        double offset; // set elsewhere for offset calculation
}

application()
{
        Axis ax1;
        myAxis ax2;
        double val=1.0;
        double offset =10.0;

        ax1.setX(val+offset); // explicit offset in application code
        ax2.setX(val);        // offset hidden by configuration
}
```

3.2.2 SPECIALIZATION

OMAC API leverages the OO concept of inheritance to attain **specialization**. Specialization is useful for managing the scope of an API. For example, when defining a control law, many options exist including PID, Fuzzy Logic, Neural Nets, and Nonlinear. This proliferation of options begs for a compartmental approach. The OMAC API approach is to define a base class (generally corresponding to one of the OMAC Modules) and for each option derive a specialized class.
Specialization has many benefits. It helps manage the scope of capabilities, which reduces complexity. It allows differing terminology based on need (e.g., weights versus gains). Specialization provides a technique to handle evolving technology by allowing new derived class to be defined when necessary. To expedite the OMAC API effort, only options considered most important have been derived.

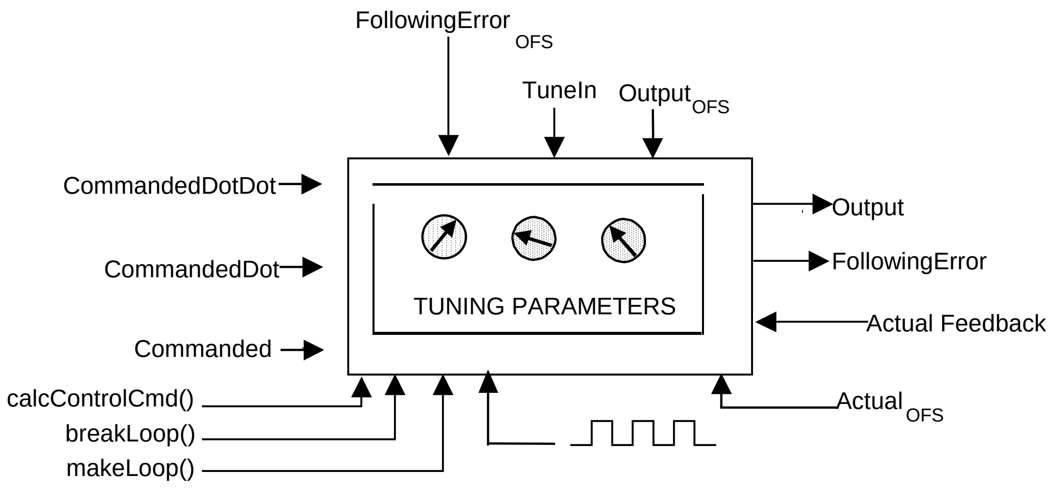


Figure 9: General Control Law

The control law module will be used to illustrate specialization. The responsibility of the Control Law module is conceptually simple - use closed loop control to cause a measured feedback variable to track a commanded setpoint value using an actuator. Figure 9 illustrates the definition of a base control law class. The concept of tuning is encapsulated within the black box and is conceptually controlled via "knob turning." The concept of accepting third party signal injection is handled by the inclusion of pre-and post-offsets (e.g., `FollowingError`). These offsets allow sensors or other process-related functionality to "tap" and dynamically modify behavior by applying some coordinate space transformation. The IDL definition of the illustrated control law module follows. The IDL keyword `interface` signifies the start of a new interface, corresponding to a C++ class.

```
interface IControlLaw
{  // Parameters
  void setCommanded(double setpoint);
  double getCommanded();

  void setCommandedDot(double setpointdot);
  double getCommandedDot();

  void setCommandedDotDot(double setpointdotdot);
  double getCommandedDotDot();

  void setOutput(double value);
  double getOutput();

  void setFeedback(double actual);
  double getFeedback();

  void setFollowingError(double epsilon);
  double getFollowingError();

  // Offsets
  void setFollowingErrorOffset(double preoffset);
  double getFollowingErrorOffset();

  void setOutputOffset(double postoffset);
  double getOutputOffset();

  void setFeedbackOffset(double postoffset);
  double getFeedbackOffset();

  void setTuneIn(double value); // enable with breakLoop
  double getTuneIn();

  void breakLoop();
  void makeLoop();
  void calcControlCommand();

};
```

Each `ControlLaw` specialization is a subtype whereby each subtype inherits the definition of the supertype. By applying this concept, an evolutionary process evolves to adapt to changes in the technology. At first, only highly-demanded subtypes, such as PID, were handled. Figure 10 conceptually illustrates the PID specialization of the control law. The IDL definition of the PID control law follows.

```
interface IPIDTuning: IControlLaw
{ // Attributes
  double getKp();
  double getKi();
  double getKd();

  void setKp(double val);
  void setKi(double val);
  void setKd(double val);

   double getKcommanded();
   double getKcommandedDot();
   double getKcommandedDotDot();
   double getKfeedback();
```

```
          void setKcommanded(double val);
          void setKcommandedDot(double val);
          void setKcommandedDotDot(double val);
          void setKfeedback(double val);
      };
```

OMAC API also uses inheritance to maintain levels of complexity. Level 0 would constitute base functionality seen in current practice. Level 2 would constitute functionality expected of advanced practices. Level 3, 4,..., n would constitute advanced capability seen in emerging technology, but unnecessary for simple applications.

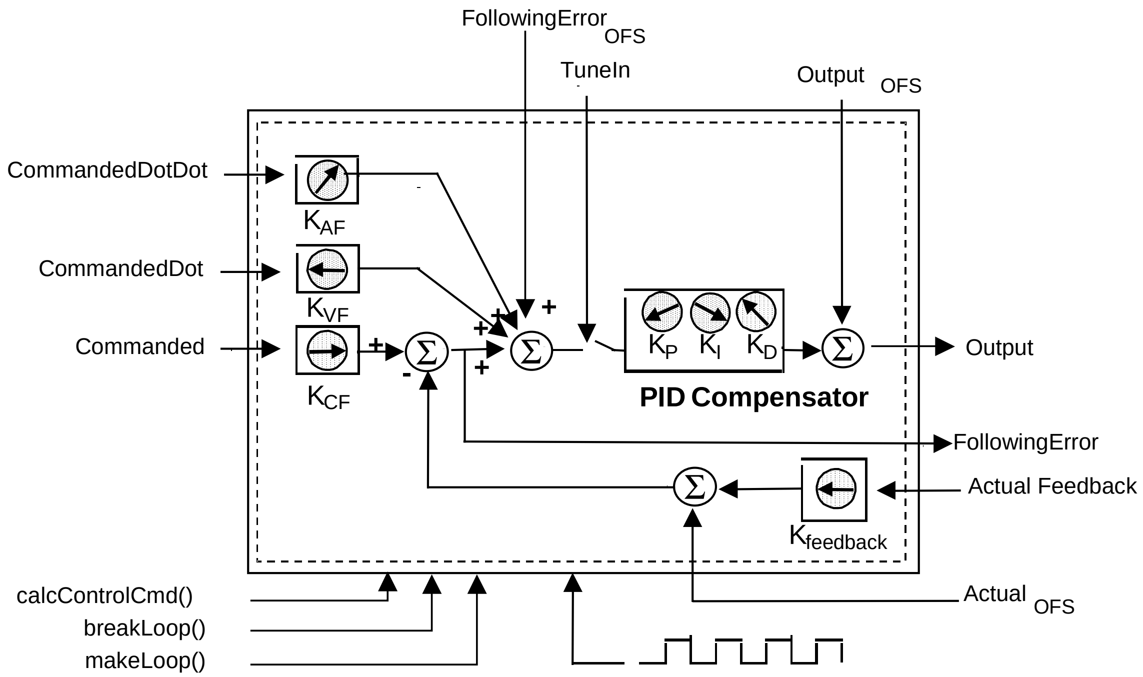


Figure 10: PID Control Law

3.3 CLIENT SERVER BEHAVIOR MODEL

OMAC API adopts a client server model for inter-object communication. In the client/server model, an object is a **server** and a user of an object is called a **client**. Objects can act as both a client and a server. Objects cooperate by having clients issue requests to the servers. The server responds to client requests. For OMAC API, a client invokes **class methods** to achieve the described cooperative behavior. A client uses **accessor methods** to manipulate data. Accessor methods hide the data's physical representation from the abstract data representation.

Standard client-server requests result in a synchronous execution of operation. The synchronous execution has a client-server **roundtrip** where the client issues a request, server receives a method invocation, performs the corresponding method implementation, and sends a reply back to the client. OMAC API defines three types of client-server requests: (1) parametric requests, (2) directive requests and (3) monitor requests. **State space logic** may be required to manage client-server interaction.

Parametric requests are the get/set methods that are, in theory, satisfied in one roundtrip. Parametric requests do not require state space logic.

Directive requests are events which cause a change in the server's state space (or state transition) and results in a new server state. These directive requests may run one or many cycles - such as, for

an Axis module completing a home() operation. Coordination between the client and server requires state space logic and is based on the server's Finite State Machine model
Monitor requests coordinate the execution of a module, for example, processServoLoop() or isDone() for Axis module. Monitor requests are coordinated by the state space logic. The processServoLoop method sends an event to Axis module execution to be interpreted by its state space logic. Invoking processServoLoop every servo loop period attains cyclic execution of the Axis module. In this cyclic mode, the Axis Module would be running as a software servomechanism: at every period, it accesses data (e.g., commanded position, actual feedback) and executes a transform function to derive a new setpoint. Status methods are necessary to monitor the progress of a directive request.
Client Directive and Monitoring requests may come from separate threads of control. Figure 11 illustrates a server with multiple clients running in two separate processes: an Axis Group process for issuing setpoints and a Periodic Updater process to coordinate execution. (These processes may be running in one or more threads.) Generally, the Directive service requests would come from an Axis Group module that is issuing setpoints to multiple axes. A Scheduling Updater module running in another thread of execution provides timing, synchronization and sequencing service for the Axis module. This Scheduling Updater module may be tied to some hardware device (such as a timer) to guarantee periodic execution behavior.

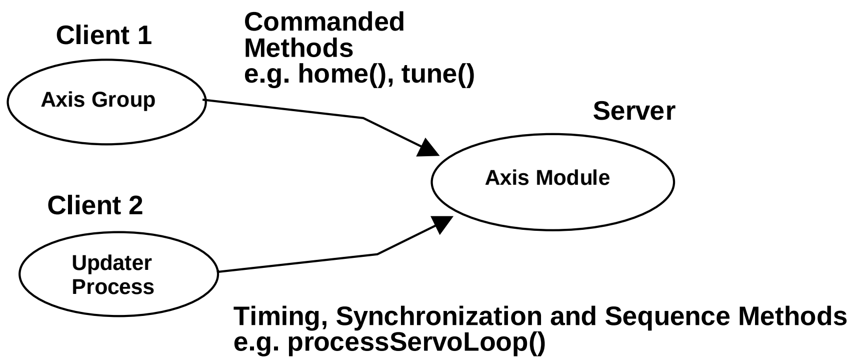

Client 1
Commanded Methods e.g. home(), tune()
Axis Group
Server
Axis Module
Client 2
Updater Process
Timing, Synchronization and Sequence Methods e.g. processServoLoop()

Figure 11: Multiple Threads of Control

3.3.1 DIRECTIVE REQUESTS DISCUSSION

Client directive requests are serviced as **client-push events**. (Server-push is a more difficult problem and is discussed in Section 5.2.) In a client-push request, events are "pushed" to the server via method calls. Client-push events may be queued and ultimately cause state transitions. Below is a code sketch of the client-push event model for an Axis class that defines two methods processServoLoop and home. An AxisFSM class is defined to handle the events caused by processServoLoop and home. Whenever the home method is invoked, it inserts a HOME_EVENT event into the Axis FSM. The FSM has an internal queue (i.e., evq) for handling events. The FSM may optionally spawn a separate thread of control (i.e., FSMThread()) for event handling. The isDone() monitor request is used to determine when the home event has completed.

```
// This is the public interface
class Axis : OmacModule
{
public:
    processServoLoop();
    home();
    boolean isDone();
private:
    AxisFSM fsm;
    boolean myDone;
};

// This is hidden in the implementers code
Axis::processServoLoop() { AxisFSM.handleEvent(AxisFSM::PROCESS_SERVO_LOOP_EVENT); }
Axis::home() { AxisFSM.handleEvent(AxisFSM::HOME_EVENT); }
Axis::isDone() { return myDone; }

class AxisFSM : FSM {
    enum { PROCESS_SERVO_LOOP_EVENT,  HOME_EVENT};
    MsgQueue evq;
    int curState;
    void handleEvent(EV_num)
    {
        evq.send(EV_NO);
    }
    void * FSMThread() // optional thread, this could be done in handleEvent
    {
        int evNum;
        evq.receive(&evNum);
        callAction(evNum, curState);
    }
    void homeUpdateAction() { /* perform homing */ }
    void processServoLoopAction() { /* evaluate state */ }
};
```

A key to the event model is to support local or remote method invocation identically. The next section on proxy agents explains how this event model provides a transparent interface.
Server request actions should be as short as possible. In the example, the simple enqueuing of events provides an efficient interface model. The rationale for short request cycles is to reduce the amount of time the client will wait while the server services the request. Evaluating system timing and performance is difficult unless the client-server round-trip time is bounded.

3.4 PROXY AGENT TECHNOLOGY

Client/server interaction can be local or distributed. In **local** interaction, the client uses a class definition to declare an object. When a client accesses data or invokes object methods, interaction is via a direct function call to the corresponding server class member. At its simplest, local interaction can be achieved with the server implemented as a class object file or library. Interaction is achieved by binding the client object to a newly created server object implementation. Such a binding could be done by static linking, with a dynamic linked library (DLL), or through a register and bind process that does not use the linker symbol table.
When **distributed** service is needed a **proxy agent** is used. A proxy agent is a set of objects that are used to allow the crossing of address-space or communication domain boundaries[M.S86]. The class describing a proxy agent uses the API of some other class (for which it is a proxy) but provides a transparent mechanism that implements that API while crossing a domain boundary. The proxy agent could use any number of lower level communication mechanisms including a network, shared memory, message queues, or serial lines.
Below is a code example to illustrate the concept of proxy agents. We will assume that we have defined an axis module by the class Axis that has but one method `setX();`. The following code would be found in the axis module header file (or API specification):

```
    class Axis : Environment
    {
    public:
      void setX();
    private:
      double myX;
    }
```

A user would then develop code to connect or bind to the axis module server, which in this case has the name “Axis1.” The `_bind` service is similar to a constructor method, but returns a server reference pointer rather than an address reference pointer. The `_bind` keeps track of the number of client pointer references to the server. The bind establishes a client/server relationship with the axis module. The application code is the client, and when Axis methods are invoked, a message is sent to the server. In the following code, the application sets the x variable to 10.0:

```
	application(){
		Axis * a1;
		a1 = Axis::_bind("Axis1");
		a1->setX(10.0);
	}
```

If the server is co-located with the application, it is trivial to implement the object server. The `Axis::setX` implements the value store.

```
	Axis::setX(double _x){ myX = _x; }
```

However, for distributed communication, `Axis::setX` is defined twice - once on the client side and once on the server side. On the client side we set up the remote communication, which in this case, is an overview of a remote procedure call.

```
	Axis::setX(double _x){
		callrpc(host, prognum, versnum, procnum, inproc, in, outproc, out)
	}
```

On the server side, a server waits for service events (such as the `bind`, and the `setX` method). A corresponding `Axis::setX` is defined to handle the x variable store. The server technology could handle events in the background or use explicit event handling. In either case, the actions of the server are transparent to the client.

```
	Axis::setX(double _x){ myX = _x; }

	server(){
		/* register rpc server name */
		while(1) { /* service events */ }
	}
```

3.5 INFRASTRUCTURE

The infrastructure deals primarily with the computing environment including platform services, operating system, and programming tools. Platform services include such items as timers, interrupt handlers, and inter-process communications. The operating system (OS) includes the collection of software and hardware services that control the execution of computer programs and provide such services as resource allocation, job control, device input/output, and file management. Real Time Operating System Extensions can be considered platform services since these extensions are required for semaphoring, and pre-emptive priority scheduling, as well as local, distributed, and networked interprocess communication. Programming tools include compilers, linkers, and debuggers.

The OMAC API does not specify an infrastructure because many of the infrastructural issues are outside the controller domain, and it would be better handled by the domain experts. Further, it is more cost-effective to leverage industry efforts rather than to reinvent these technologies. For example, commercial implementations of proxy agent technology are available. Microsoft has developed and released DCOM (Distributed Common Object Model) [DCO] for Windows 95 and Windows NT. Many implementations of CORBA (Common Object Request Broker Architecture) [COR91] are available and Netscape incorporates an Internet Interoperable ORB Protocol (IIOP) inside its browser. The question concerning the hard-real-time capability of such products remains. But, industry is acting to solve this problem. In the interim, control standards that could provide a real-time infrastructure are available [OSA96].

Because there are so many competing infrastructure technologies, OMAC API has chosen to let the market decide the course of the infrastructure definition. As such, to achieve plug-and-play module interchangeability, a commitment to a **Platform + Operating System + Compiler + Loader + Infrastructure suite** is necessary for it to be possible to swap OMAC object modules.

3.6 BEHAVIOR MODEL

For the OMAC API, **behavior** in the controller is embodied in Finite State Machines (**FSM**). OMAC API uses state terminology from IEC1131[IEC93]. An FSM **step** represents a situation in which the behavior, with respect to inputs and outputs, follows a set of rules defined by the associated **actions** of the step. A step is either **active** or **inactive**. **Action** is a step a user takes to complete a task that may invoke one or more functions, but need not invoke any. A **transition** represents the **condition** whereby control passes from one or more steps preceding the transition to one or more successor steps.

For the OMAC API, the following concepts apply. The receipt of a message causes an **event** that is evaluated with the FSM and may cause a state transition. An object **method invocation** is the mechanism in which messages are sent to cause an event. For distributed communication, OMAC API makes the assumption that the proxy agent does the encoding of methods into messages and the decoding of the transmitted message into the corresponding method calls.

3.6.1 LEVELS OF FINITE STATE MACHINES

For an OMAC API module, there can be nesting of FSMs. OMAC API does not dictate the number of levels of FSM. In general, an outer **administrative** FSM exists to handle activities that include initialization, startup, shutdown, and, if relevant, power enabling. The administrative FSM must follow established safety standards. When the administrative FSM is in the READY state, it is possible to descend into a lower level FSM.

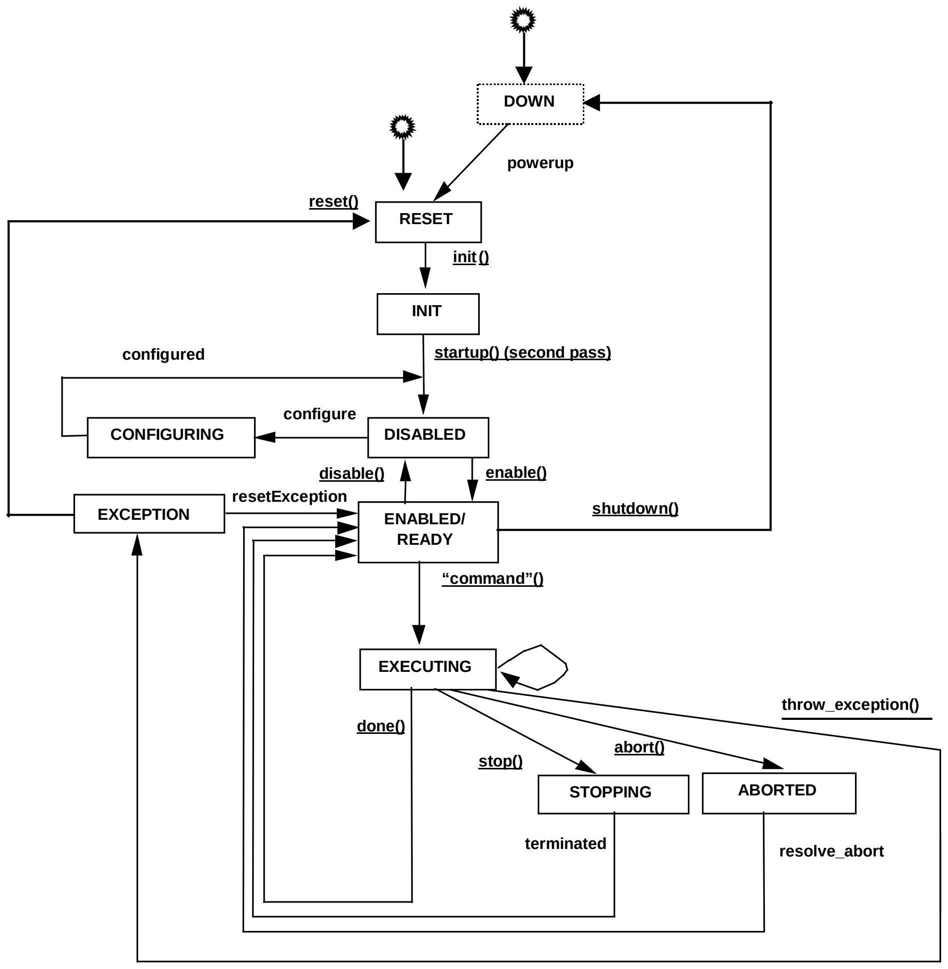


Figure 12: Generalized State Diagram

OMAC API defines the OMAC Base Class module to provide a uniform administrative state model across modules. The OMAC Base Class state model is illustrated in Figure 12. The administrative state model describes the start-up, shutdown, enabled/ready, configured, aborted, and initialization operations that form the baseline of a module state space. States have methods (e.g., `init()`, `startup()`) to cause state transitions.

To enter into a lower FSM, the module enters into the "executing" state as shown Figure 12. In the "executing" state, client/server coordination uses a lower FSM for coordination. This lower FSM is module- and application-dependent. This lower FSM, in turn, can have an FSM embedded within it so that further nesting of embedded FSMs is possible.

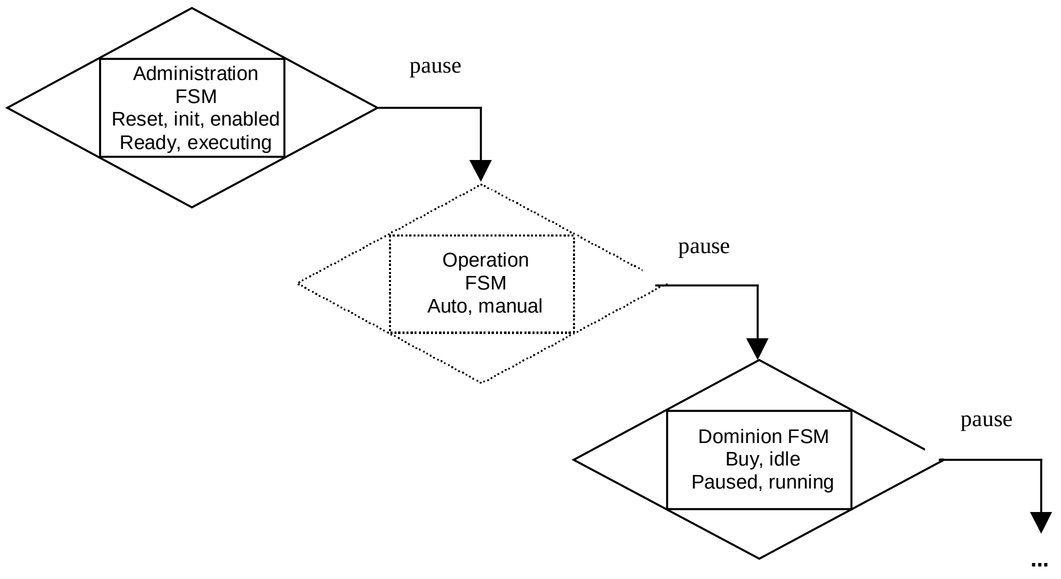


Figure 13: Levels of FSM

Figure 13 shows the nesting of FSM levels. Within the figure, the FSM icon is represented by a rectangle inside a diamond. The dotted FSM icon represents an optional FSM. The nesting of one or more lower level **operation** FSMs is possible depending on system complexity. Within the nesting of the FSM shown in Figure 13, an "operational" FSM may handle different NC modes corresponding to "auto," "manual," or "MDI". For example, at the operation level for part programming, there may be another level of FSM to handle a family of parts. The designer of a particular control system determines the number of nested FSM levels, depending upon the complexity and organization of the controlled system. The lowest level FSM or **dominion** FSM monitors the current focus of control. The **dominion** FSM "rule" over lower level objects. There may be one or more dominion FSM at the lowest level within an OMAC module.

For OMAC API, method invocations result in events to be propagated from the client to the server that may cause server state transitions. Events are evaluated within the highest level FSM and then recursively propagated through each lower level FSM. For example, in Figure 13 a `pause` event is received at the highest Administration level and is evaluated. If the Operation FSM supports a `pause` method then this method is invoked and the event evaluated. This event evaluation and recursive cascading of the event may cross module boundaries and propagate all the way to the "bottom" FSM in the application controller.

A major assumption concerning event propagation is the availability of the event method in a lower FSM. In the previous example, there was an underlying assumption that all lower-level FSM supported the `pause` method. This underlying assumption may or may not hold. For the interim, the following rules characterize the FSM behavior with regard to specifying an event space:

- an OMAC module Administrative FSM supports all the events within the OMAC API Base FSM
- any lower level FSM within an OMAC module supports both the OMAC Base FSM event space as well any event specializations that an OMAC module supports. For example, the Axis Group module defines events for `hold, pause, resume` and these would have to be supported by lower level FSM contained within the Axis Group.
- Control Plan Units may have their own event model. It is unclear if they must support the complete OMAC Base Class set of events.

- optionally, an introspective query of an FSM could be specified to see if an event is supported (e.g., `canPause()`). This mechanism is similar to that of reusable component functionality of JavaBeans that provides for run-time and design-time methods. In addition to handling event space matching, introspection could be useful as a safety feature to insure that cooperating FSM understand each other.

3.6.2 COMPUTATIONAL MODEL

A general computational model exists for characterizing all OMAC control modules. Figure 14 illustrates the general computational model. Each OMAC module can support levels of nesting FSM as part of general computational model. The OMAC API module may also have one or more FSM simultaneously executing on a dominion FSM list. Each FSM on the dominion list is conceptually equivalent to a concurrent thread of state logic. FSM on the dominion list can operate independently or have dependencies between them.

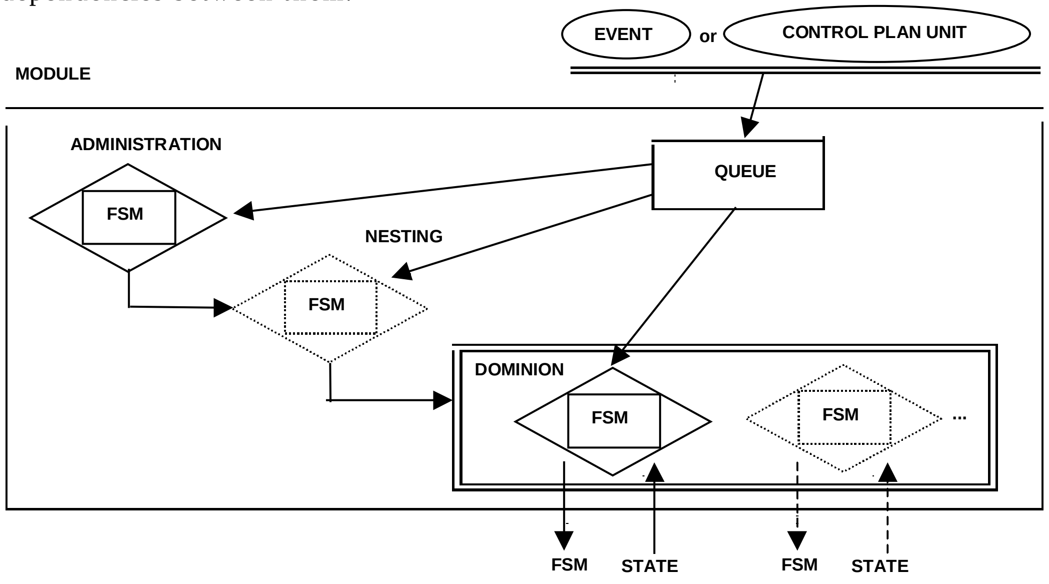


Figure 14: Module Computational Paradigm

Within the FSM paradigm, different OMAC API modules have different FSM dominion list sizes. In general, the OMAC modules exhibit the following computational model characteristics. The Discrete Logic module generally has a multi-item dominion FSM list analogous to a scan list, (some active, some not active), to coordinate IO points. The Axis Group has a multi-item dominion list, one or more motion FSM and none, one, or more Process FSM. The Axis module has one FSM derived from the OMAC Base Class and an embedded FSM to support Axis functionality.

In the general computational model, **FSM are used for controlling behavior and also serve as data.** When events are sent from the client to the server and contain FSM as data, the FSM data is called a **ControlPlanUnit** (CPU). A ControlPlanUnit is an FSM, but the internal representation is not important to the OMAC API. Instead, a CPU is defined with a simple state management API hiding messy FSM details. The following is a sketch of the ControlPlanUnit API.

