ADVANCED Motion Controls constantly strives to improve all of its products. We review the information in this document regularly and we welcome any suggestions for improvement. We reserve the right to modify equipment and documentation without prior notice.

For the most recent software, the latest revisions of this manual, and copies of compliance and declarations of conformity, visit the company’s website at www.a-m-c.com. Otherwise, contact the company directly at:

ADVANCED Motion Controls • 3805 Calle Tecate Camarillo, CA • 93012-5068 USA

Agency Compliances

The company holds original documents for the following:

- UL 508c, file number E140173
  - EN61000-6-2:2001
  - EN61000-6-4:2001
  - EN61000-3-2:2000
- Electrical Safety, Low Voltage Directive - 2006/95/EC
  - EN 60 204-1 (IEC 60 204-1)
- Reduction of Hazardous Substances (RoHS), 2002/95/EC

Trademarks

ADVANCED Motion Controls™, the combined isosceles trapezoid/right triangle logo, DIGIFLEX®, DIGIFLEX® Performance™ and DriveWare™ are either registered trademarks or trademarks of ADVANCED Motion Controls in the United States and/or other countries. All other trademarks are the property of their respective owners.

Related Documentation

- DriveWare Software Manual, available for download at www.a-m-c.com
- Product manual or datasheet specific for your drive, available for download at www.a-m-c.com.
Attention Symbols

The following symbols are used throughout this document to draw attention to important operating information, special instructions, and cautionary warnings. The section below outlines the overall directive of each symbol and what type of information the accompanying text is relaying.

Note - Pertinent information that clarifies a process, operation, or ease-of-use preparations regarding the product.

Notice - Required instruction necessary to ensure successful completion of a task or procedure.

Caution - Instructs and directs you to avoid damaging equipment.

Warning - Instructs and directs you to avoid harming yourself.

Danger - Presents information you must heed to avoid serious injury or death.

Revision History

<table>
<thead>
<tr>
<th>Document ID</th>
<th>Revision</th>
<th>Date</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNDGDLRD-11</td>
<td>1.0</td>
<td>1/2018</td>
<td>Revised to reflect release 1.0 of DriveLibrary</td>
</tr>
<tr>
<td>MNDGDLRD-12</td>
<td>1.1.0</td>
<td>8/2018</td>
<td>- Added Chapter 5 - Axis Setup Functions</td>
</tr>
</tbody>
</table>

© 2018 ADVANCED Motion Controls. All rights reserved.
## Contents

### 1 Introduction

- 1.1 General Safety Overview ........................................... 2
- 1.2 Overview of the Defined Functions ............................... 4

### 2 Installation and Getting Started

- 2.1 Pre-Requisites .......................................................... 5
- 2.2 Supported Compilers .................................................. 5
- 2.3 Installing ............................................................... 5
- 2.4 Configuration .......................................................... 6
  - 2.4.1 Compiler ............................................................ 6
  - 2.4.2 Linker ............................................................... 7
- 2.5 Using DriveLibrary .................................................... 9
  - 2.5.1 Library Naming Conventions ................................. 9
- 2.6 Examples ............................................................... 10
  - 2.6.1 Hello Drive.cpp ................................................ 10
  - 2.6.2 Status.cpp ....................................................... 10
  - 2.6.3 Io Test.cpp ....................................................... 10
  - 2.6.4 Initialize.cpp .................................................. 10
  - 2.6.5 Position Moves.cpp .......................................... 10
  - 2.6.6 Velocity Moves.cpp .......................................... 10

### 3 General Model

- 3.1 Setting Up the Axis Network ...................................... 11
- 3.2 The Axis State Diagram ............................................ 11
6.1.5 Move Absolute .................................................. 26
6.1.6 Move Relative .................................................. 28
6.1.7 Move Velocity .................................................. 30
6.1.8 Halt ............................................................... 32
6.1.9 Stop .............................................................. 33
6.1.10 Dwell ............................................................ 34
6.1.11 Set Position .................................................... 35
6.1.12 Set Override ................................................... 35
6.2 Error Handling Functions ......................................... 36
   6.2.1 Select Error Stop ............................................. 36
   6.2.2 Activate Errors ............................................... 36
   6.2.3 Configure Error Stop ........................................ 36
   6.2.4 Error .......................................................... 37
   6.2.5 Get Stopping Errors ......................................... 37
   6.2.6 Get Logic Errors ............................................. 37
   6.2.7 Reset .......................................................... 37
6.3 Status Functions .................................................. 38
   6.3.1 Read State .................................................... 38
   6.3.2 Read Events .................................................. 38
   6.3.3 Clear Event .................................................. 39
   6.3.4 In Velocity ................................................... 39
   6.3.5 Read Parameter .............................................. 39
   6.3.6 Read Boolean Parameter ................................... 40
   6.3.7 Write Parameter .............................................. 40
   6.3.8 Write Boolean Parameter .................................. 40
   6.3.9 Read Digital Input .......................................... 41
   6.3.10 Read Digital Output ....................................... 41
   6.3.11 Write Digital Output ..................................... 41
   6.3.12 Read Actual Position ..................................... 42
   6.3.13 Read Actual Velocity ..................................... 42
   6.3.14 Read Actual Current ...................................... 42
<table>
<thead>
<tr>
<th>Appendix</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Appendix A - System Config Files</td>
<td>43</td>
</tr>
<tr>
<td>B</td>
<td>Appendix B - Error Identifiers</td>
<td>45</td>
</tr>
<tr>
<td>C</td>
<td>Appendix C - Parameter Enumerations</td>
<td>47</td>
</tr>
<tr>
<td>D</td>
<td>Appendix D - State Enumeration</td>
<td>48</td>
</tr>
<tr>
<td>E</td>
<td>Appendix E - Drive Setup Methods</td>
<td>49</td>
</tr>
<tr>
<td>F</td>
<td>Appendix E - Event Enumerizations</td>
<td>52</td>
</tr>
<tr>
<td>G</td>
<td>Appendix F - Error Handling</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>G.1 Scenarios</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>G.1.1 Firmware</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>G.1.2 Internal Errors</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>G.1.3 Fault Handling</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>G.2 Strategy</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>G.2.1 Error Reporting</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>Control Errors</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>Logic Errors</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>G.2.2 Drive Initialization</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>G.2.3 Error Stop Configuration</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>Condition Selection</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>Stop Types</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>Stop Parameters</td>
<td>58</td>
</tr>
</tbody>
</table>
Trigger Logic ....................................................... 59
G.2.4 ERROR STOP Recovery ................................. 59
G.2.5 Error Entry and Error Report ......................... 59
ADVANCED Motion Controls offers DriveLibrary as a high-level interface that delivers the full advantages of high-speed networking and distribution of control for developers of motion control solutions. Expertise in drive control is not a requirement for users developing DriveLibrary applications. Rather, the developer thinks in terms of the dynamic behavior of the system to be controlled: its position, velocity and acceleration.

The DriveLibrary API defines a collection of control functions that have been time-tested and proven to support the majority of motion control applications. The power of the ADVANCED Motion Controls’ approach is that complex motions can be broken down into self-contained and reusable modules using the techniques of object-oriented programming. Time-critical control and coordination are handled by ADVANCED Motion Controls digital drives, allowing the developer to focus on the goals of the operation. DriveLibrary will constrain the function sequence to prevent invalid sequences. This release of DriveLibrary focuses on simple single-axis moves.

DriveLibrary applications are written and developed in C++. Users should be familiar with the C++ language in order to properly develop motion control applications with DriveLibrary.

This chapter discusses characteristics of using DriveLibrary with ADVANCED Motion Controls digital drives to raise user awareness of potential risks and hazards. The severity of consequences ranges from frustration of performance, through damage to equipment, injury or death. These consequences, of course, can be avoided by good design and proper installation into your mechanism.
1.1 General Safety Overview

In order to develop a DriveLibrary application, you must have a thorough knowledge and understanding of basic electronics, computers and mechanics as well as safety precautions and practices required when dealing with the possibility of high voltages or heavy, strong equipment.

Observe your facility’s lock-out/tag-out procedures so that work can proceed without residual power stored in the system or unexpected movements by the machine.

---

**Notice**

You must install and operate motion control equipment so that you meet all applicable safety requirements. Ensure that you identify the relevant standards and comply with them. Failure to do so may result in damage to equipment and personal injury.

Read this entire manual prior to attempting to install or operate the drive. Become familiar with practices and procedures that allow you to operate these drives safely and effectively. You are responsible for determining the suitability of this product for the intended application. The manufacturer is neither responsible nor liable for indirect or consequential damages resulting from the inappropriate use of this product.

---

**Notice**

High-performance motion control equipment can move rapidly with very high forces. Unexpected motion may occur especially during product commissioning. Keep clear of any operational machinery and never touch them while they are working.

---

**Warning**

Keep clear of all exposed power terminals (motor, DC Bus, shunt, DC power, transformer) when power is applied to the equipment. Follow these safety guidelines:

- Always turn off the main power and allow sufficient time for complete discharge before making any connections to the drive.
- Do not rotate the motor shaft without power. The motor acts as a generator and will charge up the power supply capacitors through the drive. Excessive speeds may cause over-voltage breakdown in the power output stage. Note that a drive having an internal power converter that operates from the high voltage supply will become operative.
- Do not short the motor leads at high motor speeds. When the motor is shorted, its own generated voltage may produce a current flow as high as 10 times the drive current. The short itself may not damage the drive but may damage the motor. If the connection arcs or opens while the motor is spinning rapidly, this high voltage pulse flows back into the drive (due to stored energy in the motor inductance) and may damage the drive.
- Do not make any connections to any internal circuitry. Only connections to designated connectors are allowed.
- Do not make any connections to the drive while power is applied.
Do not reverse the power supply leads!
Severe damage will result!

Caution

Use sufficient capacitance!
Pulse Width Modulation (PWM) drives require a capacitor on the high voltage supply to store energy during the PWM switching process. Insufficient power supply capacitance causes problems particularly with high inductance motors. During braking much of the stored mechanical energy is fed back into the power supply and charges its output capacitor to a higher voltage. If the charge reaches the drive's over-voltage shutdown point, output current and braking will cease. At that time energy stored in the motor inductance continues to flow through diodes in the drive to further charge the power supply capacitance. The voltage rise depends upon the power supply capacitance, motor speed, and inductance.

A 2\mu H motor at 20 amps can charge a 2000 \mu F capacitor an additional 30 VDC. An appropriate capacitance is typically 2000 \mu F/A maximum output current for a 50 VDC supply.

Caution

Make sure minimum inductance requirements are met!
Pulse Width Modulation (PWM) servo drives deliver a pulsed output that requires a minimum amount of load inductance to ensure that the DC motor current is properly filtered. The minimum inductance values for different drive types are shown in the individual data sheet specifications. If the drive is operated below its maximum rated voltage, the minimum load inductance requirement may be reduced. Most servo-motors have enough winding inductance. Some types of motors (e.g. “basket-wound”, “pancake”, etc.) do not have a conventional iron core rotor, so the winding inductance is usually less than 50 \mu H.

If the motor inductance value is less than the minimum required for the selected drive, use an external filter card.
1.2 Overview of the Defined Functions

The following table gives an overview of the defined functions, divided into administrative (not driving motion) and motion related categories. For single axis functions include the header "Axis.h".

**TABLE 1.1 Overview of the Defined Functions**

<table>
<thead>
<tr>
<th>Motion Control Single Axis</th>
<th>Error Handling Single Axis</th>
<th>Status Single Axis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move Absolute*</td>
<td>Select Error Stop</td>
<td>Read State</td>
</tr>
<tr>
<td>Move Relative*</td>
<td>Activate Errors</td>
<td>Read Events</td>
</tr>
<tr>
<td>Move Velocity*</td>
<td>Configure Error Stop</td>
<td>Clear Event</td>
</tr>
<tr>
<td>Home*</td>
<td>Error</td>
<td>In Velocity</td>
</tr>
<tr>
<td>Stop*</td>
<td>Get Stopping Errors</td>
<td>Read Parameter</td>
</tr>
<tr>
<td>Halt*</td>
<td>Get Logic Errors</td>
<td>Read Boolean Parameter</td>
</tr>
<tr>
<td>Dwell</td>
<td>Reset*</td>
<td>Write Parameter</td>
</tr>
<tr>
<td>Power*</td>
<td></td>
<td>Write Boolean Parameter</td>
</tr>
<tr>
<td>Busy</td>
<td></td>
<td>Read Digital Input</td>
</tr>
<tr>
<td>Done</td>
<td></td>
<td>Read Digital Output</td>
</tr>
<tr>
<td>Set Position</td>
<td></td>
<td>Write Digital Output</td>
</tr>
<tr>
<td>Set Override</td>
<td></td>
<td>Read Actual Position</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read Actual Velocity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Read Actual Current</td>
</tr>
</tbody>
</table>

*Functions marked with an asterisk cause a state change when called
2.1 Pre-Requisites

Prior to installing DriveLibrary some pre-requisites on the target system have to be met.

