Instruction Manual for Motion Server Products & Binary Command Interpreter

March, 2001

Douloi Automation, Inc.
3517 Ryder Street
Santa Clara, CA 95051-0714

Voice (408) 735-6942
Fax (408) 735-6946
EMail info@douloi.com
Site: www.douloi.com

douloi
AUTOMATION

Table of Contents

Chapter 1

1) Introduction

Welcome!

Welcome to Motion Server and Douloi Automation's Motion Control software components, tools to simplify and accelerate the creation of motion control applications.

Douloi Automation wants to encourage your project's success. Free technical support is available to answer your questions, assist you through trouble-spots in product use, and to recommend strategies and approaches for solving different aspects of a motion control problem. Sample code, application prototypes, and application notes can be provided to response to specific questions you may have. We would much rather have you call and get answers than to be frustrated or slowed in your automation project. Please feel free to contact us at:

- voice (408) 735-6942
- fax (408) 735-6946
- EMail info@douloi.com

Objective of Document

This document provides information on the use of the Binary Command Interpreter for directing Motion Server controllers. Binary commands are supported through different communication methods including computer backplane (for in-computer format controllers) or over networks such as ethernet (for stand-alone format controllers). Controller checkout and system analysis is performed through Douloi's Servo Application Workbench software, a Windows application. Although Windows is not necessary for the use of the Binary Command Interpreter, application development and testing can be accelerated when Douloi's Windows tools can be brought to bear on machine setup and diagnostics. Saw can also be used to place into controllers application specific programs. For instructions on setting up the controller please consult the setup chapter in the Instruction Manual for Motion Server and Servo Application Workbench.

Motion Server Specs (DMS in PC)

Motion System

- 128 MHz 32 bit processor with on-chip cache memory
- 4, 8, 12, or 16 axes per system
- Servo or Stepper on per-axis basis
- Multiple independent axis groups
- Trapezoidal and S-Curve profiling
- Custom profiling at application level
- 32 bit position management
- Sample rates from 1 to 4 kHz
- Linear, circular, curve interpolation
- Electronic gearing with phase adjust
- Electronic camming
- Tangent servo
- Master/slave coordination
- High speed registration
- Kinematics
- Motion superposition
- Coordination tailoring

Servo Specifications

- 486 class processor
- On-board real-time operating system supporting 12 seperate activities as well as motion control
- 4 to 16 axis of coordinated motion
- Mixed servo and stepper motor control
- 32 bit position resolution
- 48 general purpose configurable I/O
- 1 Capture signal per axis
- User Disable signal
- 2 amp enable signals per axis, one active high, the other active low
- watchdog safety system

Servo Capabilities

When configured to run a servo motor the hardware provides

- 4 MHz quadrature inputs with 3 bit filters for 4 axis, 1 MHz quadrature rate for 16 axis
- high speed position capture
- high speed position compare
- +/- 10 volt command signal with 12 bit resolution

Stepper Capabilities

When configured to run a stepper motor the hardware provides

- 2 Mhz step rate for 4 axis, 500 kHz step rate for 16 axis
- configurable step pulse polarity

Motion Server Block Specs (Standalone)

Motion System

- 128 MHz 32 bit processor with on-chip cache memory
- 2 to 10 axes per system
- Servo or Stepper in pairs configured at factory
- Multiple independent axis groups
- Trapezoidal and S-Curve profiling
- Custom profiling at application level
- 32 bit position management
- Sample rates from 1 to 8 kHz
- Linear, circular, curve interpolation
- Electronic gearing with phase adjust
- Electronic camming
- Tangent servo
- Master/slave coordination
- High speed registration
- Kinematics
- Motion superposition
- Coordination tailoring

Servo Specifications

- 486 class processor
- On-board real-time operating system supporting 12 seperate activities as well as motion control
- 2 to 10 axis of coordinated motion
- Mixed servo and stepper motor control
- 32 bit position resolution
- 18 Isolated inputs, 24 volts
- 8 Isolated outputs, 24 volts
- 48 general purpose configurable I/O
- 4 Capture signals per block
- EStop signal
- dual watchdog safety system

Servo Capabilities

When configured to run a servo motor the hardware provides

- 2 MHz quadrature inputs with 3 bit filters for 4 axis, 1 MHz quadrature rate for 10 axis
- high speed position capture
- high speed position compare
- +/- 10 volt command signal with 12 bit resolution

Stepper Capabilities

When configured to run a stepper motor the hardware provides

- 2 Mhz step rate for 4 axis, 1 Mhz step rate for 6 axes
- Step and Direction presented as differentially driven pairs

Common Specs (DMS and MSB)

Timer Event

Motion Server provides motion control functions by responding to a timer which occurs generally at 1 kHz although the frequency is programmable. This timer event performs three major functions.

The first function is control law execution. Servo control is accomplished with the familiar zero, pole, integrator filter used in many motion control systems. This digital control law operates at a 1 kHz sample rate providing comfortable closed loop system frequencies of 100 Hz and below. Stepper motor control is accomplished by updating pulse generating electronics at a frequency of of 1 kHz providing continuous velocity control of stepper motors.

The second function of the timer event is motion profiling. Motion Server is able to profile motion for up to 16 physical axes. Motion Server Block is able to coordinate up to 10 axes. These axes can be combined in different arrangements to form various coordinated multi-axis groups. Any particular axis group can perform coordinated motion along an arbitrary path. Multiple axis groups can perform motion concurrently and independently. The motion profiler uses a dynamic profiling technique which permits changing profile parameters on the fly including acceleration, deceleration, slew speed, and in some cases destination and motion type. This permits motion mode "splicing" without stopping. For example a positioning move can be changed to a jog at a new speed on the fly.

The third timer event function is multitasking. Multiple user-written motion application programs may be resident in Motion Server. The timer event contains a multitasker which activates and manages the operation of these programs.

Multiple Motion Application Threads

As many as 12 separate motion application "threads" or programs (which are distinct from motion profiles) can be running concurrently and independently at any particular time. These programs are written in Douloi Pascal, a dialect of Object Pascal. Programs can communicate to each other through

shared data structures. They can also access the motion control system, communicate with Windows applications created by the Servo Application Workbench, and to the disk file system if SAW is present.

Microsoft Windows

Microsoft Windows serves as the most common development and target environment for motion control applications using Douloi products. The familiar interface aids both developers and users of the resulting applications reducing the developers learning curve and the operators training time. Motion Server can be used with other operating systems through various communication methods available. Applications programs for downloading into Motion Server are prepared with Servo Application Workbench under Microsoft Windows. Once downloaded and retained in on-board FLASH memory, these functions can be invoked from other communication methods.

Servo Application Workbench

Servo Application Workbench (SAW) is a Windows application which greatly simplifies the creation of multithreading motion application programs and operator control elements to direct them. Applications may contain conventional Windows controls such as buttons and text items as well as more specialized controls such as components available in the on-line software catalog.

Inside Servo Application Workbench is a high level language compiler. The compiler changes the descriptions of the motion applications into native 32 bit 486 object code which executes on Motion Server very quickly. The compiler "knows" about the motion system, the multithreading system, and Windows. This permits the application developer to access different system resources in a consistent way without having to worry about how these resources are being provided.

Servo Application Workbench allows the developer to construct motion applications in a "clip art" fashion by pasting pre-fabricated parts and assemblies into the application. After "screen painting" the application and filling in the program's behavior Servo Application Workbench compiles the motion application programs and creates the associated Windows application to operate them. This ability to create new real-time behavior and download into Motion Server is constrained to the Windows environment because the language compiler is a Windows DLL. However, new motion controller capabilities (beyond the standard command set) can be created in SAW, downloaded into Motion Server, and remembered in "flash" memory for use under another operating system.

Binary Command Interpreter

Description

The Binary Command Intepreter is a command set that can be sent to Motion Server Controllers from a host. Commands might be coming from a Windows language such as Visual Basic, an Soft PLC software program such as Think-and-Do software, or from some other host device. BCI commands can be used to access standard controller functions such as independent and coordinated axis movement. BCI commands can also be used to control onboard application specific programs that have been written and downloaded with Servo Applicaion Workbench software.

Methods of Use

This manual describes Binary Commands use through several methods. One method is a Dynamic Link Library (32 bit Windows DLL). Another method of use is through an ActiveX control. Currently The DLL supports the DMS incomputer controllers and the ActiveX control supports Motion Server Block controllers through an Ethernet Modbus TCP/IP connection (Modicon Ethernet). However the means of communication and the controllers being targeted are subject to change while maintaining the communication interface described.

Chapter 2

2) Binary Communication Protocol

Purpose

The following chapter describes low-level details required to write a software interface to Motion Server through the ISA bus (DMS format). These details are only required if you are using a language Douloi does not provide a driver for, or if you have a particular interest in how the driver operates. These details are concealed within the commands described in chapter 3.

Board Addressing

Motion Server communicates through the ISA bus I/O space. The base address of the board is $330 hex and occupies 32 consecutive bytes. This resides in what is commonly used for network adaptors, but is not a typical default address for a network adaptor. For binary communication the board is managed exclusively as a 16 bit resource. Only 16 bit, word sized transfers are performed to and from Motion Server.

Communication Model

The following section describes different aspects of Motion Server's communication system, but not the sequence of use. The examples section shows the particular sequence of commands required to achieve communication.

The following paragraphs describe the general interface that should be followed to insure compatibility with future software versions. However the full functionality described is not fully realized in the current shipping version. Differences are noted in italics.

Motion Server communicates to multiple client programs through independent I/O space FIFOs (hence the name Motion Server). The most typical case for multiple clients is operating diagnostic instruments (i.e. software storage scopes and I/O monitors) while the machine control application is running.

Software wishing to communicate with Motion Server must be able to discover "what phone line is available" so as to not break in to an already present conversation. Channels are allocated and released through channel management registers.

FIFOs have 16 bit data registers for reading and writing information. They also have control and status registers. The FIFO is implemented as a single memory array with a single address control which is used for both outgoing and incoming information. Motion Server operates in a reactive, rather than proactive, manner. The host computer resets the FIFO, writes an instruction into the FIFO, and then sets a flag indicating that data is ready. Unlike conventional hardware FIFOs that indicate data is present as soon as one write has been performed, Motion Server allows the host to manipulate the FIFO before Motion Server is informed that data is present. This simplifies communication and prevents Motion Server from expecting information before it has been completely submitted by the host.

The host can reset the FIFO, write an instruction, reset the FIFO again, and read what has just been written before submitting the information to Motion Server.

All of the channels in Motion Server perform in the same manner and have the same bit layout.

As well as multple clients at one time, Motion Server supports multiple communication techniques at one time. Techniques available include:

- System Binary
- User Binary
- ASCII

Advanced users can create their own dialects with Servo Application Workbench if required. A new channel starts up in ASCII. The host program can then select the command style by using the SelectProtocol command. Communication from that point on conforms to the chosen protocol.

At this time, channel allocation is static, not dynamic. Channel 1 is fixed to operate as SystemBinary and is used by SAW and SERVOLIB driven applications. Channel 2 is fixed to operate as the Binary Command Interpreter channel. ASCII interpretation is not provided in the current release. Channels start up in their fixed protocols.

Register Map

The following registers are present in the Motion Server I/O footprint:

Address	Read	Write
BaseAddress	Channel Request	Channel Control
BaseAddress+2	Reserved	Reserved
BaseAddress+4	Channel 1 Data	Channel 1 Data
BaseAddress+6	Channel 1 Status	Channel 1 Control
BaseAddress+8	Channel 2 Data	Channel 2 Data
BaseAddress+10	Channel 2 Status	Channel 2 Control
BaseAddress+12	Channel 3 Data	Channel 3 Data
BaseAddress+14	Channel 3 Status	Channel 3 Control

Register Descriptions

Channel Request (read)

The value read from the Channel Request register is an offset, with respect to the board's base address, of the next available communication channel. The value read is the literal binary offset number, as a 16 bit word. After being read, the Channel Request register "increments" to the next available channel offset. Multiple reads of Channel Request produce different values and will eventually yield the value "0" indicating no more channels available.

Channel Request produces its offset through hardware in the communication chip on Motion Server, not through software behavior in the on-board processor. Channel Requests are atomic, intrinsically exclusive I/O space read operations insuring that no two client programs can "race" each other and come up with the same channel by somehow "tieing".

When a program is finished with Motion Server it needs to release the channel, or "hang up the phone" so that the channel is available to some other program which may need it.

This register is currently inactive as channels are statically allocated. A read from this location is currently undefined.

Channel Control (write)

Normally channels are released by a command in whatever communication method has been chosen. The Channel Control register is used to "clear the lines" in the event that this has not taken place. The data value written to Channel Control is not used, only the write event is necessary to initialize channel allocation. If programs are written correctly this register should not need to be used.

This register is currently inactive.

Channel Status (read)

Channel status provides a single bit of information, bit 0, which indicates that data is available to be read from the FIFO. A "high" or "1" level indicates that data is available. A "low" or "0" value indicates that no data is available. This bit is reset to "0" by the first read performed by the host although additional data may be available to read. The amount of data is based on message sizes of the command being performed, not the state of this bit while reading information from the FIFO.

Channel Control (write)

Channel control involves the following 2 bits:

- bit 0 - Reset Address
- bit 1 - Send Message

Bits are used by setting the value to "1". A value of "0" has no effect. Only one bit should be used at a time, so legal values include 1 and 2.

Bit 0, Reset Address, is used to reset the address of the FIFO to the beginning of the FIFO memory. Subsequent writes to the FIFO data register then places information in the FIFO with the FIFO memory automatically incrementing. Reset Address is not destructive. The FIFO can be written and the address reset prior to sending the message.

Send Message is used to tell Motion Server that the command has been prepared and is available for reading. Once Send Message has been set the host program MUST NOT manipulate the FIFO again until the status bit indicate that information is available to be read.

Fifo Hardware Abstraction

Given the structure of the hardware and data types used in the binary protocol, the following hardware abstraction is recommended for manipulating the FIFOs. The abstraction is presented in Pascal and provided as source code in both C and Pascal under the name BIN_CMND.CPP and BIN_CMND.PAS

Pascal has more native strict data types than C such as a native boolean type. The drivers support these various types with functions and procedures that reflect the types being handled. In C, these type distinctions disappear and the code bodys of some routines degenerate to the same content but are retained in the driver for consistency.

In this dialect of pascal, the I/O space of the computer is modelled as a byte-ordinal array which performs 16 bit transfers named PortW.

AllocateChannel

This command retreives the offset of the next available channel for use by the host.

Pascal Implementation

```
var FifoAddress:word;  {persistent variable}

function AllocateChannel:boolean;
   var OffsetValue:integer;
   begin
   OffsetValue:=PortW($330);
   if OffsetValue > 0 then
      begin
      FifoAddress:=PortW($330)+$330;
      AllocateChannel:=true;
      end
   else
      AllocateChannel:=false;
   end;
```

Current Pascal Implementation

```
var FifoAddress:word;  {persistent variable}

function AllocateChannel:boolean;
   begin
   FifoAddress:=$338; {static allocation}
   AllocateChannel:=true; {stub behavior}
   end;
```

FifoReset

This command resets the FIFO's internal address by writing a "1" bit into the FIFO control register located 2 bytes above the channel base address. Remember that physical addresses are in "byte space". The adjacent 16 bit word in "word space" is 2 bytes away.

Pascal Implementation

```
procedure FifoReset;
   begin
   PortW[FifoAddress+2]:=1;
   end;
```

FifoWriteWord

This command appends a 16 bit word value to the FIFO contents.

Pascal Implementation

```
procedure FifoWriteWord(Value:word);
  begin
  PortW[FifoAddress]:=Value;
  end;
```

FifoWriteInteger

This command does the same operation as FifoWriteWord with the convenience of a typecast.

Pascal Implementation

```
procedure FifoWriteInteger(Value:integer);
  begin
  PortW[FifoAddress]:=Value;
  end;
```

FifoWriteBoolean

Booleans are represented as 16 bit values that are either 0, representing false, or all 1s, ($FFFF) representing true. Other values should be regarded as illegal values.

Pascal Implementation

```
procedure FifoWriteBoolean(ParamIsTrue:boolean);
  begin
  if ParamIsTrue then
    PortW[FifoAddress]:=$FFFF
  else
    PortW[FifoAddress]:=0;
  end;
```

FifoWriteLongint

Following the Intel heritage of "little endian", the low word is submitted to the FIFO first followed by the high word. This presumes the functions LowWord and HiWord such as found in the Windows API, or the use of a record "overlay" which provides access to the internal structure of the longint data.

Pascal Implementation

```
Type TLongintToInteger=record
  LowInteger:integer;
  HighInteger:integer;
  end;

procedure FifoWriteLongint(Value:longint);
  begin
  PortW[FifoAddress]:=
      TLongintToInteger(Value).LowInteger;
  PortW[FifoAddress]:=
      TLongintToInteger(Value).HighInteger;
  end;
```

FifoWriteSingle

Singles are 32 bit floating point values. These are sent little end first also.

Pascal Implementation

```
procedure FifoWriteSingle(Value:single);
   begin
   PortW[FifoAddress]:=
      TLongintToInteger(longint(Value)).LowInteger;
   PortW[FifoAddress]:=
      TLongintToInteger(
         longint(Value)).HighInteger;
  end;
```

FifoSendMessageAndWaitForResponse;

The simplest way to communicate to MotionServer is to prepare a message, send the message, and wait for the answer. This procedure performs these steps. Based on exception handling capabilities of your language system it may be advisable to place a timeout in the loop waiting for the controller's response. Note that some commands can take as long as a physical movement (i.e. T1AxisMoveBy). Timeout durations would need to take this into account.

Pascal Implementation

```
procedure FifoSendMessageAndWaitForResponse;

   var UnusedWord:word;
   var UserNumber:integer;

   begin
   {clear possibly pending response}
   UnusedWord:=PortW[FifoAddress];
   {Set message send flag}
   PortW[FifoAddress+2]:=2;
   repeat
      {do nothing except possibly manage a timeout}
   until (PortW[FifoAddress+2] and 1) = 1;

   {reset fifo so as to read from start}
   PortW[FifoAddress+2]:=1;
   end;
```

FifoReadWord

This function assumes that the host has identified information available to read in the FIFO and returns the next 16 bits as a word.

Pascal Implementation

```
function FifoReadWord:integer;
   begin
   FifoReadWord:=Portw[FifoAddress];
   end;
```

FifoReadInteger

This function assumes that the host has identified information available to read in the FIFO and returns the next 16 bits as an integer.

Pascal Implementation

```
function FifoReadInteger:integer;
   begin
   FifoReadInteger:=Portw[FifoAddress];
   end;
```

FifoReadBoolean

This function assumes that the host has identified information available to read in the FIFO and interprets the next 16 bits as boolean.

Pascal Implementation

```
function FifoReadBoolean:boolean;
   begin
   if FifoReadWord=0 then
      FifoReadBoolean:=false
   else
      FifoReadBoolean:=true;
   end;
```

FifoReadLongint

This function assumes that the host has identified information available to read in the FIFO and interprets the next 32 bits as longint with the "little end" being read first.

Pascal Implementation

```
function FifoReadLongint:longint;

   var Answer:longint;

   begin
   TLongintToInteger(Answer).LowInteger:=
      FifoReadWord;
   TLongintToInteger(Answer).HighInteger:=
      FifoReadWord;
   FifoReadLongint:=Answer;
   end;
```

General Command Abstraction

Commands are sent in various "patterns" that can be shared across commands. The following section discusses the general structure of sending and receiving messages as well as more specific structures that are used multiple times in the command set.

Transmit Structure

The next level of communication can be built using the hardware fifo abstraction. Messages have the following general format:

- Command ID (16 bit number)
- Command Parameters (various number and type)

When dealing with motor operations the format becomes more specifically the following

- Command ID (16 bit number)
- AxisNumber or GroupNumber (16 bit number)
- Command Parameters (various number and type)

AxisNumbers are in the range 1 to 16 and represent the motors in the system. GroupNumbers are provided by the routines T2AxisInit, T3AxisInit, T4AxisInit, T5AxisInit, and T6AxisInit. These routines take a list of AxisNumbers and return a unique GroupNumber or

"handle" to the collection described. Coordinated motion commands then operate on this defined group.

In many cases, GroupNumbers and AxisNumbers can be interchanged. For example the command SetMotor, which turns a motor on or off, operates on a single motor if the GroupNumber parameters is in the range 1 to 16, or operates on all the motors in a group as a set.

Command Ids are DECLARED in the BINARY.INC include file. Command Ids have the form bc_XXXXX where "bc" stands for binary command, and XXXXX is replaced by the corresponding command name shown in Chapter 3. DO NOT CIRCUMVENT THE CONSTANT INCLUDE FILE AND USE LITERAL COMMAND NUMBERS. Douloi reserves the right to update command-value associations at a future time along with updated include files. Any hard-coded Command Ids will reference the wrong commands.