```
interface ControlPlanUnit
{ // Option 1:
  ControlPlanUnit executeUnit();    // return next ControlPlanUnit
  // Option 2:
  // boolean isDone();              // state query
  // ControlPlanUnit getNextUnit(); // actually fetch next CPU when done
  void setActive();                 // set when “executing”
  void setInactive();
  boolean isActive();            // for HMI to determine when active
```

```
        // ... methods for persistence data in binary or neutral format
        // ... methods for graph representation for navigation purposes,
        //     such as when performing lookahead
      };
```

The general computational model supports a mechanism to queue client requests - either events or CPU. A CPU received by a server is queued and is eventually inserted into the dominion list. Three types of CPU can exist on the dominion list:
Transient CPUs perform a fixed amount of work within a certain period. Transient CPUs execute cyclically and are removed from the dominion list when an internal condition is satisfied. An example of a transient CPUs is a motion segment CPU that has a beginning and an end. When the CPU `isDone()` returns true, the CPU is removed from the dominion list.
Resident Cyclic CPUs execute "forever" and perform a function periodically. Resident cyclic CPUs execute repeatedly with no internal completion condition. One example of a resident cyclic CPU is a PLC operation to turn the oil/slides pump on/off every five minutes.
Resident Event-driven CPUs execute once when an event triggers their execution. An example of a Resident Event-driven CPU is turning an IO point on or off.
The ability to have multiple CPU executing concurrently can be especially useful for Process Model enhancement. Within the Axis Group for example, one can have a **transient** CPU for motion as well as a **resident cyclic** CPU to handle data logging.
Equivalent application functionality can be achieved with different distributions of CPU within a controller. Depending on the circumstances, **tight coupling** or **loose coupling** can be used to coordinate logic and motion. Tight coupling is achieved by placing RESIDENT FSM on the dominion list. Loose coupling is achieved by placing RESIDENT FSM in a separate thread under same scheduler for all the other OMAC modules (which are resident FSM.)

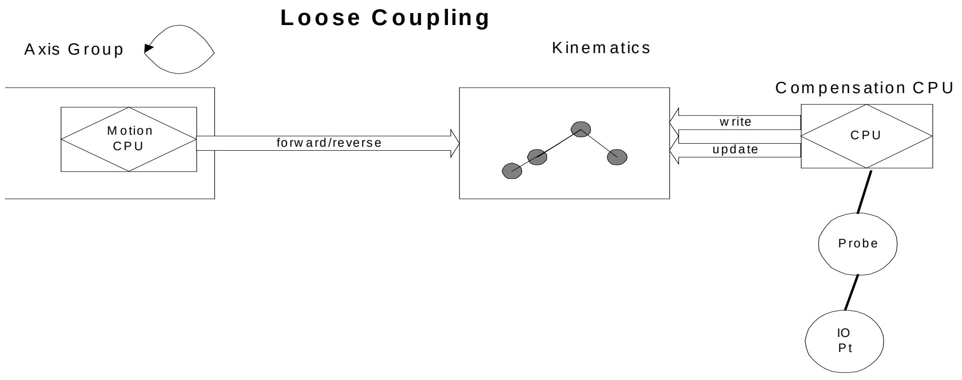


Figure 15: Example Loose Coupling Probe Architecture

As an example, consider the integration of a Probe with an Axis Group to modify motion control. Several ways exist for incorporating the Probe CPU into the system.

- The Probe CPU is placed in the Discrete Logic module to be run at a given period. The probe could running at the same period as the Axis Group or be oversampled. This is an example of loose coupling.
- The probe could run as standalone resident CPU scheduled like any other OMAC module. The probe CPU could run at a slower, faster or the same frequency as the Axis Group. This is an example of loose coupling and is illustrated in Figure 15.
- The Probe could be a Process Model resident CPU that runs inside of the Axis Group at the same frequency as the Axis Group. This is an example of tight coupling and is illustrated in Figure 16.

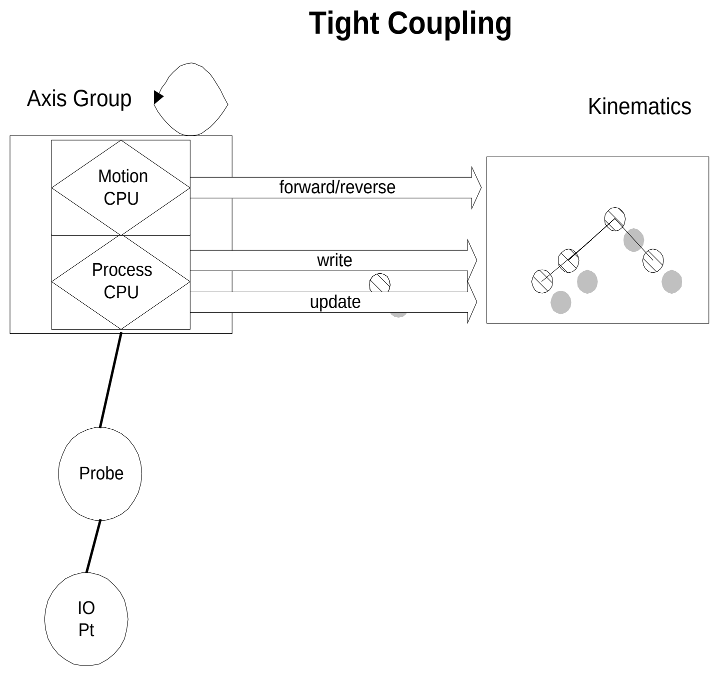


Figure 16: Example Tight Coupling Probe Architecture

3.6.2 Control Plan Unit Abstractions

The CPU is the base class, but the OMAC API defines several uses and specializations. Figure 17 illustrates the ControlPlanUnits hierarchy of possible ControlPlanUnit specializations. CPU specialization is the mechanism to add extensions. For example, the NURB MotionSegment is derived from the MotionSegment CPU. Specialization of CPU include:

Capabilities

correspond to different machine modes (manual, auto). When the Capability FSM is in the READY state, the Capability can descend into a lower FSM or ControlPlanUnit. For example, once in the auto Capability FSM, a lower level FSM for the "cycle" ControlPlanUnit can be used to sequence through a series of ControlPlanUnits.

MotionSegments

corresponds to the FSM input for an Axis Group module. In addition to the FSM directive and parameter methods, a MotionSegment includes such information as rate, geometry, and a reference to a velocity profile generator that are necessary for trajectory planning.

DiscreteLogicUnits

corresponds to the FSM input for a Discrete Logic module. DiscreteLogicUnits coordinate and control an aggregation of IO points. In addition to the FSM directive and parameter methods, a DiscreteLogicUnit contains the information necessary to either define asynchronous logic - the event or condition trigger, or to define synchronous logic - the scan rate and FSM.

ProgramLogic

CPU for decision making, (e.g., statement, loops, end program and if/then/else).

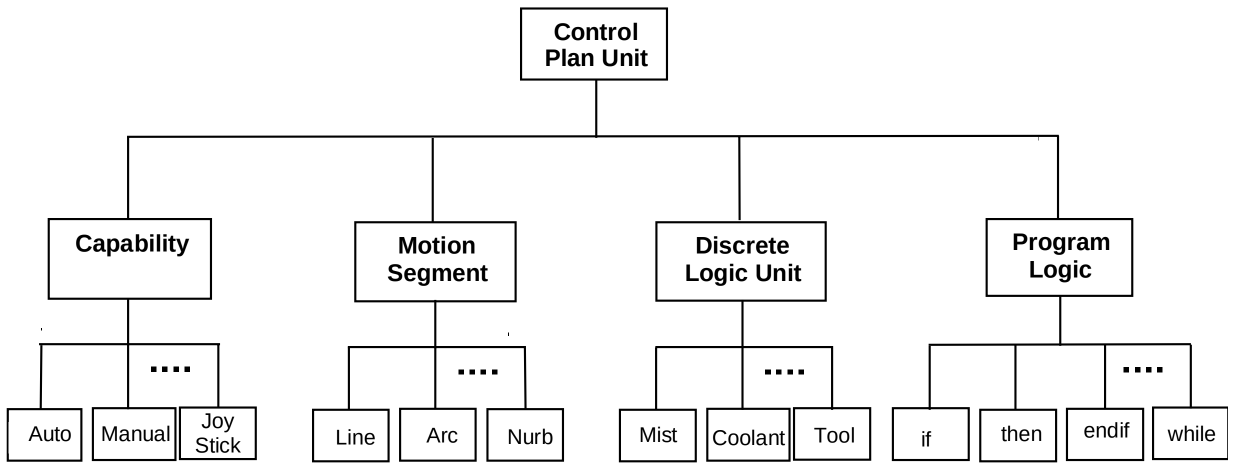


Figure 15: Examples of Different Types of Control Plan Units

A `ControlPlanUnit` is responsible for its own branching. For this reason, the method `executeUnit()` returns a reference to the next ControlPlanUnit. A `ControlPlanUnit` may embed other `ControlPlanUnits`. A series of `ControlPlanUnit(s)` is a `ControlPlan`. A `ControlPlan` can be a simple list to represent sequential behavior or a complex tree. Figure 18 illustrates some possible connections of ControlPlanUnits. Through the use of ProgramLogic CPU, one can achieve a mapping from computer programming control constructs into a list representation.
To coordinate the ControlPlan (which is a graph of ControlPlanUnits) for outside observers (such as the Human Machine Interface), there is a central ControlPlan header. The ControlPlan header monitors navigation through the graph as ControlPlanUnit are activated and deactivated. As activity in the ControlPlan occurs, the ControlPlan header points to active ControlPlanUnits. Traversal methods are defined within a ControlPlanUnit so that external modules, such as the HMI, can monitor progress of ControlPlan via the `isActive()` method.

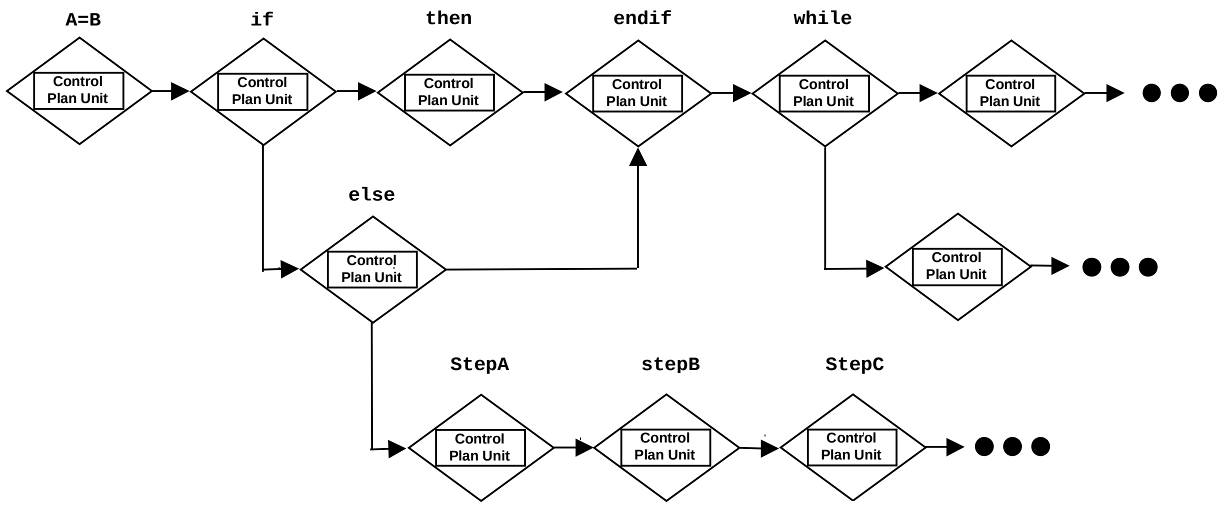


Figure 18: Control Plan built from Series of Control Plan Units

3.6.3 CONTROL PLAN UNIT NESTING

A `ControlPlanUnit` can contain other ControlPlanUnits. When activated, a CPU can send embedded CPU to lower level servers. Thus, CPUs contain "intelligence" and understand how to coordinate and sequence the lower level logic and motion modules.

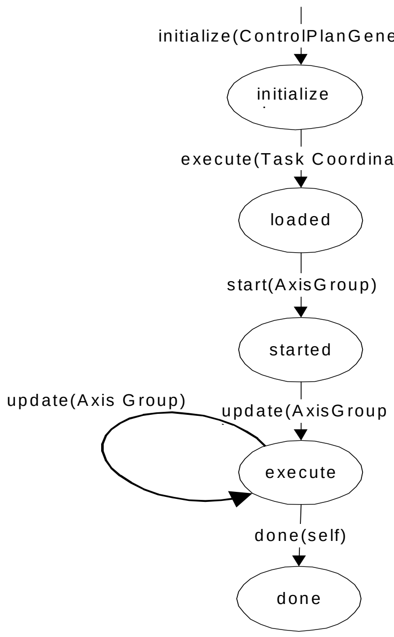


Figure 19: Example Control Plan State Transitions

Figure 19 illustrates an example of the relationship between a CPU, its states, and its travel through a control system. In this example, a ControlPlanGenerator, such as one for RS247D or IEC1131, initially generates Control Plans from part programs most likely using a CPU constructor. During execution of a Control Plan, the CPU is becomes the next active CPU in the Task Coordinator. The Task Coordinator does an `executeUnit` on this CPU. The CPU determines if it can append an embedded Motion Segment CPU onto the Axis Group motion queue. If for example, a tool change is desired, then assume the CPU should wait until all current motion must be completed first. This requires the CPU do synchronize with lower level modules. The synchronization would occur inside the CPU and could be done with or without blocking. The code for a blocking CPU would look like this:

```
CPU execute_unit()
{ axgrp->wait_for_motion_idle();  // blocks until this is true
  axgrp->setNextMotionSegment(moveToToolChangerMS);
// pass change tool CPU to discrete logic
 return nextCPU;
}
```

The code for a non-blocking CPU would look like this and assumes that the Task Coordinator periodically performs an `executeUnit` on the CPU.

```
CPU executeUnit()
{ if(!axgrp->isIdle())  return this;
  axgrp->setNextMotionSegment(moveToToolChangerMS);
// pass change tool CPU to discrete logic
return nextCPU;
}
```

Once the CPU is free to continue, embedded CPU(s) are passed to subordinate modules and loaded onto their event queues. That is, the CPU running in the Task Coordinator passes the next Motion

Segment CPU to the Axis Group module and passes a Tool Change Discrete Logic Unit to the Discrete Logic module.

Once the Motion Segment CPU is loaded onto the Axis Group queue, it waits for activation. Activation can occur if the CPU is first on the queue and no CPU are on the dominion list running, or the previous CPU already running on the dominion list returns a true to `startNextCPU()`.

If ready for activation, the Axis Group moves the MotionSegment method from the motion queue to the dominion list and calls `start`, which places the CPU in the `started` state. Herein, the MotionSegment is in the `executing` state and the Axis Group periodically calls the Motion Segment CPU `update()` method until the `isDone()` condition is true.

The transition from `executing` to `done` does not result from an externally-generated event, but rather is achieved by the CPU satisfying an internal termination condition (hence the reference to `self`).

Figure 20 illustrates the propagation of CPU through a controller. The Control Plan Generator generates a top-level ControlPlanUnit $\texttt{CPU}_1$ for the Task Coordinator. $\texttt{CPU}_1$ contains embedded MotionSegment CPU $\texttt{MotionSegment}_a$ and DiscreteLogicUnit CPU $\texttt{DiscreteLogicUnitCPU}_b$. Consider the coordination required for a tool change. The top-level $\texttt{CPU}_1$ forwards $\texttt{CPU}_{1b}$ or $\texttt{DiscreteLogicUnitCPU}_b$ to the DiscreteLogic module to be placed on its scanning list. For simplicity, assume the top-level CPU waits until the DiscreteLogic reports that it is done with the tool change. Once the tool change motion is completed, the top-level $\texttt{CPU}_1$ can then forward $\texttt{CPU}_{1a}$ or $\texttt{MotionSegment}_a$ to the AxisGroup.

It is important to understand the nesting of CPU and subsequent propagation of CPU. It is the fundamental mechanism for passing data through an OMAC API controller.

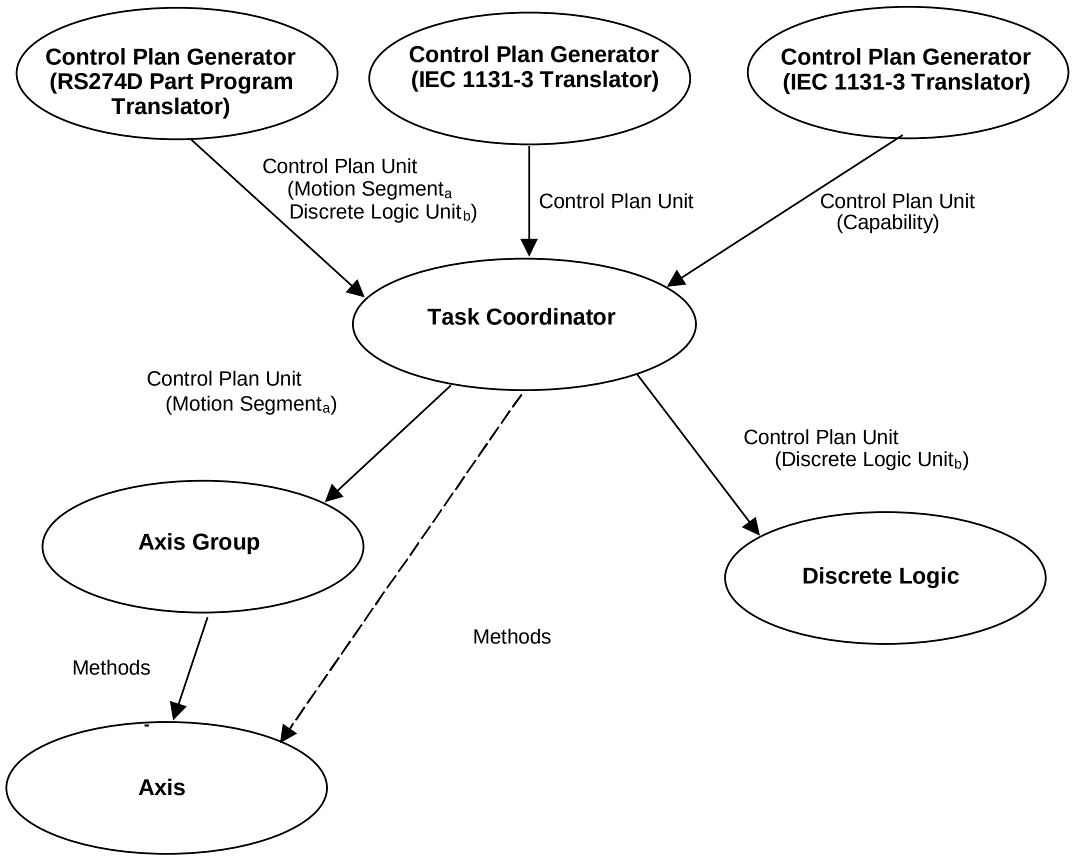
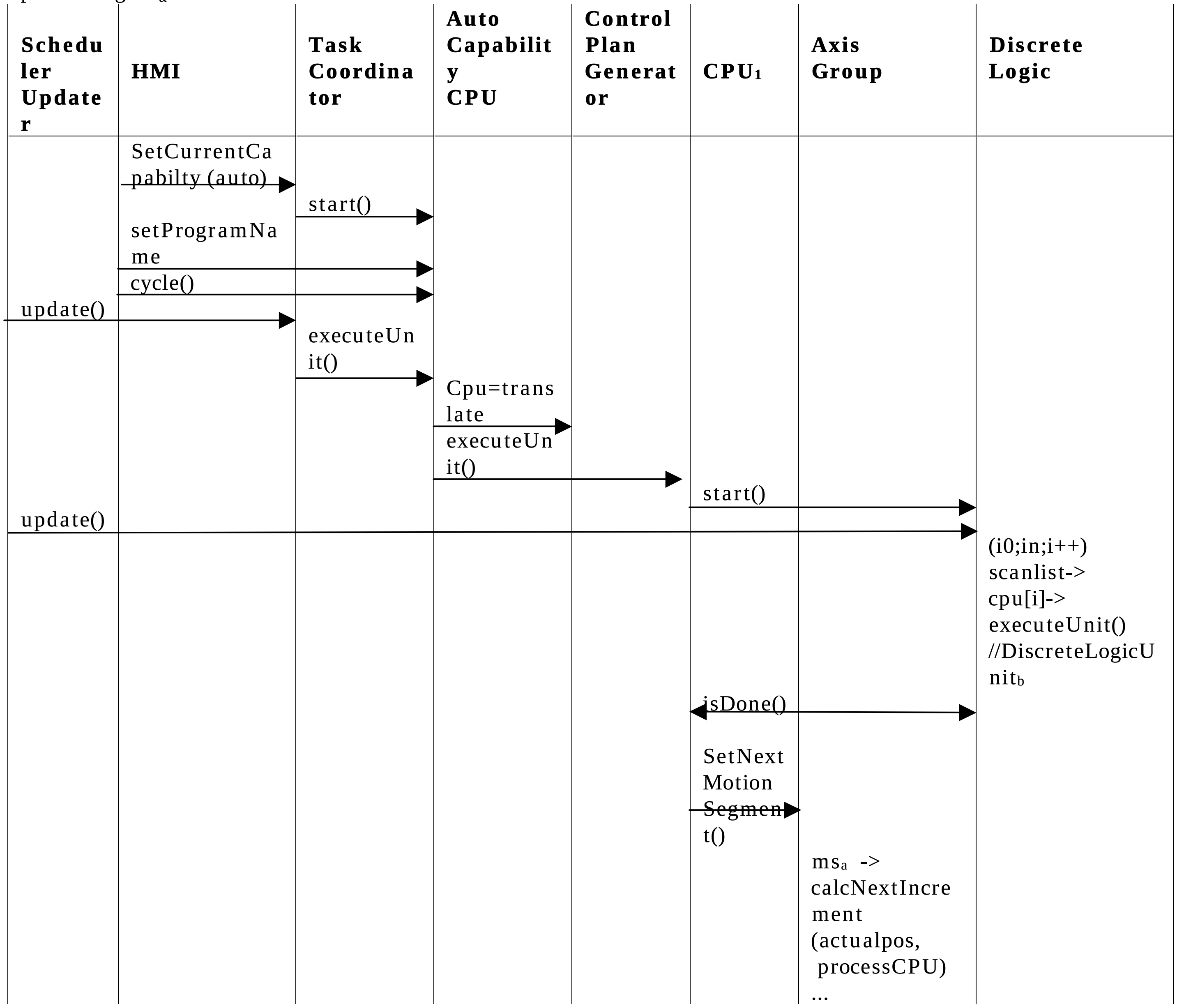

Control Plan Generator (RS274D Part Program Translator)
Control Plan Generator (IEC 1131-3 Translator)
Control Plan Generator (IEC 1131-3 Translator)
Control Plan Unit (Motion Segment$_a$ Discrete Logic Unit$_b$)
Control Plan Unit
Control Plan Unit (Capability)
Task Coordinator
Control Plan Unit (Motion Segment$_a$)
Control Plan Unit (Discrete Logic Unit$_b$)
Axis Group
Discrete Logic
Methods
Methods
Axis

Figure 17: Intelligent CPU Spawning Lower Level CPU

Figure 18 is an Object Interaction Diagram for the following propagation scenario. Assume a Human Machine Interface will set the current Capability to `Auto` mode. Then, the HMI interacts with the Auto Capability to load a program name and then start the cycle. This will cause the Task Coordinator to request the Control Plan Generator to translate the part program into a Control Plan. Once translated, CPU$_1$ will be executed via the `executeUnit` method. While CPU$_1$ is executing, it will forward two new Control Plan Units - first a Discrete Logic Unit `dlu`$_b$ to perform a tool change and afterwards a Motion Segment `ms`$_1$. When it's time, the scheduler or updater will cause the DiscreteLogic module to execute. The DiscreteLogic module will then process its scan list and in turn execute `dlu`$_b$. When the `dlu`$_b$ tool change `isDone`, CPU$_1$ will forward Motion Segment `ms`$_a$. At the appropriate time, the scheduler or updater will cause the AxisGroup to execute and it will start processing `ms`$_a$.



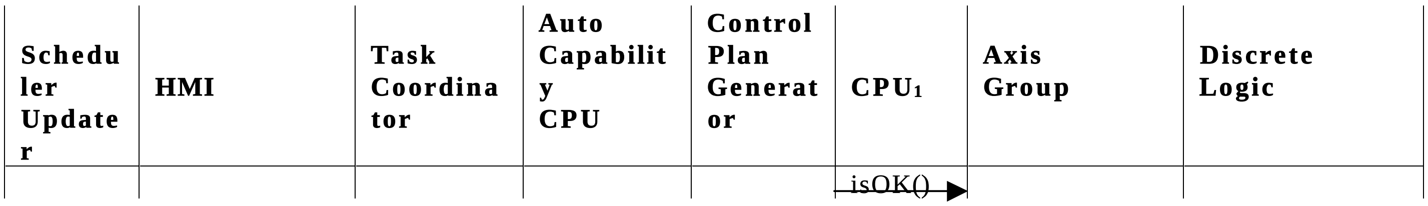


Figure 18: Embedded CPU Forwarding Object Interaction Diagram

The OMAC API specifies that `ControlPlanUnit` objects can embed module references and direct method calls. On the surface, this approach appears implausible. However, because of proxy agent technology, it is not hard to create a "forward reference" assuming one can dynamically bind to an object. This dynamic binding is beneficial since it eliminates static encoding of methods (e.g., with id numbers) necessary for methods to execute across domains (i.e., address spaces). To enable forward references, the requirement does exist for the infrastructure to support some "`lookup()`" method to map object names to addresses. Consider the following C++ code to handle generic Axis Group control within the Task Coordinator.

```
class G0CPU : ControlPlanUnit
{
  void setMotionSegment(MotionSegment _msA);  // parameters set by the CPG

  setAxisGroup(char * axgroupname) { ag=lookup(axgroupname); }
  setAxisGroup(AxisGroup * axgrp)  { ag=axgrp; }

  CPU executeUnit()
  {
    if(!firstTime++)
        ag->setNextMotionSegment(msA);      // message passing!
    if(!ag->isDone()) return this;         // not done
    else return NULL;                       // return NULL or done CPU
  }

  private:
    MotionSegment msA;
    long firstTime;
};
```

In the example, a ControlPlanGenerator will create a `G0CPU` that contains a MotionSegment (i.e., `msA`). When the `TaskCoordinator is executing the G0 CPU`, the `executeUnit` method uses explicit calls to an Axis Group object, (i.e. `ag`). In early binding, a "forward reference" must be fulfilled by the ControlPlanGenerator to the Axis Group object is required. In late binding, the TaskCoordinator could do the lookup of the AxisGroup reference. However, late binding can unnecessarily slow down the "block throughput" of CPU, hence only early binding will be considered. To achieve early binding, suppose the Control Plan Generator (`CPG`) constructor receives the name "**axisgroup1**" for an Axis Group object. The CPG can lookup the object "**axisgroup1**" to retrieve a reference address. Upon receiving a reference address to "**axisgroup1**," the CPG passes this reference address to a CPU, in this example, with the method `setAxisGroup`.
The degree of difficulty to do a reference address lookup depends on the execution environment. For modules running as one or more threads in a process, the reference address is trivial. For reference addresses that cross domain boundaries, proxy agent technology is required. Proxy agents must encode reference addresses with a more sophisticated scheme to capture the domain (e.g., machine, process) and encode the object reference and the methods. Proxy agent technology should hide the reference address encoding from the programmer.

3.7 DATA REPRESENTATION

Exchange of information between modules relies on standard information representation. Such control domain information includes units, measures, data structures, geometry, kinematics, as well

as the framework component technology. OMAC API has chosen two levels of compliance for data definitions.
The first level defines named data types to allow type-checking. The OMAC API uses the IDL primitive data types and builds on these data types to develop the foundation classes and framework components. For control domain data modeling, the OMAC API used data representations found in STEP Part Models for geometry and kinematics [Inta, Intb]. Internally, any desired representation could be used. The STEP data representations were translated from EXPRESS[EXP] into IDL. Representation units are assumed to be in International System of Units, universally abbreviated SI. Below is the basic set of data types, which use STEP terminology for data names but reference other terms for clarification.

Primitive Data

- IDL data types include **constants**, **basic data types** (float, double, unsigned long, short, char, boolean, octet, any), **constructed types** (struct, union and enum), **arrays** and **template types** bounded or unbounded sequence and string.
- IEC 1131 types - 64 bit numbers
- bounded string

Time

Length

- Plane angle
- Translation commonly referred to as position
- Roll Pitch Yaw (RPY) commonly referred to as orientation
- STEP notion of a Transform which is composed of a translation + rpy, also commonly referred to as a “pose.”
- Coordinate Frame which is defined as a Homogeneous Matrix

Dynamics

- Linear Velocity, Acceleration, Jerk
- Angular Velocity, Acceleration, Jerk
- Force
- Mass
- Moment
- Moment of Inertia
- Voltage, Current, Resistance

The second level provides for more data semantics. The OMAC API adopted the following strategy to handle data typing, measurement units, and permissible value ranges. Distinct data representations were defined for specific data types. For example, the following types were defined in IDL to handle linear velocity.

```
// Information Model - for illustrative purposes
typedef Magnitude double;

// Declaration
interface LinearVelocity : Units  {

        Magnitude  value; // should this value be used?
        // Upperbound and Lowerbound, both zero ignore
        Magnitude ub, lb; // which may be ignored