- DriveLibrary runs on Windows 7, 8, and 10 operating systems.
- A compatible version of DriveWare must be installed. For a windows installation the installer will verify the compatibility automatically.
- The communication hardware must be installed. For instance, on a CANopen system, the interface card must be installed and the device drivers must be loaded.
- The compiler installed must be supported.

2.2 Supported Compilers

DriveLibrary is distributed as a set of pre-compiled library binaries (lib files). Therefore in order to use DriveLibrary with a particular compiler you must have a set of lib files that were created for the compiler.

**TABLE 2.1**

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Compiler Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Visual C++</td>
<td>14.0 aka; Visual Studio 2015 or greater with Service Release 1</td>
</tr>
</tbody>
</table>

If your compiler is not in this list, it may be possible to add it; please contact AMC Sales.

2.3 Installing

After the pre-requisites have been satisfied it’s time to run setup.exe. The install program will guide you step by step through the installation. The installation path is:
After installation you will find a directory structure similar to the below descriptions:

C:\Program Files (x86)\Advanced Motion Controls\DriveLibrary<Ver>\n
\Config - Example input files describing the machine configuration (see Appendix A)
\Demos - Sample programs
\Docs
\Include - Header files needed to invoke DriveLibrary
\Library - Library files needed for linking DriveLibrary solutions, and dlls that must be deployed with the application
/Debug - for developers
/Release - for customer installs

On a Windows installation you will find a folder in the start menu containing the start menu and links to the documentation. You can also find a link to the documentation on your desktop.

2.4 Configuration

Now it’s time to configure your compiler to use DriveLibrary. Since Microsoft’s compiler is by far the dominant one we’ll use it for this example. If you are using a different compiler the basic requirements are the same but the procedure will be different. Please refer to your compiler’s documentation for the exact procedures.

2.4.1 Compiler

After creating your target project the first thing that we need to do is tell the compiler where it can find the header files (include files). This screen shot shows a typical configuration of the compiler tab. This will need to be done for every project where you want to use DriveLibrary.
As you can see on the “Additional Include Directories” text box, the path to the DriveLibrary Include directory has been added. No other settings needed to be changed here.

Notice

2.4.2 Linker

Now we need to tell the linker where it can find the precompiled binaries (lib files). This screen shot shows a typical configuration of the linker tab. This will need to be done for every project where you want to use DriveLibrary.

Notice

The current version of DriveLibrary is compiled against Boost 1.60. The required modules are included in the library delivery. Developers also relying upon boost must use version 1.60 for their own development.
As you can see on the “Additional Library Directories” cell the path to the 
"../DriveLibrary/Library" directory has been added. No other settings need to be changed here.

The path given in the above example is representative only! You need to use the path to where DriveLibrary was installed on your system.
2.5 Using DriveLibrary

Now that the compiler knows where to find DriveLibrary, using it is as simple as an include statement.

```c
#include <Axis.h>
```

If your compiler supports "autolinking", as many windows compilers do, then the job is done. Special code in the SingleAxis header file detects your compiler options and uses that information to encode the name of the correct library into your object files; the linker selects the library with that name from the directories you've told it to search. If the compiler does not support "autolinking" then you need to move on to the next section. If you're not sure, try it this way first, it will either work or it won't.

2.5.1 Library Naming Conventions

Some compilers will require you to explicitly specify the name of the precompiled binary (lib file) to link to. You will need to refer to your compiler's documentation to determine the proper method of doing this. The correct lib file for your application can be found using the following naming convention.

**FIGURE 2.3 Library Naming**

DriveLibrary does not support static run-time. This is due to its dependency on the boost libraries.
2.6 Examples

While there are a number of sample programs, the examples in this section are tailored to introduce a first time user to DriveLibrary. It is recommended for first time DriveLibrary users to read and understand the given examples. Each of the following examples focuses on a specific topic. NOTE: If you build the examples, don’t forget your configuration steps in section two.

If you choose to actually build and compile the examples, don’t forget the configuration steps given in “Configuration” on page 6.

2.6.1 Hello Drive.cpp
Covers the basics for connecting to a drive and passing a message.

2.6.2 Status.cpp
Covers reading the drive’s status and a number of the different ways you can work with the status command.

The ReadState and ReadEvents commands sound alike but have very different meanings. See “Read State” and “Read Events” on page 38 to make sure you understand the difference before deciding which is best for you.

2.6.3 Io Test.cpp
Covers reading and writing to the digital I/O ports.

2.6.4 Initialize.cpp
Covers power-up, homing, and reading the drive’s position.

2.6.5 Position Moves.cpp
Covers MoveAbsolute, MoveRelative, and buffering. Examine buffering versus aborting when queuing up multiple move sequences.

2.6.6 Velocity Moves.cpp
Covers MoveVelocity and stopping a motor.
3.1 Setting Up the Axis Network

DriveLibrary leverages the distributed processing capability of ADVANCED Motion Controls' drives to maximize throughput on each of its supported networks. Planning capacity on your network is essential to ensure satisfactory performance.

Each network has a type, connects to a host machine on a physical port, and operates at a certain baud rate. Each axis on the network listens for packets designated to its address. Data transmission from the host can be managed by adjusting the time steps for:

- PVT control point calculations
- Status updates
- Drive control point processing updates

In addition, DriveLibrary requires that each axis have a unique name assigned to it.

"Appendix A- System Config Files" on page 43 describes how to create files that define the communication parameters for the system. These files must exist before DriveLibrary is used. The header file "Control/SystemConfig.h" defines the functions used to load the system configuration.

3.2 The Axis State Diagram

To ensure safe and reliable execution of motion programs, DriveLibrary imposes rules on the order of execution of the defined functions. Figure 3.1 on page 13 defines the behavior of the axis at a high level. The state model allows the developer to combine functions to build a more complicated profile or to handle exceptions within a program. The basic rule is that motion functions always act on the axis' state diagram.

The axis is always in one of the defined states. Any motion function that causes a transition changes the state of the axis and as a consequence modifies the way the current motion is
computed. A change of state is reflected immediately when calling the corresponding motion function.

Any robust motion control application cannot simply fail when an error/fault condition occurs. The state diagram defines startup and recovery sequences that must be observed. They also define simple high-level abstractions that can be used to track the state of the system being controlled. For more on error/fault handling, see “Appendix F - Error Handling” on page 55.

The state **DISABLED** describes the initial state of the axis. In this state the movement of the axis is not influenced by the motion functions, though the axis feedback is operational. If the Power function is called with Enable = true while being in **DISABLED**, this either leads to **STAND STILL** if there is no error inside the axis, or to **ERROR_STOP** if an error exists.

Calling the Power function with Enable = false in any state, the axis goes to the state **DISABLED**, either directly or via any other state. If a motion generating function controls an axis and the Power function with Enable = false is called, the motion is aborted (CommandAborted).

The intention of the **ERROR_STOP** state is that the axis goes to a stop, if possible. With the exception of the Power function with Enable = false, no further functions will be accepted until a reset has been done from the **ERROR_STOP** state.

Calling the Home function in a state other than **STAND STILL** will go to **ERROR_STOP**.

Busy is set after a function is called and cleared when the control operation completes (return value of Done function is true) or when an error occurs. The outputs of the functions Busy, Done, and Error are mutually exclusive: only one of them can be true at a time. One of these functions has to be true.

Functions that are not listed in the State Diagram do not affect the state of the axis; meaning that whenever they are called the state does not change.

Calling the function Stop in state **STAND STILL** changes the state to **STOPPING** and back to **STAND STILL**.
**FIGURE 3.1 Axis States and Transition Methods**

Note 1: In the states ErrorStop or Stopping, all functions can be called (although they will simply return) except Reset and Error - they will generate the transition to StandStill or ErrorStop respectively.

Note 2: Power.Enable = true and there is an error in the Axis.

Note 3: Power.Enable = true and there is no error in the Axis.

Note 4: Stop.Done
3.3 Function interface

3.3.1 AMC Namespace

DriveLibrary is written in C++ using platform-independent libraries (specifically, the boost libraries www.boost.org). This means that developers of motion control applications can create more specific axis types on top of the AMC classes. These axes might define methods that use sequences of primitive moves to perform more complex operations. By encapsulating these procedures in a C++ class, that sequence can be reused in many motion control applications.

In order to prevent possible collisions between class and type names defined by the application developer, AMC has written all of its externally visible types into AMC namespaces. This means that all types must be prefaced, for example, by:

'amc::motion::control' or another AMC namespace, or a proper C++ using declaration must be visible in the referencing scope.

The following is a list of AMC namespaces, along with a corresponding alias that is used throughout the documentation to refer to the respective namespace:

1. namespace ctrlNS = amc::motion::control;
2. namespace hostNS = amc::motion::host;
3. namespace statNS = amc::motion::drive::model::status;
4. namespace singleNS = amc::motion::drive::control::single;

As an example, the following piece of code in the manual:

```cpp
ctrlNS::<Type>
hostNS::SystemConfig
ctrlNS::ErrorReport  rpt;
void errorAccept(ctrlNS::Axis &p_rAxis)
```

...would be written as follows in the actual C++ program:

```cpp
amc::motion::control::<Type>
amc::motion::host::SystemConfig
amc::motion::control::ErrorReport  rpt;
void errorAccept(amc::motion::control::Axis &p_rAxis)
```

3.3.2 Error Handling

All functions except status functions (Busy, Error, ReadState, and InVelocity) return the boolean value true to indicate that the request was interpreted successfully or false if a logic error occurred. If the function has failed, the programmer may read the error code for the axis using the GetLogicError function. It is recommended that the programmer always check the boolean return value.

Note that “success” may result when the axis is being placed in the ERROR_STOP condition, so the Error condition should also be checked as appropriate.
Techniques for dealing with errors vary widely across applications. Please refer to the application examples for general techniques that may be used to deal with errors. The application developer should recognize that DriveLibrary is linked code. Certain types of system-level exceptions, propagating from the memory manager or communications drivers, must be caught by the application.

Three types of error functions are provided to support analysis of errors detected by DriveLibrary.

- Error - returns a boolean value that indicates that the axis is in `ERROR_STOP` state.
- GetStoppingErrors - describes errors that force the axis into `ERROR_STOP`.
- GetLogicErrors - describes errors generated when the Library cannot interpret a request.

Errors typically fall into one of the following three categories:

- Function Error - logic errors caused by calling a function with parameters out of range, or parameters that result in a violation of application limits.
- Communication Error - caused by a failure to communicate with the drive, placing the axis into `DISABLED` state.
- Drive Errors - hardware errors such as over-voltage, under-voltage, over-temperature, etc.

See “Appendix F - Error Handling” on page 55 for more information on Error Handling.

### 3.3.3 User Units

DriveLibrary does not make any assumptions regarding the system to be controlled. By default, it assumes that the position unit at the API layer will match the drive units, which are encoder counts. Encoder counts are not the natural units for motion planning. However, DriveLibrary is not a user interface development environment. This means that there is no need to match time units familiar to workers. In fact, for motion planning, velocities and accelerations are often derived as time derivatives of the position. In those calculations, the natural time unit is always second. In the context of its intended use, DriveLibrary need only support conversion of distance units to drive counts.

\[
\text{counts} = K_d \times \text{distance}
\]

Since homing allows the drive to establish an absolute location, we do not need an additional offset to convert from counts to position.

The value of \(K_d\) can be provided through `ctrlNS::Axis::SetDistanceScale` during axis configuration. The conversion will be applied to all values (derived from distance) that are passed across the interface in either direction. Obviously, this applies to distance, position, velocity, acceleration, deceleration, and jerk.
4.1 Loading the Axis Network

This function loads the system configuration.

```cpp
bool hostNS::System::LoadSystem (std::string const & configFile);
```

**TABLE 4.1**

<table>
<thead>
<tr>
<th>PARAMETERS</th>
<th>Full path to the system (&quot;asf&quot;) file.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConfigFile</td>
<td>std::string const &amp;</td>
</tr>
</tbody>
</table>

**OUTPUT**

- bool
  - true if the file was interpreted successfully

**NOTES:**

Errors generated during processing of the system file will appear in “SystemErrors.log” in the containing directory.