Response Structure

Responses have the following structure:

- Error Code (16 bit value)
- Additional Response Data (various types)

Error codes are elaborated in the error code listing in the include files. An error code of 0 indicates no problem has occurred and that associated response information is valid. How much response information to gather depends on the command that requested that data. There is no "end of data" information in the response. If the error code is not 0, the associated data is not valid.

dms_Procedure

dms_Procedure is the simplest case. A global "dmsErrorCode" is used to record any problem that may occur during command processing. If an error has occurred, the commands are ignored (through the early exit) rather than operative on a system which has sustained an error response. In this case, there is no parameter information that accompanies the commands.

```
procedure dms_Procedure(CommandNumber:integer);
   begin
   if ErrorCode <> 0 then
      exit;
   FifoReset;
   FifoWriteWord(CommandNumber);
   FifoSendMessageandWaitForResponse;
   ErrorCode:=FifoReadWord;
   end;
```

dms_BooleanFunction

The dms_BooleanFunction represents the simplest case of getting an answer back from Motion Server. In the event that an error has occurred, the routine exits immediately and returns a value of false.

```
function dms_BooleanFunction(
   CommandNumber:word):boolean;

   begin
   if ErrorCode <> 0 then
      begin
      dms_BooleanFunction:=false;
      exit;
      end;
   FifoReset;
   FifoWriteWord(CommandNumber);
   FifoSendMessageandWaitForResponse;
   ErrorCode:=FifoReadWord;
   if ErrorCode= 0 then
      dms_BooleanFunction:=FifoReadBoolean
   else
      dms_BooleanFunction:=false;
   end;
```

dms_AxisProcedure

Most of the routines involve communicating to axes. These routines begin with dms_Axis... to indicate that they include an AxisNumber or a GroupNumber in their parameter list for referring to motors. This axis-indicating parameter is immediately after the command number and is written as a 16 bit value.

```
procedure dms_AxisProcedure(
   CommandNumber:integer; GroupNumber:integer);
   begin
   if ErrorCode <> 0 then
      exit;
   FifoReset;
   FifoWriteWord(CommandNumber);
   FifoWriteWord(GroupNumber);
   FifoSendMessageandWaitForResponse;
   ErrorCode:=FifoReadWord;
   end;
```

dms_AxisProcedureIntegerParam

This routine is used to send to a particular axis or axis group an integer parameter.

```
procedure dms_AxisProcedureIntegerParam(
   CommandNumber:integer;
   GroupNumber:integer;
   Param:Integer);

   begin
   if ErrorCode <> 0 then
      exit;
   FifoReset;
   FifoWriteWord(CommandNumber);
   FifoWriteWord(GroupNumber);
   FifoWriteInteger(Param);
   FifoSendMessageandWaitForResponse;
   ErrorCode:=FifoReadWord;
   end;
```

dms_AxisProcedureLongintParam

The following routines sends messages which involve axis information and a 32 bit longint parameter.

```
procedure dms_AxisProcedureLongintParam(
   CommandNumber:integer;
   GroupNumber:integer;
   Param:longint);

   begin
   if ErrorCode <> 0 then
      exit;
   FifoReset;
   FifoWriteWord(CommandNumber);
   FifoWriteWord(GroupNumber);
   FifoWriteLongint(Param);
   FifoSendMessageandWaitForResponse;
   ErrorCode:=FifoReadWord;
   end;
```

dms_AxisProcedureBooleanParam

Axis often require boolean parameters to turn control functions on and off. This procedure provides a boolean parameter written as a 16 bit value.

```
procedure dms_AxisProcedureBooleanParam(
   CommandNumber:integer;
   GroupNumber:integer;
   Param:Boolean);

   begin
   if ErrorCode <> 0 then
      exit;
   FifoReset;
   FifoWriteWord(CommandNumber);
   FifoWriteWord(GroupNumber);
   FifoWriteBoolean(Param);
   FifoSendMessageandWaitForResponse;
   ErrorCode:=FifoReadWord;
   end;
```

dms_AxisIntegerFunction

The following integer function is used to return 16 bit information from a particular axis, such as the CommandedTorque.

```
function dms_AxisIntegerFunction(
   CommandNumber:integer;
   GroupNumber:integer):longint;

   begin
   if ErrorCode <> 0 then
      begin
      dms_AxisIntegerFunction:=0;
      exit;
      end;
   FifoReset;
   FifoWriteWord(CommandNumber);
   FifoWriteWord(GroupNumber);
   FifoSendMessageandWaitForResponse;
   ErrorCode:=FifoReadWord;
   if ErrorCode=0 then
      dms_AxisIntegerFunction:=FifoReadInteger
   else
      dms_AxisIntegerFunction:=0;
   end;
```

dms_AxisLongintFunction

Many axis functions return 32 bits of information. This routine serves as a common pattern for axis commands such as ActualPosition and ErrorPosition.

```
function dms_AxisLongintFunction(
   CommandNumber:integer;
   GroupNumber:integer):longint;

   begin
   if ErrorCode <> 0 then
      exit;
   FifoReset;
   FifoWriteWord(CommandNumber);
   FifoWriteWord(GroupNumber);
   FifoSendMessageandWaitForResponse;
   ErrorCode:=FifoReadWord;
   if ErrorCode=0 then
      dms_AxisLongintFunction:=FifoReadLongint
   else
      dms_AxisLongintFunction:=0;
   end;
```

dms_AxisBooleanFunction

Boolean answers regarding motor condition are answered through the following function. Commands that make use of this function format include MotorIsOn and MoveIsFinished.

```
function dms_AxisBooleanFunction(
   CommandNumber:integer;
   GroupNumber:integer):boolean;

   begin
   if ErrorCode <> 0 then
      exit;
   FifoReset;
   FifoWriteWord(CommandNumber);
   FifoWriteWord(GroupNumber);
   FifoSendMessageandWaitForResponse;
   ErrorCode:=FifoReadWord;
   if ErrorCode=0 then
      dms_AxisBooleanFunction:=FifoReadBoolean
   else
      dms_AxisBooleanFunction:=false;
   end;
```

dms_T2AxisVectorProcedure

Certain groups of commands involve sending a set of 32 bit parameters to an axis group. Most often this information has to do with coordinated positioning. The following procedure manages a 2 dimensional transfer. After writing the command number and the group number, the parameters are written in order to complete the message.

```
procedure dms_T2AxisVectorProcedure(
   CommandNumber:integer;
   GroupNumber:integer;
   Param1:longint;
   Param2:longint);

  begin
  if ErrorCode <> 0 then
    exit;
  FifoReset;
  FifoWriteWord(CommandNumber);
  FifoWriteWord(GroupNumber);
  FifoWriteLongint(Param1);
  FifoWriteLongint(Param2);
  FifoSendMessageandWaitForResponse;
  ErrorCode:=FifoReadWord;
  end;
```

dms_T3AxisVectorProcedure

This procedure provides 3 longints of parameter information for use with 3 dimensional motion.

```
procedure dms_T3AxisVectorProcedure(
   CommandNumber:integer;
   GroupNumber:integer;
   Param1:longint;
   Param2:longint;
   Param3:longint);

   begin
   if ErrorCode <> 0 then
      exit;
   FifoReset;
   FifoWriteWord(CommandNumber);
   FifoWriteWord(GroupNumber);
   FifoWriteLongint(Param1);
   FifoWriteLongint(Param2);
   FifoWriteLongint(Param3);
   FifoSendMessageandWaitForResponse;
   ErrorCode:=FifoReadWord;
   end;
```

dms_T4AxisVectorProcedure

```
procedure dms_T4AxisVectorProcedure(
   CommandNumber:integer;
   GroupNumber:integer;
   Param1:longint;
   Param2:longint;
   Param3:longint;
   Param4:longint);

   begin
   if ErrorCode <> 0 then
      exit;
   FifoReset;
   FifoWriteWord(CommandNumber);
   FifoWriteWord(GroupNumber);
   FifoWriteLongint(Param1);
   FifoWriteLongint(Param2);
   FifoWriteLongint(Param3);
   FifoWriteLongint(Param4);
   FifoSendMessageandWaitForResponse;
   ErrorCode:=FifoReadWord;
   end;
```

dms_T5AxisVectorProcedure

This 5 parameter procedure helps manage 5 dimensional motion.

```
procedure dms_T5AxisVectorProcedure(
   CommandNumber:integer;
   GroupNumber:integer;
   Param1:longint;
   Param2:longint;
   Param3:longint;
   Param4:longint;
   Param5:longint);

   begin
   if ErrorCode <> 0 then
      exit;
   FifoReset;
   FifoWriteWord(CommandNumber);
   FifoWriteWord(GroupNumber);
   FifoWriteLongint(Param1);
   FifoWriteLongint(Param2);
   FifoWriteLongint(Param3);
   FifoWriteLongint(Param4);
   FifoWriteLongint(Param5);
   FifoSendMessageandWaitForResponse;
   ErrorCode:=FifoReadWord;
   end;
```

dms_T6AxisVectorProcedure

```
procedure dms_T6AxisVectorProcedure(
   CommandNumber:integer;
   GroupNumber:integer;
   Param1:longint;
   Param2:longint;
   Param3:longint;
   Param4:longint;
   Param5:longint;
   Param6:longint);

   begin
   if ErrorCode <> 0 then
      exit;
   FifoReset;
   FifoWriteWord(CommandNumber);
   FifoWriteWord(GroupNumber);
   FifoWriteLongint(Param1);
   FifoWriteLongint(Param2);
   FifoWriteLongint(Param3);
   FifoWriteLongint(Param4);
   FifoWriteLongint(Param5);
   FifoWriteLongint(Param6);
   FifoSendMessageandWaitForResponse;
   ErrorCode:=FifoReadWord;
  end;
```

Chapter 3

3) Command Reference

Command Summary

Notes

Commands are shown by name. Command names in the DLL are prefixed with "dms_" to insure uniqueness. When used with the ActiveX control, the names are proceeded by the Control name which is generally "MSB" followed by a period.

DLL Specific Commands

AllocateChannel ..... Performs initialization steps for DLL and make connection

ActiveX Specific Commands

Init ..... Initializes communication resources in ActiveX control
SetBuffer ..... Turns on or off command buffering for higher throughput
PerformBuffer ..... Performs command buffer when buffering is on
Busy ..... Indicates that command buffer is still busy performing commands

Communications

ErrorCode ..... Returns the result of the last command after completed

IO Operations

InputBit ..... Return level of specified input
ConfigureIOBitAsOutput ..... Instructs I/O bit to behave as output signal
SetOutputEnable ..... Tell output bits to become active (DMS only)
SetOutputBit ..... Change state of output bit on hardware

Safety

ResetWatchdog ..... Allow tripped safety system to resume servo activity
WatchdogHasTripped ..... Returns status of watchdog system
UserHasDisabled ..... Indicates if any disable input is asserted

Axis and AxisGroup Commands

Configuration
T2AxisInit ..... Associate 2 axes into coordinated group
T3AxisInit ..... Associate 3 axes into coordinated group
T4AxisInit ..... Associate 4 axes into coordinated group
T5AxisInit ..... Associate 5 axes into coordinated group

T6AxisInit ..... Associate 6 axes into coordinated group
TNAxisDispose ..... Release axis group relationship
ResetAllocation ..... Clear all group relationships
SetMotorType ..... Configures motor for servo or stepper operation
SetEnable ..... Allow amplifier to power motor
SetMotor ..... Turns motor operation on and off
SetLoopInversion ..... Include an additional sign inversion in control law
SetCoordinateInversion ..... Reverse which way is regarded as the positive direction
SetAccel ..... Set acceleration rate for trapezoidal moves
SetDecel ..... deceleration rate for trapezoidal moves
SetSpeed ..... Set speed of slew phase of trapezoidal moves
SetGain ..... Set compensation parameter for servo
SetZero ..... compensation parameter for servo
SetIntegrator ..... compensation parameter to eliminate steady state position error
SetErrorLimit ..... Set permissible tracking error before disable occurs
SetPositiveLimit ..... Set boundary for movement in the positive direction
SetNegativeLimit ..... Set boundary for movement in the negative direction
SetActualPosition ..... Define current position coordinate
SetCommandedPosition ..... Set commanded position for non-trapezoidal moves
ArmInputCapture ..... Prepares axis to latch position based on input signal
ArmIndexCapture ..... Prepares axis to latch position based on index signal
SetCommandedTorque ..... Set output voltage when not servoing

Motion
TNAxisMoveTo ..... Move to absolute coordinate
TNAxisMoveBy ..... Move to relative coordinate
TNAxisBeginMoveTo ..... Start move to absolute coordinate
TNAxisBeginMoveBy ..... Start move to relative coordinate
MoveAlongCurve ..... Perform coordinated multiaxis motion along curve
BeginMoveAlongCurve ..... Begin coordinated curved motion
TNAxisAppendMoveTo ..... Add absolute vector segment to curve description
TNAxisAppendMoveBy ..... Add vector segment to curve description relative to last segment
TNAxisAppendArc ..... Add circular or helical arc description to continuous path curve
Clear ..... Erase any established motion curve info
TNAxisLinkToBuffer ..... Assocate curve buffer with axis group
Jog ..... Move indefintely at constant speed
Stop ..... Gently stops any motion that may be in progress
BeginStop ..... Begins to stop but immediately does next instruction
Abort ..... Suddenly aborts any motion that my be in progress

Query
Gain ..... Return current compensator gain value
Zero ..... Return current compensator zero value
Integrator ..... Return current compensator integrator value
Accel ..... Return current acceleration in counts per second squared
Decel ..... Return current deceleration in counts per second squared
Speed ..... Return current speed in counts per second
ActualPosition ..... Return current actual motor position
CommandedPosition ..... Return ideal or target position for motor

DestinationPosition ..... Return absolute coordinate of end of move
ErrorPosition ..... Return discrepency between current and ideal position
CapturePosition ..... Return position recorded when latch event occurred
MoveIsFinished ..... Return true if move has finished
CommandedTorque ..... Return current analog output value
ProfileVelocity ..... Return current ideal profile velocity
CaptureHasTripped ..... Indicate if latch event has occurred
MotorIsOn ..... Return true if motor is currently powered and active
EnableIsOn ..... Return true if amplifier is powered
IBit ..... Return level of encoder index signal
ABit ..... Return level of encoder A channel
BBit ..... Return level of encoder B channel

User Task Control

BeginUserTask ..... Spawn independent application behavior in controller
ScheduleUserTask ..... Spawn periodically recurring independent behavior in controller
AbortUserTask ..... Terminate an independent behavior in controller
SuspendUserTask ..... Cause independent behavior in controller to become inactive
ResumeUserTask ..... Cause suspended activity to become active again
UserTaskPresent ..... Indicates if particular task is currently present in controller

User Variable Control

SetUserBoolean ..... Assigns user variable in controller
SetUserLongint ..... Assigns user variable in controller
SetUserSingle ..... Assigns user variable in controller
UserBoolean ..... Retreives user variable from controller
UserLongint ..... Retreives user variable from controller
UserSingle ..... Retreives user variable from controller

ABit

ActiveX Syntax

```
Public Function ABit(ByVal AxisNumber as Integer) As Boolean
```

C Syntax

```
long dms_ABit(int AxisNumber);
```

Pascal Syntax

```
function dms_ABit(AxisNumber:integer):longint;
```

Description

dms_Abit returns the high/low level of the encoder A channel for the specified axis. If the axis is not using the encoder, the A, B, and I signals can be used as general purpose inputs. It is also useful to use dms_ABit function to check the operation of an encoder. A constantly high or low level, regardless of encoder rotation, can indicate a broken encoder or wire. AxisNumber must be in the range 1 to 16.

Binary Command Implementation

```
function dms_ABit(AxisNumber:integer):longint;
   begin
   dms_ABit:=dms_AxisLongintFunction(bc_ABit,AxisNumber);
   end;
```

ActiveX Example

```
if Msb.Abit(1) then
   MsgBox "Axis 1 Encoder Signal A is high"
End If
```

See Also

dms_BBit
dms_IBit
dms_CaptureBit

Abort

ActiveX Syntax

```
Public sub Abort(ByVal GroupNumber as Integer)
```

C Syntax

```
void dms_Abort(int GroupNumber)
```

Pascal Syntax

```
procedure dms_Abort(GroupNumber:integer);
```

Description

Abort immediately and abruptly stops motion without a controlled decel. Abort is generally for emergency use. Note that an abort at high speeds will most likely cause a servo tracking error resulting in the servos shutting down. This problem can be solved by increasing the error limit with SetErrorLimit or using the dms_Stop command instead.

Binary Command Implementation

```
procedure dms_Abort(GroupNumber:integer);
   begin
   dms_AxisProcedure(bc_Abort,GroupNumber);
   end;
```

ActiveX Example

```
Msb.Abort 1'Aborts motion on X axis
Msb.Abort XYPair 'Group number references Coordinated pair
Msb.PerformBuffer
```

See Also

BeginStop
Stop

Accel

ActiveX Syntax

```
Public Function Accel(ByVal GroupNumber as Integer) As Long
```

C Syntax

```
long dms_Accel(int GroupNumber)
```

Pascal Syntax

```
function dms_Accel(GroupNumber:integer):longint;
```

Description

Accel returns the current setting of the acceleration that will be used by this axis group during trapezoidal moves. The units are in counts per second squared.

Binary Command Implementation

```
function dms_Accel(GroupNumber:integer):longint;
   begin
   dms_Accel:=dms_AxisLongintFunction(bc_Accel,GroupNumber);
   end;
```

ActiveX Example

```
Status.Caption=Msb.Accel(1)
```

SeeAlso

TNAxis.Decel
TNAxis.Speed
TNAxis.SetAccel
TNAxis.SetDecel
TNAxis.SetSpeed

ActualPosition

ActiveX Syntax

```
Public Function ActualPosition(ByVal AxisNumber as Integer) As Long
```

C Syntax

```
long dms_ActualPosition(int AxisNumber)
```

Pascal Syntax

```
function dms_ActualPosition(AxisNumber:integer):longint;
```

Description

ActualPosition returns the current position coordinate of the T1Axis receiver. This is often used when producing plots of the dynamic response of the motor. Note that this may well be different from the CommandedPosition of the motor, ie the theoretical position of where the motor should be. In some cases it may be more desirable to use the CommandedPosition rather than the Actual position. The value returned is in units of counts.SeeAlso

Binary Command Implementation

```
function dms_ActualPosition(AxisNumber:integer):longint;
   begin
   dms_ActualPosition:=
      dms_AxisLongintFunction(bc_ActualPosition,AxisNumber);
   end;
```

ActiveX Example

```
Status.Caption=Msb.ActualPosition(2)
```

See Also

CommandedPosition

ArmCompare

ActiveX Syntax

```
Public Sub ArmCompare(ByVal AxisNumber as Integer)
```

C Syntax

```
void dms_ArmCompare(int AxisNumber, int State)
```

Pascal Syntax

```
procedure dms_ArmCompare(AxisNumber:integer; State:boolean);
```

Description

The procedure dms_ArmCompare is used to prepare the compare bit for transitioning at a specific compare position. This high-speed compare function is the opposite of capture operation. During capture, the encoder position is recorded when an external hardware event occurs. During high speed compare, an output bit is changed when the encoder realizes a specific position established with the dms_SetComparePosition command. The boolean parameter to dms_ArmCompare is used to establish the initial, pre-triggered state of the compare output.

Binary Command Implementation

```
procedure dms_ArmCompare(AxisNumber:integer; State:boolean);
   begin
   dms_AxisProcedureBooleanParam(bc_ArmCompare,AxisNumber,State);
   end;
```

ActiveX Example

```
Msb.ArmComare 2
```

Example

```
dms_SetComparePosition(1,4000);
dms_ArmCompare(false);   {compare output signal now low}
```

See Also

dms_SetComparePosition

ArmIndexCapture

ActiveX Syntax

```
Public Sub ArmIndexCapture(ByVal AxisNumber as Integer)
```

C Syntax

```
void dms_ArmIndexCapture(int AxisNumber)
```

Pascal Syntax

```
procedure dms_ArmIndexCapture(AxisNumber:integer);
```

Description

ArmIndexCapture is used to setup the system to respond to an index pulse that is anticipated. ArmIndexCapture resets the capture latches for the axis associated with the TNAxis machine. When CaptureHasTripped the CapturePosition information is valid and can be used by the motion application.

Binary Command Implementation

```
procedure dms_ArmIndexCapture(AxisNumber:integer);
   begin
   dms_AxisProcedure(bc_ArmIndexCapture,AxisNumber);
   end;
```

ActiveX Example

```
Msb.ArmIndexCapture 4
```

See Also

ArmInputCapture
Capture
CaptureHasTripped
CapturePosition

ArmInputCapture

ActiveX Syntax

```
Public Sub ArmInputCapture(ByVal AxisNumber as Integer)
```

C Syntax

```
void dms_ArmInputCapture(int AxisNumber)
```

Pascal Syntax

```
procedure dms_ArmInputCapture(AxisNumber:integer);
```

Description

ArmInputCapture is used to setup the system to respond to an input pulse that is anticipated. ArmInputCapture resets the capture latches for the axis associated with the TNAxis machine. When CaptureHasTripped the CapturePosition information is valid and can be used by the motion application. Each axis has a specific input used for high speed capture.

Binary Command Implementation

```
procedure dms_ArmInputCapture(AxisNumber:integer);
   begin
   dms_AxisProcedure(bc_ArmInputCapture,AxisNumber);
   end;
```

ActiveX Example

```
Msb.ArmInputCapture 2
```

See Also

Capture Inputs
ArmInputCapture
Capture
Capture Inputs
CaptureHasTripped
CapturePosition

BBit

ActiveX Syntax

```
Public Function BBit(ByVal AxisNumber as Integer) As Boolean
```

C Syntax

```
long dms_BBit(int AxisNumber);
```

Pascal Syntax

```
function dms_BBit(AxisNumber:integer):longint;
```

Description

dms_Bbit returns the high/low level of the encoder B channel for the specified axis. If the axis is not using the encoder, the A, B, and I signals can be used as general purpose inputs. It is also useful to use dms_BBit function to check the operation of an encoder. A constantly high or low level, regardless of encoder rotation, can indicate a broken encoder or wire. AxisNumber must be in the range 1 to 16.