        disabled();
        enabled();
};
```

```
        // Application
        LinearVelocity vel;
```

In this case, linear velocity is a special class. Unit representation is inherited from a general unit's model. Permissible values are defined as a range from lowerbound to upperbound. The units and range information are optional and may not be used by the application.
Another data typing problem that must be resolved concerns the use of a parameter. Not all parameters are required or set by every algorithm. For example, setting the jerk limit may not be necessary for many control algorithms. It was decided to use a special value to flag a parameter as "not-in-use". This approach seems simpler than having a `useXXX` type method for each parameter. For now, OMAC API has decided that setting a parameter to an unrealistic "Not in use Number" (but not actually "Not a Number") value - such as `MAXDOUBLE` or 1.79769313486231570e+308 - renders a `double` parameter to be ignored or not-in-use. A similar number would be required for an integer. This works for level 1 and level 2. Within level 2, the methods `enable` and `disable` were added to explicitly indicate use of a parameter.

4 MODULE OVERVIEW

4.1 TASK COORDINATOR

The general characteristics of the Task Coordinator module include:

- act as central point for coordination
- initiate startup and shutdown since it understands the controller configuration - what modules are in the system and how to start up the modules
- act as the highest level Finite State Machine within the controller.
- change frequently. The leaf nodes in the OMAC API architecture will be most stable. As such, each system change should not require an entire rewrite of the TC. Instead, TC should be flexible to accommodate change.

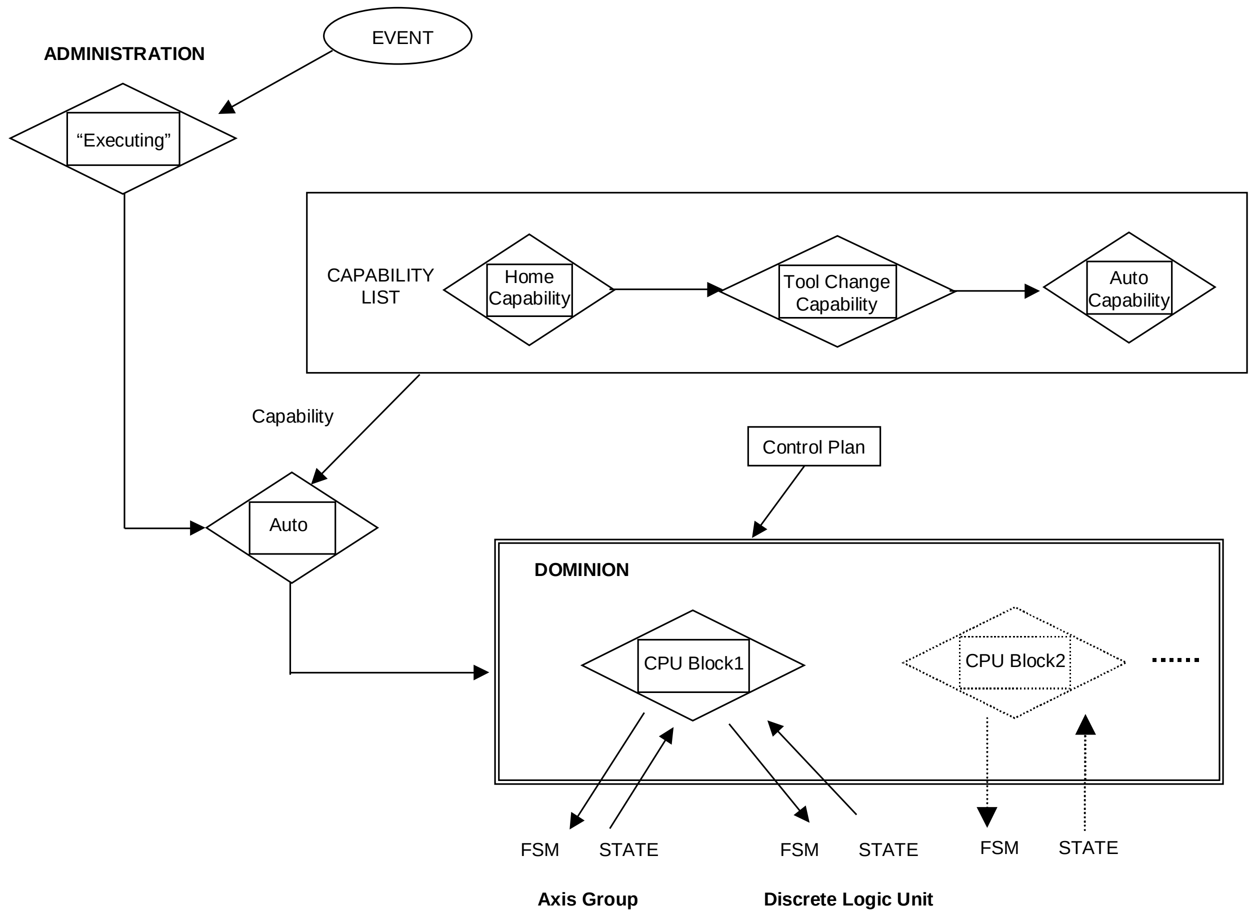


Figure 19: Task Coordinator Computational Model

The Task Coordinator module is an FSM. The Task Coordinator FSM functionality is defined by ControlPlanUnits, called a `Capability`, that are received from clients. The Task Coordinator has a one-element FSM dominion list to manage these Capabilities. The `Capability` class supports `stop, start, execute,` and `isDone` methods.

For an application controller, there is list of capabilities that a Task Coordinator can use. Figure 19 illustrates a CNC application with `Capability` instances. When a `Capability` is executing, it coordinates the servicing of requests from the HMI. When the `Auto Capability` FSM is executing, it interacts with the Control Plan Generator.

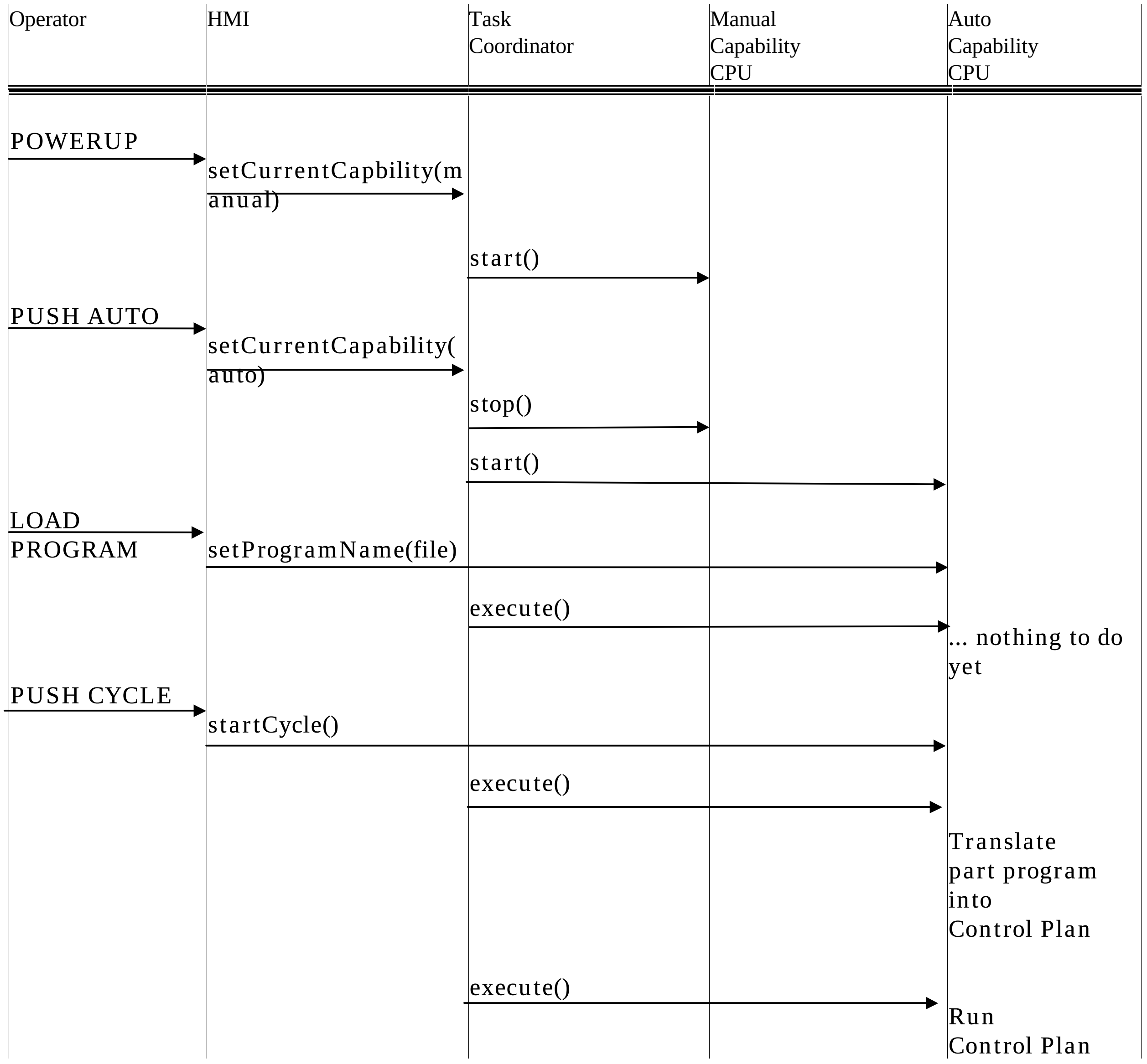


Figure 20: Task Coordinator and Capability Object Interaction Diagram

Figure 20 illustrates a sequence of operations that takes a milling CNC from manual mode to automatic mode. The diagram shows the use of `Capability start, stop,` and `execute` FSM methods. In the scenario, the controller comes up in the `manual` mode as loaded by the HMI at startup. Then, the operator pushes the `auto` button that causes the HMI to execute the `Manual Capability stop` method, and load the `Auto Capability` onto the Task Coordinator queue. That cycle, the Task Coordinator will see that the `Manual Capability` boolean `isDone` is True and will swap the `Auto Capability` FSM into the dominion FSM list. The operator action to Load Program will result in a program name loaded into the Control Plan Generator. When the operator pushes the cycle start button, it will cause the Auto `Capability` FSM to translate a part program and then start sequencing a ControlPlan generated by the Control Plan Generator.

4.2 DISCRETE LOGIC

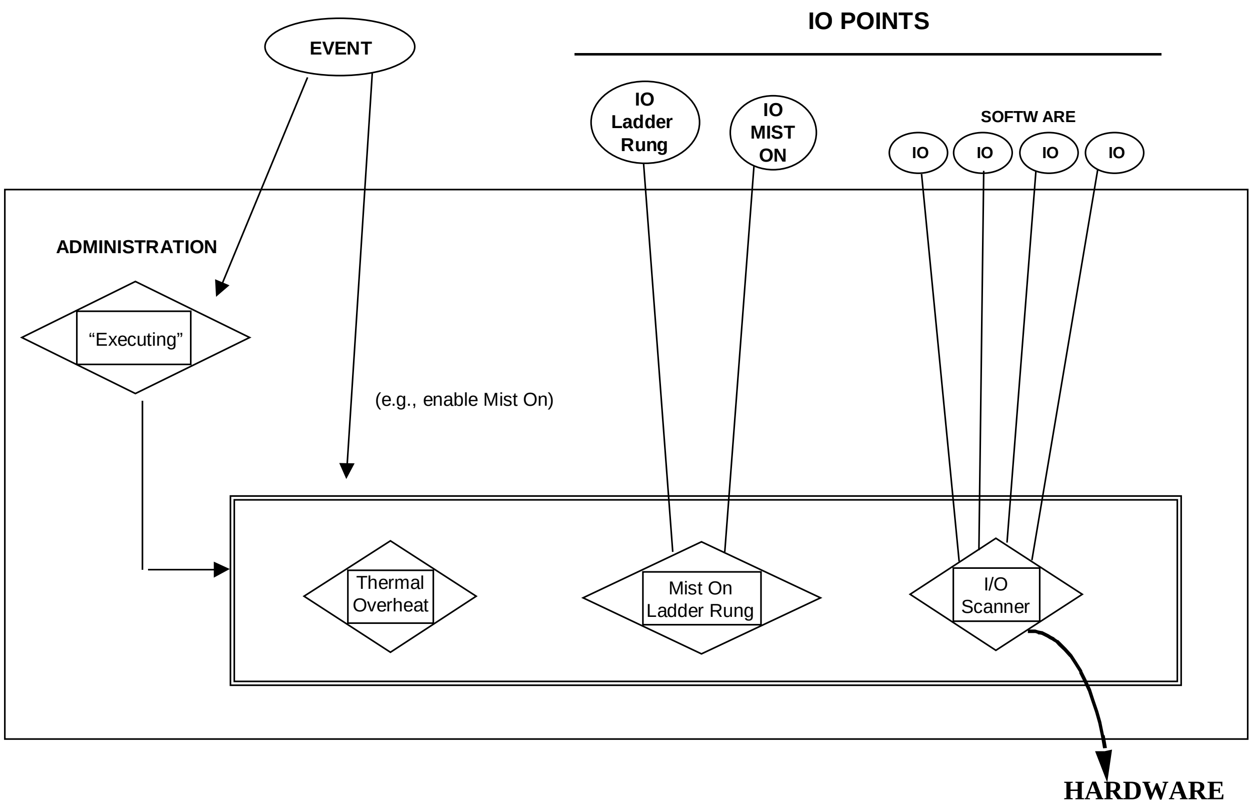


Figure 21: Discrete Logic Computational Model

The Discrete Logic module is similar to the Task Coordinator module in that it sequences and coordinates actions through dominion FSM. However, instead of a one-element dominion FSM, the Discrete Logic module has a multi-item dominion FSM list that is analogous to a scan list. In general, a Discrete Logic FSM could be coded in any of IEC-1131 languages and translated into ControlPlanUnits. Figure 21 illustrates the types of FSM that may be found on the Discrete Logic dominion list for a typical CNC milling application. An FSM to handle IO scanning would be expected. An FSM implemented as a Ladder Rung could be expected to handle a relay for turning a Mist pump on. Below is a sketch of the activity for turning the IO mist pump on.

```
mistPumpOnRung()
execute()
{ logic:  trigger relay to turn pump on
          wait till IO/pt says pump is on
          IOmist<- on;
}
```

At a higher level, a hardware-independent Mist FSM would be required to coordinate turning Mist `on` and `off`. Below is a sketch of pseudo code to sequence the Mist `on` operation. For coordination between FSM logic, polling or event-drive alternatives exist to wait for the IO Mist on activity to complete.

```
mistOnFsm()
{ "MistOn LR IO <- on" to turn LR=ladder rung on
```

```
          "subscribe to event that IO Mist On ==on"
          "wait for event or poll for IO point for Mist On == on "
          "done - deactivate FSM for scanning"
        }
```

4.3 AXIS

Axis module contains classes encapsulating the features pertaining to a single axis in a multi-axis control system. Figure 22a diagrams the relationship of the various classes. Classes are defined to provide a variety of setpoint control (e.g., following `AxisPositionServo, AxisVelocityServo, AxisAccelerationServo, AxisForceServo`), actions (e.g., `AxisHoming, AxisJogging`) and data (e.g., `AxisCommandedOutput, AxisRates, AxisLimits, AxisSensedState`). Figure 22b diagrams the finite state model of execution.

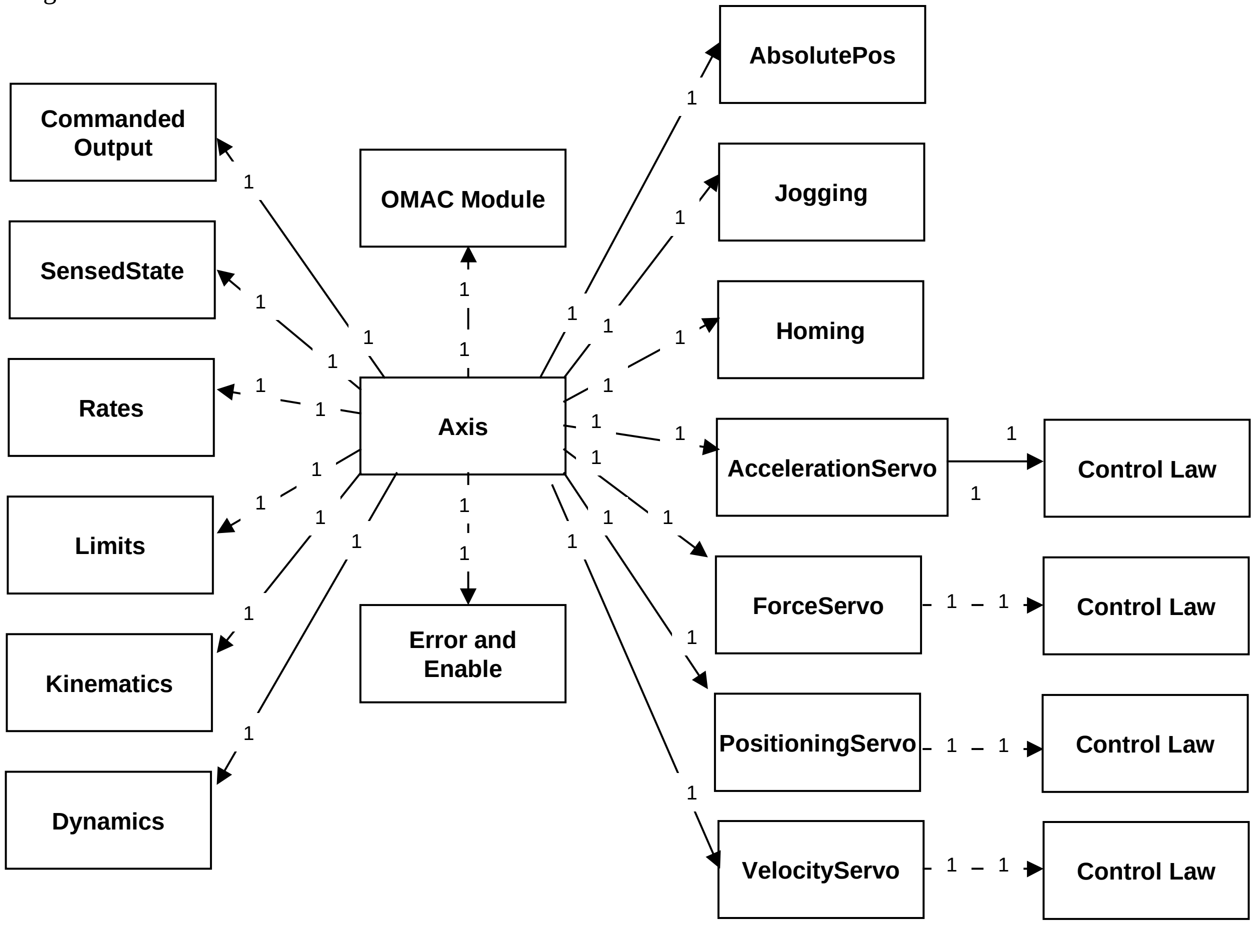


Figure 22a: Axis Class Diagram

The following list itemizes some basic open architecture requirements the axis module must support:

- nested control loops (e.g. position and velocity) using either derived feedback or additional sensors (e.g. encoders and tachometers)
- perform backlash compensation
- ability to incorporate any appropriate sensors and actuators available in the system

- provide settable error limits and "clamping" of various quantities in the loop. If error limits are exceeded, the loop will "safe" itself, and inform of an error condition.

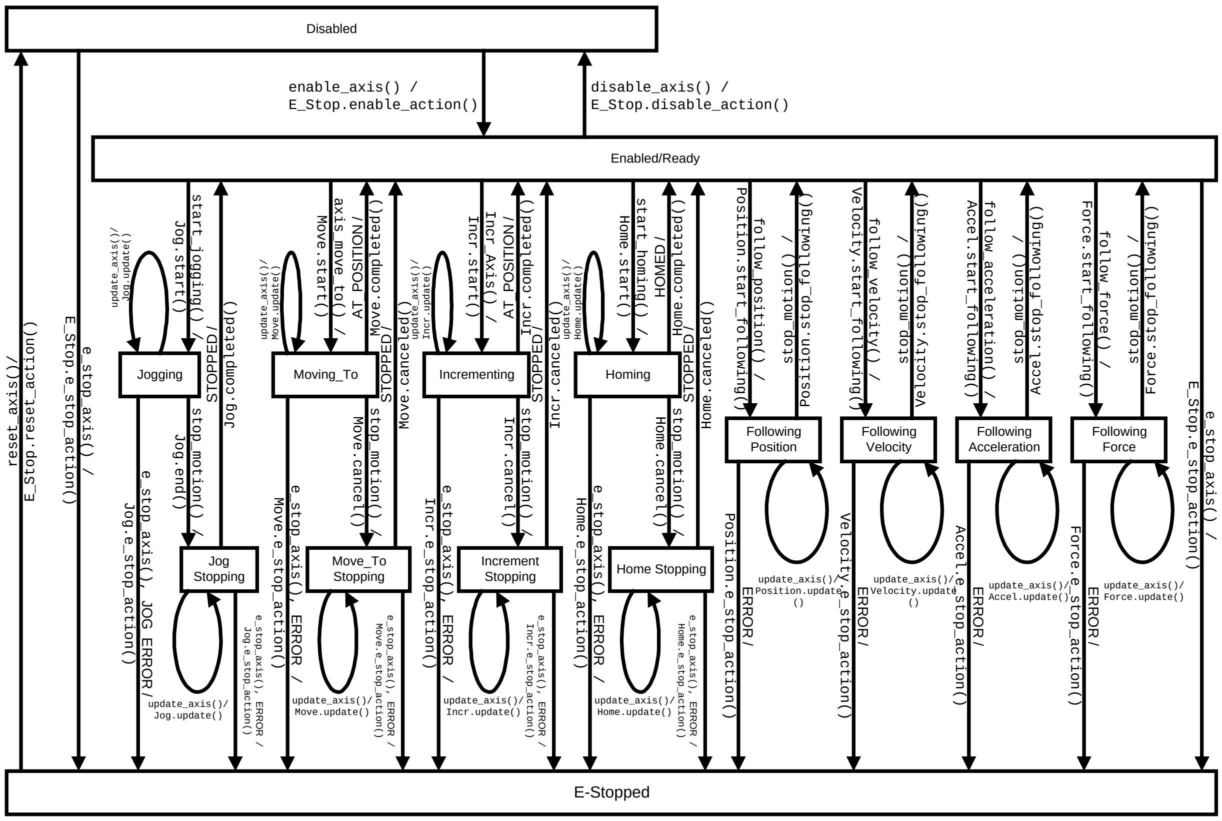

Disabled
enable_axis() / E_Stop.enable_action()
disable_axis() / E_Stop.disable_action()
Enabled/Ready
reset_axis()/ E_Stop.reset_action()
E_Stop.e_stop_axis() / e_stop_action()
start_jogging() / Jog.start()
STOPPED/ Jog.completed()
update_axis()/ Jog.update()
Jogging
stop_motion() / Jog.end()
e_stop_axis(), JOG ERROR/ Jog.e_stop_action()
Jog Stopping
update_axis()/ Jog.update()
e_stop_axis(), ERROR / Jog.e_stop_action()
axis_move_to() / Move.start()
AT_POSITION / Move.completed()
STOPPED/ Move.canceled()
update_axis()/ Move.update()
Moving_To
stop_motion() / Move.cancel()
e_stop_axis(), ERROR / Move.e_stop_action()
Move_To Stopping
update_axis()/ Move.update()
e_stop_axis(), ERROR / Move.e_stop_action()
Incr_Axis() / Incr.start()
AT_POSITION / Incr.completed()
STOPPED/ Incr.canceled()
update_axis()/ Incr.update()
Incrementing
stop_motion() / Incr.cancel()
e_stop_axis(), ERROR / Incr.e_stop_action()
Increment Stopping
update_axis()/ Incr.update()
e_stop_axis(), ERROR / Incr.e_stop_action()
start_homing() / Home.start()
HOMED / Home.completed()
STOPPED/ Home.canceled()
update_axis()/ Home.update()
Homing
stop_motion() / Home.cancel()
e_stop_axis(), ERROR / Home.e_stop_action()
Home Stopping
update_axis()/ Home.update()
e_stop_axis(), ERROR / Home.e_stop_action()
follow_position() / Position.start_following()
stop_motion() / Position.stop_following()
Following Position
update_axis()/ Position.update()
ERROR/ Position.e_stop_action()
follow_velocity() / Velocity.start_following()
stop_motion() / Velocity.stop_following()
Following Velocity
update_axis()/ Velocity.update()
ERROR/ Velocity.e_stop_action()
follow_acceleration() / Accel.start_following()
stop_motion() / Accel.stop_following()
Following Acceleration
update_axis()/ Accel.update()
ERROR/ Accel.e_stop_action()
follow_force() / Force.start_following()
stop_motion() / Force.stop_following()
Following Force
update_axis()/ Force.update()
ERROR/ Force.e_stop_action()
e_stop_axis() / E_Stop.e_stop_action()
E-Stopped

Figure 22b: Axis Module State Diagram

Within the Axis module definition, several issues exist.
One issue that occurs is mapping a single axis to multiple actuators. At this time, actuators are not an OMAC API module. The current resolution to the single axis-multiple actuator problem is to define specializations of the Axis base class to handle the multiple actuators.
Another issue is exposing the FSM methods. The reason for exposing the FSM methods is so that such FSM classes (such as AxisAccelerationServo) can be a replaceable component within the system. Different implementations of the class definition would adhere to the interface.
Another issue is what happens when a method is invoked in the wrong state? For example, suppose an ACCEL_EVENT occurs when in the HOMING state and there is no defined transition? The first possible action is to ignore the event, but this is poor system design. The preferable option is to throw an exception, but OMAC API has not enumerated exceptions yet.
Another issue is how is a Control Law attached to a servo class such as Position, Velocity, Acceleration, or Force? The answer is to use class specialization to extend the base class to contain Control Law component. For example, `AxisAccelerationServo` may not need a control law component if connected to SERCOS drive so that it uses the specified **Base Class**:

```
        interface AxisAccelerationServo(){}
```

For software servoing, an Axis class specialization would be defined that incorporates a control law component using a **Derived Class**:

```
        interface CLAxisAccelerationServo() : AxisAccelerationServo
        { ControlLaw controllaw;
        };
```

4.4 AXIS GROUP

The Axis Group module is responsible for transforming an incoming MotionSegment into a sequence of equi-time-spaced setpoints, incorporating mechanism and process knowledge, and coordinating the motions of individual axes.

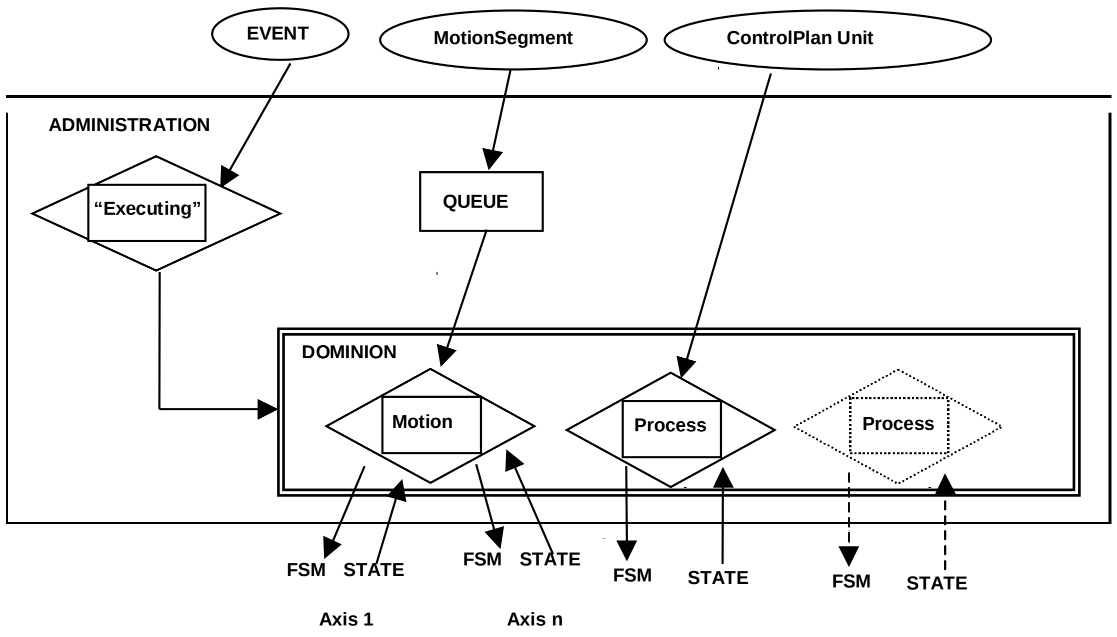


Figure 23: Axis Group Module

Figure 24 shows the class diagram for the Axis Group module. The Axis Group module consists of the following classes:

AxisGroup
is the coordination module that has the following responsibilities:

- kinematics coordination transformation
- dynamic offset (e.g. sensing inputs) and overrides
- multi-axis coordination
- blending and block look-ahead
- feedhold
- operation stop
- execution on compensation look-up tables

- path or rate-control modification based on sensor-feedback (including operator overrides)

PathElement is the class definition to define the motion geometry.

Rate is the class definition to define the motion rates and limits along a path.

VelocityProfileGenerator is generates time-based steps along a path. Time-scaling of motions is performed along a path based on rate-control (desired velocities, accelerations) or time-duration. Includes control of acceleration/deceleration.

MotionSegment is derived from ControlPlanUnit to define a motion-control FSM. Contains references to VelocityProfileGenerator, PathElement and Rate classes.

Figure 23 illustrates AxisGroup computational model. The AxisGroup receives MotionSegment CPUs that define the motion. MotionSegments are queued to allow blending or lookahead. Process CPUs are required for integrating sensing and mechanism knowledge. Process CPUs have tightly-coupled associations with the Kinematics Module (for mechanism knowledge) and the Process Model (for sensing and application specific knowledge).
The Kinematics module describes the relationship of the machine and part to a world coordinate system. Such information could include a relative offset to the machining bed and another offset to a part origin. Obstacles such as fixtures would also be included within this description. The Process Model integrates operator and sensor feedback into the trajectory motion. This feedback can be used to modify the rate-control.

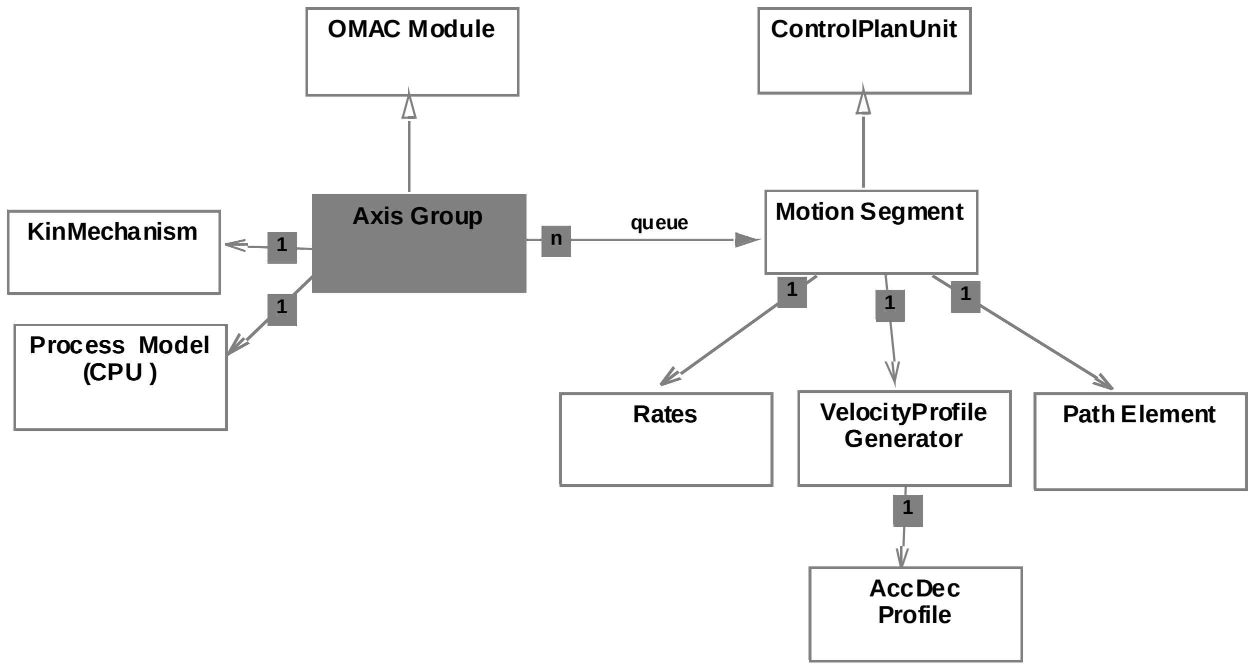


Figure 24: Axis Group Class Diagram

Discussion on some issues and procedures common to Axis Group operation follows.
Concerning the issue of power management, it is assumed to be user-specifiable by the ControlPlanUnit within some timing constraint. For example, a sequence to set a bit, wait 3 seconds and then check brakes can be embodied with a ControlPlanUnit.
A common Axis Group procedure is to stop running, change a broken tool and then resume operation. For this Axis Group module has API to save motion queue context and then restore it. An underlying

assumption is that if there are other queues internal to the Axis Group (e.g., lookahead, blending) that these too will be saved and restored.
The issue of standard stopping procedures is fundamental to a standard Axis Group API. OMAC API proposes three modes to stop:

hard stop is a stop with max deceleration rate. Also called abort.

pause is a stop on the path as defined by the KinematicPath in the MotionSegment.

hold is a stop at end of segment as defined as the next increment provided by the Velocity Profile Generator.

There are four recovery modes from stop:

resume start motion from the current point

skip skips to the next segment

flush flushes all segments on the motion queue

restore after a motion queue save after stopping, with possible intervening motions (such as to change a broken tool or backing out), the motion queue can have its previous context restored.

A standard Axis Group `estop` is not addressed because of the many different interpretations of estop. For most purposes, a hard stop and estop are identical.
An issue of axis grouping and creating higher level objects can be resolved by defining a higher level AxisGroup module. Some grouping issues include:

- error grouping - the AxisGroup has an `inhibit()` API for error recovery (e.g., 2 live axis with 3 dead axis)
- power sequencing - TBD
- power chain grouping - TBD
- kinematic grouping is done with the Kinematics module.

4.5 PROCESS MODEL

The Process Model is responsible for dynamic control modifications. The Process Model exists to encapsulate the application- or domain-dependent knowledge. For example, the Process Model for machining would incorporate feedrate override, but the Process Model for a pick-and-place robot would probably not. Some typical Process Model dynamic modifications associated with machining include:

- feedrate override
- spindle speed override
- path offset (normal adjustment for cutter compensation)
- tool length offset (dynamically modified based on tool wear, not just tool change)
- data logging flag
- cycle interruptions (e.g., estop, hard stop, feed hold)

The Process Model is generally associated with the Axis Group in order to modify the current motion. The relationship between the Process Model, Axis Group and MotionSegment modules can vary. This variation greatly affects the openness flexibility.
In the dependent relationship, the Axis Group and the Process Model know each other's API a priori. For example, suppose the Axis Group understands that the Process Model supports feedrate override via a `getFeedrateOverride()` API. Then, the Axis Group can retrieve the current feedrate override value in order to modify the current MotionSegment's feedrate.

The dependent relationship is flexible if all the required shared variables between the Axis Group and the Process Model exist. For example, if the feedrate override had been under operator-control, a user may replace the Process Model with a custom module to change the feedrate override based on some force/torque sensing. However, problems arise if the user wants to add a cutter compensation normal to a MotionSegment and a pre-defined API does not exist. Now, the Axis Group or each MotionSegment must be rewritten to incorporate this modification.
In an independent relationship, the Axis Group and Process Model coexist without a priori knowledge of each other. For this case, OMAC API is proposing to allow the Process Model to send CPU to Axes Group so that these CPU can modify the current motion CPU (i.e., MotionSegment). Consider the following alternatives where the user wants to integrate a new probe into the control system and coordinate when the motion controller to start recording points.

1. For the dependent relationship, a solution is to rewrite the Axis Group to accept a "log data" flag and then record data.
2. Another possibility is to mandate that every control plan be rewritten to contain a "log flag."
3. In the proposed independent relationship, the Process Model would generate a CPU that is sent to the Axes Group which is executed every cycle to actually log data based on an external reference to a "log flag."

In the independent relationship, countless other real-time modifications could be applied by ControlPlanUnits within the AxisGroup (as well as the Kinematics Module). The ability to extend the controller based on evolving sensor-based applications was a primary OMAC requirement. Hence, the necessity to support the Process Model independent relationship.

4.6 KINEMATICS

Kinematics refers to all the geometrical and time-based properties of motion[Cra86]. The OMAC API uses a graph representation to model the geometrical aspect of kinematics. The model is flexible enough to handle kinematic chains and kinematic hierarchies. Figure 25 illustrates the terminology used to model the geometric kinematics. A **KinStructure** describes the geometry of an axis link. A KinStructure has a **Base Frame** (generally used to model compensation) and a **Placement Frame** to model the axis link transformation. The BaseFrame is useful as an offset to model spindle growth or other compensation variables. When no compensation is planned, the BaseFrame location equals the placement frame location. A **Connection** models the relationship between two KinStructures using a **from** KinStructure and a **to** KinStructure. A **KinMechanism** models a kinematic chain as a series of connections. The OMAC API kinematic model allows recursive kinematic definition. A KinStructure can itself be a kinematic chain modeled as a KinMechanism. This recursive definition allows a static kinematic chain to collapse into a single pre-computation.

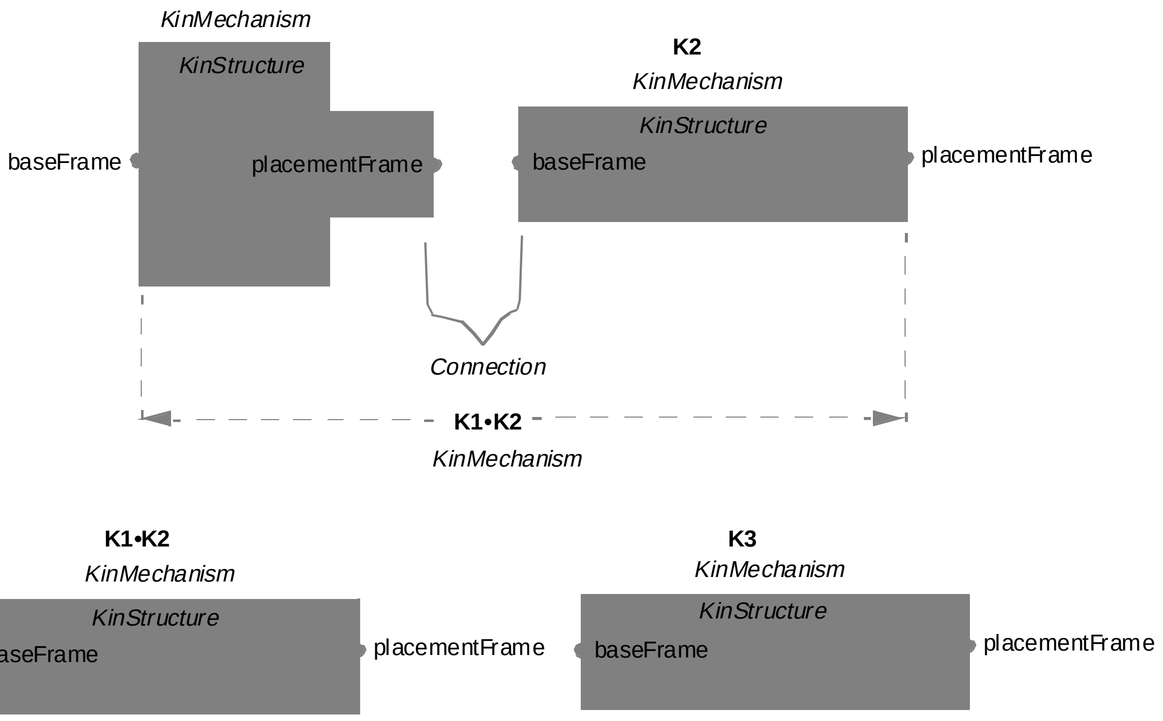


Figure 25: Kinematics Model

A KinMechanism is responsible for computing the forward and inverse kinematics. A KinStructure contains the following information necessary for these calculations:

- transform
- static or dynamic link
- home state
- link model - translational, prismatic, rotational

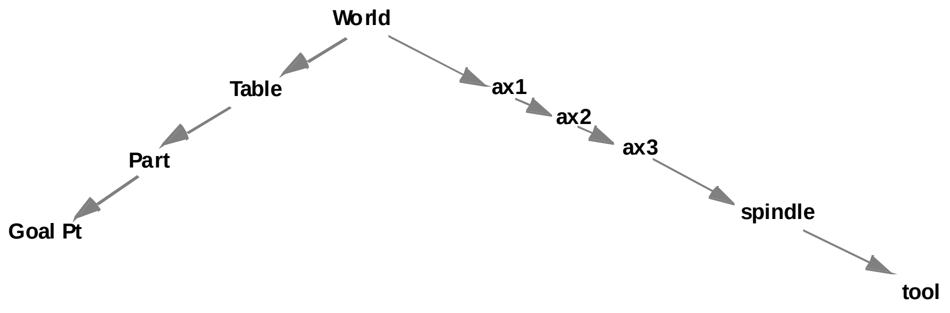


Figure 26: Kinematics Example

As an example, consider the case of a three axis machine with tool to mill parts on a table given a part offset. The machine tool kinematic chain contains a spindle KinMechanism to model spindle growth. Figure 26 illustrates the chain of KinStructures `World, Table, Part, Goal Pt, a1, a2, a3, spindle, and tool` to model this example. We will assume the table is motionless. The following code sketches an OMAC API kinematic model for this example.