4.2 Unloading the Axis Network

This function unloads the system configuration. Changing the system configuration is a means for ensuring that axes are available only when needed.

```cpp
void hostNS::System::UnLoadSystem ();
```

4.3 Axis Constructor

This constructor creates the “axis” object. Its prototype is as follows:

```cpp
ctrlNS::Axis (std::string const & axisName);
```
Axis Destructor / Construction and Destruction

### Table 4.2

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AxisName</td>
<td>std::string const &amp; The name of the axis (as defined in the system config file; see “Appendix A- System Config Files”) to prepare for use by DriveLibrary.</td>
</tr>
</tbody>
</table>

Notes:
- When the constructor returns, use IsInitialized to determine whether it is ready for use.
- Should initialization fail, error codes and messages can be obtained through GetStoppingErrors and GetLogicErrors.
- Throws an std::exception on error. See Table 4.3 below.

There are at least two methods for creating an axis using the Axis constructor. Each of the following methods creates a new object called MyAxis:

- **Method One:**
  ```cpp
  ctrlNS::Axis myAxis( "X Axis" );
  ```

- **Method Two:**
  ```cpp
  pMyAxis = new ctrlNS::Axis( "X Axis" );
  ```

In addition, DriveLibrary has been written so that Axis can be sub-classed without recompilation:

- **Method Three:**
  ```cpp
  class MySpecialAxis : public ctrlNS::Axis
  {
  MySpecialAxis(std::string const & axisName)
   : ctrlNS (axisName){};
  ~MySpecialAxis();
  // C++ allows the addition of new member functions and data.
  
  MySpecialAxis MyAxis( "X Axis" );
  ```

### 4.4 Axis Destructor

This destructor deletes the axis. As it is virtual, it will be automatically called when any sub-class is freed.

```cpp
virtual ~Axis ();
```

### 4.5 Initialize

This method establishes the connection with the AMC drive and prepares to monitor and control the axis. If initialization fails, GetLogicErrors will report the cause. The error ID will be:

```cpp
ctrlNS::Error::CANNOT_CONNECT_TO_DRIVE;
```

The error description will contain one of the messages in the following table.
### TABLE 4.3

<table>
<thead>
<tr>
<th>MESSAGE</th>
<th>CAUSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Could not communicate with the drive</td>
<td>DriveLibrary is unable to communicate with the drive. Check the communication settings within DriveWare.</td>
</tr>
<tr>
<td>Cannot assume axis control when the bridge is enabled</td>
<td>The bridge must be disabled before assuming axis control. Use the Command Toolbar in DriveWare to disable the bridge.</td>
</tr>
<tr>
<td>Axis firmware does not support DriveLibrary operations.</td>
<td>The drive firmware is an older version that does not support DriveLibrary. Download the latest firmware and load onto the drive.</td>
</tr>
<tr>
<td>The DriveWare project file is not configured for DriveLibrary operations.</td>
<td>Go to the Command Source block on the Main Block Diagram and select the Configure for DriveLibrary checkbox. This will automatically configure the project file for use with DriveLibrary.</td>
</tr>
<tr>
<td>Error downloading the DriveWare project file</td>
<td>An error occurred while trying to download the project file. Check the drive’s power supply and the communication settings within DriveWare.</td>
</tr>
<tr>
<td>Unable to connect to the drive</td>
<td>Check the drive’s power supply connection and the communication settings within DriveWare.</td>
</tr>
<tr>
<td>Drive is under the control of another application</td>
<td>Another application, such as DriveWare, is connected to the drive with Read/Write access. To allow DriveLibrary to access the drive, connect Read-Only or close the application.</td>
</tr>
</tbody>
</table>

### 4.6 Set Distance Scale

Establishes the value used to convert from distance to drive counts. SetDistanceScale should be set after drive initialization, but before any type of moves are sent to the drive. Once set, SetDistanceScale should not be changed. The default value is 1, which means that the default library units are counts. See “User Units” on page 15 for discussion.

```cpp
bool SetDistanceScale ( FLOAT64 scale );
```

### TABLE 4.4

<table>
<thead>
<tr>
<th>PARAMETERS</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale</td>
<td>FLOAT64</td>
</tr>
<tr>
<td>Scaling factor</td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td>true if value is acceptable (must be &gt;= 1)</td>
</tr>
</tbody>
</table>
The Axis Setup class provides access to DriveWare configuration parameters that may be useful to modify as machine operation changes. Appropriate applications may include machines with variable loads, or gross and fine positioning modes.

ADVANCED Motion Controls’ command set includes features that support two distinct operational settings. For DriveLibrary applications, the most common use will be the second gain set, but the user may also define a second control configuration. The Axis Setup API allows switching between these predefined settings.

For machines with more complex demands, the Axis Setup class allows direct manipulation of the control loop gains and limits.

Because DriveLibrary allows multiple instances of a named axis, creation of an Axis Setup instance is shielded to ensure that the parameter values are consistent throughout the application.
### 5.1 Axis Setup

#### 5.1.1 Prepare for Axis Setup

Obtain the setup object for the named axis.

```cpp
ctrlNS::AxisSetup * ctrlNS::AxisSetup::PrepareSetup(
    ctrlNS::Axis *pAxis);
```

<table>
<thead>
<tr>
<th>TABLE 5.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARAMETERS</td>
</tr>
<tr>
<td>pAxis</td>
</tr>
<tr>
<td>OUTPUT</td>
</tr>
<tr>
<td>ctrlNS::AxisSetup*</td>
</tr>
</tbody>
</table>

**Notes:**
The parameter must be fully initialized (see "Initialize" on page 17).

#### 5.1.2 Release Setup Resources

End use of the setup object for the named axis.

```cpp
void ctrlNS::AxisSetup::RemoveSetup(
    ctrlNS::Axis *pAxis);
```

<table>
<thead>
<tr>
<th>TABLE 5.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARAMETERS</td>
</tr>
<tr>
<td>pAxis</td>
</tr>
<tr>
<td>Notes:</td>
</tr>
<tr>
<td>If the AxisSetup instance is held in the context of two instances of the named axis, it will become invalid upon the first call to RemoveSetup.</td>
</tr>
</tbody>
</table>

#### 5.1.3 Activate a Gain Set

Activates the selected gain set, changing the loop gains for the velocity and position loops.

```cpp
bool activateGainSet(
    unsigned int gainSet);
```

<table>
<thead>
<tr>
<th>TABLE 5.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARAMETERS</td>
</tr>
<tr>
<td>gainSet</td>
</tr>
<tr>
<td>OUTPUT</td>
</tr>
<tr>
<td>bool</td>
</tr>
</tbody>
</table>
5.1.4 Activate a Configuration

Activates the selected configuration, replacing the command sources.

```cpp
bool activateConfiguration(
    unsigned int configuration);
```

<table>
<thead>
<tr>
<th>TABLE 5.4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PARAMETERS</strong></td>
</tr>
<tr>
<td>configuration</td>
</tr>
<tr>
<td><strong>OUTPUT</strong></td>
</tr>
<tr>
<td>bool</td>
</tr>
</tbody>
</table>

Notes:
For continued motion planning, the configuration must use DriveLibrary as the command source. The only side-effect is to replace the command limiter parameters.

5.1.5 Get the Current Loop Parameterization

Provides a reference to the current loop parameters, which are common to the two gain sets.

```cpp
mdm::control::CurrentLoop * getCurrentLoop();
```

<table>
<thead>
<tr>
<th>TABLE 5.5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OUTPUT</strong></td>
</tr>
<tr>
<td>mdm::control::CurrentLoop*</td>
</tr>
</tbody>
</table>

Notes:
The signature for CurrentLoop is defined in AMC/Motion/Drive/Model/Control/CurrentLoop.h (see "Appendix E – Drive Setup Methods" on page 49).

5.1.6 Get Gain Set

Produces the selected gain set, wrapper for velocity and position loop parameters.

```cpp
mdm::control::GainSet * getGainSet(
    unsigned int gainSet);
```

<table>
<thead>
<tr>
<th>TABLE 5.6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PARAMETERS</strong></td>
</tr>
<tr>
<td>gainSet</td>
</tr>
<tr>
<td><strong>OUTPUT</strong></td>
</tr>
<tr>
<td>mdm::control::GainSet*</td>
</tr>
</tbody>
</table>

Notes:
The signature for GainSet is defined in AMC/Motion/Drive/Model/Control/GainSet.h (see "Appendix E – Drive Setup Methods" on page 49).
5.1.7 Update Gains

 Writes all changes to the CurrentLoop, PositionLoop, or VelocityLoop to the Axis.

 \[\text{bool } \text{updateGains}()\]

<table>
<thead>
<tr>
<th>TABLE 5.7</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OUTPUT</strong></td>
</tr>
<tr>
<td>bool True if transfer succeeded.</td>
</tr>
</tbody>
</table>

5.1.8 Get Current Limits

 Obtains a modifiable copy of the current limits to be implemented when updateLimits is invoked. Upon first access, the attribute values are the original drive settings.

 \[\text{mdm::status::CurrentIndication } * \text{getCurrentLimits}();\]

<table>
<thead>
<tr>
<th>TABLE 5.8</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OUTPUT</strong></td>
</tr>
<tr>
<td>mdm::status::CurrentIndication* Reference to an object containing the target values of the parameterization of the current limit algorithm.</td>
</tr>
<tr>
<td><strong>Notes:</strong></td>
</tr>
<tr>
<td>The signature for CurrentIndication is defined in AMC/Motion/Drive/Model/Status/CurrentIndication.h (see “Appendix E – Drive Setup Methods” on page 49).</td>
</tr>
</tbody>
</table>

5.1.9 Get Position Limits

 Obtains a modifiable copy of the position limits to be implemented when updateLimits is invoked. Upon first access, the attribute values are the original drive settings.

 \[\text{mdm::status::PositionIndication } * \text{getPositionLimits}();\]

<table>
<thead>
<tr>
<th>TABLE 5.9</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OUTPUT</strong></td>
</tr>
<tr>
<td>mdm::status::PositionIndication* Reference to an object containing the target position limits and windows.</td>
</tr>
<tr>
<td><strong>Notes:</strong></td>
</tr>
<tr>
<td>The signature for PositionIndication is defined in AMC/Motion/Drive/Model/Status/PositionIndication.h (see “Appendix E – Drive Setup Methods” on page 49).</td>
</tr>
</tbody>
</table>

5.1.10 Get Velocity Limits

 Obtains a modifiable copy of the velocity limits to be implemented when updateLimits is invoked. Upon first access, the attribute values are the original drive settings.

 \[\text{mdm::status::VelocityIndication } * \text{getVelocityLimits}();\]

<table>
<thead>
<tr>
<th>TABLE 5.10</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OUTPUT</strong></td>
</tr>
<tr>
<td>mdm::status::VelocityIndication* Reference to an object containing the target velocity limits and windows.</td>
</tr>
<tr>
<td><strong>Notes:</strong></td>
</tr>
<tr>
<td>The signature for VelocityIndication is defined in AMC/Motion/Drive/Model/Status/VelocityIndication.h (see “Appendix E – Drive Setup Methods” on page 49).</td>
</tr>
</tbody>
</table>
5.1.11 Update Limits

 Writes all changes to CurrentIndication, PositionIndication, or VelocityIndication to the Axis.

```cpp
bool updateLimits();
```

**TABLE 5.11**

<table>
<thead>
<tr>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
</tr>
<tr>
<td>True if transfer succeeded.</td>
</tr>
</tbody>
</table>
6.1 Motion Control Functions

These functions are used for controlling axis motion. It is important when calling these functions to understand how the user units are defined as they relate to motion. See "User Units" on page 15 for more information. Some general definitions are given below.

- "Position" is a value defined within a coordinate system. "Distance" is a relative measure related to user units. "Distance" is the difference between two positions.
- "Velocity", "Acceleration", "Deceleration", and "Jerk" are always positive values.
- "Position" and "Distance" can be both positive and negative.

For motion control functions operating in "BufferMode", the following modes are available:

- **Aborting**: aborts the current motion and begins the next motion immediately.
- **Buffering**: begins the next motion upon completion of the previous motion.
- **BlendingLow**: begins the next motion upon completion of the previous motion by blending the velocity of the lowest velocity between the previous and next motion at the end of the first move.
- **BlendingPrevious**: begins the next motion upon completion of the previous motion by blending the velocity of the previous motion at the end of the first move.
- **BlendingNext**: begins the next motion upon completion of the previous motion by blending the velocity of the next motion at the end of the first move.
- **BlendingHigh**: begins the next motion upon completion of the previous motion by blending the velocity of the highest velocity between the previous and next motion at the end of the first move.
6.1.1 Power

This function controls the power bridge of the drive (on or off).

```c
bool Power {
    bool enable,
    bool enablePositive,
    bool enableNegative);
```

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
<th>UNITS</th>
<th>RANGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>enable</td>
<td>bool</td>
<td>If <code>true</code>, the power bridge of the drive is enabled</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>enablePositive</td>
<td>bool</td>
<td>If <code>true</code>, permits motion in the positive direction</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>enableNegative</td>
<td>bool</td>
<td>If <code>true</code>, permits motion in the negative direction</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Notes:

- If the Power function is called with the Enable `true` while being in `DISABLED`, this either leads to `STAND_STILL` if there is no error in the axis, or to `ERROR_STOP` if an Error exists.
- If power fails (also during operation) it will generate a transition to the `ERROR_STOP` state.
- When Power is called with Enable false the axis goes to state `DISABLED` for every state including `ERROR_STOP`.