Binary Command Implementation

```
function dms_BBit(AxisNumber:integer):longint;
   begin
   dms_BBit:=dms_AxisLongintFunction(bc_BBit,AxisNumber);
   end;
```

ActiveX Example

```
Status.Caption=Msb.BBit(2)
```

See Also

dms_BBit
dms_IBit
dms_CaptureBit

BeginMoveAlongCurve

ActiveX Syntax

```
Public Sub BeginMoveAlongCurve(ByVal GroupNumber as Integer)
```

C Syntax

```
void dms_BeginMoveAlongCurve(int GroupNumber)
```

Pascal Syntax

```
dms_AxisProcedure(bc_BeginMoveAlongCurve,GroupNumber);
```

Description

BeginMoveAlongCurve performs continuous path motion over an arbitrary, multiaxis curve description which was previously setup. This routine is not implemented by a T1Axis single axis. Program execution immediately continues after the motion has started.

Binary Command Implementation

```
procedure dms_BeginMoveAlongCurve(GroupNumber:integer);
   begin
   dms_AxisProcedure(bc_BeginMoveAlongCurve,GroupNumber);
   end;
```

ActiveX Example

```
Msb.BeginMoveAlongCurve XYTable
```

SeeAlso

Curved Trajectories
TNAxis.MoveAlongCurve
TNAxis.MoveIsFinished

BeginUserTask

ActiveX Syntax

```
Public Sub BeginUserTask(ByVal TaskNumber as Integer)
```

C Syntax

```
void dms_BeginUserTask(int TaskNumber)
```

Pascal Syntax

```
procedure dms_BeginUserTask(TaskNumber:integer);
```

Description

BeginUserTask causes an application task in the controller to begin executing as an independent thread. Tasks are saved in controller FLASH memory through the use of the SAVE_APP catalog component. It is necessary to "connect" application tasks to user task numbers through an assignment procedure for access through the binary command interpreter. Consult the chapter on "Using Flash Memory" for more information.

Binary Command Implementation

```
procedure dms_BeginUserTask(TaskNumber:integer);
   begin dms_ProcedureIntegerParam(bc_BeginUserTask,TaskNumber);

   end;
```

ActiveX Example

```
Msb.BeginUserTask 12
```

See Also

dms_AbortUserTask
dms_ScheduleUserTask
dms_UserTaskPresent
dms_SuspendUserTask
dms_ResumeUserTask

BeginStop

ActiveX Syntax

```
Public Sub BeginStop(ByVal GroupNumber as Integer)
```

C Syntax

```
void dms_BeginStop(int GroupNumber)
```

Pascal Syntax

```
procedure dms_BeginStop(GroupNumber:integer);
```

Description

BeginStop directs the axis group to slow down at the specified decel rate and stop motion. A TNAxis group will remain coordinated during the stop. The calling program will not wait until after the stop has finished before continuing but will immediately execute the next statement.

Binary Command Implementation

```
procedure dms_BeginStop(GroupNumber:integer);
   begin
   dms_AxisProcedure(bc_BeginStop,GroupNumber);
   end;
```

ActiveX Example

```
Msb.BeginStop 2
```

See Also

TNAxis.Stop
TNAxis.Abort

Busy

ActiveX Syntax

```
Public Function Busy() As Boolean
```

Description

Buffered commands sent to the controller may take some time to execute, particularly if the command buffer uses commands such as Delay or MoveTo where the next command in the buffer is not performed until the prior delay or move finishes completely. It is necessary to use the Busy function to make sure that the command buffer has finished before submitting new commands.

ActiveX Example

```
while Msb.Busy
  DoEvents
WEnd
Msb.T1AxisBeginMoveT0, 1, 50000
```

CaptureBit

ActiveX Syntax

```
Public Function CaptureBit(ByVal AxisNumber as Integer) As Boolean
```

C Syntax

```
long dms_CaptureBit(int AxisNumber);
```

Pascal Syntax

```
function dms_CaptureBit(AxisNumber:integer):longint;
```

Description

dms_Capturebit returns the high/low level of the capture signal for the specified axis. The capture signal, when used in conjunction with the dms_SetCaptureTrip, dms_ArmInputCapture, and dms_CaptureHasTripped functions provides latching of the encoder position on an input event. However the signal can also be used as a general purpose input or homing input through the use of the dms_CaptureBit command which returns the current level.

Binary Command Implementation

```
function dms_CaptureBit(AxisNumber:integer):longint;
   begin
   dms_CaptureBit:=
      dms_AxisLongintFunction(bc_CaptureBit,AxisNumber);
   end;
```

ActiveX Example

```
Status.Caption=MSb.CaptureBit(2)
```

See Also

dms_ABit
dms_BBit
dms_IBit
dms_SetCaptureTrip
dms_ArmInputCapture
dms_CaptureHasTripped

CaptureHasTripped

ActiveX Syntax

```
Public Function CaptureHasTripped(ByVal AxisNumber as Integer) As
Boolean
```

C Syntax

```
int dms_CaptureHasTripped(int AxisNumber)
```

Pascal Syntax

```
function dms_CaptureHasTripped(AxisNumber:integer):boolean;
```

Description

CaptureHasTripped returns true if the index or input event, configure by ArmIndexCapture or ArmInputCapture, has occurred. If CaptureHasTripped then CapturePosition is valid and contains the position where the event occurred.

Binary Command Implementation

```
function dms_CaptureHasTripped(AxisNumber:integer):boolean;
   begin
   dms_CaptureHasTripped:=
      dms_AxisBooleanFunction(bc_CaptureHasTripped,AxisNumber);
   end;
```

ActiveX Example

```
Status.Caption=Msb.CaptureHasTripped(2)
```

See Also

TNAxis.ArmInputCapture
TNAxis.ArmIndexCapture

CapturePosition

ActiveX Syntax

```
Public Function CapturePosition(ByVal AxisNumber as Integer) As Long
```

C Syntax

```
long dms_CapturePosition(int AxisNumber)
```

Pascal Syntax

```
function dms_CapturePosition(AxisNumber:integer):longint;
```

Description

CapturePosition returns the position the axis experienced the capture event, either an index pulse of an input, which was anticipated using the ArmIndexCapture or ArmInputCapture instructions. The CapturePosition is only valid if CaptureHasTripped.

Binary Command Implementation

```
function dms_CapturePosition(AxisNumber:integer):longint;
   begin
   dms_CapturePosition:=
      dms_AxisLongintFunction(bc_CapturePosition,AxisNumber);
   end;
```

ActiveX Example

```
Status.Caption=Msb.CapturePosition(2)
```

See Also

ArmIndexCapture
ArmInputCapture
Capture
Capture Inputs
CaptureHasTripped

Clear

ActiveX Syntax

```
Public Sub Clear(ByVal GroupNumber as Integer)
```

C Syntax

```
void dms_Clear(int GroupNumber)
```

Pascal Syntax

```
procedure dms_Clear(GroupNumber:integer);
```

Description

dms_Clear removes any previous curve information and prepares the TNAxis to receive a new curve description with Append commands..

Binary Command Implementation

```
procedure dms_Clear(GroupNumber:integer);
   begin
   dms_AxisProcedure(bc_Clear,GroupNumber);
   end;
```

ActiveX Example

```
Msb.Clear XYTable
```

SeeAlso

Curved Trajectories
TNAxis.MoveAlongCurve
TNAxis.AppendMoveBy
TNAxis.AppendMoveTo
TNAxis.AppendMoveToVector
TNAxis.AppendMoveBy
TNAxis.AppendMoveByVector

CommandedPosition

ActiveX Syntax

```
Public Function CommandedPosition(ByVal AxisNumber as Integer) As Long
```

C Syntax

```
long dms_CommandedPosition(int GroupNumber)
```

Pascal Syntax

```
function dms_CommandedPosition(GroupNumber:integer):longint;
```

Description

CommandedPosition returns the theoretical position of the motor, i.e. the desired position of the motor. During the course of a profiled motion this number will smoothly change to represent the trajectory of the motor. Actual motor trajectory will differ from this theoretical expectation due to system dynamics and power limits realized in physical, real-world machines. The commanded position only exists when the motor is servoing. If the servo is not active the CommandedPosition is a meaningless number. For multidimensional axis groups the commanded position is the vector path length into the move or curve relative to the beginning of the curve. This can be used to perform events at particular positions along a multidimensional trajectory.

Binary Command Implementation

```
function dms_CommandedPosition(GroupNumber:integer):longint;
   begin
   dms_CommandedPosition:=
      dms_AxisLongintFunction(bc_CommandedPosition,GroupNumber);
   end;
```

ActiveX Example

```
Status.Caption=Msb.CommandedPosition(2)
```

See Also

T1Axis.ActualPosition
TNAxis.GetActualPositionVector
TNAxis.GetCommandedPositionVector

CommandedTorque

ActiveX Syntax

```
Public Function CommandedTorque(ByVal AxisNumber as Integer) As
Integer
```

C Syntax

```
int dms_CommandedTorque(int AxisNumber)
```

Pascal Syntax

```
function dms_CommandedTorque(AxisNumber:integer):integer;
```

Description

CommandedTorque returns the current amount of torque the servo controller is requesting for the receiving axis. This information is returned as an integer and is in the range of MaxTorque to MinTorque. This function can be used to determine if the axis is continually applying torque to a load or undergoing saturation, (ie constantly requesting the maximum or minimum torque).

Binary Command Implementation

```
function dms_CommandedTorque(AxisNumber:integer):integer;
   begin
   dms_CommandedTorque:=
      dms_AxisIntegerFunction(bc_CommandedTorque,AxisNumber);
   end;
```

ActiveX Example

```
Status.Caption=Msb.CommandedTorque(1)
```

SeeAlso

T1Axis.SetCommandedTorque

ConfigureIOBitAsOutput

ActiveX Syntax

```
Public Sub ConfigureIOBitAsOutpout(ByVal BitNumber as Integer, Param
as boolean)
```

C Syntax

```
void dms_ConfigureIOBitAsOutput(int BitNumber, int Param)
```

Pascal Syntax

```
procedure dms_ConfigureIOBitAsOutput(Bit:integer; IsOut:boolean);
```

Description

Motion Server I/O can be configured as inputs or outputs in 4-bit nibble sized groups. Groups are indicated in the connector diagram later in this document. After reset I/O defaults to inputs with 4.7k pullups to prevent asserting an active output signal. Nibble-groups can become outputs by using this command with the Bit parameter being the bit number of any bit in the group, and the IsOut parameter being set to "true". An output group can become an input group by indicating IsOut as false.

Binary Command Implementation

```
procedure dms_ConfigureIOBitAsOutput(Bit:integer; Param:boolean);
   begin
   if ErrorCode <> 0 then
      exit;
   FifoReset;
   FifoWriteWord(bc_ConfigureIOBitAsOutput);
   FifoWriteWord(Bit);
   FifoWriteBoolean(Param);
   FifoSendMessageandWaitForResponse;
   ErrorCode:=FifoReadWord;
   end;
```

ActiveX Example

```
Msb.ConfigureIOBitAsOutput 1,True
```

SeeAlso

dms_SetOutputBit
dms_SetOutputEnable

Decel

ActiveX Syntax

```
Public Function Decel(ByVal AxisNumber as Integer) As Long
```

C Syntax

```
long dms_Decel(int GroupNumber)
```

Pascal Syntax

```
function dms_Decel(GroupNumber:integer):longint;
```

Description

Decel returns the current setting of the deceleration that will be used by this axis group during trapezoidal moves. The units are in counts per second squared.

Binary Command Implementation

```
function dms_Decel(GroupNumber:integer):longint;
   begin
   dms_Decel:=
      dms_AxisLongintFunction(bc_Decel,GroupNumber);
   end;
```

ActiveX Example

```
Status.Caption=Msb.Decel(2)
```

SeeAlso

TNAxis.Accel
TNAxis.Speed
TNAxis.SetAccel
TNAxis.SetDecel
TNAxis.SetSpeed

DestinationPosition

ActiveX Syntax

```
Public Function DestinationPosition(ByVal AxisNumber as Integer) As
Long
```

C Syntax

```
long dms_DestinationPosition(int AxisNumber)
```

Pascal Syntax

```
function dms_DestinationPosition(AxisNumber:integer):longint;
```

Description

DestinationPosition returns the absolute coordinate of where a move will finish. This can be used to calculate the distance remaining in a move, move often used to overlap motion and reduce cycle time.

Binary Command Implementation

```
function dms_DestinationPosition(AxisNumber:integer):longint;
   begin
   dms_DestinationPosition:=
      dms_AxisLongintFunction(bc_DestinationPosition,AxisNumber);
   end;
```

ActiveX Example

```
Status.Caption=Msb.DestinationPosition(2)
```

See Also

CommandedPosition
ActualPosition

EnableIsOn

ActiveX Syntax

```
Public Function EnableIsOn(ByVal GroupNumber as Integer) As Boolean
```

C Syntax

```
int dms_EnableIsOn(int GroupNumber)
```

Pascal Syntax

```
function dms_EnableIsOn(GroupNumber:integer):boolean;
```

Description

EnableIsOn returns true (non-0) if all of the axis in the axis group are enabled. This should generally follow the state requested by SetServo, however servo tracking error can cause one or more axis to automatically shutdown. When a servo is turned off, it's enabled is also turned off to shutdown the amplifier. You can re-enable an amplifier without requesting servo activity by using SetEnable

Binary Command Implementation

```
function dms_EnableIsOn(GroupNumber:integer):boolean;
   begin
   dms_EnableIsOn:=
      dms_AxisBooleanFunction(bc_EnableIsOn,GroupNumber);
   end;
```

ActiveX Example

```
if Msb.EnableIsOn(3) then
   MsbBox "Axis 3 is enabled"
End If
```

See Also

Servo States
SetEnable
SetServo
ServoIsOn

ErrorCode

ActiveX Syntax

```
Public Function ErrorCode As Integer
```

C Syntax

```
int dms_ErrorCode
```

Pascal Syntax

```
function dms_ErrorCode:integer
```

Description

ErrorCode returns the result of the last command executed. If an error occurs in the processing of a DLL call commands will not be executed until the ResetErrorCode command is issued. In the ActiveX control, commands remaining in the buffer will not be performed until the next command buffer is submitted.

A value of "0" means no error occurred. Other error codes are described in the Error Code list found in the SAW.HLP file under Error Codes.

ActiveX Example

```
Msb.SetMotor 1, true
Msb.BeginMoveTo 1, 2000
Msb.PerformBuffer
While Msb.Busy do
   DoEvents
WEnd
if Msb.ErrorCode <> 0 then
   MsgBox "Error In Last Command"
End If
```

ErrorLimit

ActiveX Syntax

```
Public Function ErrorLimit(ByVal AxisNumber as Integer) As Long
```

C Syntax

```
long dms_ErrorLimit(int AxisNumber)
```

Pascal Syntax

```
function dms_ErrorLimit(AxisNumber:integer):longint;
```

Description

ErrorLimit reports the current error limit setting for a single axis. The error limit is used to establish how much servo tracking error is permitted before the system should consider the motor inoperative. If the tracking error is greater than the error limit, the controller shuts down the motor.

Binary Command Implementation

```
function dms_ErrorLimit(AxisNumber:integer):longint;
   begin
   dms_ErrorLimit:=
      dms_AxisLongintFunction(bc_ErrorLimit,AxisNumber);
   end;
```

ActiveX Example

```
Status.Caption=Msb.ErrorLimit(2)
```

See Also

dms_SetErrorLimit

ErrorPosition

ActiveX Syntax

```
Public Function ErrorPosition(ByVal AxisNumber as Integer) As Long
```

C Syntax

```
long dms_ErrorPosition(int AxisNumber)
```

Pascal Syntax

```
function dms_ErrorPosition(AxisNumber:integer):longint;
```

Description

ErrorPosition returns the difference between where the servo is commanded to be and its actual position. This difference is monitored. If it is found to be greater than the error limit, the servo is turned off.

Binary Command Implementation

```
function dms_ErrorPosition(AxisNumber:integer):longint;
  begin
  dms_ErrorPosition:=
   dms_AxisLongintFunction(bc_ErrorPosition,AxisNumber);
  end;
```

ActiveX Example

```
Status.Caption=Msb.ErrorPosition(2)
```

See Also

CommandedPosition
ActualPosition

Gain

ActiveX Syntax

```
Public Function Gain(ByVal AxisNumber as Integer) As Integer
```

C Syntax

```
int dms_Gain(int AxisNumber)
```

Pascal Syntax

```
function dms_Gain(AxisNumber:integer):integer;
```

Description

Motion Server implements PID control. This function returns the current value of the control law gain, one of the primary compensation parameters.

Binary Command Implementation

```
function dms_Gain(AxisNumber:integer):integer;
   begin
   dms_Gain:=dms_AxisIntegerFunction(bc_Gain,AxisNumber);
   end;
```

ActiveX Example

```
Status.Caption=Msb.Gain(2)
```

See Also

Integrator
SetGain
SetIntegrator
SetZero
Zero

IBit

ActiveX Syntax

```
Public Function IBit(ByVal AxisNumber as Integer) As Boolean
```

C Syntax

```
long dms_IBit(int AxisNumber);
```

Pascal Syntax

```
function dms_IBit(AxisNumber:integer):longint;
```

Description

dms_Ibit returns the high/low level of the encoder I channel for the specified axis. If the axis is not using the encoder, the A, B, and I signals can be used as general purpose inputs. The dms_IBit function returns the current value of the index pulse. As index pulses are quite narrow it is possible to miss the pulse if it is not latched. The dms_SetCaptureTrip, dms_ArmIndexCapture, and dms_CaptureHasTripped functions can be used to latch the index uplse. AxisNumber must be in the range 1 to 16.

Binary Command Implementation

```
function dms_IBit(AxisNumber:integer):longint;
   begin
   dms_IBit:=dms_AxisLongintFunction(bc_IBit,AxisNumber);
   end;
```

ActiveX Example

```
Status.Caption=Msb.ABit(2)
```

See Also

dms_ABit
dms_BBit
dms_CaptureBit
dms_SetCaptureTrip
dms_ArmIndexCapture
dms_CaptureHasTripped

Init (ActiveX only)

ActiveX Syntax

```
Public Sub Init
```

Description

Init puts the ActiveX control into a known condition for handling command buffer operations. This command should be called from the Form.Load event of the application

ActiveX Example

```
Msb.Init
```

InputBit

C Syntax

```
int dms_InputBit(int InputBitNumber)
```

Pascal Syntax

```
function dms_InputBit(InputBitNumber:word):boolean;
```

Description

InputBit returns true if the input level is high and false if the level is low. Values for bit number are 1 through 48.

Errors

If a bit number is requested beyond the range for the system then a ParameterOutOfRangeEscapeCode occurs.

Binary Command Implementation

```
function dms_InputBit(InputBitNumber:word):boolean;
   begin
   if ErrorCode <> 0 then
      exit;
   FifoReset;
   FifoWriteWord(bc_InputBit);
   FifoWriteWord(InputBitNumber);
   FifoSendMessageandWaitForResponse;
   ErrorCode:=FifoReadWord;
   if ErrorCode=0 then
      dms_InputBit:=FifoReadBoolean
   else
      dms_InputBit:=false;
   end;
```

ActiveX Example

```
if Msb.InputBit(12) = True then
   MsbBox "Input 12 is high"
End If
```

DLL Example

Consider wanting to perform some instructions if Input 1 had a high level. The following code would make this check:

```
...
if InputBit(1) then
  Writeln('Bit 1 is on')
else
  Writeln('Bit 1 is off');
...
```

See Also

InputLong1
InputLong2
InputLong3
SetOutputBit

Integrator

ActiveX Syntax

```
Public Function Integrator(ByVal AxisNumber as Integer) As Integer
```

C Syntax

```
int dms_Integrator(int AxisNumber)
```

Pascal Syntax

```
function dms_Integrator(AxisNumber:integer):integer;
```

Description

Motion Server implements PID control. This function returns the current value of the control law integrator, one of the primary compensation parameters.

Binary Command Implementation

```
function dms_Integrator(AxisNumber:integer):integer;
   begin
   dms_Integrator:=
      dms_AxisIntegerFunction(bc_Integrator,AxisNumber);
   end;
```

ActiveX Example

```
Status.Caption=Msb.Integrator(2)
```

See Also

Gain
Zero
Integrator
SetGain
SetZero
SetIntegrator

Jog

ActiveX Syntax

```
Public Sub Jog(ByVal AxisNumber as Integer, ByVal Speed as Long)
```

C Syntax

```
void dms_Jog(int AxisNumber, long Speed)
```

Pascal Syntax

```
procedure dms_Jog(AxisNumber:integer; Speed:longint);
```

Description

Jog directs an axis to move at the specified speed indefinitely. if the magnitude of aSpeed is smaller than the magnitude of the current speed the axis will slow down at the decel rate. If the magnitude is greater it will speed up at the accel rate. It is possible to jog in the opposite direction as the current speed. It is possible to jog at 0 speed. Jog may supersede a move, changing it into a continuous motion.

Although it is possible to use jog to produce movement when searching for home switches or other input events, it is generally a better idea to move a distance which should include the event so that the behavior of the machine if the event is not found is to stop rather than to travel indefinitely.

Jogging is NOT protected by positive and negative limits. Jogging, by it's nature, is a continuous move. To realize jogging velocity to the edge of a machine movement, perform a move to the positive or negative limit at the required jog speed.

Binary Command Implementation

```
procedure dms_Jog(AxisNumber:integer; Speed:longint);
   begin
   dms_AxisProcedureLongintParam(bc_Jog,AxisNumber,Speed);
   end;
```

ActiveX Example

```
Msb.Jog 2, 500
```

SeeAlso

TNAxis.Stop

LinkToBuffer

ActiveX Syntax

```
Public Sub LinkToBuffer(ByVal GroupNumber as Integer)
```

C Syntax

```
void dms_LinkToBuffer(int GroupNumber)
```

Pascal Syntax

```
procedure dms_LinkToBuffer(GroupNumber:integer);
```

Description

Motion Server supports continuous path motion. Curves are described by appending vectors and arcs to a list associated with the axis group that will perform the curve. To indicate to an axis group that space should be made available for this list use the dms_LinkToBuffer command. There are 2 lists in the standard binary command interpreter. Each list can support up to 500 elements and up to a T6Axis group. If both of these buffers have been used the error code will be set to be_OutOfCurveBuffers. If this occurs the most likely explanation is that dms_TNAxisRelease was not used to deallocate the buffers. Refer to this command, or use the dms_ResetAllocation command to provide a "clean slate" on startup.