```
// Declarations
KinMechanism worldKM, axKM[3], spindleKM, toolKM;
KinMechanism  overallKM;                // collection w-a1-a2-a3-spindle-tool kinematic chain
KinStructure * worldKS, * axKS[3], * spindleKS, * toolKS;
Transform Identity = new Transform (1, 0 , 0, 0, 0 , 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1);

// Define KinStructures and embed in KinMechanism
  worldKS= new KinStructure();
  worldKS->setBaseFrame(&Identity);
  worldKS->setPlacementFrame(&Identity);
  worldKM.setConnections(NULL); // trivial case, does not contain KinMechanisms
  worldKM.setKinMechanisms(NULL); // trivial case
  worldKM.setKinStructure(worldKS);

  axKS[0]= new KinStructure();
  axKS[0]->setBaseFrame(&Identity);
  axKS[0]->setPlacement(/*some transform*/);
  axKM[0].setConnections(NULL);
  axKM[0].setKinMechanism(NULL);
  axKM[0].setKinStructure(axKS[1]);
  ...

// Set connections
  Connection c[5]
  Connections connections;
  c[0] = setFrom(w);
  c[0] = setTo(axKM[0]);
  c[1] = setFrom(axKM[0]);
  c[1] = setTo(axKM[1]);
  c[2] = setFrom(axKM[1]);
  c[2] = setTo(axKM[2]);
  for(int i=0; i< 5; i++) connections.add(c[i]);

//Define overall KinMechanism
  overallKM.setConnections(connections);

// Modification of axis values
  axKM[0]->getKinStructure()->setPlacementFrame(&newFrame1);
  axKM[1]->getKinStructure()->setPlacementFrame(&newFrame2);
  axKM[2]->getKinStructure()->setPlacementFrame(&newFrame3);
```

The importance of the Kinematics module is not only calculating the forward and inverse solutions, but also providing a mechanism to perform offsets and compensation. A few examples will be considered.

Relative Positioning The equivalent to the RS274D Absolute and Relative positioning cases are handled by two separate KinMechanisms.

Change Tool Suppose a tool table is to be maintained. A KinMechanism for each tool in the table will need to be defined. For a tool change, a new reference to the new tool is substituted for the `tool` KinMechanism in the `overall` kinematic chain.

```
KinMechanisms tool[100];
toolKM = &tool[2];
```

Tool Length Offset Consider the case in which tool length offset is changed to compensate for tool wear, reconditioning, depth of cut (rough, finish), or dry run. In this case, the tool KinStructure PlacementFrame is modified to reflect the change. For example, changing column 4 row 3 (i.e., the z value) of Tool displacement frame will change the offset.

```
toolKM->getKinstructure->setPlacementFrame(newFrame);
```

Spindle Growth A majority of variation during machining is attributable to spindle growth. The example kinematic chain contained a Spindle KinMechanism to model spindle growth. Modifying the spindle BaseTransform based on spindle growth achieves good correction at a modest cost.

```
spindleKM->getKinstructure->setPlacementFrame(newFrame);
```

Axial Growth Consider the case in which an axis is growing in length as the leadscrew mounting bearings heat up during machining. In this case, the axial member is growing in length. Next to the spindle, axis growth is the most common and cost-effective compensation technique. In this case, an axis KinStructure baseFrame is changed.

```
axKM[0]->getKinStructure()->setBaseFrame();
```

Cutter Radius Compensation Consider the scenario in which path modification is based on cutter radius compensation. Assume the need to apply a normal offset to the pre-defined curvilinear kinematic path from point A to point B.

In the static case, the entire kinematic path can be recomputed as specified based on a flag. In this case, responsibility is delegated to the CPU to handle the change from the nominal path to the compensated path.
In a quasi-static case, suppose the cutter radius is computed on-line by some process controller or sensor to do radial compensation to adjust the path. In this case, a radial compensation value is input to the Kinematic Path class and it returns a corrected value.

In the dynamic case, the modification is to the next increment of the interpolated path of the current MotionSegment. This would be achieved by calling the KinematicPath (i.e., `KP`) with the normal offset.

```
KP->applyNormalOffset(&normalOffset);
```

Configuration Solution rules for configuration such as up/down elbow or redundant links are handled by class specializations.

Update Unresolved is the responsible module and mechanism used to update dynamic (e.g., axis) values.

4.7 IO SYSTEM

The purpose of the IO system is to provide a uniform interface to **Physical** or **Virtual** input or output points in the system. The `IOPoint` class defines the uniform interface and hides the details of the underlying hardware interactions. An example of an IO Point is a DAC on a multiple DAC digital to analog output card. The `IOPoint` base class manages a single value, and provides an interface for reading and writing that value. The IO Point base class contains `readValue()` and `writeValue()` methods.
Each `IOPoint` may be accessed individually but `IOPoints` are controlled by an IO System. An IO System is a module consisting of one or more IO Points, grouped together because they share some resource (either hardware or software). There can be many IO systems in a controller (e.g., Sercos, D/A board, etc.)

4.7.1 IO NOTIFICATION

Each IO System may optionally contain Callback Notification and Callback Handlers.
Callback Notification object(s) provide a mechanism for other modules to be informed when some internal activity has taken place in the IO System. Each Callback Notification object contains a list of **Callback Handlers** to be activated on the desired event. This allows multiple modules to be informed on an IO System state change. The Callback Handlers are entered into the Callback Notification object's list at system integration time. For example, a Callback Notification might exist to inform other modules when the values associated with an IO System's IO Points have changed.

The IO System may also by notified by Callback Handlers. A callback by other modules would inform the IO System that some event has occurred. For example, the IO System may contain a Callback Handler to be activated when it is time to sample all of its IO Points' inputs.

4.7.2 IO CONFIGURATION

OMAC API uses a **Presentation IO model** in which each IO system (as one of many in the system) creates a series of `IOPoints` that other objects in the system access via references (or handles). This differs from an **Attachment IO model**, where each object in the system creates an `IOPoint` and attempts to attach the `IOPoint` to some hardware.
As an analogy to differentiate between the Presentation and Attachment models, consider an IO Point filled with bytes from a file. In the Attachment model one opens a file, and uses a copy of a device driver to read bytes from the file. To read bytes within the Presentation Model, the assumption exists that a separately running IO System module has already opened the file and has presented a byte IOPoint for system-wide access. In IOPoint presentation, any number of objects in the system can access the byte IOPoint buffer, which is updated by its IO System.
The Presentation IO model assumes that an object uses an ASCII naming and lookup service to connect to an `IOPoint`. This IOPoint connection is performed at configuration time. However, at this time the OMAC API does specify a configuration API for IO Point connection.

4.7.3 IO CUSTOMIZATION

Clients of I/O modules may wish to customize their interaction. OMAC API has defined `IOPoint` classes for the major types (e.g., `short, long, float, double`). THE FOLLOWING SECTION DISCUSSES ISSUES OF IO CUSTOMIZATION, HOWEVER, IO CUSTOMIZATION IS NOT WITHIN THE SCOPE OF THE OMAC API SPECIFICATION EFFORT.
Customized IOPoint classes can be derived based on specializations (such as a read-only IOPoint) as well as methods to manipulate the value's units, name, type, and other properties. These methods may be further supplemented with additional IO system-specific methods to configure IO waiting, synchronization, as well as low-level communication protocols.

IO mechanism Since IO Systems will probably represent a particular piece of hardware plugged into the system, customization of the io mechanism is also desirable to provide non-generic, hardware specific interfaces. These interfaces are referred to as **Control Interfaces**, and are somewhat analogous to the Unix `ioctl()` function calls. Unlike the other interfaces provided by the IO System, there is no fixed form for these interfaces. They exist to provide access, by knowledgeable software modules, to low level hardware functions that cannot be put into the generic forms used by the other interfaces. They would probably be used primarily by diagnostic software. Use of these interfaces by other modules, which are intended to be generic, is not recommended, since their use would prevent the module from using any other IO System that did not provide an identical interface.

IO Data Handling Customization of data handling requires some special characteristics. For example, the IO module tailors the service to offer different sampling strategies, transfer protocol and data age. The following is a list of customized IO data and protocol characteristics:

Sampling Event IO system characteristic

- ON-DEMAND,
- ON-TRANSITION,
- ON CLOCK

Data Age

- Sample Num

- Sample Num + N
- Current Reading
- Current Reading + N

Transfer Type

- Synchronous : wait until complete
- Synchronous : wait up to specific time
- Asynchronous : initiate and specify complete event handler
- Asynchronous : continuous with completion event handler

4.7.4 IO META DATA

A major issue with handling IO is the aspect of **IO Meta data**. IO Meta data correlates the IO to the device, for example, "what board is this IO Pt associated?" IO meta data incorporates knowledge useful for maintenance and diagnostics. In many ways, IO Meta data is the bigger part of IO. OMAC API has not specified a formal IO Meta data. OMAC supports the notion of an **IO registry** that would include such IO Meta knowledge as:

- IO as shared across the system
- IO as used by different clients
- IO as defined from a physical aggregation
- IO grouping for efficiency (e.g., an IO group is clustered on one board)
- physical device to logical IO mapping (e.g., a device has 4 analog inputs, 4 analog outputs, 16 discrete IO).

Overall, IO registry would consist of a container of devices as well as a container of IOPoints. Each IO point keeps a reference to a device as well as a device specific set of data which is needed to access that IO point (e.g. which bit, how wide, what type). This format information is retrieved at start-up and is returned in the form of a reference handle. This could allow a configuration utility to build a GUI and supply the data, which is then stored in the registry.

Interaction with an IO registry is as follows. At configuration-time, IO registry functions include service to bind a device to IO name (i.e., device maps into a board, point, type) and this builds the internal tables. At initialization, the IO registry return handles for names for efficient access during execution. At runtime, IO has facilities for the `read` and `write` of grouped outputs and single outputs; as well as the `read` of grouped inputs and single inputs.

4.7.5 IO ISSUES

The OMAC API has not specified a solution to the issue of whether an IOPoint tells whether it is input or output. A simple resolution would have an IO derived type from IO_PT used by configuration for mode differentiation and type checking.

The OMAC API has not specified a solution to the issue of forcing IO and machine simulations through IO points.

4.8 CONTROL PLAN GENERATOR

The Control Plan Generator is responsible for reading and translating programs, which represent machine operation and tooling. The Control Plan Generator can either translate the entire file or provide instructions a statement at a time. The Application Programming Interface to the Control Plan Generator is not concerned with the format of the part program itself, but with syntax and translating program elements into Control Plan Units. Functionality of the Control Plan Generator includes:

- reading existing program files, which contain statements in the format understood by the translator but not standardized by the OMAC API
- translating part program statements into ControlPlanUnits
- correlating source knowledge about a program, (e.g., current line number, active statement) with a ControlPlanUnit.

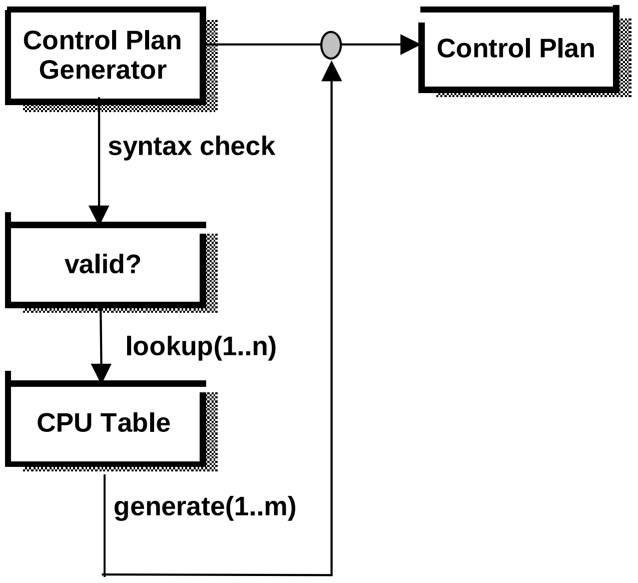


Figure 27: Control Plan Generator

Figure 27 presents an overview of the Control Plan Generator. The Control Plan Generator is responsible for syntax checking of the part program. If the syntax is valid, the Control Plan Generator generates one or more Control Plan Units for each line of the part program. The Control Plan Generator is responsible for correlating part program source information (such as line numbers) with each ControlPlanUnits. Multiple source lines may be active with one ControlPlanUnit.
Table lookup to translate a part program statement into a ControlPlanUnit can be done in a number of ways. OMAC API does not specify a standard lookup technique. One option to perform this lookup would be to associate each part program statement with a separate translation object that queries or is given the knowledge it requires. Each translation object would support an identical `translate()` interface. Another possibility is to use “flat” canonical functions instead of “object-oriented” translation classes. Any number of indexing or bidding schemes is also possible.
It would be desirable for Control Plan Generators to generate generic machine-independent Control Plans. Then, translation from generic ControlPlan Unit to a machine specific ControlPlanUnit could be done based on the specific objects in the system. For Control Plan machine-independence, adding a machine profile (e.g., 3-axis versus 5-axis) and a Control Plan should produce identical results.
Concerning the issue of part program portability, OMAC API does not expect the ControlPlanGenerator to produce a machine-independent ControlPlan. This flexibility is difficult to attain and the OMAC API determined that defining a Neutral Language Definition was outside the scope of the current effort.

4.9 HUMAN MACHINE INTERFACE

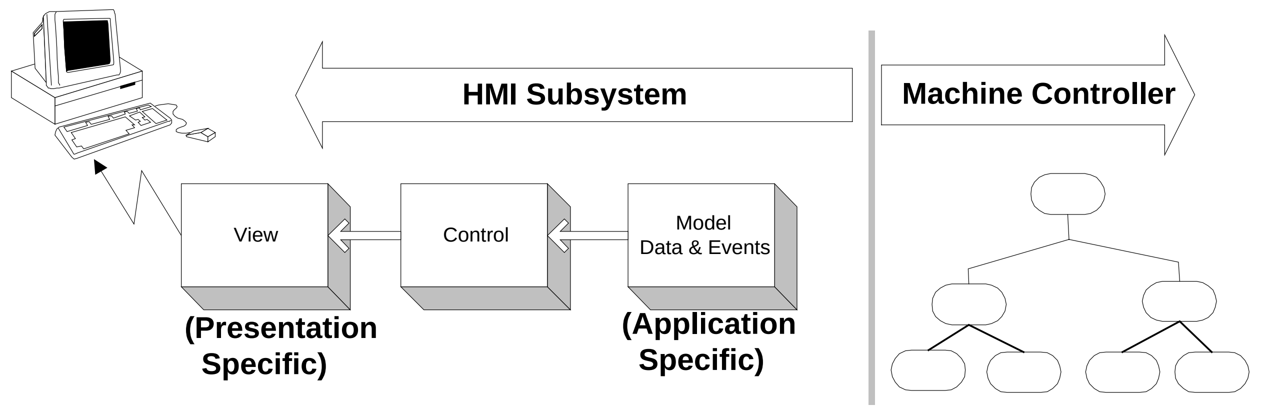


Figure 31: MVC Design Pattern

The Human Machine Interface is responsible for the connection between the controller and a human-monitoring subsystem. The object-oriented design pattern called the **model-view-controller (MVC)** will be used as the HMI reference model [GHJV94]. Figure 31 shows the relationship of the different control and human aspects within the MVC pattern. The MVC **model** "M" defines the state of the HMI objects. The MVC **View** "V" corresponds to the front-end or visual presentation with which the user interacts. The MVC **controller** "C" is not the same as the motion controller, but refers to an object that controls a View object in such a way that it responds to user input and delivers output.

Some clarifying objectives concerning the OMAC API HMI are in order. The goal of the OMAC API is to define an HMI specification that is independent of the visualization medium (i.e., V), the data entry mechanism, the operating system, or the programming language. The primary OMAC API objective is to specify a technology-neutral data and event model (i.e., M) for exchange of information between the Human subsystem and the Application Controller. The OMAC API would like to encourage the bundling of a control component with an HMI viewing component (i.e., supply component plus V & C). The OMAC API is not concerned with the "look and feel" of a HMI. The "look and feel" of an HMI is generally application-specific.

To understand the HMI for OMAC API, the elements M,V, and C will each be reviewed.

Model

The primary emphasis of the OMAC API is to define a model "M" API that allows the exchange of data and events. The traditional standardization effort for "M" relates to the data collection or back end that would be defined as a Dynamically (or Shared) Linked Library.

The desired HMI "M" functionality is best understood in the context of simple problems. Three canonical "M" problems exist that an HMI module must be able to handle. First, the HMI must have the capability for **solicited information reports** about the state of the controller, such as current axes position. Second, the user must have **command capabilities** such as the ability to set manual mode, select an axis, and then jog an axis. Third, the user must be alerted when an exception arises, in other words, handle **unsolicited information reports.**

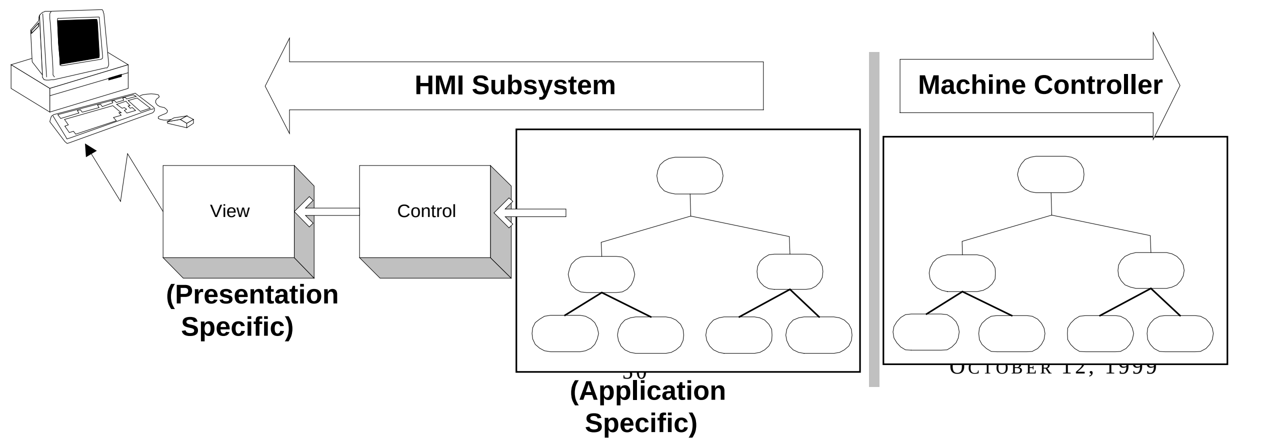


Figure 32: HMI “M” Mirrors Controller

For “M” functionality, OMAC API specifies that every controller object has a corresponding HMI object “mirror”. Figure 32 illustrates an “M” that mirrors an application controller where each mirror object in the HMI has a reference to its companion object in the controller. The mirror object can then use the reference to get/set data, or to invoke methods to initiate events. In other words, these HMI and controller objects have identical interfaces for data manipulation and event-initiation. For event-notification (unsolicited reports), this is a special problem that really has to deal with the infrastructure. (See section on event-handling.) Compared to a conventional “M”, the use of get data mimics a data base copying the desired viewable values from the controller.

The major mirror assumption is that HMI objects communicate to control objects via proxy agents. An analysis of how the HMI mirror works will be developed.

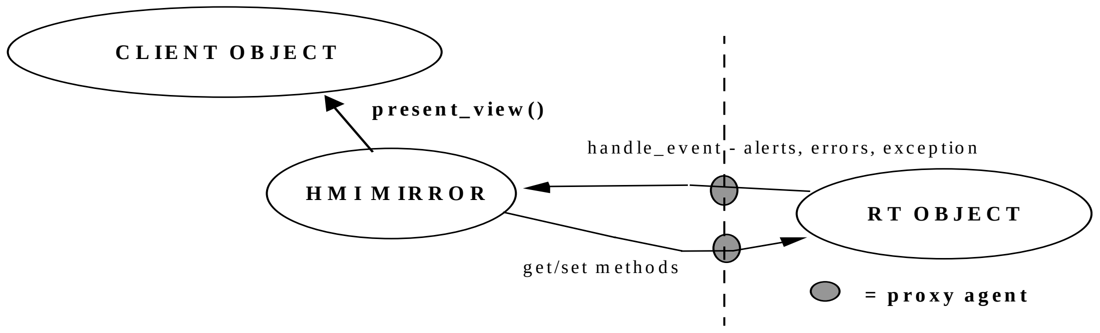


Figure 33: Close-up of HMI Proxy Interaction

1. To handle the information report functionality, an HMI mirror acts as a remote data base that replicates the state and functionality of the controller object and then adds different presentation views of the object. These HMI mirrors are not exact mirrors of the controller state, but rather contain a “snapshot” of the controller state. Figure 33 illustrates the interaction of the HMI mirror and the control object. In the basic scenario of interaction, the control object is the server and the HMI mirror object is the client. Each HMI mirror uses the accessor functions of “get” and “set” to interact with the control object. Notice that each host controller object and its corresponding HMI mirror have a proxy agent to mediate communication.

2. To handle command functionality, the HMI mirror contains the same methods as the controller object so that a command is issued by invoking a method remotely.

3. To handle abnormal events when not polling, an HMI mirror must serve as a client to the control object so that it can post alert events. For such unsolicited information reports, the control object uses an event notification function, `updateCurrentView`, in which to notify the HMI mirror that an event has occurred. This notification in turn may be propagated to a higher-authority object.

View

The MVC view “V” deals with the presentation medium, for example, whether it is a “V” for a GUI or a teach pendant. As previously stated, the OMAC API is not concerned with the “V” aspect pertaining to “look or feel” of a HMI.

Of importance to the OMAC API specification pertaining to the MVC control "V" is the aspect that deals with **data views**. Different data views correspond to different modes of presentation. For example, there can be a view for configuration, calibration, error handling - as well as normal operation. In addition, the view "V" can be used to offer different screens to different levels of authority, such as for operator, maintenance, or systems engineer.

Given this emphasis on data views, the OMAC API defines the following "V" methods to handle the different expected data views.

```
interface HMI
{
  void presentErrorView();
  void presentOperationalView();
  void presentSetupView();
  void presentMaintenanceView();
};
```

The association of data views along with a control component offers a strong potential for "complete" off-the-shelf integration. Instead of buying a control component with a standalone calibration program, a control component would come with a control view component. Then, just as the control component can be integrated into the application controller, so too can its corresponding control view component be automatically integrated into the controller presentation. As an example of this technology, a tuning package can provide a Windows-based GUI to do some knob turning. Another example, is a tuning package that offers this capability to be plugged inside a Web browser. With this development, unlimited component-based opportunities are available.

The MVC controller "C" discussion will further explore the coupling of a control component with a view component for automated system development.

Controller

The MVC controller "C" is responsible for controlling the views presented to the user. In Figure 33 the control object is represented by the Client which changes views based upon the use of different MVC "V" methods (i.e., the different types of `presentView` methods - see above). However, the Client is not bound to use the mirror "V" methods when constructing presentation views. There exists a range of approaches that the MVC "C" Client can use when controlling the user presentation - from **least** to **most customized**.

In Figure 33, the Client is using the HMI mirrors to present the view. Exclusive use of the HMI mirrors presentation views could be considered the **least customized** option. The Client is bound to the view that the control vendor supplies. However, the benefit is that Client-builder has the least amount of work to do. In the least-customized, the following concepts apply.

- each object contains methods which can display the object in one of several views
- each of these methods can be given display real-estate by the caller
- each object may recursively use its real-estate to display objects which it uses
- users may override these methods, if desired, for minor customizations

At the other extreme, a more monolithic, all-powerful Client could ignore the HMI mirror presentation views altogether. This approach could be considered the **most customized** option. In this case, the monolithic Client uses the HMI mirrors for data manipulation purpose only and the Client presents its own view of the data. The Client can develop any view it wishes. However, the Client-builder has the greatest amount of work in doing so. In the most-customized, the following concepts apply.

- a “super-object”, which is aware of all of the other objects (and their types) is created
- the “super-object” contains all code needed to create displays
- the “super-object” may use the default methods if desired
- the “super-object” may implement exactly the screens desired

Today, the MOST CUSTOMIZED approach with its monolithic, all-encompassing, micro-management of the controller presentation is most prevalent. This monolithic approach is most common mostly out of default because few, if any, control components provide HMI views. It is hoped that OMAC API MVC “V” methods will help change this situation.

4.10 MACHINE TO MACHINE INTERFACE

MMS (Manufacturing Message Specification) is an OSI application layer protocol designed for the remote control and monitoring of industrial devices such as PLCs, NCs or RCs. It provides remote manipulation of a controller that includes the following services:

Variables can be simple (booleans, integers, strings...) or structured (arrays or records). MMS variables can be read or written individually, in lists (predefined or explicitly defined).

Programs can be remotely started, stopped, resumed, killed.

Transfer allows for the download or upload of areas called domains, which can contain code, data or both.

Semaphores define two classes of semaphores, which can be used to ensure mutual exclusion or synchronization of processes.

Events provide services for attachment of an action to an event and enrollment of calling or another process to receive the event notifications.

The goal of the OMAC API is to provide an object oriented programming interface for remote functionality. It is expected that the baseline functionality would be the primary MMS capabilities. The following MMS functionality was determined to be mandatory:

- initiate
- conclude
- cancel
- unsolicited status
- solicited status
- getnamelist
- identify
- read
- write
- information report
- get variable access attribute
- initiate download sequence
- download segment
- terminate download sequence
- initiate upload sequence

- terminate upload sequence
- delete domain
- get domain attributes

It is expected that the implementation of an OMAC API MMI interface would offer a convenient programming interface that is not restricted to use MMS for its underlying communication technology. As envisioned, the internal controller infrastructure could be an ORB, while the external communication could be ORB or MMS based.

5 DISCUSSION

OMAC API has developed an API specification that is scaleable for the system design, integration and programming for systems ranging from a single-axis device to a multi-arm robot. The OMAC API working group's initial focus was to establish programming requirements for precision machining. Applicability to other control environments may be possible but is not guaranteed. The OMAC API primary focus has been to define Application Programming Interfaces for certain modules that the ICLP community routinely wants to upgrade. In addition, the workgroup has defined an assembly framework with which to connect these modules.

OMAC API has posted other papers to describe related information on life cycle, general computation models, and control models. For more information, see the Wide World Web at the Universal Resource Locator address:

`http://isd.cme.nist.gov/info/omacapi`

Within the OMAC API home page, there are hyperlinks to send comments, and to review comments and responses.

The OMAC API effort is not finished. The focus of effort has been to develop module APIs and to create a methodology for assembling and reconfiguring modules. Areas outside the OMAC API initial thrust areas or areas of disagreement include:

- performance evaluation
- validation and verification
- resource profiling
- configuration construction
- error handling and error propagation
- scheduling
- module timing profile
- event handling
- machine-to-machine interface (MMI) is outlined but incomplete.

The remaining sections will discuss some of the issues in dispute or issues that remain unresolved.

5.1 SCHEDULING AND UPDATING

Hard real-time is fundamental to a controller operation and falls under the auspices of the Real-Time Operating System. Often, commercial RTOS only support priorities to manage task scheduling. This technique is flawed. It would be preferable if one could perform periodic updating by assigning periods and a time quantum to tasks. However, the OMAC API could not agree on a single solution to this problem. This section will discuss one of many solutions.

OMAC modules can run as asynchronous or synchronous tasks. Asynchronous tasks are event-driven which is discussed in the next section. Synchronous tasks are expected to run periodically at a fixed frequency and bounded duration. Execution of a synchronous task can be either handled externally by a scheduling updater or internally by self-clocking. The remainder of this section will develop the concept of a Scheduling Updater module.

OMAC API has defined an Updater API for task execution. It is an optional API that can be useful as a reference. The Update API contains `Updatable, AsynchUpdater, and PeriodicUpdater` classes. If an OMAC module is periodic, it may derive the method `update()` by inheriting it from the Scheduling Updater class `Updatable`. For the Axis Module, the method `update()` is a wrapper that calls `processServoLoop()`. The `update()` method simplifies invocation, since the `updater` can go down a list of modules and invoke one signature.

An example to illustrate the multi-client/server interaction will be developed. First, the object naming and constructor definition that is done at configuration time will be sketched. The integration creates object references (i.e., `io1`, `io2`, `ax1`, `axgrp1`) and then binds addresses to the created objects through some name registration. Since `ax1` and `axgrp1` are periodic updating OMAC modules, they have inherited a method `update()` and register with the PeriodicUpdater `updater` using its `registerUpdatable()` method. The second parameter field in `registerUpdatable()` method is the clock divisor.