6.1.2 Home

This function commands the axis to perform a search for home sequence. The selection of the homing sequence is performed during drive commissioning using DriveWare (see the DriveWare Software Guide for more information). The "position" parameter is used to set the absolute position when the reference signal is detected. When the homing operation completes the axis state is changed to `STAND_STILL`.

```c
bool Home (FLOAT64 position);
```

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
<th>UNITS*</th>
<th>RANGE*</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>FLOAT64</td>
<td>Absolute position when the reference signal is detected, in drive units</td>
<td>counts</td>
<td>([-2^{31}] - [2^{31}-1])</td>
</tr>
</tbody>
</table>

*See "User Units" on page 15 for conversion to user units.

6.1.3 Busy

This function returns `true` if the axis is busy (e.g. executing a move commanded by a previously called function).

```c
bool Busy (void);
```

6.1.4 Done

This function returns `true` if the axis has completed its last commanded motion.

```c
bool Done (void);
```
### 6.1.5 Move Absolute

This function commands a controlled motion to a specific absolute position. It is not required to provide a jerk value. Including the jerk in the function call will execute an S-Curve, while a trapezoidal move will be performed if the jerk is not provided. The prototype is as follows:

```c
bool MoveAbsolute(
    FLOAT64 position,
    FLOAT64 velocity,
    FLOAT64 acceleration,
    FLOAT64 deceleration,
    FLOAT64 jerk,
    ctrlNS::Direction direction,
    ctrlNS::BufferMode bufferMode);
```

#### Table 6.3

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
<th>UNITS*</th>
<th>RANGE*</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>FLOAT64</td>
<td>Target position for the motion (negative or positive)</td>
<td>counts</td>
<td>[-2^{31}], [2^{31}-1]</td>
</tr>
<tr>
<td>velocity</td>
<td>FLOAT64</td>
<td>Value of the maximum velocity (always positive) (not necessarily reached)</td>
<td>counts/sec</td>
<td>0 - [2^{32}-1]</td>
</tr>
<tr>
<td>acceleration</td>
<td>FLOAT64</td>
<td>Value of the acceleration (always positive) (increasing energy of the motor)</td>
<td>counts/sec²</td>
<td>0 - [2^{32}-1]</td>
</tr>
<tr>
<td>deceleration</td>
<td>FLOAT64</td>
<td>Value of the deceleration (always positive) (decreasing energy of the motor)</td>
<td>counts/sec²</td>
<td>0 - [2^{32}-1]</td>
</tr>
<tr>
<td>jerk</td>
<td>FLOAT64</td>
<td>Value of the Jerk (always positive)</td>
<td>counts/sec³</td>
<td>0 - [2^{32}-1]</td>
</tr>
<tr>
<td>direction</td>
<td>ctrlNS::Direction</td>
<td>Enum type (1-of-4 values: ctrlNS::Positive_Direction, ctrlNS::Shortest_Way, ctrlNS::Negative_Direction, ctrlNS::Current_Direction)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>bufferMode</td>
<td>ctrlNS::BufferMode</td>
<td>Enum type (1-of-6 values: see &quot;Motion Control Functions&quot; on page 24). If omitted defaults to aborting.</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Notes:
- This action completes with velocity zero if no further actions are pending.
- The velocity, acceleration, deceleration and jerk must all be greater than zero.
- If there is only one mathematical solution to reach the commanded position (like in linear systems), the value of the input Direction is ignored.
- The Enum type 'shortest_way' is focused to a trajectory that will go through the shortest route. The decision which direction to go is based on the current position when the command is issued.
- If the jerk is provided in the function call, an S-Curve move will be executed. A trapezoidal move will be executed if the jerk is not provided.

* See "User Units" on page 15 for conversion to user units.

The following figure illustrates the use of MoveAbsolute, and the behavior of aborting moves. "Abort" in this case does not mean "stop moving". Rather, it means to recalculate the trajectory using new values for position, velocity, acceleration, deceleration, and jerk.

The example illustrates a move to a waypoint at position 6000, with a test signal that triggers a move to a working position at 10000. Two cases are shown. On the left, the waypoint is
reached before the test signal triggers the second move. On the right, the test signal occurs before the waypoint is reached.

Note that in both cases the final position is 10000. In the second case, however, the slower velocity of the second move is achieved even before the waypoint at 6000 is reached.

**FIGURE 6.1 Timing Diagram for MoveAbsolute**

*MoveAbsolute - Example*

---

```c
ctrlNS::Axis myX(0x1, "C:\\Drive_1.adf", false);
........
myX.MoveAbsolute(6000, 3000, 10000, 10000,
ctrlNS::Positive_Direction);
while (!test())
{
    MILLI_SECOND_SLEEP( 10 );
}
myX.MoveAbsolute(10000, 2000, 10000, 10000,
ctrlNS::Positive_Direction);
while (myX.Busy())
{
    MILLI_SECOND_SLEEP( 10 );
}
```
6.1.6 Move Relative

This function commands a controlled motion of a specified distance relative to the actual position at the time of the execution. It is not required to provide a jerk value. Including the jerk in the function call will execute an S-Curve, while a trapezoidal move will be performed if the jerk is not provided.

```c
bool MoveRelative(
    FLOAT64 distance,
    FLOAT64 velocity,
    FLOAT64 acceleration,
    FLOAT64 deceleration,
    FLOAT64 jerk,
    ctrlNS::BufferMode bufferMode);
```

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
<th>UNITS*</th>
<th>RANGE*</th>
</tr>
</thead>
<tbody>
<tr>
<td>distance</td>
<td>FLOAT64</td>
<td>Relative distance for the motion (in drive units)</td>
<td>counts</td>
<td>-2^{31} - [2^{31}-1]</td>
</tr>
<tr>
<td>velocity</td>
<td>FLOAT64</td>
<td>Value of the maximum velocity (not necessarily reached)</td>
<td>counts/sec</td>
<td>0 - [2^{32}-1]</td>
</tr>
<tr>
<td>acceleration</td>
<td>FLOAT64</td>
<td>Value of the acceleration (increasing energy of the motor)</td>
<td>counts/sec^2</td>
<td>0 - [2^{32}-1]</td>
</tr>
<tr>
<td>deceleration</td>
<td>FLOAT64</td>
<td>Value of the deceleration (decreasing energy of the motor)</td>
<td>counts/sec^2</td>
<td>0 - [2^{32}-1]</td>
</tr>
<tr>
<td>jerk</td>
<td>FLOAT64</td>
<td>Value of the Jerk</td>
<td>counts/sec^3</td>
<td>0 - [2^{32}-1]</td>
</tr>
<tr>
<td>buffer Mode</td>
<td>ctrlNS::BufferMode</td>
<td>Enum type [1-of-6 values: see &quot;Motion Control Functions&quot; on page 24]. If omitted defaults to aborting.</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Notes:
- This action completes with velocity zero if no further actions are pending.
- The velocity, acceleration, deceleration and jerk must all be greater than zero.
- If the jerk is provided in the function call, an S-Curve move will be executed. A trapezoidal move will be executed if the jerk is not provided.

*See "User Units" on page 15 for conversion to user units.

The following figure revisits Figure 6.1, but this time using relative moves by distances of 6000 and 4000. Again, two examples are shown. The application might be to chase after a part on a slow-moving conveyor belt, followed by a slower move to track the part while a tool is used. A test signal indicates that we have caught the part, and should synchronize our motion with it. This is a difficult problem in real-world applications; to be practicable, the example would have to have extremely high accelerations and jerk.

On the left, we reach a waypoint before the part is detected. On the right, the part is detected before the waypoint. In the second case, the final position is significantly less than the sum of the two distances.
FIGURE 6.2 Timing Diagram for MoveRelative

MoveRelative - Example

```c
ctrlNS::Axis myX(0x1, "C:\Drive_1.adf", false);

....... myX.MoveRelative(6000, 3000, 100, 100, 4000);
while ( !test() )
{
    MILLI_SECOND_SLEEP( 10 );
}
myX.MoveRelative(4000, 2000, 100, 100, 4000);
while ( myX.Busy() )
{
    MILLI_SECOND_SLEEP( 10 );
}
```
6.1.7 Move Velocity

This function commands a never ending controlled motion at a specified velocity. It is not required to provide a jerk value. Including the jerk in the function call will execute an S-Curve, while a trapezoidal move will be performed if the jerk is not provided.

```c
bool MoveVelocity(
    FLOAT64 velocity,
    FLOAT64 acceleration,
    FLOAT64 deceleration,
    FLOAT64 jerk,
    ctrlNS::Direction direction,
)
```

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
<th>UNITS*</th>
<th>RANGE*</th>
</tr>
</thead>
<tbody>
<tr>
<td>velocity</td>
<td>FLOAT64</td>
<td>Value of the maximum velocity (not necessarily reached)</td>
<td>counts/sec</td>
<td>0 - ([2^{32}] - 1)</td>
</tr>
<tr>
<td>acceleration</td>
<td>FLOAT64</td>
<td>Value of the acceleration (increasing energy of the motor)</td>
<td>counts/sec²</td>
<td>0 - ([2^{32}] - 1)</td>
</tr>
<tr>
<td>deceleration</td>
<td>FLOAT64</td>
<td>Value of the deceleration (decreasing energy of the motor)</td>
<td>counts/sec²</td>
<td>0 - ([2^{32}] - 1)</td>
</tr>
<tr>
<td>jerk</td>
<td>FLOAT64</td>
<td>Value of the jerk</td>
<td>counts/sec³</td>
<td>0 - ([2^{32}] - 1)</td>
</tr>
<tr>
<td>direction</td>
<td>ctrlNS::Direction</td>
<td>Enum type (1-of-3 values: ctrlNS::Positive_Direction, ctrlNS::Negative_Direction, ctrlNS::Current_Direction)</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Notes:

- If the second form is used, the motion is trapezoidal rather than S-Curve.
- To stop the motion, the function has to be interrupted by another function issuing a new command.
- The velocity, acceleration, deceleration and jerk must all be greater than zero.

*See "User Units" on page 15 for conversion to user units.

The following figure shows a two-speed motion, the transition again triggered by an external signal.

For velocity moves, the axis is Busy only until the acceleration phase completes. The axis will then hold that velocity until another command is issued.

On the left side of the figure, we show the typical case, which is that the initial velocity acceleration to 3000 is complete when the test signal goes high. The second call to MoveVelocity controls the deceleration to 2000.

On the right side, we illustrate the behavior when the first velocity move is interrupted before reaching terminal speed. The acceleration is interrupted, and the axis will slow down to the second speed.
FIGURE 6.3 Move Velocity Timing Diagram

MoveVelocity - Example

Sequence of two complete motions
Second motion interrupts First motion

Motion

Test
Busy

Velocity

Motion Control Functions / Single-Axis Functions

ctrlNS::Axis myX(0x1, "C:\\Drive_1.adf", false);
........
myX.MoveVelocity(3000, 10000, 10000, 4000);
while (!test())
{
    MILLI_SECOND_SLEEP(10);
}
myX.MoveVelocity(2000, 10000, 10000, 4000);
while (!myX.Busy())
{
    MILLI_SECOND_SLEEP(10);
}
6.1.8 Halt

This function commands a controlled motion stop. It aborts any ongoing function execution. The axis moves to the state **DISCRETE_MOTION**, until the velocity is zero. When the velocity reaches zero the state of the axis is changed to **STAND_STILL**.

```cpp
bool Halt(
    FLOAT64 deceleration;
    FLOAT64 jerk);
```

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
<th>UNITS*</th>
<th>RANGE*</th>
</tr>
</thead>
<tbody>
<tr>
<td>deceleration</td>
<td>FLOAT64</td>
<td>Value of the deceleration</td>
<td>counts/sec²</td>
<td>0 - [2³²-1]</td>
</tr>
<tr>
<td>jerk</td>
<td>FLOAT64</td>
<td>Value of the jerk</td>
<td>counts/sec³</td>
<td>0 - [2³²-1]</td>
</tr>
</tbody>
</table>

Notes:
- Halt is used to stop the axis under normal operation conditions. In non-buffered mode: during deceleration of the axis it is possible to call another motion function, which will abort the Halt and will be executed immediately.
- The deceleration and jerk must both be positive and non-zero.
- If this command is active the next command can be issued. E.g. a driverless vehicle detects an obstacle and needs to stop. Halt is issued. Before the standstill is reached the obstacle may be removed and the motion may be continued by setting another motion command (the vehicle does not stop).
- Setting the deceleration and jerk to a value that is less than the current acceleration when the command is issued may cause overshoot. It is recommended that the stop, decel, and jerk value be greater than any values used for moves.