Binary Command Implementation

```
procedure dms_LinkToBuffer(GroupNumber:integer);
   begin
   dms_ProcedureIntegerParam(GroupNumber);
   end;
```

ActiveX Example

```
Msb.LinkToBuffer XYTable
```

Example

```
procedure Trace2DPattern;
   var xy:integer;

   begin
   dms_SetMotorType(1,ServoMotor);
   dms_SetMotorType(2,ServoMotor);
   xy:=dms_T2AxisInit(1,2);
   dms_SetAccel(xy,20000);
   dms_SetDecel(xy,20000);
   dms_SetSpeed(xy,2000);
   dms_LinkToBuffer(xy);    {now xy can support curves}
   dms_Clear;
   dms_T2AxisAppendMoveBy(xy,1000,0);
   dms_T2AxisAppendArc(xy,2000,0,90);
   dms_BeginMoveAlongCurve(xy);
   .....
```

SeeAlso

dms_T2AxisAppendArc
dms_T3AxisAppendArc
dms_T2AxisAppendMoveBy
dms_T2AxisAppendMoveTo
dms_ReleaseAllocation
dms_TNAxisDispose

MotorIsOn

ActiveX Syntax

```
Public Function MotorIsOn(ByVal GroupNumber as Integer) As Boolean
```

C Syntax

```
int dms_MotorIsOn(int GroupNumber)
```

Pascal Syntax

```
function dms_MotorIsOn(GroupNumber:integer):boolean;
```

Description

MotorIsOn returns true (non-0) if all of the motors in the receiver are active (servoing if configured for servo) and false (0) if at least one of the motors is not active. Motors must be active before a move can take place.

Binary Command Implementation

```
function dms_MotorIsOn(GroupNumber:integer):boolean;
   begin
   dms_MotorIsOn:=
      dms_AxisBooleanFunction(bc_MotorIsOn,GroupNumber);
   end;
```

ActiveX Example

```
if Msb.MotorIsOn 2 then
   MsgBox "Motor 1 is holding position"
End If
```

SeeAlso

SetMotor

MoveAlongCurve

ActiveX Syntax

```
Public Sub MoveAlongCurve(ByVal GroupNumber as Integer)
```

C Syntax

```
void dms_MoveAlongCurve(int GroupNumber)
```

Pascal Syntax

```
procedure dms_MoveAlongCurve(GroupNumber:integer);
```

Description

MoveAlongCurve performs continuous path motion over an arbitrary, multiaxis curve description which was previously setup. This routine is not implemented by a T1Axis single axis. Program execution does not continue past MoveAlongCurve until the curve has been completed.

Binary Command Implementation

```
procedure dms_MoveAlongCurve(GroupNumber:integer);
   begin
   dms_AxisProcedure(bc_MoveAlongCurve,GroupNumber);
   end;
```

ActiveX Example

```
Msb.MoveAlongCurve XYTable
```

SeeAlso

Curved Trajectories
TNAxis.BeginMoveAlongCurve
TNAxis.MoveIsFinished

MoveIsFinished

ActiveX Syntax

```
Public Function MoveIsFinished(ByVal AxisNumber as Integer) As Boolean
```

C Syntax

```
int dms_MoveIsFinished(int GroupNumber)
```

Pascal Syntax

```
function dms_MoveIsFinished(GroupNumber:integer):boolean;
```

Description

MoveIsFinished indicates if the TNAxis is currently moving or if the move has completed. The DLL function returns 0 to represent false and non-0 to represent true. This would normally be used after starting motion with a procedure that had a name starting with BeginMove.....

Because of the multitasking options with Motion Server and SI-3000 it is sometimes more convenient to seperate functions into two parts, a motion part which uses "synchronous" motion commands that start with Move, and another part which performs the "background" activity, and to have both functions running at the same time.

Binary Command Implementation

```
function dms_MoveIsFinished(GroupNumber:integer):boolean;
   begin
   dms_MoveIsFinished:=
      dms_AxisBooleanFunction(bc_MoveIsFinished,GroupNumber);
   end;
```

ActiveX Example

```
if Msb.MoveIfFinished 2 then
   MsgBox "Axis 2 Move Is Completed"
End If
```

DLL Example

Imagine you would like to move a fixed distance with the expectation of hitting a switch along the way. The following routine would perform this check:

```
...
XAxis.BeginMoveBy(20000);
repeat
  yield;
  if XAxis.MoveIsFinished then
    Escape(SwitchNotFoundEscapeCode);
until InputBit(12);
XAxis.Stop;
...
```

The switch closure is supposed to be found before the end of the move. If the move finishes first there was a problem and an escape is performed. Otherwise the move is prematurely stopped near the switch point.

SeeAlso

BeginMoveTo
BeginMoveBy
TNAxis.BeginMoveToVector
TNAxis.BeginMoveByVector
TNAxis.BeginMoveAlongCurve

NegativeLimit

ActiveX Syntax

```
Public Function NegativeLimit(ByVal AxisNumber as Integer) As Long
```

C Syntax

```
long dms_NegativeLimit(int AxisNumber)
```

Pascal Syntax

```
function dms_NegativeLimit(AxisNumber:integer):longint;
```

Description

dms_NegativeLimit reports the current negative limit setting for a single axis. The negative limit establishes the most negative legal value for a move destination. If a move is requested that would produce a motor position more negative than the negative limit, an error is produced and no motion is performed. Note that position limits are not used when performing a jog. Jogs are by nature indefinite moves.

Binary Command Implementation

```
function dms_NegativeLimit(AxisNumber:integer):longint;
   begin
   dms_NegativeLimit:=
      dms_AxisLongintFunction(bc_NegativeLimit,AxisNumber);
   end;
```

ActiveX Example

```
Status.Caption=Msb.NegativeLimit(2)
```

See Also

dms_SetPositiveLimit

PerformBuffer (ActiveX only)

ActiveX Syntax

```
Public Sub PerformBuffer()
```

Description

In order to communicate more efficiently commands can be buffered and sent in one transaction to the controller. Once these commands have been buffered the PerformBuffer command sends them to the controller. It is important to wait for the buffer to be finished before attempting to queue up any more commands.

ActiveX Example

```
...
Msb.T1AxisMoveTo 1, 2000
Msb.T1AxisMoveTo 2, 4000
PerformBuffer
While Msb.Busy
   DoEvents
WEnd
Msb.T1AxisMoveTo 1, 0
.....
```

See Also

Busy
SetBuffer

PositiveLimit

ActiveX Syntax

```
Public Function PositiveLimit(ByVal AxisNumber as Integer) As Long
```

C Syntax

```
long dms_PositiveLimit(int AxisNumber)
```

Pascal Syntax

```
function dms_PositiveLimit(AxisNumber:integer):longint;
```

Description

dms_PositiveLimit reports the current positive limit setting for a single axis. The positive limit establishes the most positive legal value for a move destination. If a move is requested that would produce a motor position greater than the positive limit, an error is produced and no motion is performed. Note that position limits are not used when performing a jog. Jogs are by nature indefinite moves.

Binary Command Implementation

```
function dms_PositiveLimit(AxisNumber:integer):longint;
   begin
   dms_PositiveLimit:=
      dms_AxisLongintFunction(bc_PositiveLimit,AxisNumber);
   end;
```

ActiveX Example

```
Status.Caption=Msb.PositiveLimit(2)
```

See Also

dms_SetNegativeLimit

ProfileVelocity

ActiveX Syntax

```
Public Function ProfileVelocity(ByVal AxisNumber as Integer) As Long
```

C Syntax

```
long dms_ProfileVelocity(int GroupNumber)
```

Pascal Syntax

```
function dms_ProfileVelocity(GroupNumber:integer):longint;
```

Description

ProfileVelocity returns the current commanded speed (signed magnitude) that is being used to generate the trapezoidal motion trajectory. During slew, the magnitude of ProfileVelocity is the same as Speed. During accel and decel the profile velocity varies according to the point in the profile.

Binary Command Implementation

```
function dms_ProfileVelocity(GroupNumber:integer):longint;
   begin
   dms_ProfileVelocity:=
      dms_AxisLongintFunction(bc_ProfileVelocity,GroupNumber);
   end;
```

ActiveX Example

```
Status.Caption=Msb.ProfileVelocity(2)
```

See Also

TNAxis.CommandedPosition
T1Axis.ActualPosition

ResetAllocation

ActiveX Syntax

```
Public Sub ResetAllocation(ByVal AxisNumber as Integer)
```

C Syntax

```
void dms_ResetAllocation
```

Pascal Syntax

```
procedure dms_ResetAllocation;
```

Description

Motion Server is designed to provide motion services to several clients at one time. In the course of providing these services resources are allocated through the dms_T2AxisInit..dms_T6AxisInit commands and through the dms_LinkToBuffer routine. If a client program terminates and does not dispose of these resources with the dms_TNAxisDispose command, eventually the resources will be consumed and Motion Server will report errors. The procedure dms_ResetAllocation is used to provide a "clean slate" for Motion Server resources. In a multiple client situation, dms_ResetAllocation should not be used as it would "pull the resources out from under" another client program which may be active. If you are developing an application that only involves a single client, dms_ResetAllocation can be used in the startup code to insure a full set of resources is available.

Binary Command Implementation

```
procedure dms_ResetAllocation;
   begin
   dms_Procedure(bc_ResetAllocation);
   end;
```

ActiveX Example

```
Msb.ResetAllocation
```

See Also

dms_TNAxisDispose

ResetWatchdog

ActiveX Syntax

```
Public Sub ResetWatchdog(ByVal AxisNumber as Integer)
```

C Syntax

```
void dms_ResetWatchdog()
```

Pascal Syntax

```
procedure dms_ResetWatchdog;
```

Description

The motion system operates under the supervision of a watchdog system. If for any reason the processor should be delayed in responding to the motion system's timer event the watchdog system will shutdown the power amplifiers to insure that no undesired motion occurs. ResetWatchdog allows servo activity to occur again.

Binary Command Implementation

```
procedure dms_ResetWatchdog;
   begin
   dms_Procedure(bc_ResetWatchdog);
   end;
```

Errors

If ResetWatchdog discovers that the watchdog did not reset a WatchdogFailedToResetEscapeCode will occur.

ActiveX Example

```
Msb.ResetWatchdog
```

See Also

WatchdogHasTripped

SampleRate

ActiveX Syntax

```
Public Function SampleRate As Integer
```

C Syntax

```
int dms_SampleRate
```

Pascal Syntax

```
function dms_SampleRate:integer;
```

Description

The function dms_SampleRate reports the current sample rate frequency of the controller. Expected values are in the 1000 to 4000 Hz range. The sample rate can be adjusted with the dms_SetSampleRate procedure

Binary Command Implementation

```
function dms_SampleRate:integer;
   begin
   dms_IntegerFunction(bc_SampleRate);
   end;
```

Errors

If ResetWatchdog discovers that the watchdog did not reset a WatchdogFailedToResetEscapeCode will occur.

ActiveX Example

```
Status.Caption=Msb.SampleRate
```

See Also

WatchdogHasTripped

SetAccel

ActiveX Syntax

```
Public Sub SetAccel(ByVal AxisNumber as Integer, ByVal Param As Long)
```

C Syntax

```
void dms_SetAccel(int GroupNumber, long Param)
```

Pascal Syntax

```
procedure dms_SetAccel(GroupNumber:integer; Param:longint);
```

Description

SetAccel is used to set the acceleration of a profiled move in counts per second squared. If the receiver is a T1Axis the acceleration is for the movement of that motor when operating alone. If the receiver is an axis group , for example a T2Axis or T4Axis, the acceleration applies to the coordinated motion profile of the group. SetDecel may be used to independently set the deceleration of the TNAxis.

Binary Command Implementation

```
procedure dms_SetAccel(GroupNumber:integer; Param:longint);
   begin
   dms_AxisProcedureLongintParam(bc_SetAccel,GroupNumber,Param);
   end;
```

ActiveX Example

```
Msb.SetAccel 1, 50000
```

SeeAlso

TNAxis.SetDecel
TNAxis.SetSpeed
TNAxis.Accel
TNAxis.Speed
TNAxis.Decel

SetActualPosition

ActiveX Syntax

```
Public Sub SetActualPosition(ByVal AxisNumber as Integer, ByVal Param
as long)
```

C Syntax

```
void dms_SetActualPosition(int AxisNumber, long Param)
```

Pascal Syntax

```
procedure dms_SetActualPosition(AxisNumber:integer; Param:longint);
```

Description

SetActualPosition is used to define what the current physical position should be. In Douloi Pascal this procedures takes as many parameters as the dimension of the axis group, ie for a single axis this takes one parameter. For a four axis machine four parameters are required, for X,Y,Z, and U axis.

Binary Command Implementation

```
procedure dms_SetActualPosition(AxisNumber:integer; Param:longint);
   begin
   dms_AxisProcedureLongintParam(
      bc_SetActualPosition,AxisNumber,Param);
   end;
```

ActiveX Example

```
Msb.SetActualPosition 1, 0
```

DLL Example

```
SetActualPosition(1,1000,ErrorCode);
SetActualPosition(2,2000,ErrorCode);
```

See Also

T1Axis.ActualPosition
TNAxis.GetActualPositionVector

SetBuffer (ActiveX only)

ActiveX Syntax

```
Public Sub SetBuffer(ByVal State As Boolean)
```

Description

SetBuffer turns on and off the buffering of commands. If the buffer is off commands are sent immediately. If the buffer is on, commands are stored in the host until PerformBuffer is called or until the number of commands is approaching the buffer size of approximately 20 commands. The commands are then sent to the controller once which is more efficient.

On initialization the buffer is set on. If the buffer is on remember to use the PerformBuffer command to get results.

ActiveX Example

```
Msb.SetBuffer true
```

See Also

PerformBuffer

SetCaptureTrip

ActiveX Syntax

```
Public Sub SetCaptureTrip(ByVal AxisNumber as Integer, ByVal State as
boolean)
```

C Syntax

```
void dms_SetCaptureTrip(int AxisNumber, int State)
```

Pascal Syntax

```
procedure dms_SetCaptureTrip(AxisNumber:integer; State:boolean);
```

Description

The procedure dms_SetCaptureTrip is used to establish what signal transition constitutes a capture event. A low-to-high transition would be indicated with a true parameter. A high-to-low transition would be indicated with a false parameter. dms_SetCaptureTrip is used with dms_ArmIndexCapture arm dms_ArmIputCapture.

Binary Command Implementation

```
procedure dms_SetCaptureTrip(AxisNumber:integer; State:boolean);
   begin
   dms_AxisProcedureBooleanParam(
      bc_SetCaptureTrip,AxisNumber,State);
   end;
```

ActiveX Example

```
Msb.SetCaptureTrip 1, true
```

DLL Example

```
dms_SetCaptureTrip(1,true);
dms_ArmIndexCapture; {controller waiting for low-to-high change}
```

See Also

dms_ArmIndexCapture
dms_ArmInputCapture

SetCommandedPosition

ActiveX Syntax

```
Public Sub SetCommandedPosition(ByVal AxisNumber as Integer, ByVal
Param as long)
```

C Syntax

```
void dms_SetCommandedPosition(int AxisNumber, long Param)
```

Pascal Syntax

```
procedure dms_SetCommandedPosition(
   AxisNumber:integer; Param:longint);
```

Description

SetCommandedPosition is used to set the desired setpoint for the servo. During normal profiled moves the commanded position is set for you by the profiler whilch calculates a smooth sequences of commanded positions. However there are some situations where the criteria for where the motor should servo is custom, for example electronic gearing. SetCommandedPosition "goes around" the profiler allowing you to directly set the servo setpoint. Note that a discontinuity in setpoint positions will directly map into an attempted discontinuity in motor position resulting in a substantial jerk.This procedure takes as many parameters as the dimension of the axis group, ie for a single axis this takes one parameter. For a four axis machine four parameters are required, for X,Y,Z, and U axis.

Binary Command Implementation

```
procedure dms_SetCommandedPosition(
   AxisNumber:integer; Param:longint);

   begin
   dms_AxisProcedureLongintParam(
      bc_SetCommandedPosition,AxisNumber,Param);
   end;
```

ActiveX Example

```
Msb.SetCommandedPosition 1, 0
```

SeeAlso

T1Axis.ActualPosition
TNAxis.GetActualPositionVector
TNAxis.CommandedPosition
TNAxis.GetCommandedPositionVector
TNAxis.SetCommandedPositionVector

SetCommandedTorque

ActiveX Syntax

```
Public Sub SetCommandedTorque(ByVal AxisNumber as Integer, ByVal Param
as Integer)
```

C Syntax

```
void dms_SetCommandedTorque(int AxisNumber, int Param)
```

Pascal Syntax

```
procedure dms_SetCommandedTorque(
   AxisNumber:integer; Param:integer);
```

Description

SetCommandedTorque is used in situations where the motor is being run "open loop". This procedure sets the value for the digital to analog converter for the physical axis related to this T1Axis. The parameter value may range from MinTorque to MaxTorque which is the range -2040 to 2039. The actual torque is the result of adding this parameter to the current TorqueOffset for the axis. The resulting sum will be truncated within the bounds MinTorque to MaxTorque. SetCommandedTorque can only be used if the axis is not currently servoing and the axis is enabled.

Binary Command Implementation

```
procedure dms_SetCommandedTorque(
   AxisNumber:integer; Param:integer);

   begin
   dms_AxisProcedureIntegerParam(
      bc_SetCommandedTorque,AxisNumber,Param);
   end;
```

Errors

SetCommandedTorque will escape if the parameter value is greater than MaxTorque or less then MinTorque with a ParameterOutOfRangeEscapeCode .

ActiveX Example

```
Msb.SetCommandedTorque 1, 1000
```

SeeAlso

CommandedTorque
OffsetTorque

SetCompareBit

ActiveX Syntax

```
Public Sub SetCompareBit(ByVal AxisNumber as Integer, ByVal Param as
Boolean)
```

C Syntax

```
void dms_SetCompareBit(int AxisNumber, int Param)
```

Pascal Syntax

```
procedure dms_SetCompareBit(AxisNumber:integer; Param:boolean);
```

Description

dms_SetCompareBit is used to manipuate the axis compare bit signal as if it was a general purpose output. This is used to provide an output when the high-speed compare function for that axis is not required. If Param is true, the compare output signal is set to a logic high level. If Param is false, the signal is set to a logic low level.

Binary Command Implementation

```
procedure dms_SetCompareBit(AxisNumber:integer; Param:boolean);
   begin
   dms_AxisProcedureBooleanParam(bc_SetCompare,AxisNumber,Param);
   end;
```

ActiveX Example

```
Msb.SetCompareBit 1, true
```

DLL Example

```
dms_SetCompareBit(1,true);
dms_SetCompareBit(2,false);
```

See Also

dms_SetOutputBit

SetCoordinateInversion

ActiveX Syntax

```
Public Sub SetCoordinateInversion(ByVal AxisNumber as Integer, ByVal
Param as Boolean)
```

C Syntax

```
void dms_SetCoordinateInversion(int AxisNumber, int Param)
```

Pascal Syntax

```
procedure dms_SetCoordinateInversion(
   AxisNumber:integer; Param:boolean);
```

Description

SetCoordinateInversion is used to change the direction a motor regards as positive. Axis direction is influenced by mechanical transmission reversals, encoder phase definition, and wiring conventions. If the motor does not move in the direction regarded as positive this procedure may be used to invert the direction by calling with a parameter value of true. Using the predefined booleans On and Off may improve the readability of the code. A better design option is to change the wiring, most likely of the A and B channels of the encoder so that the axis moves in the correct direction from the default values rather than having to be "setup" by this procedure call. If that wiring is inconvenient this procedure may be the simplest option. Changing the encoder wires also requires changing the motor wires so as to preserve the loop sign. Note that changing the wires of the motor alone will not have the desired effect but will instead cause the servo loop to go unstable.

This command operates in an incremental manner by inverting the coordinate frame about the current actual position rather than 0. The best time to use this command is during initial setup before homing has been performed. This is not intended to be used during motion.

Binary Command Implementation

```
procedure dms_SetCoordinateInversion(
   AxisNumber:integer; Param:boolean);

   begin
   dms_AxisProcedureBooleanParam(
      bc_SetCoordinateInversion,AxisNumber,Param);
   end;
```

ActiveX Example

```
Msb.SetCoordinateInversion 1, true
```

See Also

T1Axis.SetLoopInversion

SetDac

ActiveX Syntax

```
Public Sub SetDac(ByVal AxisNumber as Integer, ByVal Param as integer)
```

C Syntax

```
void dms_SetDac(int AxisNumber, int Param)
```

Pascal Syntax

```
procedure dms_SetDac(AxisNumber:integer; Param:integer);
```

Description

The dms_SetDac command is used to operate the motor command output signal for a particular axis as if it was a general purpose Digital to Analog converter output. The signal is made available for use by setting the motor type for that axis to be ServoMotorNoDAC. This is most often used at the motion controller application level when creating new application level control laws. dms_SetDac is a low-level command and operates the dac regardless of whether the dms_MotorIsOn or the dms_EnableIsOn for a particular axis.

Binary Command Implementation

```
procedure dms_SetDac(AxisNumber:integer; Param:integer);

   begin
   dms_AxisProcedureIntegerParam(
      bc_SetDac,AxisNumber,Param);
   end;
```

Errors

SetCommandedTorque will escape if the parameter value is greater than MaxTorque or less then MinTorque with a ParameterOutOfRangeEscapeCode .