```
        integrationProcessInit(){
                // initialize parameters
                PeriodicUpdater updater;

                IOPoint  io1= new IOPoint(“encoder1”);
                IOPoint  io2= new IOPoint(“actuator1”);}
                Axis ax1= Axis(“Axis1”, io1, io2);
                AxisGroup axgrp1= AxisGroup(“AxisGroup1”, ax1);

                updater.setTimingInterval(.01);  // 10 millisecond period
                updater.registerUpdatable((Updatable *) axgrp, 2);
                updater.registerUpdatable((Updatable *) ax1, 1);
        }
```

Next, a sequence of operations will highlight the connection between the Scheduling Updater (`Updater`), the Axis Group module (`AxGrp`), the Axis module (`Axis`) and the actuator and encoder IO points. Within the `Axis` module, references to the component classes `AxisVelocityServo`, `AxisCommandOutput` and `Control Law` module will be made. (Readers are referred to Section 4.0 to further review Axis components.)

Figure 30 presents an Object Interaction Diagram to track the sequence of axis operation as triggered by a Scheduling Updater. The Updater calls the AxisGroup, which sets followingVelocity servo control and sends a commanded velocity setpoint. The Updater then triggers the Axis which in turn causes a `processServoLoop()` to perform a servo cycle. Since velocity servoing is enabled, the AxisVelocityServo is responsible to get the velocity command, read the axis actual velocity (as retrieved from io1), computes the next acceleration setpoint using a Control Law and then output a commanded acceleration to io2.

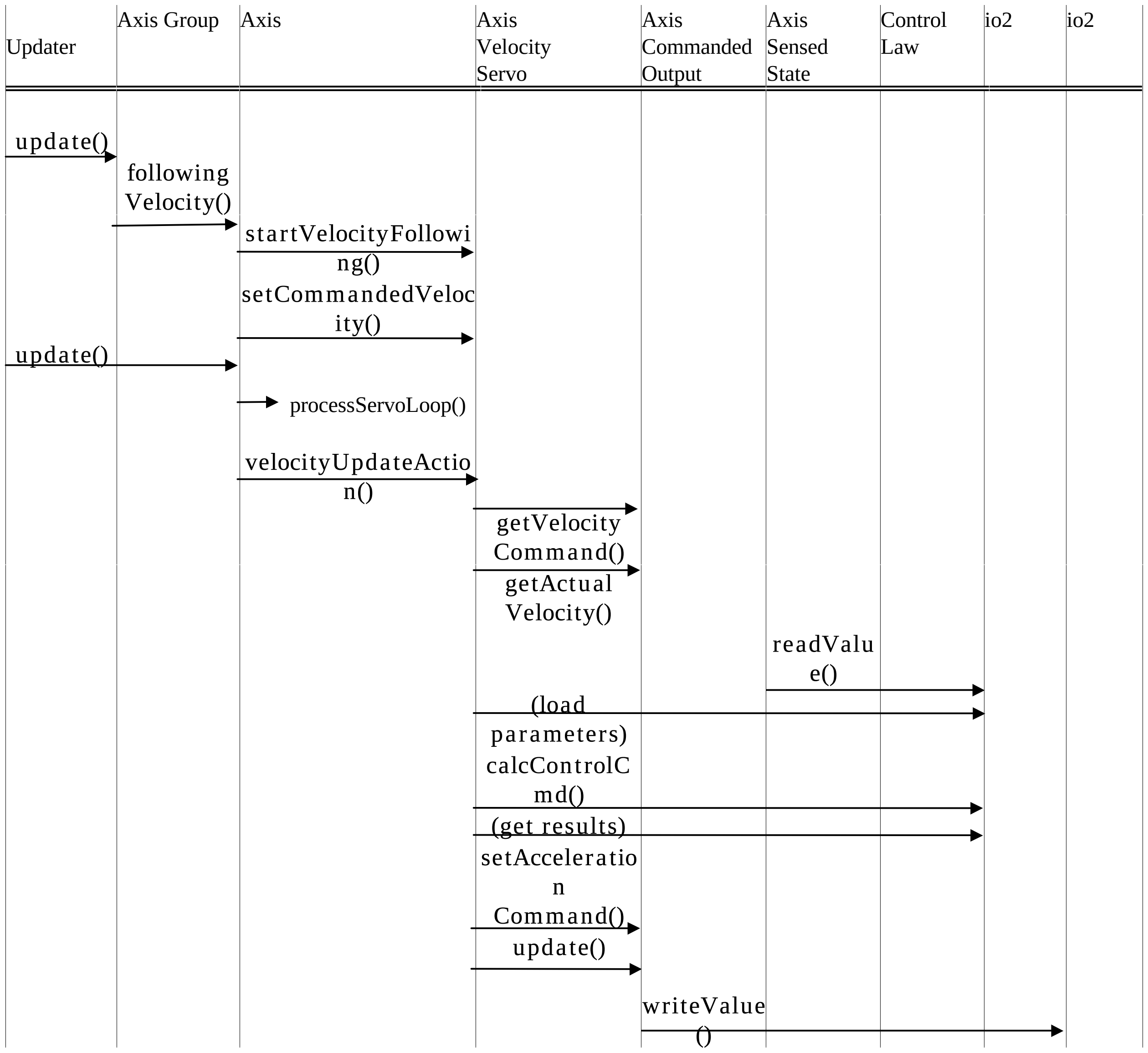


Figure 30: Schedule Updating Axis Object Interaction Diagram

As seen, the Axis module method `processServoLoop` performs the basic inputs, computes and outputs expected of a cyclic process. This functionality includes state interpretation so that an Axis module typically has a reference to an Axis FSM. Within the Axis FSM, the calls to `AxisVelocityServo` are made.

As stated earlier, one assumption within the object interaction is that a state transition, such as `followingVelocity`, is permissible. If not, either the method invocation is ignored or an exception is thrown.

Overall, the Scheduling Updater method `update()` is really a wrapper that calls `processServoLoop`. Hence, it isn't necessary to use an Updater. However, the `update()` wrapper does provide a generic interface to simplify scheduling of a variety of modules.

5.2 EVENT HANDLING

Standard client object requests to a server object result in synchronous execution of operation. In this case, the client sends the request and awaits a server response. This synchronous model includes the standard **client-push** model that sends an event through a method invocation. Section 3.3.1 has more on the client-push model.

Many times client-server interaction requires a more decoupled communication model. Of interest is the client-server interaction, called the **server-push** model, in which the server can spontaneously (asynchronously) issue an event to the client. For example, it is desirable to send an asynchronous `informDone()` event to the Task Coordinator when a CPU finished execution in the Axes Group. The question arises, "How is the Task Coordinator informed that the Axis Group is finished?" There are several options:

- The Task Coordinator polls the Axis Group with the `isDone()` method. This is the **client-pull** event method.
- Use cross-reference pointers between the communicating objects. In this case, the AxisGroup has a reference pointer back to the Task Coordinator, and it invokes a method (e.g., `informDone()`) to alert the Task Coordinator. There still must be some programming mechanism to tell the AxisGroup that it needs to call the Task Coordinator. Most likely, `informDone()` is mirrored in the TaskCoordinator and the AxisGroup to achieve this programming. The TaskCoordinator calls the AxisGroup `informDone()` to set the event, and the AxisGroup calls the TaskCoordinator `informDone()` when the event occurs. A simple event model is to add to all `isXstate()` query methods an `informXstate()` corollary.
- Another approach is to have the Task Coordinator call an AxisGroup method `waitUntilDone()` that blocks until the AxesGroup is done.

No agreement has been reached at this time regarding any standard server-push event model(s) or any server-push events.

The following general-purpose sequence has been proposed as the server-push event model:

- clients register what events it cares about with the server capable of detecting the event
- server send unique event id to client as part of registration
- when server detects an event it looks in a table (linked list) of clients which care about that event and sends the event id to each client (id will be unique for each client)
- clients use and unregister events using the id not the name.

5.3 CONFIGURATION

As a part of the open architecture life cycle, **configuration** and **integration** are important elements. Configuration is defined as module specification that maps it into a specific solution. Integration is defined as the capability to allow the connection and cooperation of two or more modules within a system. Readers are urged to review an OMAC API document concerning the open architecture Life Cycle that can be found at URL http://isd.cme.nist.gov/info/omacapi/Bibliography/oalifecycle.pdf. Briefly summarizing, the following steps outline the major configuration and integration steps.

1. distribution of modules to processes
2. distribution of processes to CPU
3. assignment of interprocess communication via proxy manager to processes
4. module/object construction and connection

This section will review the module construction phase because of the crucial role of global naming within the open architecture paradigm.
The construction phase is responsible for building the name data base and registering names with the appropriate lookup-information (e.g., address pointer or server information such as host id and server name). Within the Object Oriented paradigm there is a constructor phase wherein all the static application objects (in this case modules) must be constructed.
At this time, no agreement has been reached regarding configuration for module constructors. Herein a couple of alternatives for module constructors will be discussed.

Advertisement Model - The constructor is an advertisement for what a module needs. As an example, an OMAC API Configurator would construct a directed graph of modules in the system. The Task Coordinator would use the directed graph to construct the system. In a pure approach only the constructor would contain configuration information, as in the following example.

```
X_AXIS = new Axis(new PID_CL());
 Y_AXIS = new Axis(new PID_ControlLaw());
 AG1 = new AxisGroup(X_AXIS,  Y_AXIS);
```

One problem with the pure constructor approach is resolving circular references. For example, suppose the Axis and Axis Group modules' constructor need a reference to each other.

Another problem with pure constructors for configuration is handling combinatorial explosion of constructor possibilities. For example, if the system is not doing force control, does one need a set of special constructors to allow AxisForceServo control law references? To handle the combinatorial explosion, one could either define a monolithic constructor that accepts null references, or define constructors for each potential configuration.

The use of SETPARAMETERREFERENCE (e.g., `setControlLaw` below) helps reduce the combinatorial constructor possibilities. However, in this case, configuration is now based on selectively configuring parameters. The following example illustrates configuring the X and Y positioning servo control law.

```
X_AXIS = new Axis();
Y_AXIS = new Axis();
X_AXIS->AxisPositioningServo->setControlLaw(new PID_ControlLaw());
Y_AXIS->AxisPositioningServo->setControlLaw(new PID_ControlLaw());
...
AG1 = new AxisGroup(X_AXIS, Y_AXIS);
if((s=AG1->isSatisfied)!=NULL) cout << “Missing Parameter”<< s << endl;
```

Although flexible, selectively configuring parameters is vague so that it can be unclear what parameters must be specified. The potential for chaos can arise without some formalism. Does the AxisForceServo control law need to be configured? How does one determine when the AxisForceServo control law needs to be configured? To avoid confusion, a configuration method such as `isSatisfied()` that returns a string array of missing parameter definitions is essential.

Registry Model – In this case, the constructor plays a small role and system generation is name-driven. It is expected that names would be maintained in a globally accessible registry either a simple table or data base. Resolving object references would use a setParameterReference - although this time the method signature would be string-oriented.

Naming is divided into two categories - **local naming** and **global naming**.

Local naming is responsible for the names associated with a particular module. A vendor would be responsible for distributing a local naming table associated with each module. For example, the following table sketches a local naming table for an Axis module.

Local Name	*Type*	*Configured*
“ENCODER”	“IO_FLOAT”	Y
“ACTUATOR”	“IO_FLOAT”	Y
“POSITION_CONTROL_LAW”	“OMAC_CONTROL_LAW”	y
“VELOCITY_CONTROL_LAW”	“OMAC_CONTROL_LAW”	y
…		

Global naming is responsible for mapping local names to global names. Global naming serves two purposes. First, the global naming allows system access to local address references. Second, global naming enables familiar naming conventions. For example, a three axis mill would have three instances of the parameter `ENCODER` that could be resolved into corresponding global names of `X-ENCODER`, `Y-ENCODER`, and `Z-ENCODER`.

Global Name	Module	Local Name
“X-AXIS-ENCODER”	“X_AXIS”	“ENCODER”
“X-POSITION-CONTROL_LAW”	“X_AXIS”	“POSITION_CONTROL_LAW”
…	…	…
“Y-AXIS-ENCODER”	“Y_AXIS”	“ENCODER”
“Y-POSITION-CONTROL_LAW”	“Y_AXIS”	“POSITION_CONTROL_LAW”
…	…	…

There would be several steps in configuring a global naming scheme, including:

1. Create with “`new`” and `constructor(string NAME)`. In this case, the constructor takes a unique name, registers the name and module type in the global registry, and uses recursion to back through the object's parents to add type/name for registry (or self-discovery).

```
Axis X_AXIS = new Axis(“X-AXIS”);
Axis Y_AXIS = new Axis(“Y-AXIS”);
ControlLaw CL1 = new PID_ControlLaw(“CL1”);
ControlLaw CL2 = new PID_ControlLaw(“CL2”);
```

   Recursion is necessary because modules (i.e., objects) may be specialized and other modules may need a less specialized object. For example, a “SercosAxis” module is also a derived type of “Axis” and “OMAC Module”. **Self-discovery** of an object such as “SercosAxis” would recursively descend its parents until it reached some base class, in this case “OMAC Module”. To provide a flexible naming service, lists for types and objects should exist to provide object references. Figure 35 illustrates the relationship between each module base and derived types which have a pointer to a list of object names, which in turn, contains the actual object reference. This table could preexist in some data base.

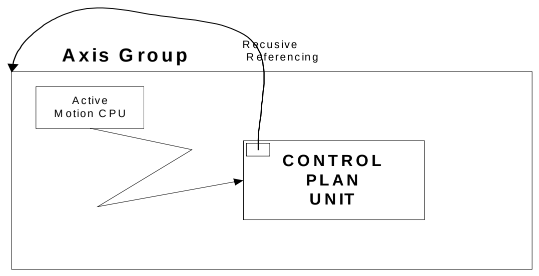


Figure 35: Type and Object Reference Lists from Recursive

2. Initialize objects. This initialization scope is directed at objects' local variables such as zeroing private variables. No external references should be used as these references may not have been resolved yet.

3. Connect objects by assigning names to different internal references. The general method signature would be:

```
setReference(string localName, string  globalName);
```

The following illustrates the registering some Axis and Axis Group names.

```
AG1->setReference("AXIS1", "X-AXIS");
AG1->setReference("AXIS2", "Y-AXIS");
X_AXIS->setReference("PositioningServoControlLaw", "CL1");
Y_AXIS->setReference("PositioningServoControlLaw", "CL2");
if((s=AG1->isSatisfied)!=NULL) cout << "Missing Parameter"<< s << endl;
```

Within a module, the `setReference` method would do a symbolic lookup of the type based on the local name, and then use the type to retrieve the actual reference. The following code sketches this approach.

```
class Axis {
...
IOFloat Encoder;
string itemType;

  void setReference(LocalName localName, GlobalName globalName){
       itemType=typelookup(localName);
       switch(localName){
          case "encoder":
            encoder= (IOFLoat) lookup(globalName, itemType);
            break;
          ...
       }
   }
 }
```

As an alternative to hard coding the connections, a module could read a file or data base to derive the references it needs. The table could contain other performance parameters as well. Below is a sketch of the information that could be expected using a file registry.

```
#
# Global Name     Type           Period   Timing       Local Names
#
  AxGrp1        AxisGroup        .01      .002         Ax1="X"
```

```
                                                  Ax2="Y"
                                                  Ax3="Z"
  X             Axis            .001     .0002    Output= "act1"
                                                  Feedback= "enc1"
                                                  Position= "PIDControlLaw"
                                                  Velocity= "Sercos1"
                                                  Acceleration= NULL

  Y             Axis            .001     .0002    Output= "act2"
                                                  Feedback= "enc2"
                                                  Position= "PIDControlLaw"
                                                  Velocity= "Sercos2"
                                                  Acceleration= NULL

  Z             Axis            .001     .0002    Output= "act3"
                                                  Feedback= "enc3"
                                                  Position= "PIDControlLaw"
                                                  Velocity= "Sercos3"
                                                  Acceleration= NULL
  Sercos1   SERCOSControlLaw
  Sercos2   SERCOSControlLaw
  Sercos3   SERCOSControlLaw

# This is sketch of an Abstract to Physical IO Map
# IOPTs    Type    Board      Address        Bytes
  act1     IO-W    D/A1       0xFFFFFF00      8
  enc1     IO-R
  act2     IO-W
  enc2     IO-R
  act3     IO-W
  enc3     IO-R
```

4. Reinitialization of objects. The second pass assumes that all external references are resolved, so that an object can access external objects as part of its initialization sequence.

5.4 ERROR HANDLING, ERROR PROPAGATION

"Exception and error handling is 90% of the aggravation on the shop floor." Attempting to resolve errors/exceptions as they propagate through the system is difficult. Errors can be hard to anticipate and/or resolve. However, errors and exceptions are really just server-push events (clients don't push errors on the servers). Infrastructure support for server-push event handling is weak.
As an intermediary solution, a simple error propagation technique is to allow object cross-references so that for every pair of objects, each one has a reference to the other object. In this case, each invokes methods in the other to propagate and event.
Within OMAC API, a proposal for handling errors is for each OMAC module to support an error CPU with a `setErrorCPU(cpu)` method. In the event an error occurs, an `error(errcode)` method could be invoked. For example, in the case that a Task Coordinator received an error event, it could then dispatch the ERROR Capability. The ERROR Capability could be passed an error code or be smart enough to analyze the system and determine the error.
As another example, consider the handling of thermal overload on a drive. How does it trickle up? A straightforward solution is to add a CPU to the Discrete Logic to monitor this event. If the overload occurs and the Discrete Logic can not rectify the error it could then notify the Task Coordinator of an error which will then initiate the ERROR Capability.

REFERENCES

COR91
Object Management Group, Framingham, MA. OBJECT MANAGEMENT ARCHITECTURE GUIDE, DOCUMENT 92.11.1, 1991.

Cra86
John J. Craig. INTRODUCTION TO ROBOTICS MECHANICS AND CONTROL. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

DCOM
Distributed Common Object Model.

See Web URL: http://www.microsoft.com/oledev/olemkt/oledcom/dcom95.htm.

EXP
International Organization for Standardization. ISO-10303-11 DESCRIPTION METHODS: THE EXPRESS LANGUAGE REFERENCE MANUAL.

IEC93
International Electrical Commission, IEC, Geneva. PROGRAMMABLE CONTROLLERS PART 3 PROGRAMMING LANGUAGES, IEC 1131-3, 1993.

IEC95
IEC. IEC1491 - SERCOS (SERIAL REAL-TIME COMMUNICATIONS SYSTEM) INTERFACE STANDARD. International Electrical Commission, Geneva, 1995.

Inta
International Organization for Standardization. ISO 10303-42 INDUSTRIAL AUTOMATION SYSTEMS AND INTEGRATION PRODUCT DATA REPRESENTATION AND EXCHANGE - PART 42: INTEGRATED RESOURCES: GEOMETRIC AND TOPOLOGICAL REPRESENTATION.

Intb
International Organization for Standardization. ISO 10303-42 INDUSTRIAL AUTOMATION SYSTEMS AND INTEGRATION PRODUCT DATA REPRESENTATION AND EXCHANGE - PART 105: INTEGRATED APPLICATION RESOURCES: KINEMATICS.

Le95
T. Lewis and et al. OBJECT ORIENTED APPLICATION FRAMEWORKS. Manning Publications Co., Greenwich, CT, 1995.

MIDL
Microsoft Corporation. Microsoft Interface Definition Language (MIDL) Reference Manual. WIN32 SDK Distribution CD, Redmond WA.

M.S86
M. Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In 6TH INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, pages 198-204. IEEE Computer Society Press, May 1986.

NGI
Next Generation Inspection System (NGIS). See Web URL: http://isd.cme.nist.gov/brochure/NGIS.html.

OMA94
Chrysler, Ford Motor Co., and General Motors. REQUIREMENTS OF OPEN, MODULAR,

ARCHITECTURE CONTROLLERS FOR APPLICATIONS IN THE AUTOMOTIVE INDUSTRY, December 1994. White Paper - Version 1.1.

OSA96
OSACA. European Open Architecture Effort. See Web URL: http://www.isw.uni-stuttgart.de/projekte/osaca/english/osaca.htm, 1996.

PM93
F. Proctor and J. Michaloski. Enhanced Machine Controller Architecture Overview. Technical Report 5331, National Institute of Standards and Technology, December 1993.

RS274
Engineering Industries Association, Washington, D.C. EIA STANDARD - EIA-274-D, INTERCHANGEABLE VARIABLE, BLOCK DATA FORMAT FOR POSITIONING, CONTOURING, AND CONTOURING/POSITIONING NUMERICALLY CONTROLLED MACHINES, February 1979.

SOS94
National Center for Manufacturing Sciences. NEXT GENERATION CONTROLLER (NGC) SPECIFICATION FOR AN OPEN SYSTEM ARCHITECTURE STANDARD (SOSAS), August 1994. Revision 2.5.

APPENDIX A – UML INTERFACE DEFINITIONS

Unified Modeling Language (UML) is a standard notation for the modeling of application objects in developing an object-oriented program. UML contains notation to model the class (of objects), object, association, responsibility, activity, interface, use case, package, sequence, collaboration, and state. The advantage of UML is that it is vendor and language neutral. However, at this time, a UML OMAC API specification has not been attempted.

APPENDIX B – MIDL API DEFINITIONS

Technical Note: These API are for review and comment only. There is no guarantee of correctness. This specification approximates the intended direction of the final API.

B.1 DISCLAIMER

This software was produced in part by agencies of the U.S. government, and by statute is not subject to copyright in the United States. Recipients of this software assume all responsibility associated with its operation, modification, maintenance, and subsequent redistribution.

B.2 NAMING CONVENTIONS

The naming convention for the IDL specification uses the Hungarian notation of separating words with CapitalLetters. (This release removed all the "_" and used concatenation of Capital letters to distinguish words.) The following conventions are being followed.

```
File Name                       : same as major class name (JAVA convention)
#define    for constants        : entire name in UPPER CASE
class name & declaration        : CapStyle with beginning C
class/variable instance         : smallCapStyleAgain
method arguments                : smallCapStyleAgain

general method signature        : nameCapStyle
        query  parameter        : getParameterName
        assignment              : setParameterName
        state query             : isStateName
```

There is consideration for adding a classifying prefix to class instances, global and static variable declarations and method arguments. In this case, d_VariableName would indicate a double variable. Note, C++ function declarations need parameter types but not parameter names, however, IDL requires both.
The use of get and set methods on these attributes, since IDL does not produce a get/set prefix to the methods. This will not work for non-IDL-like systems.

B.3 MICROSOFT COM

B.3 MICROSOFT STATUS CODES

Except in special circumstances, nearly every COM API interface member function returns a value of the type HRESULT. HRESULT is also called a "handle to a result." COM follows a naming convention for different HRESULT success and error codes. Any name with *E_* in it, which may be at the beginning as in E_FAIL or RPC_E_NOTCONNECTED means that the function failed. Any name with *S_*, as in S_TRUE, S_FALSE, or STG_S_CONVERTED, means that the function succeeded. The most common codes are listed in the following table.

Value	Meaning
S_OK	Function succeeded. Also used for functions that semantically return a Boolean TRUE result to indicate that the function succeeded.
S_FALSE	Used for functions that semantically return a Boolean FALSE result to indicate that the function succeeded.
E_NOINTERFACE	QueryInterface did not recognize the requested interface.
E_NOTIMPL	Member function contains no implementation.
E_FAIL	Unspecified failure.
E_OUTOFMEMORY	Function failed to allocate necessary memory.

B.4 BASIC TYPES

```
#ifndef DataRepresentation
#define DataRepresentation
import "oaidl.idl";
import "ocidl.idl";
// Level 1 - these will be backed out from the other API definitions
//
//interface test {

typedef long     API;
typedef double   AngularVelocity;
typedef struct     __CoordinateFrame{
    double c[4][4];
} CoordinateFrame;
//typedef struct   _FILE {int fixme; } FILE;
typedef double   Force;
typedef double   Length;
typedef double   LinearVelocity;
typedef double   LinearAcceleration;
typedef double   LinearJerk;
typedef double   LinearStiffness;
typedef struct   _LowerKinematicModel {int fixme; } LowerKinematicModel;
typedef double   Magnitude;
typedef double   Mass;
// Matrix???
typedef double   Measure;
typedef struct _OacVector{
    short  size;
    double axis[10];
} OacVector;
typedef double   PlaneAngle;
typedef struct   _RESOURCE  {int fixme; } RESOURCE ;
typedef struct   _RPY  {int fixme;} RPY;
typedef long     Status;
typedef struct   _Time { int fixme; } Time;
typedef struct   _Transform { int fixme; } Transform;
typedef struct   _UNITS {int fixme; } UNITS;
typedef struct   _UpperKinematicModel {int fixme; } UpperKinematicModel;
typedef double   Velocity;

typedef struct    _Translation  {int fixme; } Translation;
typedef Translation CartesianPoint;

/*
//?? Or you can assume numbers are flagged not active at
//?? construction time.
// Below most control parameters would be typed as double
#define doubleNotActive 1.79769313486231570e+308
#define longNotActive 0x80000000
```

```
#define shortNotActive 0x8000


// Level 2  Example - not defined here

interface LinearVelocity : Units  {
    Magnitude  value; // should this value be used?
    // Upperbound and Lowerbound, both zero ignore
    Magnitude ub, lb; // which may be ignored
    disabled();
    enabled();
};
interface Units
{ // FIXME
};
*/

#endif

```

B.5 CONNECTION TABLE FOR NAMING SERVICES

```
// ConnectionInfo.idl : IDL source for ConnectionInfo.dll
//

// This file will be processed by the MIDL tool to
// produce the type library (ConnectionInfo.tlb) and marshalling code.

import "oaidl.idl";
import "ocidl.idl";


cpp_quote("#define PublishInfoMaxNameSize 1028")
    #define PublishInfoMaxNameSize 1028
    typedef struct _PublishInfo {
        wchar_t name[ PublishInfoMaxNameSize ];
        wchar_t type[ PublishInfoMaxNameSize ];
        IUnknown *address;
    } PublishInfo;

    typedef enum tagBINDSTATES { Mandatory=1,
                                                                        Optional=2,
                                                                        Connected=4,
                                                                        Unconnected=8, any=0}
BINDSTATES;

    typedef struct _BindInfo {

        wchar_t localname[PublishInfoMaxNameSize];
        wchar_t type[PublishInfoMaxNameSize];
        wchar_t globalname[PublishInfoMaxNameSize];
        wchar_t description[PublishInfoMaxNameSize];
        unsigned long state;
        IUnknown ** ref; // store into pointer variable
    } BindInfo;
/*
    [

        uuid(6511417A-391B-11D3-AAB7-00C04FA375A6),

        helpstring("IPublishInfo Interface"),
        pointer_default(unique)
    ]
    interface IPublishInfo : IUnknown
    {
    //  HRESULT getPublishInfo([out,retval] PublishInfo * info);
    //  HRESULT setPublishInfo([in] PublishInfo  info);

    };
    */

    [
        object,
       uuid(65114180-391B-11D3-AAB7-00C04FA375A6),

       helpstring("IEnumPublishInfo Interface"),
       pointer_default(unique)
    ]
    interface IEnumPublishInfo : IUnknown
    {


      [local]

      HRESULT Next([in] ULONG celt,

                                                                   [out] PublishInfo* rgelt,

                                                                   [out] ULONG *pceltFetched);
/*
      [call_as(Next)] // Later...

      HRESULT RemoteNext([in] ULONG celt,

                                                                            [out,
size_is(celt),


length_is(*pceltFetched)] PublishInfo* rgelt,

                                                                            [out] ULONG
*pceltFetched);
*/
      HRESULT Skip([in] ULONG celt);

      HRESULT Reset();

      HRESULT Clone([out] IEnumPublishInfo **ppenum);

    };

    [
       object,
       uuid(7C045B5D-451C-11d3-AABB-00C04FA375A6),

       helpstring("IEnumBindInfo Interface"),
       pointer_default(unique)
    ]
    interface IEnumBindInfo : IUnknown
    {


      [local]

      HRESULT Next([in] ULONG celt,

                                                                   [out] BindInfo* rgelt,

                                                                   [out] ULONG *pceltFetched);
/*
      [call_as(Next)] // Later...