*See "User Units" on page 15 for conversion to user units.

The diagram shows the two cases described in the notes. On the left, an obstacle is detected and a halt command is issued. The obstacle persists and the axis reaches zero speed.

On the right, before zero speed is reached the obstacle clears and the axis is signaled (with go) to resume motion.

Another motion overrides the Halt command. Halt allows this, in contrast to Stop. The axis can accelerate again without reaching a standstill.

```cpp
ctrlNS::Axis myX(0x1, "C:\\Drive_1.adf", false);
........
myX.MoveVelocity(3000, 100, 100, 4000);
........
while (true) {
    // Check for an external interrupt.
    if (obstacle()) {
        myX.Halt(200, 400);
        break;
    }
    MILLI_SECOND_SLEEP(10);
}
// Wait for trigger on second move. May happen during Halt!
while (!go) MILLI_SECOND_SLEEP(10);
myX.MoveVelocity(3000, 100, 100, 4000);
```
6.1.9 Stop

This function commands a controlled motion stop and transfers the axis to the state `STOPPING`. While the axis is in state `STOPPING`, no other function can perform any motion on the same axis. After the axis has reached velocity zero, the axis Done status is set to `true`. The axis remains in the state `STOPPING` as long as zero velocity is not yet reached. As soon as the Done status is set to `true`, the axis goes to state `STAND_STILL`.

```c
bool Stop(
    FLOAT64 deceleration,
    FLOAT64 jerk);
```

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
<th>UNITS*</th>
<th>RANGE*</th>
</tr>
</thead>
<tbody>
<tr>
<td>deceleration</td>
<td>FLOAT64</td>
<td>Value of the deceleration</td>
<td>counts/sec²</td>
<td>0 - [2³²-1]</td>
</tr>
<tr>
<td>jerk</td>
<td>FLOAT64</td>
<td>Value of the jerk</td>
<td>counts/sec³</td>
<td>0 - [2³²-1]</td>
</tr>
</tbody>
</table>

Notes:
- The deceleration and jerk must both be greater than zero.
- Setting the deceleration and jerk to a value that is less than the current acceleration when the command is issued may cause overshoot. It is recommended that the stop, decel, and jerk value be greater than any values used for moves.

Stop should be used when the programmer wishes to ensure that motion is terminated. Because the axis enters the special state `STOPPING`, other software modules can recognize the state and avoid conflicting actions. One example might be to end machine motion when a part feeder runs out of stock.

Stop is intended to be a graceful end to motion that reflects that current condition of the axis. It is also expected that the application will be able to sustain system control without human intervention to evaluate and/or replace hardware.

The figure illustrates a triggered stop of a rotating axis using Stop. On the left, the stop completes without any further calls to the application. On the right side, the application attempts to start another move before the stop completes, generating a logic error.
6.1.10 Dwell

This function will command the drive to hold position for a fixed time interval. It can be used to initiate a move sequence, or be appended to any directed move (AbsoluteMove or RelativeMove). A dwell can be succeeded by an aborting or buffered move.

```c++
bool Dwell(
    FLOAT64   time);
```

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
<th>UNITS*</th>
<th>RANGE*</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>FLOAT64</td>
<td>Time of dwell</td>
<td>sec</td>
<td>&gt;0</td>
</tr>
</tbody>
</table>
6.1.11 Set Position

This function shifts the coordinate system of an axis by manipulating both the commanded position as well as the measured position of an axis with the same value (re-calibration). This function can only be used when the axis is in the disabled state.

```cpp
bool SetPosition(
    FLOAT64 position,
    bool mode);
```

**TABLE 6.9**

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
<th>UNITS*</th>
<th>RANGE*</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>FLOAT64</td>
<td>Position in drive units (Means 'Distance' if Mode = RELATIVE)</td>
<td>counts</td>
<td>[(-2^{31})] - [(2^{31}-1)]</td>
</tr>
<tr>
<td>mode</td>
<td>bool</td>
<td>RELATIVE = true, ABSOLUTE = false (Default)</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Notes:
RELATIVE means that the value is added to the current position. ABSOLUTE means that the current position is set to the value.

*See "User Units" on page 15 for conversion to user units.

6.1.12 Set Override

This function sets the values of override for the whole axis, and all member functions that work on that axis. The override parameters act as a factor that is multiplied to the commanded velocity, acceleration, deceleration and jerk of a move function.

```cpp
bool SetOverride(
    FLOAT64 velFactor,
    FLOAT64 accFactor,
    FLOAT64 jerkFactor);
```

**TABLE 6.10**

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
<th>UNITS*</th>
<th>RANGE*</th>
</tr>
</thead>
<tbody>
<tr>
<td>velFactor</td>
<td>FLOAT64</td>
<td>New override factor for the velocity</td>
<td>-</td>
<td>0 - 1</td>
</tr>
<tr>
<td>accFactor</td>
<td>FLOAT64</td>
<td>New override factor for the acceleration/deceleration</td>
<td>-</td>
<td>0 - 1</td>
</tr>
<tr>
<td>jerkFactor</td>
<td>FLOAT64</td>
<td>New override factor for the jerk</td>
<td>-</td>
<td>0 - 1</td>
</tr>
</tbody>
</table>

Notes:
The Input accFactor acts on positive and negative acceleration (deceleration).
The function sets the factor and this factor is applied to all subsequent motion commands (except as noted). The override factor is valid until a new override is set.
The default values of the override factor are 1.0.
The value of the overrides can be between 0.0 and 1.0. Values > 1.0 and < 0.0 are not allowed.
The function does not influence the state diagram of the axis.
If in Discrete motion, reducing the AccFactor and/or JerkFactor can lead to a position overshoot – a possible cause of damage.
This factor has no effect on PVTProfile moves.

*See "User Units" on page 15 for conversion to user units.
6.2 Error Handling Functions

These functions are used for error and fault condition processes. For more information on error handling, see "Error Handling" on page 14 and "Appendix F - Error Handling" on page 55.

6.2.1 Select Error Stop

This function allows the programmer to select the method to use when commanding the drive to stop from **ERROR_STOP** state.

```c
void SelectErrorStop(
    ctrlNS::ErrorStopType p_eStopType );
```

**TABLE 6.11**

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
<th>UNITS</th>
<th>RANGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>StopType</td>
<td>ctrlNS::ErrorStopType</td>
<td>The stop type to execute. From ctrlNS::, there is StopHard, StopPerLastMove, StopPerErrorConfig.</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Notes:**
The default is StopHard.
If StopPerErrorConfig is selected, the default behavior is similar to StopHard. To get different behavior, use ConfigureErrorStop.

6.2.2 Activate Errors

This function allows the programmer to specify the conditions that will force the drive into **ERROR_STOP** state.

```c
bool ActivateErrors(
    statNS::EventItem stopStatus,
    ctrlNS::AxisErrorList stopList );
```

**TABLE 6.12**

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
<th>UNITS</th>
<th>RANGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>stopStatus</td>
<td>statNS::EventItem</td>
<td>std::bitset large enough to cover the amc::axis::Events enumeration (see &quot;Appendix E – Event Enumerations&quot; on page 52). Detecting an active bit in the drive status (see &quot;Read Events&quot; on page 38) will cause DriveLibrary to initiate error stop procedures.</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>stopList</td>
<td>ctrlNS::AxisErrorList</td>
<td>A list of ctrlNS::Error enumeration entries (see &quot;Appendix B – Error Identifiers&quot; on page 45). When DriveLibrary generates an error in the list, it will initiate error stop procedures.</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Notes:**
To deactivate errors, pass empty lists.
To set an entry in stopStatus, e.g. - for position following error:
```cpp
stopStatus.set( ctrlNS::Axis::POSITION_FOLLOWING_ERROR);
```
To add an entry to the stopList, e.g. - for invalid acceleration parameter:
```cpp
stopList.push_back( ctrlNS::Axis::INVALID_ACCELERATION_PARAMETER);
```

6.2.3 Configure Error Stop

This function allows the programmer explicit control over how the axis will implement an error stop. By default, an error stop will be executed using the deceleration and jerk from the move in progress. Through this function, those values can be modified.
bool ConfigureErrorStop(
  FLOAT64 deceleration,
  FLOAT64 jerk,
  FLOAT64 time);

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
<th>UNITS*</th>
<th>RANGE*</th>
</tr>
</thead>
<tbody>
<tr>
<td>deceleration</td>
<td>FLOAT64</td>
<td>The deceleration to use in an error stop. If zero, the value will be taken from the last move. See notes below for details.</td>
<td>counts/sec²</td>
<td>0 - [2^32-1]</td>
</tr>
<tr>
<td>jerk</td>
<td>FLOAT64</td>
<td>The jerk to use in an error stop. See notes below for details.</td>
<td>counts/sec³</td>
<td>0 - [2^32-1]</td>
</tr>
<tr>
<td>time</td>
<td>FLOAT64</td>
<td>Maximum desired elapsed time to end of motion. The default on axis creation is 0.1sec.</td>
<td>sec</td>
<td>0 - [2^31-1]</td>
</tr>
</tbody>
</table>

Notes:
If the planned error stop must take more than 0.1sec, this function must be used to set the time parameter to a larger value.
The algorithm for arriving at the deceleration is:
- Use the error stop deceleration (defined here).
- Use the value from the last move.
- If the stop will take longer than the error stop time, use the current velocity divided by the error stop time.
The algorithm for arriving at the jerk is:
- Use the error stop jerk (defined here).
- Use the value from the last move.
- Make sure that the jerk will last at most 20% on either end of the error stop.

*See "User Units" on page 15 for conversion to user units.

### 6.2.4 Error

This function returns **true** if the axis is in **ERROR_STOP** state.

```cpp
bool Error (void);
```

### 6.2.5 Get Stopping Errors

This function publishes the list of active stopping errors. The list is cleared on Reset. See "Error Entry and Error Report" on page 59 for ErrorReport structure.

```cpp
void GetStoppingErrors(
  ctrlNS::ErrorReport        & p_rStopErrors);
```

### 6.2.6 Get Logic Errors

This function publishes the list of active logic errors. The list is always cleared on each entry to the Axis interface, except for status methods (i.e. Error, Busy, etc.). See "Error Entry and Error Report" on page 59 for ErrorReport structure.

```cpp
void GetLogicErrors(
  ctrlNS::ErrorReport        & p_rLogicErrors);
```

### 6.2.7 Reset

This function makes the transition from the state **ERROR_STOP** to **STAND_STILL** by resetting all drive or DriveLibrary errors. It returns **false** if the reset action is not possible.

```cpp
bool Reset  (void);
```
6.3 Status Functions

6.3.1 Read State

This function returns the state of the library state machine. See Table 6.14 below for a list of the state identifiers.

```cpp
bool ReadState (ctrlNS::AxisState& state);
```

**TABLE 6.14**

<table>
<thead>
<tr>
<th>state</th>
<th>ctrlNS::AxisState</th>
<th>The location where the state of the axis will be stored</th>
</tr>
</thead>
</table>

**Notes:**

The Axis states as defined in the ctrlNS::axis namespace:

- AXIS_DISABLED
- UNINITIALIZED
- STAND_STILL
- HOMING
- ERROR_STOP
- STOPPING
- DISCRETE_MOTION
- CONTINUOUS_MOTION
- SYNCHRONIZED_MOTION

6.3.2 Read Events

This function returns the status of the drive. See "Appendix E – Event Enumerations" on page 52 for a list of the status identifiers.

```cpp
bool ReadEvents (  
    ctrlNS::EventMask       &event,  
    ctrlNS::EventFilter filter);
```

**TABLE 6.15**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>ctrlNS::EventFilter</th>
<th>Enum type (1- of - 4 values, ctrlNS::History, ctrlNS::Active, ctrlNS::History_Inhibiting, ctrlNS::Active_Inhibiting) if omitted default is History</th>
</tr>
</thead>
<tbody>
<tr>
<td>filter (optional)</td>
<td>ctrlNS::EventFilter</td>
<td>ctrlNS::EventFilter values are as follows:</td>
</tr>
<tr>
<td>status</td>
<td>ctrlNS::EventMask</td>
<td>The location where the status of the axis will be stored</td>
</tr>
</tbody>
</table>

**Notes:**

Since multiple status bits can be set, ctrlNS::AxisStatus derives from std::bitset so that individual bits can be tested. The bits are enumerated in ctrlNS::AxisStatusItem (See "Appendix E – Event Enumerations" on page 52).

ctrlNS::EventFilter values are as follows:

- ctrlNS::History - All events until cleared by a Reset function
- ctrlNS::History_Inhibiting - Only events that have been inhibiting motion in the past
- ctrlNS::History_Faulting - Only events that have faulted the bridge in the past
- ctrlNS::Active - Only the events that are currently active (at the time of the call)
- ctrlNS::Active_Inhibiting - Only events that are inhibiting motion (at the time of the call)
- ctrlNS::Active_Faulting - Only events that fault the bridge (at the time of the call)
### 6.3.3 Clear Event

The programmer may wish to monitor events and initiate responsive actions independent of DriveLibrary. The events must be latched (using DriveWare). When the event is detected and the response initiated, the application must clear the latched event. This function enables that control.

```
bool ClearEvent (statNS::EventItem resetEvent);
```

**TABLE 6.16**

<table>
<thead>
<tr>
<th>PARAMETERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>resetEvent statNS::EventItem The event to reset.</td>
</tr>
</tbody>
</table>

**Notes:**

ClearEvent cannot be used on system protection events. ClearEvent can only clear non-inhibiting events.