ActiveX Example

```
Msb.SetDac 1, 1000
```

DLL Example

```
dms_SetMotorType(1,ServoMotorNoDac);
dms_SetDAC(1024);  {sets output to 5 volts}
```

SeeAlso

dms_CommandedTorque
dms_TorqueOffset

SetDecel

ActiveX Syntax

```
Public Sub SetDecel(ByVal GroupNumber as Integer, ByVal Param as long)
```

C Syntax

```
void dms_SetDecel(int GroupNumber, long Param)
```

Pascal Syntax

```
procedure dms_SetDecel(GroupNumber:integer; Param:longint);
```

Description

SetDecel establishes the deceleration rate that will be used by an axis during the ends of moves and stops. The deceleration may be different from the accelerations and is independently set with SetAccel. aDecelValue is in units of counts per second squared. Values in the range of 200,000 are gentle decelerations. Values in the range of 2,000,000 are abrupt. It is possible to change the decel during any phase of the move. The change takes immediate effect in the midst of the move.

Binary Command Implementation

```
procedure dms_SetDecel(GroupNumber:integer; Param:longint);
   begin
   dms_AxisProcedureLongintParam(bc_SetDecel,GroupNumber,Param);
   end;
```

Errors

If a move is in progress and the decel is changed on the fly to a value lower than the current decel it is possible that the current motion cannot be completed at that decel. The point where deceleration should have started may be "behind" the axis already. In this case the the axis will produce a MotionOverrunEscapeCode and come to a stop at the new decel value.

ActiveX Example

```
Msb.SetDecel 1, 50000
```

SeeAlso

TNAxis.SetAccel
TNAxis.SetSpeed
TNAxis.MoveTo
TNAxis.MoveBy

SetEnable

ActiveX Syntax

```
Public Sub SetEnable(ByVal GroupNumber as Integer, ByVal Param as
Boolean)
```

C Syntax

```
void dms_SetEnable(int GroupNumber, int Param)
```

Pascal Syntax

```
procedure dms_SetEnable(GroupNumber:integer; Param:boolean);
```

Description

If SetEnable(on) then the analog output for the receiving axis or axis group is turned on, and the amp enable is asserted. The current CommandedTorque is expressed through the analog output. If SetEnable(off) the analog voltage is set to 0 and the amp enable line is not asserted. SetEnable(off) is identical to SetMotor(off) and is provided for completeness. SetEnable(on) is distinct from SetMotor(on) in that SetMotor(on) begins the control law whereas SetEnable(on) only enables the output of the command allowing some other application criteria to determine what the CommandedTorque should be.

Binary Command Implementation

```
procedure dms_SetEnable(GroupNumber:integer; Param:boolean);
   begin
   dms_AxisProcedureBooleanParam(bc_SetDecel,GroupNumber,Param);
   end;
```

ActiveX Example

```
Msb.SetEnable 1, true
```

SeeAlso

SetCommandedTorque
CommandedTorque
SetMotor

SetErrorLimit

ActiveX Syntax

```
Public Sub SetErrorLimit(ByVal GroupNumber as Integer, ByVal Param as
long)
```

C Syntax

```
void dms_SetErrorLimit(int GroupNumber, long Param)
```

Pascal Syntax

```
procedure dms_SetErrorLimit(GroupNumber:integer; Param:longint);
```

Description

SetErrorLimit is used to describe how far a physical axis 's actual position can lag behind the commanded position without that lagging being considered an error. Ideally the motor's actual position exactly follows the commanded position however system dynamics and transient response of the motion control law means that in general this idealistic case is not achieved for arbitrary profiles although it can be closely achieved for non-accelerating profiles. Systems which have high accelerations and decelerations are also likely to incur following error during those times if the power system saturates. If the difference between the actual position and commanded positions exceeds the error limit the axis will perform a TNAxis.SetServo(Off) ;

The error limit is always being checked. Set the limit to be a large value if the SetServo(Off) behavior is not desired.

Binary Command Implementation

```
procedure dms_SetErrorLimit(GroupNumber:integer; Param:longint);
   begin
   dms_AxisProcedureLongintParam(
      bc_SetErrorLimit,GroupNumber,Param);
   end;
```

ActiveX Example

```
Msb.SetErrorLimit 2, 200
```

DLL Example

```
SetErrorLimit(1,200);
SetErrorLimit(5,500);
```

See Also

TNAxis.SetMotor
TNAxis.MotorIsOn

SetGain

ActiveX Syntax

```
Public Sub SetGain(ByVal AxisNumber as Integer, ByVal Param as
integer)
```

C Syntax

```
void dms_SetGain(int AxisNumber, int Param)
```

Pascal Syntax

```
procedure dms_SetGain(AxisNumber:integer; Param:integer);
```

Description

Motion Server and implements PID servo control. The gain of a control loop is one of the primary parameters used to set the servo's compensation. This procedure sets the control law gain to be aGainValue. Values in the range of 16 to 150 are not unusual. As the gain increases the servo system behaves more responsivly. As the gain becomes excessive the servo becomes "jittery" and tends to vibrate. The gain is used in conjunction with the zero and integrator to establish the control law for servo operation.

Binary Command Implementation

```
procedure dms_SetGain(AxisNumber:integer; Param:integer);
   begin
   dms_AxisProcedureIntegerParam(bc_SetGain,AxisNumber,Param);
   end;
```

ActiveX Example

```
Msb.SetGain 1, 30
```

See Also

Gain
Integrator
SetIntegrator
SetZero
Zero

SetIntegrator

ActiveX Syntax

```
Public Sub SetIntegrator(ByVal AxisNumber as Integer, ByVal Param as
Integer)
```

C Syntax

```
void dms_SetIntegrator(int AxisNumber, int Param)
```

Pascal Syntax

```
procedure dms_SetIntegrator(AxisNumber:integer; Param:integer);
```

Description

Motion Server aimplements PID servo control. The integrator of a control loop is one of the primary parameters used to set the servo's compensation. The integrator causes the error in a sero loop to eventually reduce to 0. How quickly the error reduces to zero is related to how large the integrator is. However if the integrator values becomes too large the system becomes unstable. In general the value of the integrator should be about 1/10th the value of the gain parameter if the integrator is being used.

Binary Command Implementation

```
procedure dms_SetIntegrator(AxisNumber:integer; Param:integer);
   begin
   dms_AxisProcedureIntegerParam(
      bc_SetIntegrator,AxisNumber,Param);
   end;
```

ActiveX Example

```
Msb.SetIntegrator 1, 4
```

See Also

Gain
Integrator
SetGain
SetZero
Zero

SetLoopInversion

ActiveX Syntax

```
Public Sub SetLoopInversion(ByVal AxisNumber as Integer, ByVal Param
as Boolean)
```

C Syntax

```
void dms_SetLoopInversion(int AxisNumber, int Param)
```

Pascal Syntax

```
procedure dms_SetLoopInversion(AxisNumber:integer; Param:boolean);
```

Description

SetLoopInversion is used to add an additional sign change in the feedback loop so as to change the total loop sign. This instruction is provided to compensate for encoder wiring or motor wiring which is not providing the correct feedback sense. A better response to the problem of unstable loop sign is to change the wiring of the motor leads (invert loop sign) or encoder A and B channels (invert coordinate frame and sign) rather than use this instruction since forgetting this instruction in a future application causes the motor to be unstable. AxisNumber must be in the range 1 to 16. Group Numbers are not allowed for this routine.

Binary Command Implementation

```
procedure dms_SetLoopInversion(AxisNumber:integer; Param:boolean);
   begin
   dms_AxisProcedureBooleanParam(
      bc_SetLoopInversion,AxisNumber,Param);
   end;
```

ActiveX Example

```
Msb.SetLoopInversion 1,true
```

See Also

T1Axis.SetErrorLimit
T1Axis.SetCoordinateInversion

SetMotor

ActiveX Syntax

```
Public Sub SetMotor(ByVal AxisNumber as Integer, ByVal Param as
Boolean)
```

C Syntax

```
void dms_SetMotor(int AxisNumber, int Param)
```

Pascal Syntax

```
procedure dms_SetMotor(AxisNumber:integer; Param:boolean);
```

Description

SetMotor is used to turn motor activity on and off for all the axis in the TNAxis. Called with a parameter value of true enables the amplifier lines. The motor servos to the current location (if configured for servo). When called with a parameter value of false the amplifier lines are disabled, the motor command is set to 0 volts (if configured for servo) and no further motor activity occurs. Readability of the program is improved by using the predefined boolean constants On and Off. SetMotor is an alias for the outdated SetServo routine, (retained for backward comaptibility) to acknowledge both stepper and servo motor capability.

Binary Command Implementation

```
procedure dms_SetMotor(AxisNumber:integer; Param:boolean);
   begin
   dms_AxisProcedureBooleanParam(bc_SetMotor,AxisNumber,Param);
   end;
```

ActiveX Example

```
Msb.SetMotor 2, true
MSB.SetMotor XYTable, true
```

DLL Example

```
SetMotor(1,On);
SetMotor(1,Off);
SetMotor(102,On);  {multiaxis group, all motors turned on}
SetMotor(104,Off); {multiaxis group, all motors turned off}
```

Escapes

SetMotor(On) will generate a WatchdogFailedToResetEscapeCode if the WatchdogHasTripped .

See Also

MotorIsOn

SetMotorType (DMS only)

C Syntax

```
void dms_SetMotorType(int AxisNumber, int Param)
```

Pascal Syntax

```
procedure dms_SetMotorType(AxisNumber:integer; Param:integer);
```

Description

SetMotorType is used to configure a particular axis to run a servo motor or a stepper motor. The AxisNumber parameter must be in the range 1 to 16. Group Numbers are not allowed for this parameter. If the configuration is for stepper, it is also possible to indicate whether the step pulse goes high to indicate a step or goes low.This information is conveyed through the bit mask parameter. The following constants are included to aid in specifying the motor configuration:

```
(ServoMotor) or
   (StepperMotor + (HighStepPulse or LowStepPulse))
```

When setting an axis for use as a servo motor, just use ServoMotor as the parameter. When specifying a StepperMotor the parameter is StepperMotor with a pulse width contant and a pulse polarity constant added to it.

Binary Command Implementation

```
procedure dms_SetMotorType(AxisNumber:integer; Param:integer);
   begin
   dms_AxisProcedureIntegerParam(bc_SetMotorType,AxisNumber,Param);
   end;
```

DLL Example

If you wanted to indicate that the XAxis was a servo motor you would say:

```
SetMotorType(1,ServoMotor);
```

If you wanted to indicate that the XAxis was a stepper motor with a high going step you would say:

```
XAxis.SetMotorType(
   StepperMotor+HighStepPulse);
```

SetNegativeLimit

ActiveX Syntax

```
Public Sub SetNegativeLimit(ByVal AxisNumber as Integer, Param As
Long)
```

C Syntax

```
void dms_SetNegativeLimit(int GroupNumber, long Param)
```

Pascal Syntax

```
procedure dms_SetNegativeLimit(GroupNumber:integer; Param:longint);
```

Description

SetNegativeLimit establishes a negative-direction boundary for movement. If the axis is asked to attempt a move beyond this boundary, a PositionLimitEscapeCode will occur and the axis will stop.

Binary Command Implementation

```
procedure dms_SetNegativeLimit(GroupNumber:integer; Param:longint);
   begin
   dms_AxisProcedureLongintParam(
      bc_SetNegativeLimit,GroupNumber,Param);
   end;
```

ActiveX Example

```
Msb.SetNegativeLimit 1, -50000
```

See Also

SetPositiveLimit

SetOutputBit

ActiveX Syntax

```
Public SetOutputBit(ByVal bitNumber as Integer, ByVal Param as
Boolean)
```

C Syntax

```
void dms_SetOutputBit(int BitNumber, int Param)
```

Pascal Syntax

```
procedure dms_SetOutputBit(BitNumber:word; Param:boolean);
```

Description

SetOutputBit is used to set output bits to a prescribed level. BitNumber should be in the range of 1 to 48. Value should be a boolean parameter. The predefined constants On and Off can help improve readability of the program. These bits will only take effect if SetOutputEnable(On) has been used since a hardware reset.

Errors

If the bit number is outside of the allowable range for the system configuraiton a ParameterOutOfRangeEscapeCode will occur.

Binary Command Implementation

```
procedure dms_SetOutputBit(BitNumber:word; Param:boolean);
   begin
   if ErrorCode <> 0 then
      exit;
   FifoReset;
   FifoWriteWord(bc_SetOutputBit);
   FifoWriteWord(bitNumber);
   FifoWriteBoolean(Param);
   FifoSendMessageandWaitForResponse;
   ErrorCode:=FifoReadWord;
   end;
```

ActiveX Example

```
Msb.SetOutputBit 14, true
```

DLL Example

```
SetOutputEnable(on);
SetOutputBit(1,On);
```

SetOutputEnable (DMS Only)

C Syntax

```
void dms_SetOutputEnable(int Param)
```

Pascal Syntax

```
procedure dms_SetOutputEnable(Param:boolean);
```

Description

After a hardware reset, the general I/O is configured as inputs and the output drives are tristated. Pullups on the signals will assert a "soft" high level as the default signal. Digital outputs on the axis connector will also be tristated after reset. SetOutputEnable activates the outputs (on signals configured to be outputs) so that SetOutputBit works. SetOutputEnable(Off) tristates the outputs in the same manner that a hardware reset would. The DLL call will escape if there are insufficient tasks available to perform the operation.

Binary Command Implementation

```
procedure dms_SetOutputEnable(Param:boolean);
   begin
   if ErrorCode <> 0 then
      exit;
   FifoReset;
   FifoWriteWord(bc_SetOutputEnable);
   FifoWriteBoolean(Param);
   FifoSendMessageandWaitForResponse;
   ErrorCode:=FifoReadWord;
   end;
```

DLL Example

```
SetOutputEnable(on);
SetOutputEnable(off);
```

See Also

SetOutputBit

```
SetOutputBit(2,Off);
```

SetPositiveLimit

ActiveX Syntax

```
Public Sub SetPositiveLimit(ByVal AxisNumber as Integer, ByVal Param
as Long)
```

C Syntax

```
void dms_SetPositiveLimit(int GroupNumber, long Param)
```

Pascal Syntax

```
procedure dms_SetPositiveLimit(GroupNumber:integer; Param:longint);
```

Description

SetPositiveLimit establishes a positive-direction boundary for movement. If the axis is asked to attempt a move beyond this boundary, a PositionLimitEscapeCode will occur and the axis will stop.

Binary Command Implementation

```
procedure dms_SetPositiveLimit(GroupNumber:integer; Param:longint);
   begin
   dms_AxisProcedureLongintParam(
      bc_SetPositiveLimit,GroupNumber,Param);
   end;
```

ActiveX Example

```
Msb.SetPositiveLimit 1, 50000
```

See Also

SetNegativeLimit

SetSampleRate

ActiveX Syntax

```
Public Sub SetSampleRate(ByVal Param as Integer)
```

C Syntax

```
void dms_SetSampleRate(int Param)
```

Pascal Syntax

```
procedure dms_SetSampleRate(Param:integer);
```

Description

SetSampleRate establishes the control sample rate frequency. The default value is 1000. High performance brushless motors should be controlled at higher sample rates, such as 2000 or 4000. Application tasks are invoked at the sample rate frequency.

Binary Command Implementation

```
procedure dms_SetSampleRate(Param:integer);
   begin
   dms_ProcedureIntegerParam(
      bc_SetSampleRate,Param);
   end;
```

ActiveX Example

```
Msb.SetSampleRate 2000
```

See Also

dms_SampleRate

SetSpeed

ActiveX Syntax

```
Public Sub SetSpeed(ByVal AxisNumber as Integer, ByVal Param as Long)
```

C Syntax

```
void dms_SetSpeed(int GroupNumber, long Param)
```

Pascal Syntax

```
procedure dms_SetSpeed(GroupNumber:integer; Param:longint);
```

Description

SetSpeed establishes the slew speed to be used during axis movement. aSpeed is in units of counts/second. Values in the range of 80,000 are brisk. Values in the range of 1000 are slow. The speed of a move may be changed on the fly at any point in a move and take immediate effect if the motion is in the slew phase. For single axis machines SetSpeed effects the speed of the axis. For multiaxis groups SetSpeed effects the vector speeed of the group.

Binary Command Implementation

```
procedure dms_SetSpeed(GroupNumber:integer; Param:longint);
   begin
   dms_AxisProcedureLongintParam(bc_SetSpeed,GroupNumber,Param);
   end;
```

ActiveX Example

```
Msb.SetSpeed 2, 20000
MSB.SetSpeed XYTable, 10000
```

SeeAlso

TNAxis.SetAccel
TNAxis.SetDecel
TNAxis.MoveTo
TNAxis.MoveBy
TNAxis.BeginMoveTo
TNAxis.BeginMoveBy

SetUserBoolean

ActiveX Syntax

```
Public Sub SetUserBoolean(ByVal Index as longint, ByVal Value as
Boolean)
```

C Syntax

```
void dms_SetUserBoolean(int Index, int value)
```

Pascal Syntax

```
procedure dms_SetUserBoolean(Index:integer; Value:boolean);
```

Description

SetUserBoolean assigns the value the boolean variable in Motion Server at the specified index. User variables are used to communicate data between the host and tasks operating on the Motion Server card.

Binary Command Implementation

```
procedure dms_SetUserBoolean(Number:longint; Value:longint);
   begin
   if ErrorCode <> 0 then
      exit;
   FifoReset;
   FifoWriteWord(bc_SetUserBoolean);
   FifoWriteWord(Number);
   FifoWriteBoolean(Value);
   FifoSendMessageAndWaitForResponse;
   ErrorCode:=FifoReadWord;
   end;
```

ActiveX Example

```
Msb.SetUserBoolean 5, true
```

See Also

UserSetBoolean

SetUserLongint

ActiveX Syntax

```
Public Sub SetUserLongint(ByVal Index as Long, ByVal Value as Long)
```

C Syntax

```
void dms_SetUserLongint(int Index, long value)
```

Pascal Syntax

```
procedure dms_SetUserLongint(Index:integer; Value:longint);
```

Description

SetUserLongint assigns the value the longint variable in Motion Server at the specified index. User variables are used to communicate data between the host and tasks operating on the Motion Server card.

Binary Command Implementation

```
procedure dms_SetUserLongint(Index:longint; Value:longint);
   begin
   if ErrorCode <> 0 then
      exit;
   FifoReset;
   FifoWriteWord(bc_SetUserLongint);
   FifoWriteWord(Index);
   FifoWriteLongint(Value);
   FifoSendMessageAndWaitForResponse;
   ErrorCode:=FifoReadWord;
   end;
```

ActiveX Example

```
Msb.SetUserLongint 2, 20000
```

See Also

UserLongint

SetUserSingle

ActiveX Syntax

```
Public Sub SetUserSingle(ByVal Index as Integer, ByVal Value as
Snigle)
```

C Syntax

```
void dms_SetUserSingle(int Index, single value)
```

Pascal Syntax

```
procedure dms_SetUserLongint(Index:integer; Value:single);
```

Description

SetUserSingle assigns the value the single variable in Motion Server at the specified index. User variables are used to communicate data between the host and tasks operating on the Motion Server card.

Binary Command Implementation

```
procedure dms_SetUserSingle(Number:longint; Value:single);
   begin
   if ErrorCode <> 0 then
      exit;
   FifoReset;
   FifoWriteWord(bc_SetUserSingle);
   FifoWriteWord(Number);
   FifoWriteSingle(Value);
   FifoSendMessageAndWaitForResponse;
   ErrorCode:=FifoReadWord;
   end;
```

ActiveX Example

```
Msb.SetUserSingle 4, 1.4
```

See Also

SetUserSingle

SetZero

ActiveX Syntax

```
Public Sub SetZero(ByVal AxisNumber as Integer, ByVal Param As
Integer)
```

C Syntax

```
void dms_SetZero(int AxisNumber, int Param)
```

Pascal Syntax

```
procedure dms_SetZero(AxisNumber:integer; Param:integer);
```

Description

Motion Server implements PID servo control. The zero of a control loop is one of the primary parameters used to set the servo's compensation and primarily relates to the damping of the system. This procedure sets the control law zero to be aZeroValue. Values in the range of 200 to 255 are not unusual. Values greater than 255 are not legal.

Binary Command Implementation

```
procedure dms_SetZero(AxisNumber:integer; Param:integer);
   begin
   dms_AxisProcedureIntegerParam(bc_SetZero,AxisNumber,Param);
   end;
```

ActiveX Example

```
Msb.SetZero 1, 232
```

See Also

Gain
Integrator
SetGain
SetIntegrator
Zero

Speed

ActiveX Syntax

```
Public Function Speed(ByVal AxisNumber as Integer) As Long
```

C Syntax

```
long dms_Speed(int GroupNumber)
```

Pascal Syntax

```
function dms_Speed(GroupNumber:integer):longint;
```

Description

Speed returns the current setting of the speed that will be used by this axis group during trapezoidal moves. The units are in counts per second.

Binary Command Implementation

```
function dms_Speed(GroupNumber:integer):longint;
   begin
   dms_Speed:=dms_AxisLongintFunction(bc_Speed,GroupNumber);
   end;
```

ActiveX Example

```
Status.Caption=Msb.Speed 1
```

SeeAlso

TNAxis.Accel
TNAxis.Decel
TNAxis.SetAccel
TNAxis.SetDecel
TNAxis.SetSpeed

Stop/StopAxis

ActiveX Syntax

```
Public Sub StopAxis(ByVal AxisNumber as Integer)
```

C Syntax

```
void dms_Stop(int GroupNumber)
```

Pascal Syntax

```
procedure dms_Stop(GroupNumber:integer);
```

Description

Stop directs the axis group to slow down at the specified decel rate and stop motion. A TNAxis group will remain coordinated during the stop. The calling program will wait until after the stop has finished before continuing.