      HRESULT RemoteNext([in] ULONG celt,

                                                                            [out,
size_is(celt),


length_is(*pceltFetched)] BindInfo* rgelt,

                                                                            [out] ULONG
*pceltFetched);
*/
      HRESULT Skip([in] ULONG celt);

      HRESULT Reset();

      HRESULT Clone([out] IEnumBindInfo **ppenum);

    };



    [

       uuid(6511417E-391B-11D3-AAB7-00C04FA375A6),

       helpstring("IConnectionTable Interface"),
       pointer_default(unique)
    ]
    interface IConnectionTable : IUnknown
    {
       HRESULT getBindCount([out,retval] long * pnCount);
       HRESULT getPublishCount([out,retval] long * pnCount);

       HRESULT getAllPublish(
            [retval][out]  IEnumPublishInfo  **ppAllConnections);

       HRESULT getAllBindings(
            [retval][out]  IEnumBindInfo  **ppAllConnections);

       HRESULT isFullyIntegrated(
            [retval][out]  boolean  *b);


       HRESULT getAllConnections(
            [retval][out]  IEnumString  **ppAllConnections);

       HRESULT getAllRequiredConnections(
            [retval][out]  IEnumString  **ppRequiredConnections);

       HRESULT getAllUnconnected(
            [retval][out]  IEnumString  **ppUnconnectedLocalNames);

       HRESULT getAllRequiredConnected(
            [retval][out]  IEnumString  **ppConnections);

       HRESULT isConnected(
            [in]  BSTR localName,
            [retval][out]  boolean  *b);

       HRESULT isConnectionRequired(
            [in]  BSTR localName,
            [retval][out]  boolean  *b);

       HRESULT getConnectionType(
            [in]  BSTR localName,
            [retval][out]  BSTR  *type);

       HRESULT getConnectionDescription(
            [in]  BSTR localName,
            [retval][out]  BSTR  *description);

       HRESULT getConnectedToName(
            [in]  BSTR localName,
            [retval][out]  BSTR  *connection);

    /* HRESULT setConnectionTo(
            [in]  BSTR  localName,
            [in]  BSTR  *registeredName);
    */
       HRESULT setConnectionTo(
            [in]  BSTR  localName,
            [in]  IUnknown  *connection);
};

    [

       uuid(6511417C-391B-11D3-AAB7-00C04FA375A6),

        helpstring("ITestEnumInterface"),
        pointer_default(unique)
    ]
    interface ITestEnumInterface : IConnectionTable
    {

    };

    [

        uuid(C9BA5F57-3BB0-11d3-AAB7-00C04FA375A6),

        helpstring("ITestEnumAggregatedInterface"),
        pointer_default(unique)
    ]
    interface ITestEnumAggregatedInterface :  IUnknown
        {

    };



[
    uuid(6511416D-391B-11D3-AAB7-00C04FA375A6),
    version(1.0),
    helpstring("Omac ConnectionInfo 1.0 Type Library")
]
library CONNECTIONINFOLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");
/*
    [
        uuid(6511417B-391B-11D3-AAB7-00C04FA375A6),
        helpstring("PublishInfo Class")
    ]
    coclass PublishInfo
    {
        [default] interface IPublishInfo;
    };
*/


    [
        uuid(65114181-391B-11D3-AAB7-00C04FA375A6),
        helpstring("EnumPublishInfo Class")
    ]
    coclass EnumPublishInfo
    {
        [default] interface IEnumPublishInfo;
    };

    [
        uuid(DEB592DF-451C-11d3-AABB-00C04FA375A6),
        helpstring("EnumBindInfo Class")
    ]
    coclass EnumBindInfo
    {
        [default] interface IEnumBindInfo;
    };

    [
        uuid(6511417F-391B-11D3-AAB7-00C04FA375A6),
        helpstring("ConnectionTable Class")
    ]
    coclass ConnectionTable
    {
        [default] interface IConnectionTable;
    };

    [
        uuid(6511417D-391B-11D3-AAB7-00C04FA375A6),
        helpstring("TestEnumInterface Class")
    ]
    coclass TestEnumInterface
    {
        [default] interface ITestEnumInterface;
    };

        [
        uuid(ECD6C0BB-3BB0-11d3-AAB7-00C04FA375A6),
        helpstring("TestEnumAggregatedInterface Class")
    ]
    coclass TestEnumAggregatedInterface
    {
        [default] interface IConnectionTable;
        interface ITestEnumAggregatedInterface;
    };
};

```

B.5 OMAC MODULE BASE CLASSES TYPES

```
// ControlLawModule.idl : IDL source for ControlLawModule.dll
//

// This file will be processed by the MIDL tool to
// produce the type library (ControlLawModule.tlb) and marshalling code.
import "oaidl.idl";
import "ocidl.idl";
import "ConnectionInfo.idl";

    [

        object,
        uuid(8CBFD25C-C72F-11d2-AAAB-00C04FA375A6),

        helpstring("IOmac Interface"),
        pointer_default(unique)
    ]
    interface IOmac : IUnknown
    {


    HRESULT _stdcall update();
    HRESULT _stdcall configToString([out,retval] BSTR * str);
    HRESULT _stdcall configure([in] BSTR inifile, [in] BSTR keyname);
    HRESULT _stdcall isConfigured([out,retval] BSTR * b);
    HRESULT _stdcall isFullyConfigured([out,retval] boolean * b);
    HRESULT _stdcall init();
    HRESULT _stdcall toString([out,retval] BSTR * str);
    HRESULT _stdcall integrate();
    HRESULT _stdcall setName([in] BSTR name);
    HRESULT _stdcall getName([out,retval] BSTR * str);

    // Event Triggersing State Transition Methods
    HRESULT _stdcall execute();
    HRESULT _stdcall startup();
    HRESULT _stdcall begin();
    HRESULT _stdcall done();
    HRESULT _stdcall stop();
    HRESULT _stdcall terminate();
    HRESULT _stdcall abort();
    HRESULT _stdcall Enable();
    HRESULT _stdcall Disable();

};
    [

        object,
        uuid(AC04B49D-E6CA-11d2-AAB0-00C04FA375A6),

        helpstring("Omac Named Factory Interface"),
        pointer_default(unique)
    ]
    interface IOmacModuleClassFactory : IClassFactory
    {
    HRESULT _stdcall CreateModule(BSTR name, REFIID riid, [out, iid_is(riid)] void ** ppvObj);

    /* ******************************** */
    /* Registration services */
    /* ******************************** */


                                                                    // get a reference to an
object of type
    HRESULT _stdcall lookupOmacObject([in] BSTR registryName, [out, retval] IUnknown ** address
);

                                                                    // get a reference to an
object of a specific type
    HRESULT _stdcall lookupTypedOmacObject( [in] BSTR registryName,
                                                                                        [in] BSTR
objectType, [out, retval] IUnknown ** address );

                                                                     // return an enumuration of
Strings of all the
                                                                     // registered object types in
the system
    HRESULT _stdcall getClassDirectory([out, retval] IEnumString ** ppEnumObjects);

                                                                     // return an enumeration of
Strings of all the
                                                                     // registered object
instances of the specified
                                                                     // type
    HRESULT _stdcall  getObjectDirectory([in] BSTR objectType, [out, retval] IEnumString **
ppEnumObjects );


};

// Create OMAC type library
[
    uuid(FF53F62B-E379-11d2-AAAF-00C04FA375A6),
    version(1.0),
    helpstring("Omac Module 1.0 Type Library")
]
library OMACMODULELib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [
        uuid(FF53F62C-E379-11d2-AAAF-00C04FA375A6),
        helpstring("Omac Class")
    ]
    coclass Omac
    {
        [default] interface IOmac;
                                                                       interface
IOmacModuleClassFactory;
         [optional] interface IConnectionTable;
    };

};
```

B.7 CONTROL PLAN

```
#ifndef _CONTROL_PLAN
#define _CONTROL_PLAN
import "oaidl.idl";
import "ocidl.idl";

interface IControlPlanUnit;
interface IEnumControlPlans;

[
    object,
    uuid(134A0282-E101-11d2-B512-AEC041D2957B),

    helpstring("Control Plan Unit Interface"),
    pointer_default(unique)
]

interface IControlPlanUnit : IUnknown
{    // approximate  a graph structure
    HRESULT _stdcall executeUnit([out,retval] IControlPlanUnit ** cpu); // return next
ControlPlanUnit
    // HRESULT _stdcall getNextUnit([out,retval]   ControlPlanUnit ** cpu);

    HRESULT _stdcall  setActive();    // set when "executing"
    HRESULT _stdcall  setInactive();
    HRESULT _stdcall  isActive([out, retval] boolean **flag);   // for HMI to determine when
active

    // persistence data a la binary image
    HRESULT _stdcall  save([in] BSTR file);
    HRESULT _stdcall  restore([in] BSTR file);

    // persistence data in neutral format (pre-configuration)
    HRESULT _stdcall  saveNeutral([in] BSTR file);
    HRESULT _stdcall  restoreNeutral([in] BSTR file);
};

[
    uuid(68B85C49-E86E-11d2-AAB1-00C04FA375A6),
    version(1.0),
    helpstring("Enumerated ControlPlan Interface")
]
interface IEnumControlPlans : IUnknown
{
    typedef [unique] IControlPlanUnit *LPENUMCONTROLPLANUNIT;

    [local]
    HRESULT Next(
        [in] ULONG celt,
        [out] IControlPlanUnit **rgelt,
        [out] ULONG *pceltFetched);

    [call_as(Next)]
    HRESULT RemoteNext(
        [in] ULONG celt,
        [out, size_is(celt), length_is(*pceltFetched)]
        IControlPlanUnit **rgelt,
        [out] ULONG *pceltFetched);

    HRESULT Skip(
        [in] ULONG celt);

    HRESULT Reset();

    HRESULT Clone(
        [out] IControlPlanUnit **ppenum);
};



const unsigned long E_SEQUENCERUNNING = 0x8004F001;

const unsigned long E_RERUN = 0x8004F002;
const unsigned long E_RESET = 0x8004F003;

const long EF_ABORT_PLAN            = 0x000000F0;
const long EF_PRODUCT_PLAN        = 0x000000F1;
const long EF_STEP_UNASSIGNABLE    = 0x000000F2;
const long EF_STEP_EXECUTING        = 0x000000F3;

typedef enum _StepStatus
{
    step_waiting        = 0x00,
    step_ready            = 0x01,
    step_executing        = 0x02
} StepStatus;


typedef enum _SequencerState
{
    uninitialized,
    dying,
    idle,
    running,
    halting,
    halted
}SequencerState;

//    IOperation interface

[
    uuid(ea6695e0-88af-11d2-a281-006097839e22),
    helpstring("IOperation Interface"),
    pointer_default(unique),
    dual
]
interface IOperation : IDispatch
{
    [helpstring("method Execute")]
    HRESULT Execute( [in, out] VARIANT *vaData);
}


[
    object,
    uuid(C59C4BAD-EDBB-11d2-AAB1-00C04FA375A6),
    dual,
    helpstring("ISequence Interface"),
    pointer_default(unique)
]
interface ISequence : IDispatch
{
    [helpstring("method InsertStep")]
    HRESULT InsertStep(
        [in] IOperation* pOperation,
        [in] BSTR strStepID,
        [in] VARIANT *vaData
        );

    [helpstring("method AddFollower")]
    HRESULT AddFollower(
        [in] BSTR strStepID,
        [in] HRESULT retVal,
        [in] BSTR strFollowerID,
        [in] IOperation* pFollowerOp,
        [in] VARIANT *vaFollowerData
        );

    [helpstring("method SetFollower")]
    HRESULT SetFollower(
        [in] BSTR strStepID,
        [in] HRESULT retVal,
        [in] BSTR strFollowerID
        );
    [helpstring("method ClearSteps")]
    HRESULT ClearSteps();

    [helpstring("method GetStepCount")]
    HRESULT GetStepCount(
        [out] ULONG* pnCount
        );

    [helpstring("method EnumerateSteps")]
    HRESULT EnumerateSteps(
        [out] VARIANT *pSteps,
        [out] ULONG* pnReturnedCount
        );
    [helpstring("method GetStepStatus")]
    HRESULT GetStepStatus(
            [in] BSTR strStepID,
            [out] StepStatus* pStatus
            );

    [helpstring("method GetPredecessorCount")]
    HRESULT GetPredecessorCount(
            [in] BSTR strStepID,
            [out] ULONG* pnCount
            );
    [helpstring("method EnumPredecessors")]
    HRESULT EnumPredecessors(
            [in] BSTR strStepID,
            [out] VARIANT *pPredecessors,
            [out] ULONG* pnReturnedCount
            );

    [helpstring("method GetSuccessorCount")]
    HRESULT GetSuccessorCount(
            [in] BSTR strStepID,
            [out] ULONG* pnCount
            );

    [helpstring("method EnumSuccessors")]
    HRESULT EnumSuccessors(
            [in] BSTR strStepID,
            [out] VARIANT *pSuccessors,
            [out] VARIANT *pResults,
            [out] ULONG* pnReturnedCount
            );

    [id(1), helpstring("method EnumWaitingSteps")]
    HRESULT EnumWaitingSteps([out] VARIANT *steps,
        [out] ULONG *pnReturnedCount);

    [id(2), helpstring("method AddPrecondition")]
    HRESULT AddPrecondition(BSTR step, BSTR preStep, HRESULT condition);

    [id(3), helpstring("method GetPreconditionCount")]
    HRESULT GetPreconditionCount([in] BSTR stepID, [out] ULONG *pnCount);

    [id(4), helpstring("method EnumPreconditions")]
    HRESULT EnumPreconditions([in] BSTR strStepID,
        [out] VARIANT *pPreconditions,
        [out] VARIANT *pConditions,
        [out] ULONG *pnReturnedCount);
};


    [
        object,
        uuid(ea6695e3-88af-11d2-a281-006097839e22),
        dual,
        helpstring("ISequencer Interface"),
        pointer_default(unique)
    ]
    interface ISequencer : IDispatch
    {
        [helpstring("method SetProductSequence")]
        HRESULT SetProductSequence( [in] ISequence* pProductPlan);

        [helpstring("method GetProductSequence")]
        HRESULT GetProductSequence( [out] ISequence** ppProductPlan);

        [helpstring("method SetAbortSequence")]
        HRESULT SetAbortSequence( [in] ISequence* pAbortPlan);

        [helpstring("method GetAbortSequence")]
        HRESULT GetAbortSequence( [out] ISequence** ppAbortPlan);

        [helpstring("method Go")]
        HRESULT Go();

        [helpstring("method Step")]
        HRESULT Step( [in] BSTR step);

        [helpstring("method Stop")]
        HRESULT Stop();

        [helpstring("method Abort")]
        HRESULT Abort();

        [id(1), helpstring("method RunThruStep")]
        HRESULT RunThruStep([in] BSTR step);

        [id(2), helpstring("method StartAt")]
        HRESULT StartAt([in] BSTR step);

        [id(3), helpstring("method Reset")]
        HRESULT Reset();

        [id(4), helpstring("method Rerun")]
        HRESULT Rerun();

        [id(5), helpstring("method SetName")]
        HRESULT SetName([in] BSTR name);

        [id(6), helpstring("method GetState")]
        HRESULT GetState([out] SequencerState *pState);
    };



[
    uuid(134A0283-E101-11d2-B512-AEC041D2957B),
    version(1.0),
    helpstring("ControlPlanModule 1.0 Type Library")
]
library CONTROL_PLAN_MODULE_Lib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [
        uuid(134A0284-E101-11d2-B512-AEC041D2957B),
        helpstring("ControlPlanUnit Class")
    ]
    coclass ControlPlanUnit
    {
        [default] interface IControlPlanUnit;
    };
    [
        uuid(134A0285-E101-11d2-B512-AEC041D2957B),
        helpstring("Enumerated Control Plans Class")
    ]
    coclass EnumControlPlans
    {
        [default] interface IEnumControlPlans;
    };

    interface IOperation;

    [
        uuid(ea6695e7-88af-11d2-a281-006097839e22),
        helpstring("Sequence component")
    ]
    coclass Sequence
    {
        [default] interface ISequence;
    };


    [
        uuid(ea6695e9-88af-11d2-a281-006097839e22),
        helpstring("Sequencer component")
    ]
    coclass Sequencer
    {
        [default] interface ISequencer;
    };

};


#endif




```

B.8 CAPABILITY

```
// CapabilityModule.idl : IDL source for CapabilityModule.dll
#ifndef _Capability
#define _Capability

import "oaidl.idl";
import "ocidl.idl";
import "OmacModule.idl";

// Each capablity is an FSM and types of capabilities include: manual, auto, estop, etc.
// FIXME: What is the relationship of manual to auto and any to estop?
// Internally the capbility is a FSM.
[
    object,
    uuid(134A0281-E101-11d2-B512-AEC041D2957B),

    helpstring("Capability Control Plan Interface"),
    pointer_default(unique)
]
interface ICapability : IUnknown
{
  HRESULT _stdcall  start();
  HRESULT _stdcall  execute();
  HRESULT _stdcall  updateCap(); //update() can call updateCap()
  HRESULT _stdcall  stop();
  HRESULT _stdcall  abort();
  HRESULT _stdcall  throwExecption();
  HRESULT _stdcall  resolveExecption();
  HRESULT _stdcall  isDone();
  HRESULT _stdcall  isActive();
};

[
    object,
    uuid(FDEC2BF7-E3AE-11d2-AAB0-00C04FA375A6),

    helpstring("Capability Control Plan Interface"),
    pointer_default(unique)
]
interface IEnumCapabilities : IUnknown
{

    typedef [unique] ICapability *LPENUMCAPABILITY;

    [local]
    HRESULT Next(
        [in] ULONG celt,
        [out] ICapability **rgelt,
        [out] ULONG *pceltFetched);

    [call_as(Next)]
    HRESULT RemoteNext(
        [in] ULONG celt,
        [out, size_is(celt), length_is(*pceltFetched)]
        ICapability **rgelt,
        [out] ULONG *pceltFetched);

    HRESULT Skip(
        [in] ULONG celt);

    HRESULT Reset();

    HRESULT Clone(
        [out] ICapability **ppenum);
};

[
    uuid(134A0286-E101-11d2-B512-AEC041D2957B),
    version(1.0),
    helpstring("Capability CPU Module 1.0 Type Library")
]
library CAPABILITY_MODULE_Lib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [
        uuid(134A0287-E101-11d2-B512-AEC041D2957B),
        helpstring("Capability CPU Class")
    ]
    coclass Capability
    {
        [default] interface ICapability;
    };
};
#endif

```

B.9 IO

```
// IOModule.idl : IDL source for IO Points.dll

#ifndef __IOModule__IDL
#define __IOModule__IDL
import "oaidl.idl";
import "ocidl.idl";
import "OmacModule.idl";
import "DataRepresentation.idl";

//typedef unsigned char byte;

// Level 1
[
    object,
    uuid(252BD0E9-EDB6-11d2-AAB1-00C04FA375A6),

    helpstring("IO Base Class Interface"),
    pointer_default(unique)
]
interface IIOPt : IOmac
{
  // Metadata
    typedef [v1_enum] enum tag_TYPE {
       DONTCARE,
       R_ONLY,
       W_ONLY,
       RW
    } TYPE;

    HRESULT _stdcall setType([in] TYPE value);
    HRESULT _stdcall getType([out, retval] TYPE ** value);
    HRESULT _stdcall setUnits([in] UNITS value);
    HRESULT _stdcall getUnits([out, retval] UNITS ** value);
    HRESULT _stdcall set([in] VARIANT value);
    HRESULT _stdcall get([out, retval] VARIANT ** value);
    HRESULT _stdcall setUpperBound([in]VARIANT value);
    HRESULT _stdcall getUpperBound([out,retval]VARIANT ** value);
    HRESULT _stdcall setLowerBound([in]VARIANT value);
    HRESULT _stdcall getLowerBound([out,retval]VARIANT ** value);
    HRESULT _stdcall enableBoundsChecking([in] boolean value);

};

[
    object,
    uuid(2CD39DE5-EDB6-11d2-AAB1-00C04FA375A6),
    helpstring("IOPtlong Interface"),
    pointer_default(unique)
]
interface IOPtlong : IIOPt
{
  HRESULT _stdcall getValue([out, retval] long ** value);
  HRESULT _stdcall setValue([in] long value);

};

[
    object,
    uuid(37B6BADF-EDB6-11d2-AAB1-00C04FA375A6),
    helpstring("IOPtshort Interface"),
    pointer_default(unique)
]
interface IOPtshort : IIOPt
{
  HRESULT _stdcall getValue([out, retval] short ** value);
  HRESULT _stdcall setValue([in] short value);

};

[
    object,
    uuid(42EAE7CD-EDB6-11d2-AAB1-00C04FA375A6),
    helpstring("IOPtbyte Interface"),
    pointer_default(unique)
]
interface IOPtbyte : IIOPt
{
  HRESULT _stdcall  getValue([out,retval] byte ** value);
  HRESULT _stdcall  setValue([in] byte value);

};

[
    object,
    uuid(4D9BF365-EDB6-11d2-AAB1-00C04FA375A6),
    helpstring("IOPtboolean Interface"),
    pointer_default(unique)
]
interface IOPtboolean : IIOPt
{
  HRESULT _stdcall getValue([out,retval] boolean ** value);
  HRESULT _stdcall setValue([in] boolean value);

};

[
    object,
    uuid(644BD3CD-EDB6-11d2-AAB1-00C04FA375A6),

    helpstring("IOPtdouble Interface"),
    pointer_default(unique)
]
interface IOPtdouble : IIOPt
{
  HRESULT _stdcall getValue([out,retval] double ** value);
  HRESULT _stdcall setValue([in] double value);

};

[
    object,
    uuid(6D8B52E7-EDB6-11d2-AAB1-00C04FA375A6),

    helpstring("IOPtfloat Interface"),
    pointer_default(unique)
]
interface IOPtfloat : IIOPt
{
  HRESULT _stdcall  getValue([out,retval]  float ** value);
  HRESULT _stdcall setValue([in] float value);

};

[
    uuid(770B317F-EDB6-11d2-AAB1-00C04FA375A6), // New GUID
    helpstring("ISubjectObserver Interface"),
    pointer_default(unique)
]
interface ISubjectObserver : IUnknown
{
    [helpstring("method SubscribeByID")]
    HRESULT SubscribeByID([in] DWORD dwSubjectID,
        [in] long lFlags,
        [in] long lNotificationFilter);

    [helpstring("method SubscribeByName")]
    HRESULT SubscribeByName([in] BSTR strname,
        [in] long lFlags,
        [in] long lNotificationFilter,
        [in, out] long *plSubscriptions);
        [helpstring("method Unsubscribe")]
    HRESULT Unsubscribe([in]DWORD dwSubjectID,[in]BOOL bAllSubjects);

    [helpstring("method IsSubscribed")]
    HRESULT IsSubscribed(DWORD dwSubjectID);

    [helpstring("method GetCountSubscriptions")]
    HRESULT GetCountSubscriptions([out] long *lCount);

    [helpstring("method GetCountSubscribers")]
    HRESULT GetCountSubscribers([out] long *lCount);

    [helpstring("method Notify")]
    HRESULT Notify([in] long lSizeNotification,
        [in, size_is(lSizeNotification)] [ptr] byte* pNotification,
        [in] long lDataType, [in] long lNotificationType,
        [in] long lExtra);

    [helpstring("method GetIDFromName")]
    HRESULT GetIDFromName([in] BSTR strname,[out]DWORD * dwObjectID);

    [helpstring("method GetObjectID")]
    HRESULT GetObjectID([out]DWORD * dwID);

    [helpstring("method GetName")]
    HRESULT GetName([out, retval]BSTR *strName);

    [helpstring("method GetNameFromID")]
    HRESULT GetNameFromID([in] DWORD dwID, [out, retval]BSTR *pbstrName);

    [helpstring("method SetName")]
    HRESULT SetName([in, string] BSTR bstrName);

    [helpstring("method GetError")]
    HRESULT GetError([out, retval] BSTR *pbstrError);
};
[
    uuid(17A90B20-8221-11d2-9AD6-00C0D15709A3),

    helpstring("IObserverNotification Interface"),
    pointer_default(unique)
]
interface IObserverNotification: IUnknown
{
    [helpstring("method OnNotify")]
    HRESULT OnNotify([in] VARIANT *pObj);

    //[helpstring("method OnNotify")] HRESULT OnNotify([in] DWORD
    // dwSubjectSender,[in] long
    // nSizeNotification,[in,size_is(nSizeNotification)] [ptr] byte*
    // pNotification);

    [helpstring("method OnNotifySubjectBroken")]
    HRESULT OnNotifySubjectBroken([in] DWORD dwSubjectID);
};



#ifdef IGNORE_THIS
    OPC  has defined this sort of interface
typedef sequence<IOPt> IOvalues;
typedef sequence<string> IOnames;
typedef sequence<string> IOmetadata;

// Or should this just be an array of IOPts?
interface IOgroup
{
  IOvalues getValues();
  void setValues(in IOvalues values);

  void addIoPtlong(in IOPtlong io);
  void addIoPtshort(in IOPtshort io);
  void addIoPtboolean(in IOPtboolean io);
  void addIoPtdouble(in IOPtdouble io);
  void addIoPtfloat(in IOPtfloat io);
  IOnames getNames();
  IOmetadata getMetadata();
};

interface IOsystem
{
  void addIoGroup(in IOgroup aIOgroup);
  IOgroup getIoGroup(in string name);
  // FIXME: how do you do this in IDL?
  // IOPt getIoPt(char * name);
};
#endif

[
    uuid(134A02A3-E101-11d2-B512-AEC041D2957B),
    version(1.0),
    helpstring("ControlPlanGenerator Module 1.0 Type Library")
]
library IO_MODULE_Lib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [
        uuid(903B079F-EDB8-11d2-AAB1-00C04FA375A6),
        helpstring("IO Point Class")
    ]
    coclass IOPt
    {
        [default] interface IIOPt;
    };



    [
        uuid(71CB82B7-EDB8-11d2-AAB1-00C04FA375A6),
        helpstring("SubjectObserver Class")
    ]
    coclass SubjectObserver
    {
        [default] interface ISubjectObserver;
        [default,source] interface IObserverNotification;
    };

};

// Level 2: Hierarchy of Common IO Points - for type checking
// See IO API Document for further details
#endif

```

B.10 TASK COORDINATOR

```
// TaskCoordinatorModule.idl : IDL source for TaskCoordinator.dll

#ifndef TaskCoordinator__IDL
#define TaskCoordinator__IDL
import "oaidl.idl";
import "ocidl.idl";
import "OmacModule.idl";
import "CapabilityModule.idl";

[
    object,
    uuid(134A0280-E101-11d2-B512-AEC041D2957B),

    helpstring("TaskCoordinator Interface"),
    pointer_default(unique)
]

// Task Coordinator accepts one capability from a list of capabilities.
interface ITaskCoordinator : IOmac /*UPDATABLE*/
{

   HRESULT _stdcall  update();  //can be inherited from UPDATER

  // Capability List Management
   HRESULT _stdcall addToList([in] ICapability * cap);
   HRESULT _stdcall removeFromList([in] ICapability * cap);
   HRESULT _stdcall getList([out, retval] IEnumCapabilities **cap);

  // Current Capability Management
   HRESULT _stdcall getCurrentCapability([out, retval] ICapability **cap);
   HRESULT _stdcall setCurrentCapability([in] ICapability * cap);
};

[
    uuid(134A0288-E101-11d2-B512-AEC041D2957B),
    version(1.0),
    helpstring("Task Coordinator Module 1.0 Type Library")
]
library TASK_COORDINATOR_MODULE_Lib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [
        uuid(134A0289-E101-11d2-B512-AEC041D2957B),
        helpstring("Task Coordinator Class")
    ]
    coclass TaskCoordinator
    {
        [default] interface ITaskCoordinator;
    };
};
#endif

```

B.11 DISCRETE LOGIC

```
//
// DiscreteLogic.idl
//
#ifndef DiscreteLogic__idl
#define DiscreteLogic__idl

import "oaidl.idl";
import "ocidl.idl";

import "OmacModule.idl";
import "ControlPlanModule.idl";

interface IDiscreteLogicUnit;

// Discrete Logic Module contains a list of logic units. A PLC like scan
// goes down the list and executes each logic unit if it is on. Logic units
// will be executed as often as its posted scan rate indicates.
// Internally each discrete logic unit is an FSM.
// Discrete Logic Units (DLUs) are grouped by scan rates.

[
    object,
    uuid(134A028C-E101-11d2-B512-AEC041D2957B),

    helpstring("Discrete Logic Interface"),
    pointer_default(unique)
]
interface IDiscreteLogic : IOmac
{

  // Logic Units Management
    HRESULT _stdcall createDiscreteLogicUnit([out, retval] IDiscreteLogicUnit ** d);
    HRESULT _stdcall addLogicUnit([in] IDiscreteLogicUnit * dlu);
    HRESULT _stdcall removeLogicUnit([in] IDiscreteLogicUnit * dlu);
    HRESULT _stdcall enableLogicUnit([in] IDiscreteLogicUnit * dlu);
    HRESULT _stdcall disableLogicUnit([in] IDiscreteLogicUnit * dlu);
};

// Derived from ControlPlanUnit, see: part program translator
[
    object,
    uuid(134A028D-E101-11d2-B512-AEC041D2957B),

    helpstring("Discrete Logic Interface"),
    pointer_default(unique)
]
interface IDiscreteLogicUnit: IControlPlanUnit
{
  HRESULT _stdcall setInterval([in] long aInterval);
  HRESULT _stdcall getInterval([out,retval] long ** val);

  HRESULT _stdcall start();
  HRESULT _stdcall scanUpdate();
  HRESULT _stdcall stop();
  HRESULT _stdcall isOn([out,retval] boolean ** flag);
  HRESULT _stdcall turnOn([out,retval] boolean ** flag);
  HRESULT _stdcall turnOff([out,retval] boolean ** flag);
};

[
    uuid(134A028E-E101-11d2-B512-AEC041D2957B),
    version(1.0),
    helpstring("DiscreteLogicModule 1.0 Type Library")
]
library DISCRETE_LOGIC_MODULE_Lib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [
        uuid(134A028F-E101-11d2-B512-AEC041D2957B),
        helpstring("DiscreteLogic Class")
    ]
    coclass DiscreteLogic
    {
        [default] interface IDiscreteLogic;
    };
    [
        uuid(134A0290-E101-11d2-B512-AEC041D2957B),
        helpstring("DiscreteLogicUnit Class")
    ]
    coclass DiscreteLogicUnit
    {
        [default] interface IDiscreteLogicUnit;
    };
};
#endif

```

B.12 CONTROL PLAN GENERATOR

```
//
// ControlPlanGenerator.idl
//
#ifndef ControlPlanGenerator__idl
#define ControlPlanGenerator__idl

import "DataRepresentation.idl";
import "ControlPlanModule.idl";

// Level 1 assuming simple File Manipulation
[
    object,
    uuid(134A02A2-E101-11d2-B512-AEC041D2957B),

    helpstring("Control Plan Generator Interface"),
    pointer_default(unique)
]
interface IControlPlanGenerator :IUnknown
{
    HRESULT _stdcall setProgramName([in] BSTR s);
    HRESULT _stdcall getProgramName([out,retval] BSTR **name );

    HRESULT _stdcall checkSyntax([out,retval] boolean **flag);

   //get error codes or returns file name or file pointer?
    HRESULT _stdcall getErrorCodes([out,retval] BSTR ** results);

    // complete translation into ControlPlan
    HRESULT _stdcall  translate([out,retval] IEnumControlPlans ** cp);

    // step by step translation
    HRESULT _stdcall  getNextControlPlanUnnit([out,retval] IControlPlanUnit ** cpu);
};

[
    uuid(134A02A3-E101-11d2-B512-AEC041D2957B),
    version(1.0),
    helpstring("ControlPlanGenerator Module 1.0 Type Library")
]
library CONTROL_PLAN_GENERATOR_MODULE_Lib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [
        uuid(134A02A4-E101-11d2-B512-AEC041D2957B),
        helpstring("Control Plan Generator Class")
    ]
    coclass ControlPlanGenerator
    {
        [default] interface IControlPlanGenerator;
    };
};

#endif

```

B.13 AXIS GROUP

There are some inconsistencies within the Axis Group module API. The major remaining problem is to resolve the use of the axis group velocity profile generator (VPG) versus having the VPG embedded within a motion segment.