---

Note that ClearEvent cannot be used for events that fault the drive, or that are selected to trigger an Error Stop using ActivateErrors.

### 6.3.4 In Velocity

This function returns `true` if the axis is running at constant velocity.

```
bool InVelocity (void);
```

### 6.3.5 Read Parameter

This function returns the value of a specific parameter.

```
bool ReadParameter( 
ctrNS::AxisParameterId parameter 
FLOAT64 & value);
```

**TABLE 6.17**

<table>
<thead>
<tr>
<th>PARAMETERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter ctrNS::AxisParameterId The selected value. See &quot;Appendix C – Parameter Enumerations&quot; on page 47 for an enumeration.</td>
</tr>
</tbody>
</table>

**OUTPUT**

| value FLOAT64 & | The value of the parameter requested. Units are defined in "Appendix C – Parameter Enumerations" on page 47. |
6.3.6 Read Boolean Parameter

This function returns the value of a specific boolean parameter.

```c
bool ReadBoolParameter(
    ctrlNS::AxisParameterId parameter
    bool & value);
```

<table>
<thead>
<tr>
<th>TABLE 6.18</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PARAMETERS</strong></td>
</tr>
<tr>
<td>parameter</td>
</tr>
<tr>
<td>OUTPUT</td>
</tr>
<tr>
<td>value</td>
</tr>
</tbody>
</table>

6.3.7 Write Parameter

This function modifies the value of a specific parameter.

```c
bool WriteParameter(
    ctrlNS::AxisParameterId parameter
    FLOAT64 value);
```

<table>
<thead>
<tr>
<th>TABLE 6.19</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PARAMETERS</strong></td>
</tr>
<tr>
<td>parameter</td>
</tr>
<tr>
<td>value</td>
</tr>
</tbody>
</table>

6.3.8 Write Boolean Parameter

This function modifies the value of a specific parameter.

```c
bool WriteBoolParameter(
    ctrlNS::AxisParameterId parameter
    bool value);
```

<table>
<thead>
<tr>
<th>TABLE 6.20</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PARAMETERS</strong></td>
</tr>
<tr>
<td>parameter</td>
</tr>
<tr>
<td>value</td>
</tr>
</tbody>
</table>
6.3.9 Read Digital Input

This function returns the value of a specific digital input.

```c
bool ReadDigitalInput(
    INT inputNumber
    bool & value);
```

<table>
<thead>
<tr>
<th>TABLE 6.21</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PARAMETERS</strong></td>
</tr>
<tr>
<td>inputNumber</td>
</tr>
<tr>
<td><strong>OUTPUT</strong></td>
</tr>
<tr>
<td>value</td>
</tr>
<tr>
<td><strong>Notes:</strong></td>
</tr>
<tr>
<td>Values are only sampled when the function is called.</td>
</tr>
<tr>
<td>Function is dependent on the number of I/O for the drive in use. See the drive datasheet for I/O details.</td>
</tr>
</tbody>
</table>

6.3.10 Read Digital Output

This function returns the value of a specific physical digital output.

```c
bool ReadDigitalOutput(
    INT outputNumber
    bool & value);
```

<table>
<thead>
<tr>
<th>TABLE 6.22</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PARAMETERS</strong></td>
</tr>
<tr>
<td>outputNumber</td>
</tr>
<tr>
<td><strong>OUTPUT</strong></td>
</tr>
<tr>
<td>value</td>
</tr>
<tr>
<td><strong>Notes:</strong></td>
</tr>
<tr>
<td>Values are only sampled when the function is called.</td>
</tr>
</tbody>
</table>

6.3.11 Write Digital Output

This function writes a value to a specific User Bit. User Bits can be mapped to a physical digital output using DriveWare.

```c
bool WriteDigitalOutput(
    INT outputNumber
    bool value);
```

<table>
<thead>
<tr>
<th>TABLE 6.23</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PARAMETERS</strong></td>
</tr>
<tr>
<td>outputNumber</td>
</tr>
<tr>
<td>value</td>
</tr>
</tbody>
</table>
6.3.12 Read Actual Position

This function returns the current position in counts.

```c
bool ReadActualPosition( 
    FLOAT64 & position);
```

**TABLE 6.24**

<table>
<thead>
<tr>
<th>OUTPUT</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
<th>UNITS</th>
<th>RANGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>FLOAT64 &amp;</td>
<td>The axis position in drive units</td>
<td>counts</td>
<td>[-2^{31}] - [2^{31} - 1]</td>
</tr>
</tbody>
</table>

6.3.13 Read Actual Velocity

This function returns the actual velocity.

```c
bool ReadActualVelocity( 
    FLOAT64 & velocity);
```

**TABLE 6.25**

<table>
<thead>
<tr>
<th>OUTPUT</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
<th>UNITS</th>
<th>RANGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>velocity</td>
<td>FLOAT64 &amp;</td>
<td>The value of the actual velocity</td>
<td>counts/sec</td>
<td>0 - [2^{32} - 1]</td>
</tr>
</tbody>
</table>

6.3.14 Read Actual Current

This function returns the actual current in amperes. This current value is the actual current that delivers power to the motor shaft.

```c
bool ReadActualCurrent( 
    FLOAT64 & current);
```

**TABLE 6.26**

<table>
<thead>
<tr>
<th>OUTPUT</th>
<th>TYPE</th>
<th>DESCRIPTION</th>
<th>UNITS</th>
<th>RANGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current</td>
<td>FLOAT64 &amp;</td>
<td>The value of the actual current</td>
<td>amperes</td>
<td>±2 * (power board peak current)</td>
</tr>
</tbody>
</table>

Notes: The output actual current can be a signed value.
DriveLibrary requires three files to load a system configuration. The first, the system file, typically has the "asf" extension. It simply names the system, and identifies its networks. The file has a standard "ini" or config file format, with sections in square brackets and key=value entries. Note that the keys are case-sensitive: they must be typed exactly as shown in the examples below.

```
# This file describes a control system to DriveLibrary. The system has a name, and a network. Note that
# comments can be added after a '#'.

[System]
name=AMC Test Kit
# Additional networks will be added as [Network2], etc.
[Network1]
# If the directory is not included in the network file path, it is assumed that it is with the system
# file.
path=CAN_network.anf
```

The second file, the network ("anf") file, provides communication settings for the network and all of the axes attached to it.

```
[Network]
name=AMC CAN
# Valid types are 0 for RS-232, 3 for CAN, and 8 for RS-485.
type=3
baudrate=1000000
# The port is actually a text string. "COM4" would be typical.
port=0

[Axis1]
# The name must be unique on the network.
name=Left Spinner
# Used by the drive to recognize its messages.
address=1
# Time interval between points on the trajectories generated by DriveLibrary (e.g. - from Move calls
# over the API). The longer this time, the less communication over the network, but also the greater the
# delay when implementing a change in motion.
pvt_step_size=75 # [milliseconds]
status_interval=20 # [milliseconds]
spooler_interval=20 # [milliseconds]
# If the directory is not included in the network file path, it is assumed that it is with the system
# file.
path=can_config.adf

[Axis2]
name=Right Spinner
path=spinner2.adf
address=2
pvt_step_size=75 # [milliseconds]
status_interval=20 # [milliseconds]
spooler_interval=20 # [milliseconds]
```
path=can_config2.adf

On an RS-485 network, the following would be used:

[Network]
name=AMC RS-485

# Valid types are 0 for RS-232, 3 for CAN, and 8 for RS-485.
type=8
baudrate=115200
port=COM4

The third file, the "adf" file, can be retrieved from the drive and modified using DriveWare. This file must be referenced in the "anf" file for each axis that is defined.

Sample files are included with the DriveLibrary distribution in the Dependencies folder. Contact ADVANCED Motion Controls technical support if assistance is required in setting these up.
The following list identifies the error conditions that may occur during axis control. These identifiers are defined in the amc::motion::control (ctrlNS) namespace, so each must be prefaced by `amc::motion::control::` when used.

```cpp
enum Error {
    NO_AXIS_ERRORS = 0,
    MIN_ERROR_CODE = 0x10040200,
    LIBRARY_DISABLED = MIN_ERROR_CODE,
    PLCOPEN_TRANSITION_FAULT,
    AXIS_DOES_NOT_EXIST,
    AXIS_HAS_ERROR_STATUS,
    AXIS_FAULTED,
    ILLEGAL_ACTION,
    COMMAND_ABORTED,
    INVALID_POSITION_PARAMETER,
    INVALID_VELOCITY_PARAMETER,
    INVALID_ACCELERATION_PARAMETER,
    INVALID_DECELERATION_PARAMETER,
    INVALID_JERK_PARAMETER,
    INVALID_OFFSET_PARAMETER,
    INVALID_MOVE_TIME_PARAMETER,
    INVALID_DIRECTION_PARAMETER,
    INVALID_IO_POINT,
    COMMAND_OR_FEATURE_NOT_IMPLEMENTED,
    COMMAND_OR_FEATURE_NOT_AVAILABLE,
    UNABLE_TO_HOME_THE_AXIS,
    CANNOT_CONNECT_TO_DRIVE,
    ERROR_DRIVE_QUERY_FAILED,
    ERROR_DRIVE_CONFIG_FAILED,
    ERROR_STOP_IN_PROGRESS,
    ERROR_UNABLE_TO_CHANGE_ENABLE_STATE,
    ERROR_UNABLE_TO_CHANGE_MODES,
    ERROR_UNABLE_TO_START_PVT_MOVE,
    ERROR_UNABLE_TO_START_POSITION_MOVE,
    ERROR_UNABLE_TO_START_VELOCITY_MOVE,
    ERROR_UNABLE_TO_WRITE_TO_THE_DIGITAL_OUTPUTS,
    ERROR_UNABLE_TO_STOP_THE_DRIVE,
};
```
ERROR_UNABLE_TO_RESET_THE_DRIVE,
ERROR_UNABLE_TO_GET_EVENTS_FROM_THE_DRIVE,
ERROR_UNABLE_TO_RESET_DRIVE_EVENTS,
ERROR_UNABLE_TO_READ_LAST_POINT,
ERROR_UNABLE_TO_TRANSFER_PVT_POINTS,
ERROR_UNABLE_TO_START_THE_PVT_SPOOLER,
ERROR_UNABLE_TO_STOP_THE_PVT_SPOOLER,
ERROR_UNABLE_TO_PURGE_THE_PVT_SPOOLER,
FAILED_TO_GENERATE_PROFILE,
NO_FRAME_BUFFER,
ERROR_EMPTY_FRAME_BUFFER,
ERROR_NULL_PVT_SEQUENCE,
ERROR_EMPTY_PVT_SEQUENCE,
ERROR_NULL_PVP_SEQUENCE,
ERROR_EMPTY_PVP_SEQUENCE,
INVALID_PARAMETER_ID,
PARAMETER_NOT_SUPPORTED,
READ_ONLY PARAMETER,
WRONG_PARAMETER_TYPE,
REQUESTED_ACTION_HAS_NO_EFFECT,
ERROR_UNABLE_TO_SET_THE_POSITION_LIMIITS,
ERROR_CONNECTION_TO_DRIVE_LOST,
ERROR_UNABLE_TO_SET_THE_POSITION,
ERROR_UNABLE_TO_READ_CAN_MESSAGE,
ERROR_UNABLE_TO_WRITE_CAN_MESSAGE,
MAX_ERROR_CODE
}
These parameters are available for use in the application program, and typically are not intended for commissioning tools like operator panels, etc. (the drive is not visible – only the axis position).