Binary Command Implementation

```
procedure dms_Stop(GroupNumber:integer);
   begin
   dms_AxisProcedure(bc_Stop,GroupNumber);
   end;
```

ActiveX Example

```
Msb.StopAxis 1
```

SeeAlso

BeginStop
Abort

T2AxisAppendArc

ActiveX Syntax

```
Public T2AxisAppendArc(ByVal GroupNumber as Integer, ByVal Radius as
long, ByVal InitialAngle as Single, ByVal DeltaAngle as Single)
```

C Syntax

```
void dms_T2AxisAppendArc(int GroupNumber,
  long Radius,
  float InitialAngle,
  float DeltaAngle)
```

Pascal Syntax

```
procedure dms_T2AxisAppendArc(GroupNumber:integer;
   Radius:longint;
   InitialAngle:single;
   DeltaAngle:single);
```

Description

AppendArc adds a circular segment to the continuous path being constructed. The first parameter is the radius of the arc. The InitialAngle indicates, in degrees, the tangent angle of the beginning of the arc. The DeltaAngle indicates how many degrees of rotation should occur. Note that DeltaAngle can indicate more than 360 degrees of rotation. Negative delta angles indicate curves to the right. Positive delta angles indicates curves to the left. Angles are measured with the X pointing in direction 0 and Y pointing in direction 90.

Binary Command Implementation

```
procedure dms_T2AxisAppendArc(GroupNumber:integer;
   Radius:longint;
   InitialAngle:single;
   DeltaAngle:single);

   begin
   if ErrorCode <> 0 then
      exit;
   FifoReset;
   FifoWriteWord(bc_T2AxisAppendArc);
   FifoWriteWord(GroupNumber);
   FifoWriteLongint(Radius);
   FifoWriteSingle(InitialAngle);
   FifoWriteSingle(DeltaAngle);
   FifoSendMessageandWaitForResponse;
   ErrorCode:=FifoReadWord;
   end;
```

ActiveX Example

```
Msb.T2AxisAppendArc XYTable, 5000, 0 ,90
```

SeeAlso

Curved Trajectories
TNAxis.MoveAlongCurve
TNAxis.AppendMoveTo
TNAxis.AppendMoveToVector
TNAxis.AppendMoveByVector

T3AxisAppendArc

ActiveX Syntax

```
Public Sub T3AxisAppendArc(ByVal GroupNumber as Integer, ByVal Radius
as long, ByVal InitialAngle As Single, ByVal DeltaAngle as Single,
ByVal DeltaZ As Long)
```

C Syntax

```
void dms_T3AxisAppendArc(int GroupNumber,
  long Radius,
  float InitialAngle,
  float DeltaAngle,
  long DeltaZ)
```

Pascal Syntax

```
procedure T3AxisAppendArc(
   GroupNumber:word;
   Radius:longint;
   InitialAngle:single;
   DeltaAngle:single;
   DeltaZ:longint);
```

Description

AppendArc adds a circular segment to the continuous path being constructed. The first parameter is the radius of the arc. The InitialAngle indicates, in degrees, the tangent angle of the beginning of the arc. The DeltaAngle indicates how many degrees of rotation should occur. Note that DeltaAngle can indicate more than 360 degrees of rotation. Negative delta angles indicate curves to the right. Positive delta angles indicates curves to the left. Angles are measured with the X pointing in direction 0 and Y pointing in direction 90. The last parameter, DeltaZ, indicates the change in Z position over the course of the arc. AppendArc, when used with a T3Axis group, allows circular interpolation to occur in X and Y while linear interpolation is occuring in Z.

Binary Command Implementation

```
procedure dms_T3AxisAppendArc(GroupNumber:integer;
   Radius:longint;
   InitialAngle:single;
   DeltaAngle:single;
   DeltaZ:longint);

   begin
   if ErrorCode <> 0 then
      exit;
   FifoReset;
   FifoWriteWord(bc_T3AxisAppendArc);
   FifoWriteWord(GroupNumber);
   FifoWriteLongint(Radius);
   FifoWriteSingle(InitialAngle);
   FifoWriteSingle(DeltaAngle);
   FifoWriteLongint(DeltaZ);
   FifoSendMessageandWaitForResponse;
   ErrorCode:=FifoReadWord;
   end;
```

ActiveX Example

```
Msb.T3Axis.AppendArc XYZTable, 5000, 0, 90, 2000
```

SeeAlsoSeeAlso

Curved Trajectories
TNAxis.MoveAlongCurve
TNAxis.AppendMoveTo
TNAxis.AppendMoveToVector
TNAxis.AppendMoveByVector

TNAxisAppendMoveBy

ActiveX Syntax

```
Public Sub T2AxisAppendMoveBy(ByVal GroupNumber as Integer,
   ByVal Delta1 As Long,
   ByVal Delta2 As Long)

Public Sub T3AxisAppendMoveBy(ByVal GroupNumber as Integer,
   ByVal Delta1 As Long,
   ByVal Delta2 As Long,
   ByVal Delta3 As Long)

Public Sub T4AxisAppendMoveBy(ByVal GroupNumber as Integer,
   ByVal Delta1 As Long,
   ByVal Delta2 As Long,
   ByVal Delta3 As Long,
   ByVal Delta4 As Long)

Public Sub T5AxisAppendMoveBy(ByVal GroupNumber as Integer,
   ByVal Delta1 As Long,
   ByVal Delta2 As Long,
   ByVal Delta3 As Long,
   ByVal Delta4 As Long,
   ByVal Delta5 As Long)

Public Sub T6AxisAppendMoveBy(ByVal GroupNumber as Integer,
   ByVal Delta1 As Long,
   ByVal Delta2 As Long,
   ByVal Delta3 As Long,
   ByVal Delta4 As Long,
   ByVal Delta5 As Long,
   ByVal Delta6 As Long)
```

C Syntax

```
void dms_T2AxisAppendMoveBy(int GroupNumber,
  long Delta1,
  long Delta2)

void dms_T3AxisAppendMoveBy(int GroupNumber,
  long Delta1,
  long Delta2,
  long Delta3)

void dms_T4AxisAppendMoveBy(int GroupNumber,
  long Delta1,
  long Delta2,
  long Delta3,
  long Delta4)

void dms_T5AxisAppendMoveBy(int GroupNumber,
  long Delta1,
  long Delta2,
  long Delta3,
  long Delta4,
  long Delta5)

void dms_T6AxisAppendMoveBy(int GroupNumber,
  long Delta1,
  long Delta2,
  long Delta3,
  long Delta4,
  long Delta5,
  long Delta6)
```

Pascal Syntax

```
procedure dms_T2AxisAppendMoveBy(GroupNumber:word;
   Delta1:longint;
   Delta2:longint);

procedure dms_T3AxisAppendMoveBy(GroupNumber:word;
   Delta1:longint;
   Delta2:longint;
   Delta3:longint);

procedure dms_T4AxisAppendMoveBy(GroupNumber:word;
   Delta1:longint;
   Delta2:longint;
   Delta3:longint;
   Delta4:longint);

procedure dms_T5AxisAppendMoveBy(GroupNumber:word;
   Delta1:longint;
   Delta2:longint;
   Delta3:longint;
   Delta4:longint;
   Delta5:longint);

procedure dms_T6AxisAppendMoveBy(GroupNumber:word;
   Delta1:longint;
   Delta2:longint;
   Delta3:longint;
   Delta4:longint;
   Delta5:longint;
   Delta6:longint);
```

Description

AppendMoveBy adds an additional descriptive point, expressed in relative coordinates, to the end of a curve description. The number of parameters corresponds to the dimension of the TNAxis.

Binary Command Implementation

The following is a 2 axis implementation. Other axis counts would be implemented by changing the T2AxisVectorProcedure to the procedure with the correct dimension.

```
procedure dms_T2AxisAppendMoveBy(GroupNumber:word;
   Delta1:longint;
   Delta2:longint);

   begin
   dms_T2AxisVectorProcedure(bc_TNAxisAppendMoveBy,GroupNumber,
      Delta1,Delta2);
   end;
```

ActiveX Example

```
Msb.T2AxisAppendMoveBy XYTable, 1000, 2000
```

See Also

Curved Trajectories
TNAxis.MoveAlongCurve
TNAxis.AppendMoveTo
TNAxis.AppendMoveToVector
TNAxis.AppendMoveByVector

TNAxisAppendMoveTo

ActiveX Syntax

```
Public Sub T2AxisAppendMoveTo(ByVal GroupNumber as Integer,
   ByVal Destination1 As Long,
   ByVal Destination2 As Long)

Public Sub T3AxisAppendMoveTo(ByVal GroupNumber as Integer,
   ByVal Destination1 As Long,
   ByVal Destination2 As Long,
   ByVal Destination3 As Long)

Public Sub T4AxisAppendMoveTo(ByVal GroupNumber as Integer,
   ByVal Destination1 As Long,
   ByVal Destination2 As Long,
   ByVal Destination3 As Long,
   ByVal Destination4 As Long)

Public Sub T5AxisAppendMoveTo(ByVal GroupNumber as Integer,
   ByVal Destination1 As Long,
   ByVal Destination2 As Long,
   ByVal Destination3 As Long,
   ByVal Destination4 As Long,
   ByVal Destination5 As Long)

Public Sub T6AxisAppendMoveTo(ByVal GroupNumber as Integer,
   ByVal Destination1 As Long,
   ByVal Destination2 As Long,
   ByVal Destination3 As Long,
   ByVal Destination4 As Long,
   ByVal Destination5 As Long,
   ByVal Destination6 As Long)
```

C Syntax

```
void dms_T2AxisAppendMoveTo(int GroupNumber,
  long Destination1,
  long Destination2)

void dms_T3AxisAppendMoveTo(int GroupNumber,
  long Destination1,
  long Destination2,
  long Destination3)

void dms_T4AxisAppendMoveTo(int GroupNumber,
  long Destination1,
  long Destination2,
  long Destination3,
  long Destination4)

void dms_T5AxisAppendMoveTo(int GroupNumber,
  long Destination1,
  long Destination2,
  long Destination3,
  long Destination4,
  long Destination5)

void dms_T6AxisAppendMoveTo(int GroupNumber,
  long Destination1,
  long Destination2,
  long Destination3,
  long Destination4,
  long Destination5,
  long Destination6)
```

Pascal Syntax

```
procedure T2AxisAppendMoveTo(GroupNumber:word;
   Destination1:longint;
   Destination2:longint);

procedure T3AxisAppendMoveTo(GroupNumber:word;
   Destination1:longint;
   Destination2:longint;
   Destination3:longint);

procedure T4AxisAppendMoveTo(GroupNumber:word;
   Destination1:longint;
   Destination2:longint;
   Destination3:longint;
   Destination4:longint);

procedure T5AxisAppendMoveTo(GroupNumber:word;
   Destination1:longint;
   Destination2:longint;
   Destination3:longint;
   Destination4:longint;
   Destination5:longint);

procedure T6AxisAppendMoveTo(GroupNumber:word;
   Destination1:longint;
   Destination2:longint;
   Destination3:longint;
   Destination4:longint;
   Destination5:longint;
   Destination6:longint;
```

Description

AppendMoveTo adds an additional descriptive point, expressed in absolute coordinates, to the end of a curve description. The number of parameters corresponds to the dimension of the TNAxis.

Binary Command Implementation

The following is a 4 axis implementation. Other axis counts would be implemented by changing the T4AxisVectorProcedure to the procedure with the correct dimension.

```
procedure dms_T4AxisAppendMoveTo(GroupNumber:word;
 Destination1:longint;
 Destination2:longint;
 Destination3:longint;
 Destination4:longint);

  begin
 dms_T4AxisVectorProcedure(bc_TNAxisAppendMoveTo,GroupNumber,
  Destination1,Destination2,Destination3,Destination4);
  end;
```

ActiveX Example

```
Msb.T2AxisAppendMoveTo XYTable, 1000, 1000
```

SeeAlso

Curved Trajectories
TNAxis.MoveAlongCurve
TNAxis.AppendMoveBy
TNAxis.AppendMoveToVector
TNAxis.AppendMoveByVector

TNAxisBeginMoveBy

ActiveX Syntax

```
Public Sub T2AxisBeginMoveBy(ByVal GroupNumber as Integer,
   ByVal Delta1 As Long,
   ByVal Delta2 As Long)

Public Sub T3AxisBeginMoveBy(ByVal GroupNumber as Integer,
   ByVal Delta1 As Long,
   ByVal Delta2 As Long,
   ByVal Delta3 As Long)

Public Sub T4AxisBeginMoveBy(ByVal GroupNumber as Integer,
   ByVal Delta1 As Long,
   ByVal Delta2 As Long,
   ByVal Delta3 As Long,
   ByVal Delta4 As Long)

Public Sub T5AxisBeginMoveBy(ByVal GroupNumber as Integer,
   ByVal Delta1 As Long,
   ByVal Delta2 As Long,
   ByVal Delta3 As Long,
   ByVal Delta4 As Long,
   ByVal Delta5 As Long)

Public Sub T6AxisBeginMoveBy(ByVal GroupNumber as Integer,
   ByVal Delta1 As Long,
   ByVal Delta2 As Long,
   ByVal Delta3 As Long,
   ByVal Delta4 As Long,
   ByVal Delta5 As Long,
   ByVal Delta6 As Long)
```

C Syntax

```
void dms_T1AxisBeginMoveBy(int GroupNumber,
  long DeltaPosition)

void dms_T2AxisBeginMoveBy(int GroupNumber,
  long DeltaPosition1,
  long DeltaPosition2)

void dms_T3AxisBeginMoveBy(int GroupNumber,
  long DeltaPosition1,
  long DeltaPosition2,
  long DeltaPosition3)

void dms_T4AxisBeginMoveBy(int GroupNumber,
  long DeltaPosition1,
  long DeltaPosition2,
  long DeltaPosition3,
  long DeltaPosition4)

void dms_T5AxisBeginMoveBy(int GroupNumber,
  long DeltaPosition1,
  long DeltaPosition2,
  long DeltaPosition3,
  long DeltaPosition4,
  long DeltaPosition5)

void dms_T6AxisBeginMoveBy(int GroupNumber,
  long DeltaPosition1,
  long DeltaPosition2,
  long DeltaPosition3,
  long DeltaPosition4,
  long DeltaPosition5,
  long DeltaPosition6)
```

Pascal Syntax

```
procedure T1AxisBeginMoveBy(AxisNumber:word;
   DeltaPosition:longint);

procedure T2AxisBeginMoveBy(GroupNumber:word;
   DeltaPosition1:longint;
   DeltaPosition2:longint);

procedure T3AxisBeginMoveBy(GroupNumber:word;
   DeltaPosition1:longint;
   DeltaPosition2:longint;
   DeltaPosition3:longint);

procedure T4AxisBeginMoveBy(GroupNumber:word;
   DeltaPosition1:longint;
   DeltaPosition2:longint;
   DeltaPosition3:longint;
   DeltaPosition4:longint);

procedure T5AxisBeginMoveBy(GroupNumber:word;
   DeltaPosition1:longint;
   DeltaPosition2:longint;
   DeltaPosition3:longint;
   DeltaPosition4:longint;
   DeltaPosition5:longint);

procedure T6AxisBeginMoveBy(GroupNumber:word;
   DeltaPosition1:longint;
   DeltaPosition2:longint;
   DeltaPosition3:longint;
   DeltaPosition4:longint;
   DeltaPosition5:longint;
   DeltaPosition6:longint);
```

Description

BeginMoveBy starts relative coordinated move by the specified position deltas but does not wait for the move to finish. In actual use the "N" in TNAxisBeginMoveBy is replaced by the dimension of the controlled group. For example, a 2 axis call would be T2AxisBeginMoveBy. The method requires as many parameters as the dimension of the receiver axis group, ie a 2 axis group requires 2 parameters, a 4 axis group requires 4 parameters. The motion is performed with a trapezoidal velocity profile based on parameters set with the SetAccel , SetDecel , and SetSpeed methods. These parameters apply to the vector path motion of the coordinated group rather than to any particular axis. BeginMoveBy returns immediatly and does not wait for the motion to finish. For cases where it is important to "blocking" program execution until the end of the move use MoveBy instead of BeginMoveBy. Use MoveHasCompleted to determine when a move started with BeginMoveBy has finished.

Group Numbers required to perform the call is provided by the TNInit functions

Binary Command Implementation

The following is a 3 axis implementation. Other axis counts would be implemented by changing the T3AxisVectorProcedure to the procedure with the correct dimension.

```
procedure dms_T3AxisBeginMoveBy(GroupNumber:word;
   Delta1:longint;
   Delta2:longint;
   Delta3:longint);

   begin
   dms_T3AxisVectorProcedure(bc_TNAxisBeginMoveBy,GroupNumber,
      Delta1,Delta2,Delta3);
   end;
```

Errors

BeginMoveBy will escape if while in motion the resulting destination specified is "behind" the vector path position or if the destination is so close that the axis group cannot accomplish the move at the specified decel rate. In these cases the group will emit a MotionOverrunEscapeCode and come to a stop.

ActiveX Example

```
Msb.T2AxisBeginMoveBy XYTable, 2000, 2000
```

SeeAlso

TNAxis.SetAccel
TNAxis.SetDecel
TNAxis.SetSpeed
TNAxis.MoveIsFinished
TNAxis.MoveBy
TNAxis.MoveTo
TNAxis.BeginMoveTo
MotionOverrunEscapeCode

dms_TNAxisBeginMoveTo

ActiveX Syntax

```
Public Sub T2AxisBeginMoveTo(ByVal GroupNumber as Integer,
   ByVal Delta1 As Long,
   ByVal Delta2 As Long)

Public Sub T3AxisBeginMoveTo(ByVal GroupNumber as Integer,
   ByVal Delta1 As Long,
   ByVal Delta2 As Long,
   ByVal Delta3 As Long)

Public Sub T4AxisBeginMoveTo(ByVal GroupNumber as Integer,
   ByVal Delta1 As Long,
   ByVal Delta2 As Long,
   ByVal Delta3 As Long,
   ByVal Delta4 As Long)

Public Sub T5AxisBeginMoveTo(ByVal GroupNumber as Integer,
   ByVal Delta1 As Long,
   ByVal Delta2 As Long,
   ByVal Delta3 As Long,
   ByVal Delta4 As Long,
   ByVal Delta5 As Long)

Public Sub T6AxisBeginMoveTo(ByVal GroupNumber as Integer,
   ByVal Delta1 As Long,
   ByVal Delta2 As Long,
   ByVal Delta3 As Long,
   ByVal Delta4 As Long,
   ByVal Delta5 As Long,
   ByVal Delta6 As Long)
```

C Syntax

```
void dms_T1AxisBeginMoveTo(int GroupNumber,
  long Destination)

void dms_T2AxisBeginMoveTo(int GroupNumber,
  long Destination1,
  long Desintation2)

void dms_T3AxisBeginMoveTo(int GroupNumber,
  long Destination1,
  long Desintation2,
  long Destination3)

void dms_T4AxisBeginMoveTo(int GroupNumber,
  long Destination1,
  long Desintation2,
  long Destination3,
  long Destination4)

void dms_T5AxisBeginMoveTo(int GroupNumber,
  long Destination1,
  long Desintation2,
  long Destination3,
  long Destination4,
  long Destination5)

void dms_T6AxisBeginMoveTo(int GroupNumber,
  long Destination1,
  long Desintation2,
  long Destination3,
  long Destination4,
  long Destination5,
  long Destination6)
```

Pascal Syntax

```
procedure T2AxisBeginMoveTo(GroupNumber:word;
   Destination1:longint;
   Destination2:longint);

procedure T3AxisBeginMoveTo(GroupNumber:word;
   Destination1:longint;
   Destination2:longint;
   Destination3:longint);

procedure T4AxisBeginMoveTo(GroupNumber:word;
   Destination1:longint;
   Destination2:longint;
   Destination3:longint;
   Destination4:longint);

procedure T5AxisBeginMoveTo(GroupNumber:word;
   Destination1:longint;
   Destination2:longint;
   Destination3:longint;
   Destination4:longint;
   Destination5:longint);

procedure T6AxisBeginMoveTo(GroupNumber:word;
   Destination1:longint;
   Destination2:longint;
   Destination3:longint;
   Destination4:longint;
   Destination5:longint;
   Destination6:longint);
```

Description

BeginMoveTo starts an absolute coordinated move to the specified absolute position destinations but does not wait for the move to finish. The "N" in TNAxisBeginMoveTo is replaced by the dimension of the group being controlled, for example T2AxisBeginMoveTo for a 2 axis group. The method requires as many parameters as the dimension of the receiver axis group, ie a 2 axis group requires 2 parameters, a 4 axis group requires 4 parameters. The motion is performed with a trapezoidal velocity profile based on parameters set with the SetAccel , SetDecel , and SetSpeed methods. These parameters apply to the vector path motion of the coordinated group rather than to any particular axis for multidimensional axis groups. BeginMoveTo returns immediatly and does not wait for the motion to finish. For cases where it is important to “block” program execution until the end of the move use MoveTo instead of BeginMoveTo. Use MoveHasCompleted to determine when a move started with BeginMoveTo has finished.

Group numbers required for this routine are provided by the TNInit functions.