```
#ifndef AxisGroup__IDL
#define AxisGroup__IDL

import "DataRepresentation.idl"
import "OmacModule.idl"
import "Kinematics.idl"
import "ControlPlan.idl"

//+ add accel mode - use instead of enum - windows problem
typedef long ACCMode;
#define  SCURVE 1
#define  TRAPEZOIDAL 2

interface                  IAxisGroup;
interface                  IMotionSegment;
interface                  IRate;
interface                  IVelocityProfileGenerator;
typedef   long             AccDecProfile;
struct  _CoordinatedAxes {
    double axis[10];
} CoordinatedAxes;

struct  _CRCMODE         { /* FIXME */ } CRCMODE;

[
    object,
    uuid(134A0292-E101-11d2-B512-AEC041D2957B),

    helpstring("Axis Group Interface"),
    pointer_default(unique)
]
interface IAxisGroup : IOmac
{
//+  enum { ERROR, HELD, HOLDING, STOPPED, STOPPING,
//          PAUSED, PAUSING, RESUME, EXECUTING, IDLE };

  // STATE LOGIC
  // ============================================

  HRESULT _stdcall  hardStopAxes();  // Stop at max deceleration rate (abort)
  HRESULT _stdcall  pauseAxes();      // stop on path
  HRESULT _stdcall  holdAxes();       // stop at end of segment
  HRESULT _stdcall  resumeAxes();     // Resumes motion from current point

//  HRESULT _stdcall     updateAxes();
  HRESULT _stdcall     update();       //+ changed for consistent interface

  HRESULT _stdcall getCurrentState([out,retval] long **value);
  HRESULT _stdcall  getCurrentStateName(BSTR statename);
  HRESULT _stdcall  isOk(boolean **flag);
  HRESULT _stdcall  isExecuting([out,retval] boolean **flag);
  HRESULT _stdcall  isHeld([out,retval] boolean **flag);
  HRESULT _stdcall  isHolding([out,retval] boolean **flag);
  HRESULT _stdcall  isPaused([out,retval] boolean **flag);
  HRESULT _stdcall  isPausing([out,retval] boolean **flag);
  HRESULT _stdcall  isStopping([out,retval] boolean **flag);
  HRESULT _stdcall  isStopped([out,retval] boolean **flag);

  // These methods could be operator Control Plan Unit
  HRESULT _stdcall  jogAxis([in] long axisNo,
       [in] Velocity speed );

  HRESULT _stdcall  homeAxis([in] long axisNo,
       [in] Velocity speed );

  HRESULT _stdcall  moveAxisTo([in] long axisNo,
       [in] Velocity speed,
       [in] Length  toPosition);

  HRESULT _stdcall  incrementAxis([in] long axisNo,
       [in] Velocity speed,
       [in] Length increment);

  // BUFFERING MANAGEMENT
  //================================================
  HRESULT _stdcall  setNextMotionSegment([in] IMotionSegment block);
  // MotionSegment getCurrentMotionBlock( );  //hazardous to your controller’s health
  HRESULT _stdcall  getMaxqsize( [out,retval] long ** val);     // largest queue size possible=n
  HRESULT _stdcall  setQlength([in] long value); // maximum number of queue members=(1..n)
  HRESULT _stdcall  getQlength([out,retval] long ** val);
  HRESULT _stdcall  getCurrentQsize([out,retval] long ** val);     // number of items in queue=i
  HRESULT _stdcall  isFull([out,retval] boolean **flag);           // number of items = n
  HRESULT _stdcall  isEmpty([out,retval] long ** val);          // number or items = 0

  HRESULT _stdcall  flush();                // flush all segments
  HRESULT _stdcall  skip();                 // skip to next segment
  HRESULT _stdcall  saveQContext();       // save current queue
  HRESULT _stdcall  restoreQContext();    // restore saved queue

  // FIXME: possibly more queue mgt functions (accessor, query, ... )

  // CONVENIENCE FUNCTIONS TO ACCESS MOTION SEGMENT DATA
  //================================================
  HRESULT _stdcall getNeighborhood([out,retval] Length ** dval);
  HRESULT _stdcall getFeedrate([out,retval] LinearVelocity  ** dval);
  HRESULT _stdcall getTraverserate([out,retval] Velocity ** dval);
  HRESULT _stdcall getFeedrateOverride([out,retval] double ** val);
  HRESULT _stdcall getSpindleRateOverride([out,retval] double ** val);
  HRESULT _stdcall getJerkLimit([out,retval] LinearJerk ** lj);
  HRESULT _stdcall getInPosition([out,retval] boolean ** flag);
  HRESULT _stdcall setInPosition([in] boolean value); /* privapte method*/

  // See Note 1
  HRESULT _stdcall getActualAxisPosition([in] long axisNo, [out,retval] Measure **value );
  HRESULT _stdcall getActualAxesPositions([out, retval] OacVector ** vector);
  HRESULT _stdcall getXformedActualPositions([out,retval] CoordinateFrame ** coord );
  HRESULT _stdcall getCommandedAxisPosition([in] long axisNo, [out,retval]  Measure ** dVal );
  HRESULT _stdcall getCommandedAxesPositions([out,retval] OacVector ** vector );
  HRESULT _stdcall  getXformedCommandedPositions([in] OacVector axisPositions, [out,retval]
CoordinateFrame ** cf );

  HRESULT _stdcall  getAccmode([out,retval] ACCMode ** accmode);

  // KINEMATIC INFORMATION
  //================================================
  // Axis under control
  HRESULT _stdcall getCoordinatedAxes([out,retval] CoordinatedAxes ** ca);
  HRESULT _stdcall getKinstructure([out,retval] IKinStructure ** kin );
  HRESULT _stdcall setKinstructure([in] IKinStructure value);
  HRESULT _stdcall getToolTransform([out,retval] Transform ** t);
  HRESULT _stdcall getBaseframe([out,retval] Transform ** t);
  HRESULT _stdcall setBaseframe([in] CoordinateFrame value);

  // recovery from fault error, sharing
  HRESULT _stdcall inhibitAxis([in]  long axisNo, [in] boolean inhibit );
  HRESULT _stdcall axisInhibitd([in] long axisNo, [out,retval] boolean ** flag );
  HRESULT _stdcall inhibitSpindle([in] boolean inhibit );
  HRESULT _stdcall spindleInhibitd([out,retval] boolean ** flag);

  // TRAJECTORY INFORMATION
  //===============================================
  HRESULT _stdcall setBlending([in] boolean flag);      // TRUE=ON, FALSE=OFF
  HRESULT _stdcall setSingleStep([in] boolean flag);   // TRUE=ON, FALSE=OFF

  // HRESULT _stdcall  setVpg([in] IVelocityProfileGenerator vpg);
  // VelocityProfileGenerator getVpg();

  // Timing is now a reference to another object
  // timeMeasure getAxisupdateinterval() const;
  // HRESULT _stdcall  setAxisupdateinterval(timeMeasure value);
    //  attribute Time timing;

  HRESULT _stdcall setPhysicalLimits([in] Rate limits); //+ 3-Jun-1997
  HRESULT _stdcall getPhysicalLimits([out,retval] Rate ** r );        //+
};

// NOTES
// 1. There is a problem in JAVA with returning data type.
// Storing into calling parameter as a side effect Side
// instead of
//       OacVector getCommandedAxesPositions( );
// use
//       getCommandedAxesPositions( OacVector positions );
// It is possible to redo above in this signature style.
// 2. Issue: There are issues as to maximum acceleration of device
// versus Control Plan Unit (Motion Segment)

// Control Plan Class Definitions- Motion Segments


[
    object,
    uuid(134A0293-E101-11d2-B512-AEC041D2957B),

    helpstring("Path Node Interface"),
    pointer_default(unique)
]

interface IPathNode
{
  HRESULT _stdcall  getControltransform([out,retval] Transform ** t);
  HRESULT _stdcall  setControltransform(Transform value);
};
[
    object,
    uuid(134A0294-E101-11d2-B512-AEC041D2957B),

    helpstring("PathElement Interface"),
    pointer_default(unique)
]
interface IPathElement :  IKinematicPath
{
  HRESULT _stdcall initAccDecProfile([in] LinearVelocity vel);
  HRESULT _stdcall setStartPoint([in] IPathNode startPoint ); // axgroup sets
  HRESULT _stdcall getStartPoint([out,retval] IPathNode ** pn );
  HRESULT _stdcall getEndPoint([out,retval] IPathNode ** pn );     // axgroup sets
  // HRESULT _stdcall  setEndPoint([in] IPathNode endPoint);    // ppt or internal use
  HRESULT _stdcall getDistanceToGo([out,retval] LengthMeasure ** len);
  HRESULT _stdcall isPathComplete([out,retval] boolean ** flag);
  HRESULT _stdcall pathLength([out,retval] LengthMeasure ** len );
  // LengthMeasure pathLength(XYZ xyz); // what is this
};


[
    object,
    uuid(134A0295-E101-11d2-B512-AEC041D2957B),

    helpstring("Rate Interface"),
    pointer_default(unique)
]
```

```
interface IRate
{
  HRESULT _stdcall setNominalFeedrate([in] double vnom);
  HRESULT _stdcall setCurrentFeedrate([in] double vmax, [out,retval] long ** pplVal );      //
includes override
  HRESULT _stdcall setMaximumAcceleration([in] double amax, [out,retval] long **pplVal);
  HRESULT _stdcall setMaximumJerk([in] double jmax, [out,retval] long **value);

  HRESULT _stdcall getNominalFeedrate([out,retval] double ** ppdVal);
  HRESULT _stdcall getCurrentFeedrate([out,retval] double ** ppdVal);             // includes
override
  HRESULT _stdcall getMaximumAcceleration([out,retval] double ** ppdVal);
  HRESULT _stdcall getMaximumJerk([out,retval] double ** ppdVal);

  HRESULT _stdcall getCurrentVelocity([out,retval] double ** val);
  HRESULT _stdcall setCurrentVelocity([in] double vcur);

  HRESULT _stdcall getFinalVelocity([out,retval] double  ** val );
  HRESULT _stdcall setFinalVelocity([in] double vcur);

  HRESULT _stdcall getCurrentAcceleration([out,retval]  double ** val);
  HRESULT _stdcall setCurrentAcceleration([in] double acur);

  HRESULT _stdcall getAccState([out,retval] long **value);
  HRESULT _stdcall setAccState([in] long val);
  HRESULT _stdcall isDone([out,retval] boolean ** flag);
  HRESULT _stdcall isAccel([out,retval] boolean ** flag);
  HRESULT _stdcall isConst([out,retval] boolean ** flag);
  HRESULT _stdcall isDecel([out,retval] boolean ** flag);

  HRESULT _stdcall setNominalSpindleSpeed([in] double spd); // why here?
  HRESULT _stdcall getNominalSpindleSpeed([out,retval] double ** val );
};
[
    object,
    uuid(134A0296-E101-11d2-B512-AEC041D2957B),

    helpstring("Kinematic Info Interface"),
    pointer_default(unique)
]
interface  IKinematicInfo
{
  HRESULT _stdcall  setToolCenter([in] Length effectiveDisplacement,
             [in] CRCMODE cutterRadiusCompensation);

  HRESULT _stdcall getCurrentFrame([out,retval] Transform ** tr);
  HRESULT _stdcall setCurrentFrame([in] Transform currentFrame );

  HRESULT _stdcall getKinematics([out,retval] IKinMechanism ** kin);
  HRESULT _stdcall setKinematics ([in] IKinMechanism kin);
};

[
    object,
    uuid(134A0297-E101-11d2-B512-AEC041D2957B),

    helpstring("VelocityProfileGenerator Interface"),
    pointer_default(unique)
]
interface IVelocityProfileGenerator
{
  HRESULT _stdcall getAccdecprofile([out,retval]  AccDecProfile ** accdec);
  HRESULT _stdcall setAccdecprofile([in] AccDecProfile value);

  HRESULT _stdcall setBlendingPointDistance([in] double distance );
  HRESULT _stdcall getBlendingPointDistance([out,retval]  double ** val);

  HRESULT _stdcall getSamplingTime([out,retval] Time ** t);
  HRESULT _stdcall  setSamplingTime([in] Time value);
  /* New  3-Jun-1997 */
  HRESULT _stdcall  holdSegment();
  HRESULT _stdcall  pauseSegment();
  HRESULT _stdcall  resumeSegment();
```

```
};
// Base Class for Motion Segment
// Derived from ControlPlanUnit  - see part program translator
[
    object,
    uuid(134A0298-E101-11d2-B512-AEC041D2957B),

    helpstring("MotionSegment Interface"),
    pointer_default(unique)
]
interface IMotionSegment  : IControlPlanUnit
{
  HRESULT _stdcall getKinematicInfo (KinematicInfo **kin);
  HRESULT _stdcall setKinematicInfo (KinematicInfo kin);

  HRESULT _stdcall setVpg([in] IVelocityProfileGenerator  aVPG);
  HRESULT _stdcall getVpg([out,retval]  IVelocityProfileGenerator ** vpg);

  HRESULT _stdcall setTranslationalRate([in] IRate rate);
  HRESULT _stdcall getTranslationalRate([out,retval]  IRate ** rate );

  HRESULT _stdcall setOrientationRate([in] Rate rate);
  HRESULT _stdcall getOrientationRate([out,retval]  Rate ** rate);

  HRESULT _stdcall setAngularRate([in] IRate rate); // does this belong in axis group?
  HRESULT _stdcall getAngularRate([out,retval]  IRate ** rate);

  // if internal velocity profile generation supply this interface
  HRESULT _stdcall setBlendingPointDistance([in] double distance );
  HRESULT _stdcall getBlendingPointDistance([out,retval]  double **val);

  HRESULT _stdcall calcDistanceRemaining([out,retval]  Length **l); // axes

  HRESULT _stdcall getIncrementalDistance([out,retval] OacVector **vector );
  HRESULT _stdcall getLengthsRemaining([out,retval]  OacVector ** vector );  // per axis
  HRESULT _stdcall calcNextIncrement([in] double feedOverride,
                [in] double spindleOverride,
                [out,retval] OacVector ** vector
                );
  HRESULT _stdcall  startNextSegment([out,retval] boolean ** flag); //? what does this mean init?
//?  int init(double cycleTime); //+ 3-Jun-1997
  HRESULT _stdcall pauseSegment();
  HRESULT _stdcall holdSegment();  /* new */
  HRESULT _stdcall stopSegment();  /* new 3-Jun-1997 set motion to done */
  HRESULT _stdcall resumeSegment();
  HRESULT _stdcall isPaused([out,retval] boolean ** flag);
  HRESULT _stdcall isHeld([out,retval] boolean ** flag);

#ifdef SKIPTHIS

  // Program information (file, line number, block) and signals(active)
  HRESULT _stdcall  setPpb( PartProgramBlock ppb );
  HRESULT _stdcall  segmentStarted();
  HRESULT _stdcall  segmentFinished();
#endif
};
//NOTES:
// 1. Handling Termination Condition:
// a. Exact Stop = blending distance=0

[
    uuid(134A0299-E101-11d2-B512-AEC041D2957B),
    version(1.0),
    helpstring("Axis Group Module 1.0 Type Library")
]
library AXIS_GROUP_MODULE_Lib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [
        uuid(134A029A-E101-11d2-B512-AEC041D2957B),
        helpstring("Axis Group Class")
    ]
    coclass AxisGroup
    {
        [default] interface IAxisGroup;
        interface IOmac;
    };
    [
        uuid(134A029B-E101-11d2-B512-AEC041D2957B),
        helpstring("PathNode Class")
    ]
    coclass PathNode
    {
        [default] interface IPathNode;
    };

    [
        uuid(134A029C-E101-11d2-B512-AEC041D2957B),
        helpstring("PathElement Class")
    ]
    coclass PathElement
    {
        [default] interface IPathElement;
        interface IKinematicPath;
    };

    [
        uuid(134A029D-E101-11d2-B512-AEC041D2957B),
        helpstring("Rate Class")
    ]
    coclass Rate
    {
        [default] interface IRate;
    };

    [
        uuid(134A029B-E101-11d2-B512-AEC041D2957B),
        helpstring(" KinematicInfo Class")
    ]
    coclass  KinematicInfo
    {
        [default] interface IKinematicInfo;
    };

    [
        uuid(134A029E-E101-11d2-B512-AEC041D2957B),
        helpstring("VelocityProfileGenerator Class")
    ]
    coclass VelocityProfileGenerator
    {
        [default] interface IVelocityProfileGenerator;
    };

    [
        uuid(134A029F-E101-11d2-B512-AEC041D2957B),
        helpstring("MotionSegment Class")
    ]
    coclass MotionSegment
    {
        [default] interface IMotionSegment;
        interface IControlPlanUnit;
    };

};


#endif

```

B.14 AXIS

```
// AxisModule.idl : IDL source for AxisModule.dll
```

```
//

// This file will be processed by the MIDL tool to
// produce the type library (AxisModule.tlb) and marshalling code.

import "oaidl.idl";
import "ocidl.idl";

import "DataRepresentation.idl";
import "OmacModule.idl";
import "ControlLawModule.idl";

interface IAxis;
interface IAxisAbsolutePos;
interface IAxisAccelerationServo;
interface IAxisCommandedInput;
interface IAxisCommandedOutput;
interface IAxisDyn;
interface IAxisErrorAndEnable;
interface IAxisForceServo;
interface IAxisHoming;
interface IAxisIncrementPos;
interface IAxisKinematics;
interface IAxisJogging;
interface IAxisLimits;
interface IAxisMaintenance;
interface IAxisPositioningServo;
interface IAxisRates;
interface IAxisSensedState;
interface IAxisSetup;
interface IAxisVelocityServo;

typedef double AxisAccelCmd;
typedef double AxisForceCmd;
typedef double AxisPositionCmd;
typedef double AxisVelocityCmd;

// {9C56BEC5-07CB-11d3-AAB2-00C04FA375A6}
cpp_quote("const CATID CATID_AxisModule = { 0x9c56bec5, 0x7cb, 0x11d3, { 0xaa, 0xb2, 0x0, 0xc0,
0x4f, 0xa3, 0x75, 0xa6 } };")

//const GUID CATID_ControlLawModule =
{0xE1D6F9F1,0xB1FE,0x11D2,{0xAA,0xA8,0x00,0xC0,0x4F,0xA3,0x75,0xA6}};

// Example: This CLSID  is specific for one vendor, (i.e., NIST) Control Law Server
// {803B45C1-07CB-11d3-AAB2-00C04FA375A6}
cpp_quote("const CLSID CLSID_NISTAxisModuleServer = { 0x803b45c1, 0x07cb, 0x11d3, { 0xaa, 0xb2,
0x0, 0xc0, 0x4f, 0xa3, 0x75, 0xa6 } };")

    [
        object,
        uuid(0A70EBB0-06D9-11D3-AAB2-00C04FA375A6),

        helpstring("IAxisModuleClassFactory Interface"),
        pointer_default(unique)
    ]
    interface IAxisModuleClassFactory : IUnknown
    {
        HRESULT _stdcall CreateModule([in] BSTR name, [in] REFIID riid, [out, iid_is(riid)] void
** ppvObj);
    };

[

 uuid(AA03FCE5-FF08-11D2-AAB2-00C04FA375A6),

 helpstring("IAxis Interface"),
 pointer_default(unique)
]
interface IAxis : IOmac
{
  // Get Reference Objects
  HRESULT _stdcall getAbsolutePos([out,retval] IAxisAbsolutePos ** val);
  HRESULT _stdcall getAccelerationServo([out,retval] IAxisAccelerationServo ** a);
  HRESULT _stdcall getCommandedInput([out,retval] IAxisCommandedInput ** a);
  HRESULT _stdcall getCommandedOutput([out,retval] IAxisCommandedOutput** a);
  HRESULT _stdcall getDynamics([out,retval] IAxisDyn ** val);
  HRESULT _stdcall getErrorAndEnable([out,retval] IAxisErrorAndEnable** a);
  HRESULT _stdcall getForceServo([out,retval] IAxisForceServo **a);
  HRESULT _stdcall getHoming([out,retval] IAxisHoming **  a);
  HRESULT _stdcall getIncrementPosition([out,retval] IAxisIncrementPos ** a);
  HRESULT _stdcall getJogging([out,retval] IAxisJogging ** a);
  HRESULT _stdcall getKinematics([out,retval] IAxisKinematics ** val);
  HRESULT _stdcall getLimits([out,retval] IAxisLimits ** val);
  HRESULT _stdcall getMaintenance([out,retval] IAxisMaintenance ** val);
  HRESULT _stdcall getPositioningServo([out,retval] IAxisPositioningServo ** a);
  HRESULT _stdcall getSensedState([out,retval] IAxisSensedState ** a);
  HRESULT _stdcall getSetup([out,retval] IAxisSetup ** val);
  HRESULT _stdcall getVelocityServo([out,retval] IAxisVelocityServo ** a);

  HRESULT _stdcall setAbsolutePos([in] IAxisAbsolutePos * val);
  HRESULT _stdcall setAccelerationServo([in] IAxisAccelerationServo * val);
  HRESULT _stdcall setCommandedInput([in] IAxisCommandedInput * val);
  HRESULT _stdcall setCommandedOutput([in] IAxisCommandedOutput * val);
  HRESULT _stdcall setErrorAndEnable([in] IAxisErrorAndEnable * val);
  HRESULT _stdcall setForceServo([in] IAxisForceServo * val);
  HRESULT _stdcall setHoming([in] IAxisHoming * val);
  HRESULT _stdcall setIncrementPosition([in] IAxisIncrementPos * val);
  HRESULT _stdcall setJogging([in] IAxisJogging * val);
  HRESULT _stdcall setKinematics([in] IAxisKinematics * val);
  HRESULT _stdcall setLimits([in] IAxisLimits * val);
  HRESULT _stdcall setMaintenance([in] IAxisMaintenance * val);
  HRESULT _stdcall setPositioningServo([in] IAxisPositioningServo * val);
  HRESULT _stdcall setSensedState([in] IAxisSensedState * val);
  HRESULT _stdcall setSetup([in] IAxisSetup * val);
  HRESULT _stdcall setVelocityServo([in] IAxisVelocityServo * val);

  HRESULT _stdcall setPositionControlLaw([in] IControlLaw * val);
  HRESULT _stdcall setVelocityControlLaw([in] IControlLaw * val);
  HRESULT _stdcall setAccelerationControlLaw([in] IControlLaw * val);
  HRESULT _stdcall getPositionControlLaw([out,retval] IControlLaw ** a);
  HRESULT _stdcall getVelocityControlLaw([out,retval] IControlLaw ** a);
  HRESULT _stdcall getAccelerationControlLaw([out,retval] IControlLaw ** a);


  HRESULT _stdcall processServoLoop( ); // the primary function.
  HRESULT _stdcall checkPreconditions([out, retval] long * val); //  checked at every servo loop.

    // State transition methods and state queries
  HRESULT _stdcall disableAxis();            // DISABLEEvent
  HRESULT _stdcall enableAxis();             // ENABLEEvent
  HRESULT _stdcall followCommandedPosition(); // FOLLOWPositionEvent
  HRESULT _stdcall followCommandedTorque();   // FOLLOWTorqueEvent
  HRESULT _stdcall followCommandedVelocity(); // FOLLOWVelocityEvent
  HRESULT _stdcall followCommandedForce();    // FOLLOWForceEvent
  HRESULT _stdcall home([in] double velocity);  // STARTHomeEvent
  HRESULT _stdcall jog([in] double velocity);   // STARTJogEvent
  HRESULT _stdcall resetAxis();              // RESETEvent
  HRESULT _stdcall stopMotion();             // CANCELEvent
  HRESULT _stdcall estop();                                                                   // ESTOPEvent
  HRESULT _stdcall updateAxis();             // UPDATEEvent

    // Instead of:
    // int currentState();
    //  DISABLED               =  1,
    //  ENABLED                =  2,
    //  EStopped              =  3,
    //  FOLLOWINGPosition     =  4,
    //  FOLLOWINGTorque       =  5,
    //  FOLLOWINGVelocity     =  6,
    //  HOMING                 =  7,
    //  JOGGING                =  8,
    //  STOPPING               =  9;  // Use accessor functions so there is no confusion about numbering
    // Also inherit state queries from OMAC Base Module

  HRESULT _stdcall isFollowingAcceleration([out,retval] boolean * b);
  HRESULT _stdcall isFollowingForce([out,retval] boolean * b);
  HRESULT _stdcall isFollowingPosition([out,retval] boolean *  b);
  HRESULT _stdcall isFollowingVelocity([out,retval] boolean *  b);
  HRESULT _stdcall isHoming([out,retval] boolean * b);
  HRESULT _stdcall isIncrementingPosition([out,retval] boolean * b);
  HRESULT _stdcall isJogging([out,retval] boolean * b);
  HRESULT _stdcall isMovingto([out,retval] boolean * b);

  HRESULT _stdcall isReset([out,retval] boolean * b);
  HRESULT _stdcall isInited([out,retval] boolean * b);
  HRESULT _stdcall isEnabled([out,retval] boolean * b);
  HRESULT _stdcall isDisabled([out,retval] boolean * b);
  HRESULT _stdcall isReady([out,retval] boolean * b);
  HRESULT _stdcall isEstopped([out,retval] boolean * b);

    // Add isStopping() which includes any stopping?
    // Returns a ASCII readable string
  HRESULT _stdcall currentStateName([out,retval] BSTR * name);

};

[

 uuid(50B62B8D-4513-11d3-AABB-00C04FA375A6),

 helpstring("IAxisAccelerationServo Interface"),
 pointer_default(unique)
]
interface IOmacAxis : IOmac
{  HRESULT _stdcall setAxisContainer([in] IAxis * a);
};

[

 uuid(AA03FCE7-FF08-11D2-AAB2-00C04FA375A6),

 helpstring("IAxisAccelerationServo Interface"),
 pointer_default(unique)
]
interface IAxisAccelerationServo : IOmacAxis
{
  // All invoked by Axis FSM
  HRESULT _stdcall stopFollowingAccelerationAction();
  HRESULT _stdcall estopFollowingAccelerationAction();
  HRESULT _stdcall startFollowingAccelerationAction();
  HRESULT _stdcall updateFollowingAccelerationAction();

  HRESULT _stdcall isDone([out,retval] boolean * b);
  HRESULT _stdcall isFollowingAccelerationError([out,retval] boolean * b);


};
[

 uuid(AA03FCE9-FF08-11D2-AAB2-00C04FA375A6),

 helpstring("IAxisCommandedInput Interface"),
 pointer_default(unique)
]
interface IAxisCommandedInput : IOmacAxis
{
  HRESULT _stdcall getPositionCmdInput([out,retval] AxisPositionCmd * a);
  HRESULT _stdcall getVelocityCmdInput([out,retval] AxisVelocityCmd * a);
  HRESULT _stdcall getAccelerationCmdInput([out,retval] AxisAccelCmd * a);
  HRESULT _stdcall getForceCmdInput([out,retval] AxisForceCmd * a);
  HRESULT _stdcall setPositionCmdInput([in] AxisPositionCmd  positioningCmd );
  HRESULT _stdcall setVelocityCmdInput([in] AxisVelocityCmd  velocityCmd );
  HRESULT _stdcall setAccelerationCmdInput([in] AxisAccelCmd accelerationCmd );
  HRESULT _stdcall setForceCmdInput([in] AxisForceCmd  forceCmd );
  HRESULT _stdcall updateCommandedInput();  // updates using connections to IO

```

```
};

[

 uuid(AA03FCEB-FF08-11D2-AAB2-00C04FA375A6),

 helpstring("IAxisCommandedOutput Interface"),
 pointer_default(unique)
]
interface IAxisCommandedOutput : IOmacAxis
{
  HRESULT _stdcall getPositionCmdOutput([out,retval] AxisPositionCmd * a);
  HRESULT _stdcall getVelocityCmdOutput([out,retval] AxisVelocityCmd * a);
  HRESULT _stdcall getAccelerationCmdOutput([out,retval] AxisAccelCmd * a);
  HRESULT _stdcall getForceCmdOutput([out,retval] AxisForceCmd * a);
  HRESULT _stdcall setPositionCmdOutput([in] AxisPositionCmd  positioningCmd );
  HRESULT _stdcall setVelocityCmdOutput([in] AxisVelocityCmd  velocityCmd );
  HRESULT _stdcall setAccelerationCmdOutput([in] AxisAccelCmd accelerationCmd );
  HRESULT _stdcall setForceCmdOutput([in] AxisForceCmd  forceCmd );
  HRESULT _stdcall updateCommandedOutput();  // updates using connections to IO

};
[

 uuid(AA03FCED-FF08-11D2-AAB2-00C04FA375A6),

 helpstring("IAxisDyn Interface"),
 pointer_default(unique)
]
interface IAxisDyn : IOmacAxis
{
  HRESULT _stdcall getAccelerationLimit([out, retval] LinearAcceleration *pVal);
  HRESULT _stdcall getAxmass([in] Mass newVal);
  HRESULT _stdcall getBacklash([out, retval] Length *pVal);
  HRESULT _stdcall getDamping([out, retval] Force *pVal);
  HRESULT _stdcall getDeadband([out, retval] Length *pVal);
  HRESULT _stdcall getDecelerationLimit([out, retval] LinearAcceleration *pVal);
  HRESULT _stdcall getInertia([out, retval] Mass *pVal);
  HRESULT _stdcall getJerkLimit([out, retval] LinearJerk *pVal);
  HRESULT _stdcall getLoadedCaseSpringRate([out, retval] LinearStiffness *pVal);
  HRESULT _stdcall getMaxVelAccLim([out, retval] LinearAcceleration *pVal);
  HRESULT _stdcall getOvershootStepInput([out, retval] Length *pVal);
  HRESULT _stdcall getQuasiStaticLoadLimit([out, retval] Force *pVal);
  HRESULT _stdcall getRisingTimeStepInput([out, retval] Time *pVal);
  HRESULT _stdcall getRunFriction([out, retval] Force *pVal);
  HRESULT _stdcall getStaticFriction([out, retval] Force *pVal);
  HRESULT _stdcall getTimeConstant([out, retval] Time *pVal);
  HRESULT _stdcall getWorstCaseSpringRate([out, retval] LinearStiffness *pVal);
  HRESULT _stdcall getZeroVelAccLim([out, retval] LinearAcceleration *pVal);

  HRESULT _stdcall setAccelerationLimit([in] LinearAcceleration newVal);
  HRESULT _stdcall setAxmass([out, retval] Mass *pVal);
  HRESULT _stdcall setBacklash([in] Length newVal);
  HRESULT _stdcall setDamping([in] Force newVal);
  HRESULT _stdcall setDeadband([in] Length newVal);
  HRESULT _stdcall setDecelerationLimit([in] LinearAcceleration newVal);
  HRESULT _stdcall setInertia([in] Mass newVal);
  HRESULT _stdcall setJerkLimit([in] LinearJerk newVal);
  HRESULT _stdcall setLoadedCaseSpringRate([in] LinearStiffness newVal);
  HRESULT _stdcall setMaxVelAccLim([in] LinearAcceleration newVal);
  HRESULT _stdcall setOvershootStepInput([in] Length newVal);
  HRESULT _stdcall setQuasiStaticLoadLimit([in] Force newVal);
  HRESULT _stdcall setRisingTimeStepInput([in] Time newVal);
  HRESULT _stdcall setRunFriction([in] Force newVal);
  HRESULT _stdcall setStaticFriction([in] Force newVal);
  HRESULT _stdcall setTimeConstant([in] Time newVal);
  HRESULT _stdcall setWorstCaseSpringRate([in] LinearStiffness newVal);
  HRESULT _stdcall setZeroVelAccLim([in] LinearAcceleration newVal);

};

[

 uuid(AA03FCF2-FF08-11D2-AAB2-00C04FA375A6),

 helpstring("IAxisForceServo Interface"),
 pointer_default(unique)
]
interface IAxisForceServo : IOmacAxis
{
  // All invoked by Axis FSM

  HRESULT _stdcall stopFollowingForceAction();
  HRESULT _stdcall estopFollowingForceAction();
  HRESULT _stdcall startFollowingForceAction();
  HRESULT _stdcall updateFollowingForceAction();

  HRESULT _stdcall isDone([out,retval] boolean * b);
  HRESULT _stdcall isFollowingForceError([out,retval] boolean * b);

};
[

 uuid(AA03FCF4-FF08-11D2-AAB2-00C04FA375A6),

 helpstring("IAxisErrorAndEnable Interface"),
 pointer_default(unique)
]
interface IAxisErrorAndEnable : IOmacAxis
{
  HRESULT _stdcall resetAxisAction();
  HRESULT _stdcall disableAxisAction();
  HRESULT _stdcall enableAxisAction();
  HRESULT _stdcall estopAxisAction();
};

[

 uuid(AA03FCF6-FF08-11D2-AAB2-00C04FA375A6),

 helpstring("IAxisHoming Interface"),
 pointer_default(unique)
]
interface IAxisHoming : IOmacAxis
{
  HRESULT _stdcall  startHomingAction([in] double startVelocity ); // prepares homing
  HRESULT _stdcall  updateHomingAction();           // called each servo cycle
  HRESULT _stdcall  stopHomingAction();             // stops homing before completion
  HRESULT _stdcall  estopHomingAction();           // On transition from homing to E-stopped
  HRESULT _stdcall  completedHomingAction();          // On transition from homing to disabled
  HRESULT _stdcall isDone([out,retval] boolean * b);                     // signals when homing is completed
  HRESULT _stdcall isStopping([out,retval] boolean * b);
  HRESULT _stdcall isHomingError([out,retval] boolean * b);                // true if error has occurred during homing
};
[

 uuid(AA03FCF8-FF08-11D2-AAB2-00C04FA375A6),

 helpstring("IAxisJogging Interface"),
 pointer_default(unique)
]
interface IAxisJogging : IOmacAxis
{
  HRESULT _stdcall  completedJoggingAction();
  HRESULT _stdcall  estopJoggingAction();
  HRESULT _stdcall  startJoggingAction([in] double targetVelocity );
  HRESULT _stdcall  stopJoggingAction();
  HRESULT _stdcall  updateJoggingAction();
  HRESULT _stdcall  updateJoggingStoppingAction();

  HRESULT _stdcall  isDone([out,retval] boolean * b);
  HRESULT _stdcall  isStopping([out,retval] boolean * b);
  HRESULT _stdcall  isJoggingError([out,retval] boolean *  b);

```