Note: that the most used parameters are accessible directly via functions, and are not listed here.

These identifiers are defined in the amc::motion::control (ctrlNS) namespace, and so must be prefaced by amc::motion::control:: when used.

(Note: PN is Parameter Number see ReadParameter / WriteParameter and boolean versions)

### TABLE C.1 Parameters for ReadParameter and WriteParameter

<table>
<thead>
<tr>
<th>PN</th>
<th>Name</th>
<th>Datatype</th>
<th>B/E</th>
<th>R/W</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CommandedPosition</td>
<td>DOUBLE</td>
<td>B</td>
<td>R</td>
<td>Commanded position</td>
</tr>
<tr>
<td>2</td>
<td>MaxMeasuredPositionLimit</td>
<td>DOUBLE</td>
<td>E</td>
<td>R/W</td>
<td>Positive Software limit switch position</td>
</tr>
<tr>
<td>3</td>
<td>MinMeasuredPositionLimit</td>
<td>DOUBLE</td>
<td>E</td>
<td>R/W</td>
<td>Negative Software limit switch position</td>
</tr>
<tr>
<td>4</td>
<td>EnablePositionLimits</td>
<td>BOOL</td>
<td>E</td>
<td>R/W</td>
<td>Enable positive software limit switch</td>
</tr>
<tr>
<td>5</td>
<td>MaxPositionLag</td>
<td>DOUBLE</td>
<td>E</td>
<td>R/W</td>
<td>Maximal position lag</td>
</tr>
<tr>
<td>6</td>
<td>EnablePosLagMonitoring</td>
<td>BOOL</td>
<td>E</td>
<td>R/W</td>
<td>Enable monitoring of position lag</td>
</tr>
<tr>
<td>7</td>
<td>MaxPositiveVelocitySystem</td>
<td>DOUBLE</td>
<td>E</td>
<td>R</td>
<td>Maximal allowed positive velocity of the axis in the motion system</td>
</tr>
<tr>
<td>8</td>
<td>MaxNegativeVelocitySystem</td>
<td>DOUBLE</td>
<td>E</td>
<td>R</td>
<td>Maximal allowed negative velocity of the axis in the motion system</td>
</tr>
<tr>
<td>9</td>
<td>MaxVelocityAppl</td>
<td>DOUBLE</td>
<td>B</td>
<td>R/W</td>
<td>Maximal allowed velocity of the axis in the application</td>
</tr>
<tr>
<td>10</td>
<td>ActualVelocity</td>
<td>DOUBLE</td>
<td>B</td>
<td>R</td>
<td>Actual velocity</td>
</tr>
<tr>
<td>11</td>
<td>CommandedVelocity</td>
<td>DOUBLE</td>
<td>B</td>
<td>R</td>
<td>Commanded velocity</td>
</tr>
<tr>
<td>12</td>
<td>MaxAccelerationSystem</td>
<td>DOUBLE</td>
<td>E</td>
<td>R</td>
<td>Maximal allowed acceleration of the axis in the motion system</td>
</tr>
<tr>
<td>13</td>
<td>MaxAccelerationAppl</td>
<td>DOUBLE</td>
<td>E</td>
<td>R/W</td>
<td>Maximal allowed acceleration of the axis in the application</td>
</tr>
<tr>
<td>14</td>
<td>MaxDecelerationSystem</td>
<td>DOUBLE</td>
<td>E</td>
<td>R</td>
<td>Maximal allowed deceleration of the axis</td>
</tr>
<tr>
<td>15</td>
<td>MaxDecelerationAppl</td>
<td>DOUBLE</td>
<td>E</td>
<td>R/W</td>
<td>Maximal allowed deceleration of the axis</td>
</tr>
<tr>
<td>16</td>
<td>MaxJerkAppl</td>
<td>DOUBLE</td>
<td>E</td>
<td>R/W</td>
<td>Maximal allowed jerk of the axis</td>
</tr>
</tbody>
</table>
The axis states are defined in the amc::motion::control (ctrlNS) namespace. When referencing these constants, they must be prefaced by `amc::motion::control::State::`.

```cpp
gen amc::motion::control::State
{
    MIN_STATES,
    UNINITIALIZED,
    AXIS_DISABLED,
    STAND_STILL,
    HOMING,
    ERROR_STOP,
    STOPPING,
    DISCRETE_MOTION,
    CONTINUOUS_MOTION,
    SYNCHRONIZED_MOTION,
    MAX_STATES
};
```
Appendix E - Drive Setup Methods

**Current Loop**
Methods:
- void SetTorqueProportionalGain(double p_dGain)
  double GetTorqueProportionalGain() const
- void SetTorqueIntegralGain(double p_dGain)
  double GetTorqueIntegralGain() const
- void SetFluxProportionalGain(double p_dGain)
  double GetFluxProportionalGain() const
- void SetFluxIntegralGain(double p_dGain)
  double GetFluxIntegralGain() const
- void SetFieldWeakeningSpeed(double p_dSpeed)
  double GetFieldWeakeningSpeed() const

**GainSet**
Methods:
- PositionLoop * getPositionLoop()
- VelocityLoop * getVelocityLoop()

**Velocity Loop**
Methods:
- void SetFeedbackDirection(int p_iDirection )
  int GetFeedbackDirection() const
- void SetProportionalGain(double p_dGain )
  double GetProportionalGain() const
- void SetIntegralGain(double p_dGain )
  double GetIntegralGain() const
- void SetDerivativeGain(double p_dGain )
  double GetDerivativeGain() const
- void SetAccelFeedForward(double p_dFeedForward )
  double GetAccelFeedForward() const
- void SetValidFilterCoefficient(bool p_bValid )
  bool HasValidFilterCoefficient() const
- void SetFilterCoefficient(double p_dCoefficient )
  double GetFilterCoefficient() const
- void SetLowSpeedSmoothing(double p_dFactor )
  double GetLowSpeedSmoothing() const
- void ActivateIntegratorDecay(bool p_bActivate )
  bool HasIntegratorDecay()
- void SetIntegratorDecayRate(double p_dRate )
  double GetIntegratorDecayRate() const
### Appendix E – Drive Setup Methods

#### Position Loop Methods:
- `void SetFeedbackDirection(int p_iDirection)`
- `int GetFeedbackDirection()` const
- `void SetProportionalGain(double p_dGain)`
- `double GetProportionalGain()` const
- `void SetIntegralGain(double p_dGain)`
- `double GetIntegralGain()` const
- `void SetDerivativeGain(double p_dGain)`
- `double GetDerivativeGain()` const
- `void SetVelFeedForward(double p_dFeedForward)`
- `double GetVelFeedForward()` const
- `void SetAccelFeedForward(double p_dFeedForward)`
- `double GetAccelFeedForward()` const
- `void ActivateIntegratorDecay(bool p_bActivate)`
- `bool HasIntegratorDecay()`
- `void SetIntegratorDecayRate(double p_dRate)`
- `double GetIntegratorDecayRate()` const
- `void SetStopDeceleration(double p_dDecel)`
- `double GetStopDeceleration()` const

#### Current Limits Methods:
- `void SetLimitMethod(LimitMethod p_eLimitMethod)`
- `LimitMethod GetLimitMethod()`
- `void SetContCurrentThreshold(double p_dContCurrentThreshold)`
- `double GetContCurrentThreshold()`
- `void SetPeakCurrent(double p_dPeakCurrent)`
- `double GetPeakCurrent()`
- `void SetPeakCurrentTime(double p_dPeakCurrentTime)`
- `double GetPeakCurrentTime()`
- `void SetContFoldbackTime(double p_dContFoldbackTime)`
- `double GetContFoldbackTime()`

#### Velocity Limits
Note that the parameters for Velocity Limits have a default unit of counts/sec. See “User Units” on page 15 for conversion to user units.

Methods:
- `void SetMotorSpeedLimit(double p_dVelocity)`
- `double GetMotorSpeedLimit()` const
- `void SetNegativeLimit(double p_dVelocity)`
- `double GetNegativeLimit()` const
- `void SetPositiveLimit(double p_dVelocity)`
- `double GetPositiveLimit()` const
- `void SetZeroSpeedWindow(double p_dWindow)`
- `double GetZeroSpeedWindow()` const
- `void SetAtSpeedWindow(double p_dWindow)`
- `double GetAtSpeedWindow()` const
- `void SetFollowErrorWindow(double p_dWindow)`
- `double GetFollowErrorWindow()` const
Position Limits  Note that the parameters for Position Limits specified as type `double` have a default unit of counts. See "User Units" on page 15 for conversion to user units.

Methods:

```cpp
void SetEnableLimits(bool p_bEnable);
bool GetEnableLimits() const;
void SetFollowErrorWindow(double p_dWindow);
double GetFollowErrorWindow() const;
void SetInPositionWindow(double p_dWindow);
double GetInPositionWindow() const;
void SetInHomePositionWindow(double p_dWindow);
double GetInHomePositionWindow() const;
void SetMaxMeasuredLimit(double p_dPosition);
double GetMaxMeasuredLimit() const;
void SetMinMeasuredLimit(double p_dPosition);
double GetMinMeasuredLimit() const;
void SetMaxTargetLimit(double p_dPosition);
double GetMaxTargetLimit() const;
void SetMinTargetLimit(double p_dPosition);
double GetMinTargetLimit() const;
//Only Valid with Valid Homing Routing
void SetHomePosition(double p_dHome);
double GetHomePosition() const;
void SetPresetPosition(double p_dPreset);
double GetPresetPosition() const;
void ActivateIntegratorDecay(bool p_bActivate);
bool HasIntegratorDecayWindow() const;
void SetIntegratorDecayWindow(double p_dWindow);
double GetIntegratorDecayWindow() const;
```
The axis status is returned as a bit mask.

\[
\text{std::bitset<MAX_EVENT_ITEM> EventMask}
\]