Binary Command Implementation

The following is a 5 axis implementation. Other axis counts would be implemented by changing the T5AxisVectorProcedure to the procedure with the correct dimension.

```
procedure dms_T5AxisBeginMoveTo(GroupNumber:word;
   Delta1:longint;
   Delta2:longint;
   Delta3:longint;
   Delta4:longint;
   Delta5:longint);

   begin
   dms_T5AxisVectorProcedure(bc_TNAxisBeginMoveTo,GroupNumber,
      Delta1,Delta2,Delta3,Delta4,Delta5);
   end;
```

Errors

BeginMoveTo will escape if while in motion the destination specified is "behind" the vector path position or if the destination is so close that the receiver cannot accomplish the move at the specified decel rate. In these cases an escape will occur with MotionOverrunEscapeCode and the receiver will stop.

ActiveX Example

```
Msb.T2AxisBeginMoveTo XYTable, 0,0
```

SeeAlso

TNAxis.SetAccel
TNAxis.SetDecel
TNAxis.SetSpeed
TNAxis.MoveIsFinished
TNAxis.MoveBy
TNAxis.MoveTo
TNAxis.BeginMoveTo
MotionOverrunEscapeCode

TNAxisDispose

ActiveX Syntax

```
Public Sub TNAxisDispose(ByVal GroupNumber as Integer)
```

C Syntax

```
void dms_TNAxisDispose(int GroupNumber)
```

Pascal Syntax

```
procedure dms_TNAxisDipose(GroupNumber:integer);
```

Description

The Motion Server command set is designed to support multiple client programs. Coordinated motion is described by using GroupNumbers provided by TNAxisInit routines. When a client program is done and exiting the client needs to tell Motion Server that it is finished with the resources that were allocated so that another program can use them. This is done with dms_TNAxisDispose. dms_TNAxisDispose is analagous to releasing memory after use so as to prevent a "memory leak". If dms_TNAxisDispose is not used, an "axis leak" will occur and eventually Motion Server will indicate that there are no more axis groups available for use.

ActiveX Example

```
Msb.TNAxisDispose XYTable
```

DLL Example

```
procedure PerformCoordinatedActivity;
   var Group:integer;

   begin
   dms_SetMotorType(1,ServoMotor);
   dms_SetMotorType(2,ServoMotor);
   Group:=dms_T2AxisInit(1,2);
   dms_SetAccel(Group,20000);
   dms_SetDecel(Group,20000);
   dms_SetSpeed(Group,1000);
   dms_SetMotor(Group,on);
   dms_T2AxisMoveBy(1000,2000);
   dms_TNAxisDispose(Group);
   end;
```

TNAxisInit

ActiveX Syntax

```
Public Function T2AxisInit(
   ByVal Axis1Number As Integer,
   ByVal Axis2Number As Integer)

Public Function T3AxisInit(
   ByVal Axis1Number As Integer,
   ByVal Axis2Number As Integer,
   ByVal Axis3Number As Integer)

Public Function T4AxisInit(
   ByVal Axis1Number As Integer,
   ByVal Axis2Number As Integer,
   ByVal Axis3Number As Integer,
   ByVal Axis4Number As Integer)

Public Function T5AxisInit(
   ByVal Axis1Number As Integer,
   ByVal Axis2Number As Integer,
   ByVal Axis3Number As Integer,
   ByVal Axis4Number As Integer,
   ByVal Axis5Number As Integer)

Public Function T6AxisInit(
   ByVal Axis1Number As Integer,
   ByVal Axis2Number As Integer,
   ByVal Axis3Number As Integer,
   ByVal Axis4Number As Integer,
   ByVal Axis5Number As Integer,
   ByVal Axis6Number As Integer)
```

C Syntax

```
int dms_T2AxisInit(
  int Axis1Number,
  int Axis2Number)

int dms_T3AxisInit(
  int Axis1Number,
  int Axis2Number,
  int Axis3Number)

int dms_T4AxisInit(
  int Axis1Number,
  int Axis2Number,
  int Axis3Number,
  int Axis4Number)

int dms_T5AxisInit(
  int Axis1Number,
  int Axis2Number,
  int Axis3Number,
  int Axis4Number,
  int Axis5Number)

int dms_T6AxisInit(
  int Axis1Number,
  int Axis2Number,
  int Axis3Number,
  int Axis4Number,
  int Axis5Number,
  int Axis6Number)
```

Pascal Syntax

```
function T2AxisInit(
   Axis1Number:word;
   Axis2Number:word):word;

function T3AxisInit(
   Axis1Number:word;
   Axis2Number:word;
   Axis3Number:word):word;

function T4AxisInit(
   Axis1Number:word;
   Axis2Number:word;
   Axis3Number:word;
   Axis4Number:word):word;

function T5AxisInit(
   Axis1Number:word;
   Axis2Number:word;
   Axis3Number:word;
   Axis4Number:word;
   Axis5Number:word):word;

function T6AxisInit(
   Axis1Number:word;
   Axis2Number:word;
   Axis3Number:word;
   Axis4Number:word;
   Axis5Number:word;
   Axis6Number):word;
```

Description

TNAxisInit is used to associate axes into a coordinated group and returns a Group Number to reference the group in the future. In actual use, the "N" in TNAxisInit is replaced by the dimension of the group being constructed, i.e. T2AxisInit for a 2 axis group or T6AxisInit for a 6 axis group. In coordinated motion commands, the group number is the "handle" that refers to this particular axis association. The axis are specified with their axis numbers ranging from 1 to 16. The routine requiers as many axis parameters as dimension of the group being constructed. The order the axes are indicated here becomes the order of parameters used to describe coordinated motion. The first axis listed here receives the first coordinate number in motion commands. Coordinated motion can only be performed on groups that have been initialized.

Binary Command Implementation

```
function dms_TNAxisInit(
   Axis1Number:word;
   Axis2Number:word;   {possibly other axis parameters}

   var ErrorCode:integer):word;

   begin
   if ErrorCode <> 0 then
      exit;
   FifoReset;
   FifoWriteWord(bc_T4AxisInit); {or T2AxisInit etc.}
   FifoWriteWord(Axis1Number);
   FifoWriteWord(Axis2Number);
   {possibly other writes for other dimensions}
   FifoSendMessageAndWaitForResponse;
   ErrorCode:=FifoReadWord;
   TNAxisInit:=FifoReadWord;
   end;
```

ActiveX Example

```
XYTable=Msb.T2AxisInit 1,2
```

DLL Example

```
GroupWith4Axes:=T4AxisInit(1,2,3,4);
GroupWith6Axes:=T6AxisInit(5,6,7,8,9,10);
```

TNAxisMoveBy

ActiveX Syntax

```
Public Sub T2AxisMoveBy(ByVal GroupNumber as Integer,
   ByVal Delta1 As Long,
   ByVal Delta2 As Long)

Public Sub T3AxisMoveBy(ByVal GroupNumber as Integer,
   ByVal Delta1 As Long,
   ByVal Delta2 As Long,
   ByVal Delta3 As Long)

Public Sub T4AxisMoveBy(ByVal GroupNumber as Integer,
   ByVal Delta1 As Long,
   ByVal Delta2 As Long,
   ByVal Delta3 As Long,
   ByVal Delta4 As Long)

Public Sub T5AxisMoveBy(ByVal GroupNumber as Integer,
   ByVal Delta1 As Long,
   ByVal Delta2 As Long,
   ByVal Delta3 As Long,
   ByVal Delta4 As Long,
   ByVal Delta5 As Long)

Public Sub T6AxisMoveBy(ByVal GroupNumber as Integer,
   ByVal Delta1 As Long,
   ByVal Delta2 As Long,
   ByVal Delta3 As Long,
   ByVal Delta4 As Long,
   ByVal Delta5 As Long,
   ByVal Delta6 As Long)
```

C Syntax

```
void dms_T1AxisMoveBy(int GroupNumber,
  long DeltaPosition)

void dms_T2AxisMoveBy(int GroupNumber,
  long DeltaPosition1,
  long DeltaPosition2)

void dms_T3AxisMoveBy(int GroupNumber,
  long DeltaPosition1,
  long DeltaPosition2,
  long DeltaPosition3)

void dms_T4AxisMoveBy(int GroupNumber,
  long DeltaPosition1,
  long DeltaPosition2,
  long DeltaPosition3,
  long DeltaPosition4)

void dms_T5AxisMoveBy(int GroupNumber,
  long DeltaPosition1,
  long DeltaPosition2,
  long DeltaPosition3,
  long DeltaPosition4,
  long DeltaPosition5)

void dms_T6AxisMoveBy(int GroupNumber,
  long DeltaPosition1,
  long DeltaPosition2,
  long DeltaPosition3,
  long DeltaPosition4,
  long DeltaPosition5,
  long DeltaPosition6)
```

Pascal Syntax

```
procedure T1AxisMoveBy(AxisNumber:word;
   DeltaPosition:longint);

procedure T2AxisMoveBy(GroupNumber:word;
   DeltaPosition1:longint;
   DeltaPosition2:longint);

procedure T3AxisMoveBy(GroupNumber:word;
   DeltaPosition1:longint;
   DeltaPosition2:longint;
   DeltaPosition3:longint);

procedure T4AxisMoveBy(GroupNumber:word;
   DeltaPosition1:longint;
   DeltaPosition2:longint;
   DeltaPosition3:longint;
   DeltaPosition4:longint);

procedure T5AxisMoveBy(GroupNumber:word;
   DeltaPosition1:longint;
   DeltaPosition2:longint;
   DeltaPosition3:longint;
   DeltaPosition4:longint;
   DeltaPosition5:longint);

procedure T6AxisMoveBy(GroupNumber:word;
   DeltaPosition1:longint;
   DeltaPosition2:longint;
   DeltaPosition3:longint;
   DeltaPosition4:longint;
   DeltaPosition5:longint;
   DeltaPosition6:longint);
```

Description

TNAxisMoveBy performs a relative coordinated move by the specified position deltas. In actual use, the "N" in TNAxis... is replaced by the dimension of the group being directed, i.e. T3AxisMoveBy for a 3 axis group. The method requires as many parameters as the dimension of the receiver axis group, ie a 2 axis group requires 2 parameters, a 4 axis group requires 4 parameters. The motion is performed with a trapezoidal velocity profile based on parameters set with the SetAccel , SetDecel , and SetSpeed methods. These parameters apply to the vector path motion of the coordinated group rather than to any particular axis. MoveBy does not return until the motion has been accomplished. “Blocking” program execution until the end of the move may be helpful for synchronizing the next event, ie don’t drill the hole until you get to the destination. Some appllications need to continue execution even though the destination has not yet been achieved. For these cases use BeginMoveBy which starts the move and immediately returns to continue with the next instruction.

Group Numbers are provided by TNAxisInit routines.

Binary Command Implementation

The following is a 2 axis implementation. Other axis counts would be implemented by changing the T2AxisVectorProcedure to the procedure with the correct dimension.

```
procedure dms_T2AxisMoveBy(GroupNumber:word;
   Delta1:longint;
   Delta2:longint);

   begin
   dms_T2AxisVectorProcedure(bc_TNAxisMoveBy,GroupNumber,
      Delta1,Delta2);
   end;
```

ActiveX Example

```
Msb.T2AxisMoveBy XYTable, 2000, 4000
```

DLL Example

```
T3AxisMoveBy(My3AxisGroup,2000,4000,6000);
T5AxisMoveBy(My5AxisGroup,-1000,0,2000,1000,5000);
```

Errors

MoveBy will escape if while in motion the new destination specified is "behind" the vector path position or if the destination is so close that the axis group cannot accomplish the move at the specified decel rate. In these cases the group will emit a MotionOverrunEscapeCode and come to a stop.

SeeAlso

TNAxis.SetAccel
TNAxis.SetDecel
TNAxis.SetSpeed
TNAxis.MoveTo
MotionOverrunEscapeCode

TNAxisMoveTo

ActiveX Syntax

```
Public Sub T2AxisMoveTo(ByVal GroupNumber as Integer,
   ByVal Destination1 As Long,
   ByVal Destination2 As Long)

Public Sub T3AxisMoveTo(ByVal GroupNumber as Integer,
   ByVal Destination1 As Long,
   ByVal Destination2 As Long,
   ByVal Destination3 As Long)

Public Sub T4AxisMoveTo(ByVal GroupNumber as Integer,
   ByVal Destination1 As Long,
   ByVal Destination2 As Long,
   ByVal Destination3 As Long,
   ByVal Destination4 As Long)

Public Sub T5AxisMoveTo(ByVal GroupNumber as Integer,
   ByVal Destination1 As Long,
   ByVal Destination2 As Long,
   ByVal Destination3 As Long,
   ByVal Destination4 As Long,
   ByVal Destination5 As Long)

Public Sub T6AxisMoveTo(ByVal GroupNumber as Integer,
   ByVal Destination1 As Long,
   ByVal Destination2 As Long,
   ByVal Destination3 As Long,
   ByVal Destination4 As Long,
   ByVal Destination5 As Long,
   ByVal Destination6 As Long)
```

C Syntax

```
void dms_T1AxisMoveTo(int AxisNumber,
  long Destination)

void dms_T2AxisMoveTo(int GroupNumber,
  long Destination1,
  long Destination2)

void dms_T3AxisMoveTo(int GroupNumber,
  long Destination1,
  long Destination2,
  long Destination3)

void dms_T4AxisMoveTo(int GroupNumber,
  long Destination1,
  long Destination2,
  long Destination3,
  long Destination4)

void dms_T5AxisMoveTo(int GroupNumber,
  long Destination1,
  long Destination2,
  long Destination3,
  long Destination4,
  long Destination5)

void dms_T6AxisMoveTo(int GroupNumber,
  long Destination1,
  long Destination2,
  long Destination3,
  long Destination4,
  long Destination5,
  long Destination6)
```

Pascal Syntax

```
procedure T1AxisMoveTo(AxisNumber:word;
   Destination:longint);

procedure T2AxisMoveTo(GroupNumber:word;
   Destination1:longint;
   Destination2:longint);

procedure T3AxisMoveTo(GroupNumber:word;
   Destination1:longint;
   Destination2:longint;
   Destination3:longint);

procedure T4AxisMoveTo(GroupNumber:word;
   Destination1:longint;
   Destination2:longint;
   Destination3:longint;
   Destination4:longint);

procedure T5AxisMoveTo(GroupNumber:word;
   Destination1:longint;
   Destination2:longint;
   Destination3:longint;
   Destination4:longint;
   Destination5:longint);

procedure T6AxisMoveTo(GroupNumber:word;
   Destination1:longint;
   Destination2:longint;
   Destination3:longint;
   Destination4:longint;
   Destination5:longint;
   Destination6:longint);
```

Description

TNAxisMoveTo performs an absolute coordinated move to the specified destination. In actual use, the "N" in TNAxis is replaced by the dimension of the group, i.e. T2AxisMoveTo. The number of parameters provided is the same as the dimension of the axis group, ie a 2 axis group requires 2 parameters, a 4 axis group requires 4 parameters. The motion is performed with a trapezoidal velocity profile based on parameters set with the SetAccel , SetDecel , and SetSpeed methods. These parameters apply to the vector path motion of the coordinated group rather than to any particular axis. TNAxisMoveTo does not return until the motion has been accomplished. “Blocking” program execution until the end of the move may be helpful for synchronizing the next event, ie don’t drill the hole until you get to the destination. Some appllications need to continue execution even though the destination has not yet been achieved. For these cases use TNAxisBeginMoveTo which starts the move and immediately returns to continue with the next instruction.

Binary Command Implementation

The following is a 3 axis implementation. Other axis counts would be implemented by changing the T3AxisVectorProcedure to the procedure with the correct dimension.

```
procedure dms_T3AxisMoveTo(GroupNumber:word;
   Destination1:longint;
   Destination2:longint;
   Destination3:longint);

   begin
   dms_T3AxisVectorProcedure(bc_TNAxisMoveTo,GroupNumber,
      Destination1,Destination2,Destination3);
   end;
```

ActiveX Example

```
Msb.T2AxisMoveTo XYTable, 1, 2
```

DLL Example

```
T2AxisMoveTo(My2AxisGroupNumber,2000,4000,ErrorCode);
T4AxisMoveTo(Another4AxisGroupNumber,0,0,ErrorCode);
```

Errors

MoveTo will escape if while in motion the destination specified is "behind" the vector path position or if the destination is so close that the axis group cannot accomplish the move at the specified decel rate. In these cases the group will emit a MotionOverrunEscapeCode and come to a stop.

SeeAlso

TNAxis.SetAccel
TNAxis.SetDecel
TNAxis.SetSpeed
MotionOverrunEscapeCode
TNAxis.MoveBy

UserBoolean

ActiveX Syntax

```
Public Function UserBoolean(ByVal Index As Long) As Boolean
```

C Syntax

```
int dms_UserBoolean(int Index)
```

Pascal Syntax

```
function dms_UserBoolean(Index:integer):boolean;
```

Description

UserBoolean queries the value of the boolean variable in Motion Server at the specified index. User variables are used to communicate data between the host and tasks operating on the Motion Server card.

Binary Command Implementation

```
function dms_UserBoolean:boolean;
   begin
   dms_UserBoolean:=
      dms_BooleanFunctionIntegerParam(bc_UserBoolean,Index);
   end;
```

ActiveX Example

```
Status.Caption=Msb.UserBoolean 4
```

See Also

SetUserBoolean

UserHasDisabled

ActiveX Syntax

```
Public Function UserHasDisabled as Boolean
```

C Syntax

```
int dms_UserHasDisabled()
```

Pascal Syntax

```
function dms_UserHasDisabled:boolean;
```

Description

If the User Disable input is not held low this function returns true indicating that the user is attempting to disable the system.

Binary Command Implementation

```
function dms_UserHasDisabled:boolean;
   begin
   dms_UserHasDisabled:=dms_BooleanFunction(bc_UserHasDisabled);
   end;
```

ActiveX Example

```
if Msb.UserHasDisabled then
   MsbBox "Release EStop Switch"
End If
```

See Also

ResetWatchdog
WatchdogHasTripped

UserLongint

ActiveX Syntax

```
Public Function UserLongint(ByVal Index As Long) As Long
```

C Syntax

```
long dms_UserLongint(int Index)
```

Pascal Syntax

```
function dms_UserLongint(Index:integer):Longint;
```

Description

UserLongint queries the value of the longint variable in Motion Server at the specified index. User variables are used to communicate data between the host and tasks operating on the Motion Server card.

Binary Command Implementation

```
function dms_UserLongint(Index:integer):longint;
   begin
   if ErrorCode <> 0 then
      begin
      dms_UserLongint:=0;
      exit;
      end;
   FifoReset;
   FifoWriteWord(bc_UserLongint);
   FifoWriteWord(Index);
   FifoSendMessageAndWaitForResponse;
   ErrorCode:=FifoReadWord;
   if ErrorCode=0 then
      dms_UserLongint:=FifoReadLongint
   else
      dms_UserLongint:=0;
   end;
```

ActiveX Example

```
Status.Caption=Msb.UserLongint 4
```

See Also

SetUserLongint

UserSingle

ActiveX Syntax

```
Public Function UserSingle(ByVal Index As Long) As Single
```

C Syntax

```
single dms_UserSingle(int Index)
```

Pascal Syntax

```
function dms_UserSingle(Index:integer):Single;
```

Description

UserSingle queries the value of the single precision floating point variable in Motion Server at the specified index. User variables are used to communicate data between the host and tasks operating on the Motion Server card.

Binary Command Implementation

```
function dms_UserSingle(Index:integer):single;

   begin
   if ErrorCode <> 0 then
      begin
      dms_UserSingle:=0;
      exit;
      end;
   FifoReset;
   FifoWriteWord(bc_UserSingle);
   FifoWriteWord(Number);
   FifoSendMessageAndWaitForResponse;
   ErrorCode:=FifoReadWord;
   if ErrorCode=0 then
      dms_UserSingle:=FifoReadSingle
   else
      dms_UserSingle:=0;
  end;
```

ActiveX Example

```
Status.Caption=Msb.UserSingle 7
```

See Also

SetUserSingle

WatchdogHasTripped

ActiveX Syntax

```
Public Function WatchdogHasTripped () as Boolean
```

C Syntax

```
int dms_WatchdogHasTripped()
```

Pascal Syntax

```
function dms_WatchdogHasTripped:boolean;
```

Definition

The watchdog safety system will shut down servo activity if the processor fails to respond to the timer event correctly. This function indicates if the watchdog system has shut down activity.

Binary Command Implementation

```
function dms_WatchdogHasTripped:boolean;
   begin
   dms_WatchdogHasTripped:=
      dms_BooleanFunction(bc_WatchdogHasTripped);
   end;
```

ActiveX Example

```
if Msb.WatchdogHasTripped then
   MsgBox "Safety System Shutdown Machine"
End If
```

DLL Example

```
....
If WatchdogHasTripped then
  Writeln('System has shutdown');
....
```

See Also

ResetWatchdog

Zero

ActiveX Syntax

```
Public Function Zero(ByVal AxisNumber as Integer) As Integer
```

C Syntax

```
int dms_Zero(int AxisNumber)
```

Pascal Syntax

```
function dms_Zero(AxisNumber:integer):integer;
```

Description

Motion Server and SI-3000 implement PID control. This function returns the current value of the control law zero, one of the primary compensation parameters.

Binary Command Implementation

```
function dms_Zero(AxisNumber:integer):integer;
   begin
   dms_Zero:=dms_AxisIntegerFunction(bc_Zero,AxisNumber);
   end;
```

ActiveX Example

```
Status.Caption=Msb.Zero 3
```

See Also

Gain
Integrator
SetIntegrator
SetGain
SetZero

Chapter 4

4) Visual Basic DLL Examples

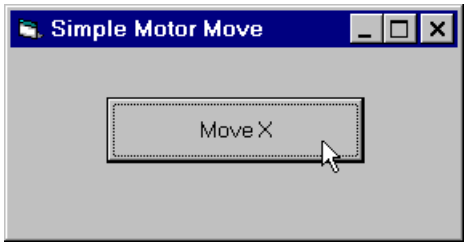
Objective

The following examples illustrate how to use the command set to perform basic motion. The basic pattern of use is to successfully allocate a channel of communication and select binary communication during machine startup, send motion commands to perform machine operation during machine operation, and release the channel when the application is finished.