```
};
[

 uuid(AA03FCFA-FF08-11D2-AAB2-00C04FA375A6),

 helpstring("IAxisKinematics Interface"),
 pointer_default(unique)
]
interface IAxisKinematics : IOmacAxis
{
  HRESULT _stdcall getKs([out, retval] double *pVal);
  HRESULT _stdcall setKs([in] double newVal);
  HRESULT _stdcall getPosFeedBackGain([out, retval] double *pVal);
  HRESULT _stdcall setPosFeedBackGain([in] double newVal);
  HRESULT _stdcall getVelFeedBackGain([out, retval] double *pVal);
  HRESULT _stdcall setVelFeedBackGain([in] double newVal);
  HRESULT _stdcall getUpperKinematicModel([out, retval] UpperKinematicModel *pVal);
  HRESULT _stdcall setUpperKinematicModel([in] UpperKinematicModel newVal);
  HRESULT _stdcall getLowerKinematicModel([out, retval] LowerKinematicModel *pVal);
  HRESULT _stdcall setLowerKinematicModel([in] LowerKinematicModel newVal);
  HRESULT _stdcall getPlacement([out, retval] CoordinateFrame *pVal);
  HRESULT _stdcall setPlacement([in] CoordinateFrame newVal);

};
[

 uuid(AA03FCFD-FF08-11D2-AAB2-00C04FA375A6),

 helpstring("IAxisLimits Interface"),
 pointer_default(unique)
]
interface IAxisLimits : IOmacAxis
{
  HRESULT _stdcall getCutOffPosition([out, retval] Length *pVal);
  HRESULT _stdcall getFollowingErrorViolationLim([out, retval] Length *pVal);
  HRESULT _stdcall getFollowingErrorWarnLim([out, retval] Length *pVal);
  HRESULT _stdcall getHardFwdOTravelLim([out, retval] Length *pVal);
  HRESULT _stdcall getHardRevOTravelLim([out, retval] Length *pVal);
  HRESULT _stdcall getJerkLimit([out, retval] LinearJerk *pVal);
  HRESULT _stdcall getMaxForceLimit([out, retval] Force *pVal);
  HRESULT _stdcall getMaxVelocity([out, retval] LinearVelocity *pVal);
  HRESULT _stdcall getOvershootViolationLim([out, retval] Length *pVal);
  HRESULT _stdcall getOvershootWarnLevelLimit([out, retval] Length *pVal);
  HRESULT _stdcall getSoftFwdOTravelLim([out, retval] Length *pVal);
  HRESULT _stdcall getSoftRevOTravelLim([out, retval] Length *pVal);
  HRESULT _stdcall getUnderreachViolationLim([out, retval] Length *pVal);
  HRESULT _stdcall getUnderreachWarnLevelLimit([out, retval] Length *pVal);
  HRESULT _stdcall getUsefulTravel([out, retval] Length *pVal);

  HRESULT _stdcall setCutOffPosition([in] Length newVal);
  HRESULT _stdcall setFollowingErrorViolationLim([in] Length newVal);
  HRESULT _stdcall setFollowingErrorWarnLim([in] Length newVal);
  HRESULT _stdcall setHardFwdOTravelLim([in] Length newVal);
  HRESULT _stdcall setHardRevOTravelLim([in] Length newVal);
  HRESULT _stdcall setJerkLimit([in] LinearJerk newVal);
  HRESULT _stdcall setMaxForceLimit([in] Force newVal);
  HRESULT _stdcall setMaxVelocity([in] LinearVelocity newVal);
  HRESULT _stdcall setOvershootViolationLim([in] Length newVal);
  HRESULT _stdcall setOvershootWarnLevelLimit([in] Length newVal);
  HRESULT _stdcall setSoftFwdOTravelLim([in] Length newVal);
  HRESULT _stdcall setSoftRevOTravelLim([in] Length newVal);
  HRESULT _stdcall setUnderreachViolationLim([in] Length newVal);
  HRESULT _stdcall setUnderreachWarnLevelLimit([in] Length newVal);
  HRESULT _stdcall setUsefulTravel([in] Length newVal);

};
[

 uuid(AA03FCFF-FF08-11D2-AAB2-00C04FA375A6),

 helpstring("IAxisMaintenance Interface"),
 pointer_default(unique)
]
```

```
interface IAxisMaintenance : IOmacAxis
{
};

[

 uuid(AA03FD01-FF08-11D2-AAB2-00C04FA375A6),

 helpstring("IAxisPositioningServo Interface"),
 pointer_default(unique)
]
interface IAxisPositioningServo : IOmacAxis
{
  // All invoked by Axis FSM
  HRESULT _stdcall stopFollowingPositionAction();
  HRESULT _stdcall estopFollowingPositionAction();
  HRESULT _stdcall startFollowingPositionAction();
  HRESULT _stdcall updateFollowingPositionAction();

  HRESULT _stdcall isDone([out,retval] boolean * b);
  HRESULT _stdcall isFollowingPositionError([out,retval] boolean * b);

};

[

 uuid(AA03FD03-FF08-11D2-AAB2-00C04FA375A6),

 helpstring("IAxisRates Interface"),
 pointer_default(unique)
]
interface IAxisRates : IOmacAxis
{
  //Specifications of travel capabilities.
  //worst-case conditions.  But to take advantage of more
  //capability provide a model that describes conditions
  //when more capability is available and the corresponding
  //values or value-functions.
  // FIXME: Problem here with typedef derivative of double, versus real class definition?
  HRESULT _stdcall getMaxAcceleration([out, retval] LinearAcceleration *pVal);
  HRESULT _stdcall getMaxJerk([out, retval] LinearJerk *pVal);
  HRESULT _stdcall getMaxTravel([out, retval] Length *pVal);
  HRESULT _stdcall getMaxVelocity([out, retval] LinearVelocity *pVal);
  HRESULT _stdcall getPosErrRatioCutMoving([out, retval] Length *pVal);
  HRESULT _stdcall getPosErrRatioIdleMoving([out, retval] Length *pVal);
  HRESULT _stdcall getPosErrRatioIdleStationary([out, retval] Length *pVal);
  HRESULT _stdcall getRepeatability([out, retval] long *pVal);

  HRESULT _stdcall setMaxAcceleration([in] LinearAcceleration newVal);
  HRESULT _stdcall setMaxJerk([in] LinearJerk newVal);
  HRESULT _stdcall setMaxTravel([in] Length newVal);
  HRESULT _stdcall setMaxVelocity([in] LinearVelocity newVal);
  HRESULT _stdcall setPosErrRatioCutMoving([in] Length newVal);
  HRESULT _stdcall setPosErrRatioIdleMoving([in] Length newVal);
  HRESULT _stdcall setPosErrRatioIdleStationary([in] Length newVal);
  HRESULT _stdcall setRepeatability([in] long newVal);

};
[

 uuid(AA03FD05-FF08-11D2-AAB2-00C04FA375A6),

 helpstring("IAxisSensedState Interface"),
 pointer_default(unique)
]
interface IAxisSensedState : IOmacAxis
{
  //if(!hardFwdOTravel) && if(!softFwdOTravel) &&if(!hardRevOTravel) &&
  //   if(!softRevOTravel)
  //then enablingPrecondition = 1;
  //else enablingPrecondition = 0;
  //   Concurrency: Sequential
  HRESULT _stdcall getEnablingPrecondition([out, retval] boolean * b);
  HRESULT _stdcall inPosition([out, retval] boolean * pVal);
  HRESULT _stdcall isSoftFwdOTravel([out, retval] boolean *pVal);
  HRESULT _stdcall isHardFwdOTravel([out, retval] boolean *pVal);
  HRESULT _stdcall isSoftRevOTravel([out, retval] boolean *pVal);
  HRESULT _stdcall isHardRevOTravel([out, retval] boolean *pVal);
  HRESULT _stdcall isFollowingErrorWarn([out, retval] boolean *pVal);
  HRESULT _stdcall isFollowingErrorViolation([out, retval] boolean *pVal);
  HRESULT _stdcall isOverShootViolation([out, retval] boolean *pVal);
  HRESULT _stdcall isEnablingPrecondition([out, retval] boolean *pVal);

  HRESULT _stdcall setAxisContainer([in] IAxis * a);

  HRESULT _stdcall getActualPosition([out,retval]  Length * a);
  HRESULT _stdcall getActualVelocity([out,retval]  LinearVelocity * a);
  HRESULT _stdcall getActualAcceleration([out,retval]  LinearAcceleration * a);
  HRESULT _stdcall getActualForce([out,retval]  Force * a);

};
[

 uuid(AA03FD07-FF08-11D2-AAB2-00C04FA375A6),

 helpstring("IAxisSetup Interface"),
 pointer_default(unique)
]
interface IAxisSetup : IOmacAxis
{
  // sets the reference to the axis rates for physical limits, software limits.
  HRESULT _stdcall getCurrentRates([out, retval] IAxisRates **pVal);
  HRESULT _stdcall getDynamicRates([out, retval] IAxisDyn **pVal);
  HRESULT _stdcall getPhysicalLimits([out, retval] IAxisRates **pVal);
  HRESULT _stdcall setCurrentRates([in] IAxisRates * newVal);
  HRESULT _stdcall setDynamicRates([in] IAxisDyn * newVal);
  HRESULT _stdcall setPhysicalLimits([in] IAxisRates * newVal);

};

[

 uuid(AA03FD09-FF08-11D2-AAB2-00C04FA375A6),

 helpstring("IAxisVelocityServo Interface"),
 pointer_default(unique)
]
interface IAxisVelocityServo : IOmacAxis
{
  // All invoked by Axis FSM
  HRESULT _stdcall stopFollowingVelocityAction();
  HRESULT _stdcall estopFollowingVelocityAction();
  HRESULT _stdcall startFollowingVelocityAction();
  HRESULT _stdcall updateFollowingVelocityAction();

  HRESULT _stdcall isDone([out,retval] boolean * b);
  HRESULT _stdcall isFollowingVelocityError([out,retval] boolean * b);

};
[

 uuid(AA03FD0B-FF08-11D2-AAB2-00C04FA375A6),

 helpstring("IAxisAbsolutePos Interface"),
 pointer_default(unique)
]
interface IAxisAbsolutePos : IOmacAxis
{
  HRESULT _stdcall  completedAbsolutePosAction();
  HRESULT _stdcall  estopAbsolutePosAction();
  HRESULT _stdcall  startAbsolutePosAction([in] double targetVelocity );
  HRESULT _stdcall  stopAbsolutePosAction();
  HRESULT _stdcall  updateAbsolutePosAction();

  HRESULT _stdcall  isDone([out,retval] boolean * b);
  HRESULT _stdcall  isStopping([out,retval] boolean * b);
  HRESULT _stdcall  isAbsolutePosError([out,retval] boolean *  b);
};

[
 object,
 uuid(AA03FD0D-FF08-11D2-AAB2-00C04FA375A6),
 helpstring("IAxisIncrementPos Interface"),
 pointer_default(unique)
]
interface IAxisIncrementPos : IOmacAxis
{
  HRESULT _stdcall  completedIncrementingAction();
  HRESULT _stdcall  estopIncrementingAction();
  HRESULT _stdcall  startIncrementingAction([in] double targetVelocity );
  HRESULT _stdcall  stopIncrementingAction();
  HRESULT _stdcall  updateIncrementingAction();

  HRESULT _stdcall  isDone([out,retval] boolean * b);
  HRESULT _stdcall  isStopping([out,retval] boolean * b);
  HRESULT _stdcall  isIncrementingError([out,retval] boolean * b);

};
[
 uuid(AA03FCD8-FF08-11D2-AAB2-00C04FA375A6),
 version(1.0),
 helpstring("AxisModule 1.0 Type Library")
]
library AXISMODULELib
{
  importlib("stdole32.tlb");
  importlib("stdole2.tlb");


    [
        uuid(0A70EBB1-06D9-11D3-AAB2-00C04FA375A6),
        helpstring("AxisModuleClassFactory Class")
    ]
    coclass AxisModuleClassFactory
    {
        [default] interface IAxisModuleClassFactory;
         interface IOmacModuleClassFactory;
    };

  [
   uuid(AA03FCE6-FF08-11D2-AAB2-00C04FA375A6),
   helpstring("Axis Class")
  ]
    coclass Axis
    {
      [default] interface IAxis;
    };
  [
   uuid(AA03FCE8-FF08-11D2-AAB2-00C04FA375A6),
   helpstring("AxisAccelerationServo Class")
  ]
    coclass AxisAccelerationServo
    {
      [default] interface IAxisAccelerationServo;
    };
  [
   uuid(AA03FCEA-FF08-11D2-AAB2-00C04FA375A6),
   helpstring("AxisCommandedInput Class")
  ]
    coclass AxisCommandedInput
    {
      [default] interface IAxisCommandedInput;
    };
  [
   uuid(AA03FCEC-FF08-11D2-AAB2-00C04FA375A6),
   helpstring("AxisCommandedOutput Class")
  ]
    coclass AxisCommandedOutput
    {
```

```
    [default] interface IAxisCommandedOutput;
  };
[
 uuid(AA03FCEE-FF08-11D2-AAB2-00C04FA375A6),
 helpstring("AxisDyn Class")
]
  coclass AxisDyn
  {
    [default] interface IAxisDyn;
  };
[
 uuid(AA03FCF0-FF08-11D2-AAB2-00C04FA375A6),
 helpstring("IAxisErrorAndEnable Class")
]
  coclass AxisErrorAndEnable
  {
    [default] interface IAxisErrorAndEnable;
  };
[
 uuid(AA03FCF3-FF08-11D2-AAB2-00C04FA375A6),
 helpstring("AxisForceServo Class")
]
  coclass AxisForceServo
  {
    [default] interface IAxisForceServo;
  };

[
 uuid(AA03FCF7-FF08-11D2-AAB2-00C04FA375A6),
 helpstring("AxisHoming Class")
]
  coclass AxisHoming
  {
    [default] interface IAxisHoming;
  };
[
 uuid(AA03FCF9-FF08-11D2-AAB2-00C04FA375A6),
 helpstring("AxisJogging Class")
]
  coclass AxisJogging
  {
    [default] interface IAxisJogging;
  };
[
 uuid(AA03FCFB-FF08-11D2-AAB2-00C04FA375A6),
 helpstring("AxisKinematics Class")
]
  coclass AxisKinematics
  {
    [default] interface IAxisKinematics;
  };
[
 uuid(AA03FCFE-FF08-11D2-AAB2-00C04FA375A6),
 helpstring("AxisLimits Class")
]
  coclass AxisLimits
  {
    [default] interface IAxisLimits;
  };
[
 uuid(AA03FD00-FF08-11D2-AAB2-00C04FA375A6),
 helpstring("AxisMaintenance Class")
]
  coclass AxisMaintenance
  {
    [default] interface IAxisMaintenance;
  };
[
 uuid(AA03FD02-FF08-11D2-AAB2-00C04FA375A6),
 helpstring("AxisPositioningServo Class")
]
  coclass AxisPositioningServo
  {
      [default] interface IAxisPositioningServo;
    };
  [
   uuid(AA03FD04-FF08-11D2-AAB2-00C04FA375A6),
   helpstring("AxisRates Class")
  ]
    coclass AxisRates
    {
      [default] interface IAxisRates;
    };
  [
   uuid(AA03FD06-FF08-11D2-AAB2-00C04FA375A6),
   helpstring("AxisSensedState Class")
  ]
    coclass AxisSensedState
    {
      [default] interface IAxisSensedState;
    };
  [
   uuid(AA03FD08-FF08-11D2-AAB2-00C04FA375A6),
   helpstring("AxisSetup Class")
  ]
    coclass AxisSetup
    {
      [default] interface IAxisSetup;
    };
  [
   uuid(AA03FD0A-FF08-11D2-AAB2-00C04FA375A6),
   helpstring("AxisVelocityServo Class")
  ]
    coclass AxisVelocityServo
    {
      [default] interface IAxisVelocityServo;
    };
  [
   uuid(AA03FD0C-FF08-11D2-AAB2-00C04FA375A6),
   helpstring("AxisAbsolutePos Class")
  ]
    coclass AxisAbsolutePos
    {
      [default] interface IAxisAbsolutePos;
    };
  [
   uuid(AA03FD0E-FF08-11D2-AAB2-00C04FA375A6),
   helpstring("AxisIncrementPos Class")
  ]
    coclass AxisIncrementPos
    {
      [default] interface IAxisIncrementPos;
    };
};

```

B.15 CONTROL LAW

```
// ControlLawModule.idl : IDL source for ControlLawModule.dll
//

// This file will be processed by the MIDL tool to
// produce the type library (ControlLawModule.tlb) and marshalling code.

import "oaidl.idl";
import "ocidl.idl";
import "OmacModule.idl";

    [
        object,
        uuid(4B179145-BC3B-11D2-AAAA-00C04FA375A6),

        helpstring("IControlLaw Interface"),
        pointer_default(unique)
    ]
    interface IControlLaw : IOmac
    {
    HRESULT _stdcall getActualOffset([out,retval] double * val);
    HRESULT _stdcall getActualPosition([out,retval] double * val);
    HRESULT _stdcall getCmdOffset([out,retval] double * val);
    HRESULT _stdcall getFollowingError([out,retval] double * val);
    HRESULT _stdcall getOutputCommand([out,retval] double * val);
    HRESULT _stdcall getOutputOffset([out,retval] double * val);
    HRESULT _stdcall getScaleOffset([out,retval] double * val);
    HRESULT _stdcall getSetpoint([out,retval] double * val);
    HRESULT _stdcall getSetpointDot([out,retval] double * val);
    HRESULT _stdcall getSetpointDotDot([out,retval] double * val);
    HRESULT _stdcall getSetpointPrime([out,retval] double * val);

    HRESULT _stdcall setActualOffset([in] double k);
    HRESULT _stdcall setActualPosition([in] double x);
    HRESULT _stdcall setCmdOffset([in]  double off) ;
    HRESULT _stdcall setOutputCommand([in]  double value);
    HRESULT _stdcall setOutputOffset([in]  double k);
    HRESULT _stdcall setScaleOffset([in]  double k) ;
    HRESULT _stdcall setSetpoint([in] double X);
    HRESULT _stdcall setSetpointDot([in]  double Xdot);
    HRESULT _stdcall setSetpointDotDot([in] double Xdotdot);
    HRESULT _stdcall setSetpointPrime([in] double Xprime);

    // This defines an abstract interface  class definition
    HRESULT _stdcall calculateOutputCommand();
        };

    [
        object,

        uuid(4B179148-BC3B-11D2-AAAA-00C04FA375A6),

        helpstring("IPIDControlLaw Interface"),
        pointer_default(unique)
    ]


    interface IPIDControlLaw : IControlLaw
    {

    HRESULT _stdcall  getCycleTime([out,retval] double * val);
    HRESULT _stdcall  getKaf([out,retval] double * val);
    HRESULT _stdcall  getKcf([out,retval] double * val);
    HRESULT _stdcall  getKd([out,retval] double * val);
    HRESULT _stdcall  getKi([out,retval] double * val);
    HRESULT _stdcall  getKp([out,retval] double * val);
    HRESULT _stdcall  getKvf([out,retval] double * val);
    HRESULT _stdcall  getKxprime([out,retval] double * val);
    HRESULT _stdcall  getintegrationLimit([out,retval] double * val);

    HRESULT _stdcall setKaf([in] double k) ;
    HRESULT _stdcall setKcf([in] double k);
    HRESULT _stdcall setKd([in] double k);
    HRESULT _stdcall setKi([in] double k);
    HRESULT _stdcall setKp([in] double k);
    HRESULT _stdcall setKvf([in] double k);
    HRESULT _stdcall setKxprime([in] double k);
    HRESULT _stdcall setIntegrationLimit([in] double integrationLimit);
    HRESULT _stdcall setCycleTime([in] double time);
    HRESULT _stdcall init();
    HRESULT _stdcall reset();
    HRESULT _stdcall calculateOutputCommand();
    HRESULT _stdcall isConfigured([out,retval] BSTR * str);
    HRESULT _stdcall debug();
    HRESULT _stdcall toString([out,retval] BSTR * str);
    HRESULT _stdcall configToString([out,retval] BSTR * str);
    HRESULT _stdcall configure(BSTR filename, BSTR section);
    };

```

```

// Now add ControlLawClassFactory so that multiple factories
// can exist to create PID, or other control laws. Clients
// look up available control law servers under  CATID_ControlLawModule
// category. Then, the client does a CLSID_IOmacClassFactory query interface on one of the
// ControlLaw module servers.

cpp_quote("const CATID CATID_ControlLawModule = {0xE1D6F9F1,0xB1FE,0x11D2,{0xAA,0xA8,0x00,0xC0,0x4F,0xA3,0x75,0xA6}};")

// Example: This CLSID  is specific for one vendor, (i.e., NIST) Control Law Server
cpp_quote("const CLSID CLSID_NISTControlLawServer = { 0x24F48688, 0xE842, 0x11D2, {0xAA, 0xB1, 0x00, 0xC0, 0x4F, 0xA3, 0x75, 0xA6}};")

    [
        object,
        // Replace this uuid with vendor-specific uuid
        uuid(0A70EBAC-06D9-11D3-AAB2-00C04FA375A6),

        helpstring("IControlLawModuleClassFactory Interface"),
        pointer_default(unique)
    ]
    interface IControlLawModuleClassFactory : IOmacModuleClassFactory
    {
    extern const IID IID_IDL_IPIDControlLaw;
    HRESULT _stdcall CreateModule([in] BSTR name, [in] REFIID riid, [out, retval, iid_is(riid)] void ** ppvObj);
//  HRESULT _stdcall CreatePIDObject([in] BSTR name, [out, iid_is(&IID_IDL_IPIDControlLaw)] void ** ppvObj);

    };

[
    uuid(4B179138-BC3B-11D2-AAAA-00C04FA375A6),
    version(1.0),
    helpstring("ControlLawModule 1.0 Type Library")
]
library CONTROLLAWMODULELib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

//  extern const GUID CATID_ControlLawModule;
//  extern const GUID CLSID_NISTControlLawServer;

    [
        uuid(E1D6F9ED-B1FE-11D2-AAA8-00C04FA375A6),
        helpstring("ControlLaw Class")
    ]
    coclass ControlLaw
    {
        [default] interface IControlLaw;
        interface IOmac;
    };
/*  [
        uuid(4B179147-BC3B-11D2-AAAA-00C04FA375A6),
        helpstring("Omac Class")
    ]

    coclass Omac
    {
        [default] interface IOmac;
    };
*/
    [
        uuid(E1D6F9F1-B1FE-11D2-AAA8-00C04FA375A6),
        helpstring("PIDControlLaw Class")
    ]
    coclass PIDControlLaw
    {
        [default] interface IPIDControlLaw;
    };

    [
        uuid(0A70EBAD-06D9-11D3-AAB2-00C04FA375A6),
        helpstring("ControlLawModuleClassFactory Class")
    ]
    coclass ControlLawModuleClassFactory
    {
        //[default] interface IControlLawModuleClassFactory;
        [default] interface IOmacModuleClassFactory;
        interface IClassFactory;
    };
};


```

B.16 HUMAN MACHINE INTERFACE

```
// HMIModule.idl : IDL source for HMI dll

#ifndef __HMIModule__IDL
#define __HMIModule__IDL
import "oaidl.idl";
import "ocidl.idl";
import "OmacModule.idl";

[

    object,
    uuid(134A02A1-E101-11d2-B512-AEC041D2957B),

    helpstring("HMI Interface"),
    pointer_default(unique)
]

interface IHMI : IOmac
{
  // Presentation Methods
  HRESULT _stdcall  presentErrorView();
  HRESULT _stdcall  presentOperationalView();
  HRESULT _stdcall  presentSetupView();
  HRESULT _stdcall  presentMaintenanceView();

  // Events - to alert HMI that something has happened
  HRESULT _stdcall  updateCurrentView();
};


#endif

```

B.17 PROCESS MODEL

```
//
// ProcessModel.idl
//


#ifndef ProcessModel__idl
#define ProcessModel__idl
import "oaidl.idl";
import "ocidl.idl";
import "DataRepresentation.idl";
// Level 1

[

    object,
    uuid(134A02A0-E101-11d2-B512-AEC041D2957B),

    helpstring("Process Model  Interface"),
    pointer_default(unique)
]
interface IProcessModel : IUnknown
{
  HRESULT _stdcall  getUserCoordinateOffsets([out,retval] OacVector ** offsets);
  HRESULT _stdcall  setUserCoordinateOffsets([in] OacVector offsets);

 HRESULT _stdcall   getAxesCoordinateOffsets([out,retval] OacVector ** axoff);     // used by
axes group
  HRESULT _stdcall  setAxesCoordinateOffsets([in] OacVector offsets);   // set by sensor process

 HRESULT _stdcall   getFeedrateOverrideValue([out,retval] Measure ** m); // used by axisgroup
 HRESULT _stdcall   setFeedrateOverrideValue([in] Measure feed);  // used by hmi
 HRESULT _stdcall   getSpindleOverrideValue([out,retval] Measure ** m);   // used by axisgroup
 HRESULT _stdcall   setSpinldeOverrideValue([in] Measure feed);   // used by hmi
};

[
    uuid(134A028A-E101-11d2-B512-AEC041D2957B),
    version(1.0),
    helpstring("Process Model Module 1.0 Type Library")
]
library PROCESS_MODEL_MODULE_Lib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [
        uuid(134A028B-E101-11d2-B512-AEC041D2957B),
        helpstring("Process Model Class")
    ]
    coclass ProcessModel
    {
        [default] interface IProcessModel;
    };
};
#endif

```

B.18 KINEMATICS

```
#ifndef _KINEMATICS__IDL
#define _KINEMATICS__IDL
import "DataRepresentation.idl";
// General Agreement: 18-Jun-1997 Sushil Birla, Steve Sorensen


[
    object,
    uuid(134A02A5-E101-11d2-B512-AEC041D2957B),


    helpstring("Kinematics Interface"),
    pointer_default(unique)
]
interface IKinStructure :IUnknown
{
  HRESULT _stdcall getPlacementFrame([out,retval]  CoordinateFrame ** cf);
  HRESULT _stdcall setPlacementFrame([in] CoordinateFrame value);

  HRESULT _stdcall getBaseframe([out,retval] CoordinateFrame ** cf);
  HRESULT _stdcall setBaseframe([in] CoordinateFrame value);
};

[
    object,
    uuid(134A02A6-E101-11d2-B512-AEC041D2957B),


    helpstring("Kinematic Connection Interface"),
    pointer_default(unique)
]
interface IKinConnection :IUnknown
{
  HRESULT _stdcall  getFrom([out,retval] IKinStructure ** value);
  HRESULT _stdcall  setFrom([in] IKinStructure * value);

  HRESULT _stdcall  getTo([out,retval] IKinStructure **value);
  HRESULT _stdcall  setTo([in] IKinStructure * value);

  HRESULT _stdcall  getPlacement([out,retval] CoordinateFrame ** frame);
  HRESULT _stdcall  setPlacement([in] CoordinateFrame value);
};


[
    object,
    uuid(6735BEA5-EDA7-11d2-AAB1-00C04FA375A6),

    helpstring("Kinematic Connections Interface"),
    pointer_default(unique)
]
interface IEnumKinConnections : IUnknown
{

    typedef [unique] IKinConnection *LPENUMCONNECTION;

    [local]
    HRESULT Next(
        [in] ULONG celt,
        [out] IKinConnection **rgelt,
        [out] ULONG *pceltFetched);

    [call_as(Next)]
    HRESULT RemoteNext(
        [in] ULONG celt,
        [out, size_is(celt), length_is(*pceltFetched)]
        IKinConnection **rgelt,
        [out] ULONG *pceltFetched);

    HRESULT Skip(
        [in] ULONG celt);

    HRESULT Reset();

    HRESULT Clone(
        [out] IKinConnection **ppenum);
};

interface IEnumKinMechanisms;

[
    object,
    uuid(134A02A7-E101-11d2-B512-AEC041D2957B),


    helpstring("KinMechanism Interface"),
    pointer_default(unique)
]
interface IKinMechanism :IUnknown
{
  HRESULT _stdcall  forwardKinematicTransform([in] IEnumKinConnections * cn);

  HRESULT _stdcall  inverseKinematicTransform([in] CoordinateFrame cf,
        DWORD size_vector,
        [out,retval,size_is(,size_vector)] double ** vector);

  HRESULT _stdcall  getConnections([out,retval] IEnumKinConnections ** c);
  HRESULT _stdcall  setConnections([in] IEnumKinConnections * value);

  HRESULT _stdcall  getKinmechanisms([out,retval] IEnumKinMechanisms ** mechs);
  HRESULT _stdcall  setKinmechanisms([in] IEnumKinMechanisms * value);
};
```

```


// FIXME: A template  would map into IDL sequence
//typedef RWTPtrSlist<KinMechanism> KinMechanisms;
// FIXME: add graph/tree traversal functions


// Notes:
// 1. For various specilizations of inverseKinematicTransform()
// Specialize KinMechanism and extend as needed.
[
    object,
    uuid(949F889D-EDA8-11d2-AAB1-00C04FA375A6),

    helpstring("Enum Kinematic Mechanisms Interface"),
    pointer_default(unique)
]
interface IEnumKinMechanisms : IUnknown
{

    typedef [unique] IKinMechanism *LPENUMKINMECHANISM;

    [local]
    HRESULT Next(
        [in] ULONG celt,
        [out] IKinMechanism **rgelt,
        [out] ULONG *pceltFetched);

    [call_as(Next)]
    HRESULT RemoteNext(
        [in] ULONG celt,
        [out, size_is(celt), length_is(*pceltFetched)]
        IKinMechanism **rgelt,
        [out] ULONG *pceltFetched);

    HRESULT Skip(
        [in] ULONG celt);

    HRESULT Reset();

    HRESULT Clone(
        [out] IKinMechanism **ppenum);
};
[
    uuid(134A02A8-E101-11d2-B512-AEC041D2957B),
    version(1.0),
    helpstring("Kinematics Module 1.0 Type Library")
]
library KINEMATICS_MODULE_Lib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [
        uuid(134A02A9-E101-11d2-B512-AEC041D2957B),
        helpstring("Kinematics Class")
    ]
    coclass KinStructure
    {
        [default] interface IKinStructure;
    };

        [
        uuid(134A02AA-E101-11d2-B512-AEC041D2957B),
        helpstring("Connection Class")
    ]
    coclass EnumKinConnection
    {
        [default] interface IEnumKinConnections;
    };

    [
        uuid(134A02AB-E101-11d2-B512-AEC041D2957B),
        helpstring("KinMechanism Class")
    ]
    coclass KinMechanism
    {
        [default] interface IKinMechanism;
    };

    [
        uuid(EBD7EEBF-EDA9-11d2-AAB1-00C04FA375A6),
        helpstring("Kin Mechanisms Collection  Class")
    ]
    coclass EnumKinMechanisms
    {
        [default] interface IEnumKinMechanisms;
    };

};


#endif


```

B.19 SCHEDULING UPDATER

```
import "oaidl.idl";
import "ocidl.idl";


[
    object,
    uuid(B64988A7-EDC3-11d2-AAB1-00C04FA375A6),

    helpstring("TaskCoordinator Interface"),
    pointer_default(unique)
]
interface IUpdatable : IUnknown
{
  HRESULT _stdcall getPeriod([out,retval] double ** value);
  HRESULT _stdcall setPeriod([in] double aPeriod);
  HRESULT _stdcall update();
};

[
    object,
    uuid(E05FAB5D-EDC3-11d2-AAB1-00C04FA375A6),

    helpstring("TaskCoordinator Interface"),
    pointer_default(unique)
]
interface IAsynchUpdater : IUnknown
{
  HRESULT _stdcall  registerUpdatable([in] IUpdatable * upd);
  HRESULT _stdcall  update();
};

[
    object,
    uuid(134A0280-E101-11d2-B512-AEC041D2957B),

    helpstring("TaskCoordinator Interface"),
    pointer_default(unique)
]
interface IPeriodicUpdater : IAsynchUpdater
{
   HRESULT _stdcall  getTimingInterval([out, retval] double ** value);
// /*no virtual*/   void update();
};


```