This is defined in the `amc::motion::drive::model::status` namespace (statNS). Individual bits can be tested:

```cpp
statNS::Event::ctrlNS::EventMask myStatus;
myX.ReadEvents( myStatus, ctrlNS::EventFilter::History );
if (myStatus.test(statNS::MOTOR_OVER_TEMP))
{
    // Turn on the chiller!!!
}
```

```cpp
enum statNS::EventItem
{
    MIN_EVENT_ITEM = 0,
    LOG_ENTRY_MISSED, // STARTS DRIVE STATUS 1
    SOFTWARE_DISABLE,
    USER_DISABLE,
    USER_POSITIVE_LIMIT,
    USER_NEGATIVE_LIMIT,
    CURRENT_LIMIT_ACTIVE,
    CONTINUOUS_CURRENT_FOLDBACK,
    CURRENT_LOOP_SATURATED,
    SOFTWARE_UNDER_VOLTAGE,
    SOFTWARE_OVER_VOLTAGE,
    NON_SINUSOIDAL_COMMUTATION,
    PHASE_DETECT_ACTIVE,
    MOTION_ENGINE_ACTIVE, // WAS COMMANDED_DYNAMIC_BRAKE
    USER_AUX_DISABLE,
    SHUNT_REGULATOR_ACTIVE,
    PHASE_DETECT_COMPLETE,
    ZERO_VELOCITY, // STARTS DRIVE STATUS 2
    AT_COMMAND,
    VELOCITY_FOLLOWING_ERROR,
    POSITIVE_VELOCITY_LIMIT,
    NEGATIVE_VELOCITY_LIMIT,
    COMMAND_LIMITER_ACTIVE,
    IN_HOME_POSITION,
    POSITION_FOLLOWING_ERROR,
    MAX_TARGET_POSITION_LIMIT,
    MIN_TARGET_POSITION_LIMIT,
    SET_POSITION,
    LOAD_TARGET,
    HOME_ACTIVE,
    SAFE_TORQUE_OFF,
    HOME_COMPLETE,
    RESERVED_DS2_15,
    PVT_BUFFER_FULL, // STARTS DRIVE STATUS 3
    PVT_BUFFER_EMPTY,
    PVT_BUFFER_THRESHOLD,
    PVT_BUFFER_FAILURE,
    PVT_BUFFER_EMPTY_STOP,
    PVT_SEQUENCE_NUMBER_ERROR,
    SOFTWARE_STOP,
}
Appendix E – Event Enumerations

USER_STOP,
CAPTURE_A,
CAPTURE_B,
CAPTURE_C,
SOFTWARE_POSITIVE_LIMIT,
SOFTWARE_NEGATIVE_LIMIT,
RESERVED_DS3_13,
RESERVED_DS3_14,
RESERVED_DS3_15,
PARAMETER_RESTORE_ERROR, // STARTS SYSTEM PROTECTION
PARAMETER_STORE_ERROR,
NULL_STATE_ERROR,
PHASE_SYNC_ERROR,
MOTOR_OVER_TEMP,
MOTOR_DETECT_ERROR,
MOTOR_OVER_SPEED,
MAX_MEASURED_POSITION_LIMIT,
MIN_MEASURED_POSITION_LIMIT,
COMM_CHANNEL_ERROR,
PWM_DIR_BROKEN_WIRE,
MOTION_ENGINE_FAULT,
MOTION_ENGINE_INTERRUPT,
RESERVED_SP_14,
RESERVED_SP_15,
DRIVE_RESET, // STARTS DRIVE PROTECTION
DRIVE_INTERNAL_ERROR,
SHORT_CIRCUIT,
OVER_CURRENT,
UNDER_VOLTAGE,
OVER_VOLTAGE,
DRIVE_OVER_TEMP,
RESERVED_DP_7,
RESERVED_DP_8,
RESERVED_DP_9,
RESERVED_DP_10,
RESERVED_DP_11,
RESERVED_DP_12,
RESERVED_DP_13,
RESERVED_DP_14,
RESERVED_DP_15,
BRIDGE_ACTIVE, // STARTS ACTION STATUS
DYNAMIC_BRAKE_ACTIVE,
STOP_ACTIVE,
POSITIVE_STOP_ACTIVE,
NEGATIVE_STOP_ACTIVE,
POSITIVE_TORQUE_INHIBIT_ACTIVE,
NEGATIVE_TORQUE_INHIBIT_ACTIVE,
BRAKE_ACTIVE,
RESERVED_BR_8,
RESERVED_BR_9,
RESERVED_BR_10,
RESERVED_BR_11,
RESERVED_BR_12,
RESERVED_BR_13,
RESERVED_BR_14,
RESERVED_BR_15,
REFERENCE_FRAME_VALID, // STARTS ACTIVE ALGORITHM STATUS
POSITIVE_STOP_HOLD,
NEGATIVE_STOP_HOLD,
JOGGING_ACTIVE,
RESERVED_AA_4,
RESERVED_AA_5,
RESERVED_AA_6,
RESERVED_AA_7,
RESERVED_AA_8,
RESERVED_AA_9,
RESERVED_AA_10,
RESERVED_AA_11,
RESERVED_AA_12,
RESERVED_AA_13,
RESERVED_AA_14,
RESERVED_AA_15,
ME_EXECUTE, // STARTS MOTION ENGINE STATUS
ME_BUSY,
ME_ACTIVE,
Appendix E – Event Enumerations

ME_MOTION_COMPLETE,
ME_SEQUENCE_COMPLETE,
ME_DONE,
ME_ABORTED,
ME_ERROR,
RESERVED_ME_8,
RESERVED_ME_9,
RESERVED_ME_10,
RESERVED_ME_11,
RESERVED_ME_12,
RESERVED_ME_13,
RESERVED_ME_14,
RESERVED_ME_15,
MAX_EVENT_ITEM
};
ADVANCED Motion Controls’ DriveLibrary seeks to present a simple programming interface to our customer’s developers, with a goal of minimizing the learning curve for development of applications that will control simple machines.

For simple directed motion, the API is flat (only one class exposed at the interface). In consequence, one challenge for end-user developers is to interpret and react to information that reflects diverse sources for errors. This document describes the DriveLibrary approach to this problem.

Any specific reaction to error conditions must be under the control of the library user. DriveLibrary can be configured to automatically handle error conditions, but the customer development team may ignore this capability and handle exceptions themselves.

G.1 Scenarios

DriveLibrary funnels into customer code exception conditions arising from several sources. Each source has independent mechanisms for managing errors.

G.1.1 Firmware

The firmware itself also has error stop capabilities. This starts with a signal from the hardware that something has gone wrong. Among those signals are:

- Motion limits (position, velocity, torque, etc.)
- Operating limits (temperature)
- Programming errors (no more data points, etc.)

These appear to DriveLibrary as bits on a status register.

When these conditions occur, the firmware can, per DriveWare configuration:

- Report the error
- Initiate a hard stop
- Disable the bridge
- Impose positive or negative motion limits
When these events occur, DriveLibrary needs to coordinate the axis state with the condition of the drive. Since the library does not continuously poll drive status, the firmware must be configured to latch its status bits. History bits can only be reset through the command interface.

### G.1.2 Internal Errors

Internal errors arise through several paths. Most important are:

- **Method parameterization errors** - These require an immediate response.
- **Sequencing dependencies (e.g. – state machine)** - The drive may not be ready for an operation.
- **Control faults** - The drive does not behave as programmed. The assumptions of the customer’s program will be invalid.

### G.1.3 Fault Handling

DriveLibrary provides a basic framework for fault handling. When the drive is not responding (e.g. – due to connection loss), the **axis disabled** state prevents the user from initiating any control of the drive until communication is re-established.

The state machine includes an **error stop** state. In this state, some mechanism must be used to bring the machine to a safe stop. Until that occurs, all DriveLibrary motion control functions are disabled. A Reset method is provided to exit from the **error stop** state into **stand still**, from which normal control functions can be resumed.

How the drive is brought to a “safe” stop is specific to the operation. If the drive is unloaded, a hard stop may be safe. If the drive powers a machine that is loaded with top-heavy canisters, a fairly gentle stop may be required. It is reasonable to assume that it would be safe to use the control parameters that described the motion in progress.

One exception to this strategy is when the firmware turns off the bridge power. In that case, the specification requires a transition to **error stop**. When power is off, no safe stop is possible.

### G.2 Strategy

With two exceptions, by default neither the firmware nor DriveLibrary takes any action when errors occur. All error handling is under customer control.

### G.2.1 Error Reporting

DriveLibrary recognizes two kinds of error conditions.

**Control Errors**  Conditions that are configured to trigger **error stop** are recognized as **control errors**. When these occur, errors are not reported when returning to customer code, because
the library has successfully handled the event. Note that landing in \texttt{ERROR\_STOP} disables motion control until the customer program takes action to correct the condition.

Control errors are reported to the user program through:

- \texttt{singleNS::SingleAxis::NodeId} identifies the drive that reported the error
- \texttt{ctrlNS::Axis::Error, true when in ERROR\_STOP}
- \texttt{ctrlNS::Axis::GetState, which will report ERROR\_STOP}
- \texttt{ctrlNS::Axis::GetStoppingErrors provides the list of active logic errors.}
- \texttt{ctrlNS::Axis::ErrorText supplies a description of the error}

The error ID and text will be held until the axis is reset (using \texttt{singleNS::Reset}).

\textbf{Logic Errors} When the library is unable to process a request, it is reported as a \textit{logic error}. In this case:

- The function will return \texttt{false}, indicating an error.
- \texttt{singleNS::SingleAxis::NodeId} identifies the drive that reported the error.
- \texttt{ctrlNS::Axis::GetLogicError} describes the problems that resulted in the failure of the request.

Both the logic error ID and text are cleared on every motion control method.

\textbf{G.2.2 Drive Initialization}

When an axis is established, the library puts it in the \texttt{AXIS\_DISABLED} state. To get the drive into \texttt{STAND\_STILL} state, the drive must be enabled with a call to \texttt{ctrlNS::Axis::Power}.

When we lose communication with the drive, control over the drive is no greater than before the axis was created. To ensure consistent programming, the library forces the axis into \texttt{DISABLED} when communication is lost.

When the customer code attempts to change the power state (through \texttt{ctrlNS::Axis::Power}), the library reads the bridge state to confirm the desired state was achieved. If the power failed to turn on, the axis moves into \texttt{ERROR\_STOP}. If the power was on and did not turn off, then DriveLibrary tries to stop motion.

Power may be commanded on while error conditions exist. The obvious case is if power is commanded off while in \texttt{ERROR\_STOP}. This forces the axis into \texttt{DISABLED}, which allows the customer program to command power on. If either an error stop status or control error is found, the axis transitions to \texttt{ERROR\_STOP} rather than \texttt{STAND\_STILL}.

If the library detects a loss of power on the bridge (due to local command by the firmware), it forces the axis into \texttt{ERROR\_STOP}, but does not attempt to stop motion.

\textbf{G.2.3 Error Stop Configuration}

Support for \texttt{ERROR\_STOP} ensures that a fall-back is always available when an unanticipated condition occurs. Further control operations are disabled.
The developer is able to specify both when and how an ERROR_STOP will be executed.

**Condition Selection** Exception conditions are indicated as:

- bits in the drive status mask
- errors diagnosed and reported directly by the library

To select from among these indicators, ctrlNS::Axis::ActivateErrors takes two parameters:

- ctrlNS::EventMask is the usual return result for drive status, but in this case is used to select bits that will trigger an ERROR_STOP. The bits are defined in the statNS::Event::EventItem enumeration.
- ctrlNS::AxisErrorList is a std::vector of ctrlNS::Error. Any error code in Error (see "Appendix B – Error Identifiers" on page 45) can be added to the list, but the following codes will be ignored:
  - ILLEGAL_ACTION
  - NO_AXIS_ERRORS
  - AXIS_HAS_ERROR_STATUS
  - AXIS_FAULTED
  - ERROR_STOP_IN_PROGRESS

The selections are cleared by passing empty conditions.

**Stop Types** Through ctrlNS::Axis::SelectErrorStop, the application can dynamically choose the type of error stop to implement. There are three settings.

- **Hard Stop**: This will command deceleration to zero speed in 0.1 seconds. The actual stop time will depend upon:
  - Frequency of polling by the motion control application (see "Appendix A - System Config Files" on page 43).
  - Communications lag in the interaction with the drive.
  - Current limits in the drive itself, which will ultimately control how fast the energy of motion is dissipated.
- **End Move**: The axis will be commanded to zero speed as though it had successfully completed the active move.
- **As Configured**: The stop parameters will be taken from those provided through configuration.

**Stop Parameters** As mentioned above, no single set of parameters is sufficient to ensure a "safe" stop. Through ctrlNS::Axis::ConfigureErrorStop, the client code can dynamically configure the stop process. The parameters are:

- Deceleration, \( d \)
- Jerk, \( j \)
- Time limit for stop, \( t \), defaulting to 0.1 seconds (effectively a hard stop)

These are cached in the library for reference during an ERROR_STOP.

Given the current velocity \( v \), the actual stop parameters are determined as follows:
Derive the deceleration:
- To hard stop, take the deceleration as ten times the current velocity plus 50 times the current acceleration.
- If the configured stop is selected, use \( d \). If the total deceleration time \( \frac{v}{d} \) is greater than \( t \), set the deceleration to \( \frac{v}{t} \).
- If stop from move is selected, use the deceleration from the move under way.

Use the jerk that goes with the chosen deceleration. If not found, use ten times the deceleration.

From this, it should be clear that the time limit is only a guide. Attempting to do better could lead to unexpected results, depending upon the current acceleration/deceleration and jerk.

**Trigger Logic** Again, ERROR_STOP can be triggered either by a firmware status bit, or an internal error condition. Except when commanding power on, an axis cannot enter ERROR_STOP from AXIS_DISABLED, and new triggers are ignored after ERROR_STOP is entered.

Because the event latching logic lifecycle is extended, the status trigger implementation is relatively complex. The library reads the drive status at the interval specified in the system setup (see “Appendix A- System Config Files” on page 43). If an error bit is detected, DriveLibrary immediately triggers ERROR_STOP.

When an error is generated, it is compared to the selection list entries just before exiting from the library. If a match is found, an error stop is triggered.

**G.2.4 ERROR STOP Recovery**

Once ERROR_STOP is entered, ctrlNS::Axis::Reset is the only exit mechanism. However, invocations are rejected until motion is brought to an end. When the reset occurs, the control error and status bits are also cleared.

While waiting for motion to end, ctrlNS::Axis::Busy is false, while ctrlNS::Error is true. If the Reset fails because the drive is not yet stopped, the call will report ERROR_STOP_IN_PROGRESS.

**G.2.5 Error Entry and Error Report**

ErrorEntry and ErrorReport are type declarations defined in the ctrlNS namespace used with Get Stopping Errors and Get Logic Errors. ErrorEntry is used for error report entry.

```c++
struct ErrorEntry{
    U_INT uisquence;
    INT iCode;
    std::string sDetail;
};
```

ErrorReport is a list of error reports.

```c++
typedef std::list<ErrorEntry> ErrorReport;
```