The examples shown are done with Visual Basic 5.0 Professional Edition. The only step needed to include the motion commands into your Visual Basic project is to select from the Visual Basic Menu Project|Add Module and choose the DMS_VB32.BAS file located in the C:\DOULOI\DMS_BC32 directory. This file contains declarations that refer to the actual motion commands found in the DMS_VB32.DLL stored in the Windows directory.

Setting Controller Parameters and Performing Motion

The following program sets motor 1 to to be a stepper motor, turns the motor on, sets the accel, decel, and speed, and moves by 2000 steps. The application looks like the following:



It is necessary to remember if a successful connection has been made to the controller to know if submitting commands is legal or not. To remember the condition of the channel a variable will be declared in the Declarations section of the module:

```
dim DmsOpen as Boolean
```

This boolean is then used in the Form Load procedure in the following way:

```
Private Sub Form_Load()
DmsOpen = dms_AllocateChannel
If Not DmsOpen Then
   MsgBox "Unable to Allocate Channel"
Else
   dms_SelectBinaryCommunication
   dms_ResetErrorCode
   dms_SetMotorType 1, 0
   dms_SetSpeed 1, 1000
   dms_SetAccel 1, 10000
   dms_SetDecel 1, 10000
   dms_SetMotor 1, 1
   If dms_ErrorCode <> 0 Then
      MsgBox "Error in Controller Setup"
End If
End Sub
```

The dms_AllocateChannel command is used first. This returns a value of zero (for false), or not-zero (for true) indicating success in allocating a channel. If the channel does not allocate then the function dms_ErrorCode explains why. The channel might not allocate for a number of reasons including:

Error 1009 - Windows driver not properly setup

Error 1010 - Communication Timeout

Error 1011 - Controller is not present in computer

After getting a "true" value for dms_AllocateChannel, the command dms_SelectBinaryCommunication is used. Presently, this function does not perform a function but is a place-holder in application programs for future compatibility with other interpreter formats.

The command dms_ResetErrorCode is used to clear any pending errors that may have been reported. Once an error occurs, as reported through the dms_ErrorCode function, subsequent motion commands will be ignored. The error code should be checked and if not 0, reset with the command dms_ResetErrorCode during program execution as part of the host program error management strategy.

The commands that follow are used to initialize the first axis in the system. SetMotorType is used to configure axis 1 to be a stepper motor. Speed, Accel, and Decel settings are made for the motor. The motor is then turned on and is ready for movement.

On the form is a Move X button which has the following click procedure:

```
Private Sub TestButton_Click()
if DmsOpen then
  dms_T1AxisBeginMoveBy 1, 400
End If
End Sub
```

This button checks to see if the channel is open and if it is performs the command, dms_T1AxisBeginMoveBy, a relative move command for axis 1 to produce movement of 400 counts. BeginMoveBy immediately returns flow to VisualBasic after starting the motion of the axis. BeginMoveBy does not wait for the move to finish.

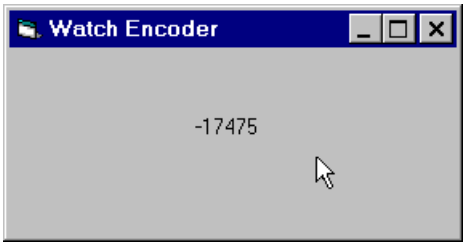
The form terminate procedure looks like the following:

```
Private Sub Form_Terminate()
If DmsOpen then
  dms_SetMotor 1, 0
  dms_ReleaseChannel
End If
End Sub
```

Here the motor is turned off and the command dms_ReleaseChannel is used to indicate that the host program is done with the controller.

Monitoring Controller Status

The following program continually reports the position of the axis 1 encoder:



The form load and form terminate methods for this are the same as for the first example. The display control is named "Status". The activity of reporting the position is performed by a timer routine with the following timer event:

```
Private Sub Timer1_Timer()
If DmsOpen Then
  Status = dms_EncoderPosition(1)
End If
End Sub
```

The dynamic link library currently being supplied is not re-entranct. Accordingly it is not appropriate to send dms commands from separate VB threads however timer events que in the Windows message loop permitting status timer events to monitor the controller while motion commands are also sent to the controller.

Chapter 5

5) C Language DLL Examples

Objective

These examples illustrate the general pattern of use of the driver interface. Examples also illustrate particular controller functions. If there is an example you would like to see that is not present, please contact Douloi for sample code (and look for your question-and-answer in the next version of the manual!).

C Example Framework

These examples were compiled with Turbo C++ for Windows. A Windows application named BIN_DEMO.CPP was made using Object Windows Library that contains a single "Start" menu selection. This start menu causes the following sections of code to operate and illustrate operation of the binary command functions. It is not necessary to understand this OWL application to benefit from the motion behavior that is consolidated in the CMStart function.

```
// Borland Turbo C++ Example Framework

// The following example is based on a Borland
// Object Windows Library demo program and must
// be compiled with OWL

#include <owl.h>
#include "g:\bin_cmnd\bin_demo.h"
#include "g:\bin_cmnd\bin_cmnd.cpp"


class TMotionApp : public TApplication {
public:
 TMotionApp(LPSTR Name, HINSTANCE hInstance,
             HINSTANCE hPrevInstance, LPSTR lpCmd,
             int nCmdShow)
            : TApplication(Name, hInstance,
                        hPrevInstance, lpCmd, nCmdShow) {};
 virtual void InitMainWindow();
};

class TExampleWindow : public TWindow {
public:
 TExampleWindow(PTWindowsObject AParent, LPSTR ATitle);
 virtual void CMStart(TMessage& Msg) = [CM_FIRST + CM_START];
};
```

```
TExampleWindow::TExampleWindow(PTWindowsObject AParent, LPSTR ATitle)
  : TWindow(AParent, ATitle)
{
 AssignMenu("COMMANDS");
}

void TExampleWindow::CMStart(TMessage&)
{
  /* Example Code goes in here */
}

void TMotionApp::InitMainWindow()
{
 MainWindow = new TExampleWindow(NULL, "Driver Example");
}

int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                LPSTR lpCmd, int nCmdShow)
{
 TMotionApp MotionApp("Driver Example", hInstance, hPrevInstance,
             lpCmd, nCmdShow);
 MotionApp.Run();
 return(MotionApp.Status);
}
```

Setting Controller Parameters and Performing Motion

This program sets motor 1 to be a stepper motor, turns the motor on, sets the acceleration, deceleration, and speed, and causes the motor to move by 2000 steps.

```
void TExampleWindow::CMStart(TMessage&)
{
 if (! dms_AllocateChannel())
    {
  MessageBox(HWindow,"Unable to Allocate Channel","Error",MB_OK);
    return;
    };

 dms_ResetErrorCode();
 dms_SelectBinaryCommunication();
 dms_SetMotorType(1,StepperMotor);
 dms_SetMotor(1,1);
 dms_SetAccel(1,10000);
 dms_SetDecel(1,10000);
 dms_SetSpeed(1,1000);
 dms_T1AxisMoveBy(1,2000);
 dms_ReleaseChannel();

  if (dms_ErrorCode != 0)
   MessageBox(HWindow,"Motion Problem","Error",MB_OK);
  else
   MessageBox(HWindow,"Test Done","Status",MB_OK);
}
```

Single Axis Motion Pattern

The following program directs Axis 1 to operate as a stepper motor and move in a multiple move pattern.

```
void TExampleWindow::CMStart(TMessage&)
{
 if (! dms_AllocateChannel())
    {
   MessageBox(HWindow,"Unable to Allocate Channel","Error",MB_OK);
    return;
    };

 dms_ResetErrorCode();
 dms_SelectBinaryCommunication();

 dms_ResetWatchdog();
 dms_SetMotorType(1,StepperMotor);
 dms_SetAccel(1,20000);
 dms_SetDecel(1,20000);
 dms_SetSpeed(1,1500);
 dms_SetMotor(1,1);

 dms_SetActualPosition(1,0);
 dms_T1AxisMoveBy(1,2000);
 dms_T1AxisMoveBy(1,-4000);
 dms_T1AxisMoveTo(1,0);
 dms_SetMotor(1,0);

 dms_ReleaseChannel();

  if (dms_ErrorCode != 0)
   MessageBox(HWindow,"Motion Problem","Error",MB_OK);
  else
   MessageBox(HWindow,"Test Done","Status",MB_OK);
}
```

Coordinated Motion

This program performs coordinated motion and produces a diamond shape with axis 1 and 2. These axis are associated into a coordianted group that is referenced with the name "DiamondAxes".

```
void TExampleWindow::CMStart(TMessage&)
{
 int DiamondAxes;

 if (! dms_AllocateChannel())
    {
  MessageBox(HWindow,"Unable to Allocate Channel","Error",MB_OK);
    return;
    };

 dms_ResetErrorCode();
 dms_SelectBinaryCommunication();

 dms_SetMotorType(1,StepperMotor);
 dms_SetMotorType(2,StepperMotor);


 DiamondAxes=dms_T2AxisInit(1,2);
 dms_SetAccel(DiamondAxes,20000);
 dms_SetDecel(DiamondAxes,20000);
 dms_SetSpeed(DiamondAxes,1500);
 dms_SetMotor(DiamondAxes,1);

 dms_T2AxisMoveBy(DiamondAxes,2000,2000);
 dms_T2AxisMoveBy(DiamondAxes,-2000,2000);
 dms_T2AxisMoveBy(DiamondAxes,-2000,-2000);
 dms_T2AxisMoveBy(DiamondAxes,2000,-2000);

 dms_SetMotor(DiamondAxes,0);
 dms_ReleaseChannel();
  if (dms_ErrorCode != 0)
   MessageBox(HWindow,"Motion Problem","Error",MB_OK);
  else
   MessageBox(HWindow,"Test Done","Status",MB_OK);
}
```

Curved Motion

This program performs coordinated motion along a continuous curved path

```
void TExampleWindow::CMStart(TMessage&)
{
 int CurveAxes;

 if (! dms_AllocateChannel())
    {
  MessageBox(HWindow,"Unable to Allocate Channel","Error",MB_OK);
   return;
    };

 dms_ResetErrorCode();
 dms_SelectBinaryCommunication();

 dms_SetMotorType(1,StepperMotor);
 dms_SetMotorType(2,StepperMotor);

 CurveAxes=dms_T2AxisInit(1,2);
 dms_SetAccel(CurveAxes,20000);
 dms_SetDecel(CurveAxes,20000);
 dms_SetSpeed(CurveAxes,1500);
 dms_SetMotor(CurveAxes,1);

 dms_Clear(CurveAxes);
 dms_T2AxisAppendMoveBy(CurveAxes,2000,0);
 dms_T2AxisAppendArc(CurveAxes,500,0,-90);
 dms_T2AxisAppendMoveBy(CurveAxes,0,2000);
 dms_T2AxisAppendArc(CurveAxes,500,90,-90);
 dms_T2AxisAppendMoveBy(CurveAxes,-2000,0);
 dms_T2AxisAppendArc(CurveAxes,500,180,-90);
 dms_T2AxisAppendMoveBy(CurveAxes,0,-2000);
 dms_T2AxisAppendArc(CurveAxes,500,270,-90);
 dms_BeginMoveAlongCurve(CurveAxes);

/* check for completion with dms_MoveIsFinished(CurveAxes) and do other
things */

 dms_SetMotor(CurveAxes,0);
 dms_TNAxisDispose(CurveAxes);
 dms_ReleaseChannel();
  if (dms_ErrorCode != 0)
   MessageBox(HWindow,"Motion Problem","Error",MB_OK);
  else
   MessageBox(HWindow,"Test Done","Status",MB_OK);
}
```

Chapter 6

6) Pascal DLL Examples

Objective

Chapter 3 explains command operation by showing the binary command structure in terms of this abstraction. Peeking ahead to commands described in chapter 3, the following examples illustrate the sequence of use of the hardware abstraction shown. Normally the channel is allocated when a program starts and released before the program closes. Here the entire process is shown together only to illustrate a complete communication exercise in a minimum example.

Setting Controller Parameters and Performing Motion

The following program sets motor to to be a stepper motor, turns the motor on, sets the accel, decel, and speed, and moves by 2000 steps.

```
program PerformStepperMoves;

include HW_ABS.INC

begin
ChannelAddress:=GetNextChannelAddress;
if ChannelAddress=0 then
   begin
   writeln('Unable to allocate channel');
   exit;
   end;
ClearErrorCode;
SelectBinaryCommunication;

dms_SetMotorType(1,ServoMotor);
dms_SetAccel(1,20000);
dms_SetDecel(1,20000);
dms_SetSpeed(2000);
dms_SetMotor(1,On);
dms_T1AxisMoveBy(1,2000);

if ErrorCode <> 0 then
 writeln('Error encountered: ',ErrorCode);
ReleaseChannel;
end.
```

Single Axis Motion Pattern

The following program directs Axis 1 to operate as a stepper motor and move in a multiple move pattern.

```
program PerformStepperMoves;

include HW_ABS.INC

begin
ChannelAddress:=GetNextChannelAddress;
if ChannelAddress=0 then
   begin
   writeln('Unable to allocate channel');
   exit;
   end;
ClearErrorCode;
SelectBinaryCommunication;

dms_SetMotorType(1,ServoMotor);
dms_SetAccel(1,20000);
dms_SetDecel(1,20000);
dms_SetSpeed(2000);

dms_SetMotor(1,On);
dms_T1AxisMoveBy(1,2000);
dms_T1AxisMoveBy(1,-4000);
dms_T1MoveBy(1,2000);

dms_SetMotor(1,off);

if ErrorCode <> 0 then
 writeln('Error encountered: ',ErrorCode);
ReleaseChannel;
end.
```

Coordinated Motion

This program performs coordinated motion and produces a diamond shape with axis 1 and 2.

```
program MoveDiamond;

include HW_ABS.INC

var AxisScanner:integer;
var DiamondAxes:integer;

begin
ChannelAddress:=GetNextChannelAddress;
if ChannelAddress=0 then
   begin
   writeln('Unable to allocate channel');
   exit;
   end;
ClearErrorCode;
SelectBinaryCommunication;

for AxisScanner:=1 to 2 do
   begin
   dms_SetMotorType(AxisScanner,ServoMotor);
   dms_SetGain(AxisScanner,30);
   dms_SetZero(AxisScanner,232);
   dms_SetErrorLimit(AxisScanner,200);
   end;

DiamondAxes:=T2AxisInit(1,2);

dms_SetMotor(DiamondAxes,on);

dms_T2AxisMoveBy(DiamondAxes, 2000, 2000);
dms_T2AxisMoveBy(DiamondAxes, 2000,-2000);
dms_T2AxisMoveBy(DiamondAxes,-2000,-2000);
dms_T2AxisMoveBy(DiamondAxes,-2000, 2000);

dms_SetMotor(DiamondAxes,off);

if ErrorCode <> 0 then
  writeln('Error encountered: ',ErrorCode);
ReleaseChannel;
end.
```

Chapter 7

7) Visual Basic ActiveX Examples

Objective

The following examples illustrate how to use the ActiveX control to perform basic motion. The steps required to setup and use the Ethernet systems are shown and described.

Preparing the Host for Ethernet Communication

The design intent is for the host computer to communicate to an ethernet-equipped Motion Server Block on an exclusive network composed of just the host, a cross-over ethernet cable, and the MSB.

The host must share the same "subnet" as the MSB. The MSB IP address is 84.83.83.1. Douloi Automation suggests that you use a host IP address of 84.83.83.2. To change the host IP address right click on "Network Neighborhood", Select the TCP/IP configuration that is associated with the ethernet card you will plug your cable into and select "Properties". From the properties page check "Specify an IP Address". Use the IP address 84.83.83.1 and the subnet mask 255.0.0.0.

Install the provided crossover cable in the host and place the other end in the Motion Server Block. (The label on side of the MSB indicates "Profibus" but it is the ethernet RJ45 connector)

Preparing Visual Basic to use the ActiveX Control

The provided ActiveX control can be placed onto the component bar for the current project by right clicking on the component tool bar and select "Components....". From the Components List select "Browse" and choose the file MSBControl.OCX found on the CD-Rom in the ActiveX Directory

Checking the Ethernet Setup

To confirm the setup is working properly run the EthernetCheck.exe program on the CD-Rom in the ActiveX directory and press the "Check" button. If all is well a message box should appear indicating that the communication test passed.

Returning to Ethernet Use After Using SAW

Servo Application Workbench is available for system setup and checkout by following instructions in the Servo Application Workbench manual. Using SAW turns off any programs running in Motion Server Block and replaces them with new programs that might be used for system checkout and diagnostics. When returning to ethernet use, power cycle the Motion Server Block. This will cause the on-board ethernet communication application to restart as the default application.

Simple Motion

The first application illustrates the minimum number of commands to perform motion. Create a standard application, place the MSBControl into the application and give it the name "Msb". In the Form Load procedure place the following code

```
Msb.Init
```

Create a button and place in the button the following commands:

```
Msb.SetMotor 1, true
Msb.SetSpeed 1, 1000
Msb.SetAccel 1, 10000
Msb.SetDecel 1, 10000
Msb.BeginMoveBy 1, 1000
Msb.PerformBuffer
While Msb.Busy
  DoEvents
WEnd
```

Monitoring Controller Status

You can monitor the position of a motor by using the following command in a timer routine.

Status.Caption=Msb.ActualPosition 1

The MsbControl is not able to handle commands from more than one activity at once. During the section of code in a timer routine that uses the MsbControl you must insure that commands are not being sent to the control elsewhere. This is most easily handled by having a "mutual exclusion" flag that you use. Before a section of code where commands will be sent, wait for the flag to become false and then set the flag true. When leaving that code section, set the flag false. If all control users check the flag and wait for it to become available then we insure that only one activity is accessing the Msb control at a a time.

Coordinated XY Motion

To perform XY coordination we must first describe a coordinated group. This is done with the T2AxisInit command. T2AxisInit returns a "handle" that we can use to reference a particular coordinated group in the controller. In the Form Load we would now have the following command:

```
Msb.Init
Msb.ResetAllocation
XYTable=Msb.T2AxisInit 1,2
```

Where XYTable is an integer that was defined earlier in the project. ResetAllocation is used to make sure that a full set of resources if available in the controller for allocation in future Init requests. XYTable now has a "handle" that will be used to represent the group in future commands.

In a button procedure type in the following commands

```
Msb.SetMotor XYTable, true
Msb.SetSpeed XYTable, 1000
Msb.SetAccel XYTable, 10000
Msb.SetDecel XYTable, 10000
Msb.BeginMoveBy XYTabe, 1000, 2000
Msb.PerformBuffer
While Msb.Busy
  DoEvents
WEnd
```

Circular Interpolation

Another command must be applied to our XYTable group to support vector and arc path descriptions. This command is "LinkToBuffer"

```
Msb.Init
Msb.ResetAllocation
XYTable=Msb.T2AxisInit 1,2
Msb.LinkToBuffer XYTable
```

Now we can describe a race-track style oval pattern using append commands and then perform the motion.

In a button procedure type in the following commands

```
Msb.SetMotor XYTable, true
Msb.SetSpeed XYTable, 1000
Msb.SetAccel XYTable, 10000
Msb.SetDecel XYTable, 10000
Msb.T2AxisAppendMoveBy XYTable, 0, 1000
Msb.T2AxisAppendArc XYTable, 1000, 0, 180
Msb.T2AxisAppendMoveBy XYTable, 0, -1000
Msb.T2AxisAppendArc XYTable, 1000, 180, 180
Msb.PerformBuffer
While Msb.Busy
  DoEvents
if Msb.ErrorCode <> 0 then
  MsbBox "Problem with Curved Motion"
WEnd
```

8) MSB Connections

Description

Cabling to Motion Server Block is performed through detachable screw terminals, ribbon cable, Mini-din, and Subminiature D cables.

Connector Layout

Most signals attach to screw terminal connectors on the front edge of Motion Server Block. The terminals are two-level. The left column describes signals on the upper level, the right column signals on the lower level. Parenthesis are for comment only and do not occupy connecton locations:

Label	Description	Label	Description
(Servo 1)		(Servo 2)	
Enc A+	Encoder 1 A +	Enc A+	Encoder 2 A+
Enc A-	Encoder 1 A-	Enc A-	Encoder 2 A-
Enc B+	Encoder 1 B+	Enc B+	Encoder 2 B+
Enc B-	Encoder 1 B-	Enc B-	Encoder 2 B-
Enc I+	Encoder 1 I+	Enc I+	Encoder 2 I+
Enc I-	Encoder 1 I-	Enc I-	Encoder 2 I-
+5V	+5 Volts	+5V	+5 Volts
Gnd	Ground	Gnd	Ground
Ena+	Enable 1 Plus	Ena+	Enable 2 Plus
Ena-	Enable 1 Minus	Ena-	Enable 2 Minus
Cmd+	Command 1+	Cmd+	Command 2+
Cmd-	Command 1 -	Cmd-	Command 2 -

(Servo 3)		(Servo 4)	
Enc A+	Encoder 3 A +	Enc A+	Encoder 4 A+
Enc A-	Encoder 3 A-	Enc A-	Encoder 4 A-
Enc B+	Encoder 3 B+	Enc B+	Encoder 4 B+
Enc B-	Encoder 3 B-	Enc B-	Encoder 4 B-
Enc I+	Encoder 3 I+	Enc I+	Encoder 4 I+
Enc I-	Encoder 3 I-	Enc I-	Encoder 4 I-
+5V	+5 Volts	+5	+5 Volts
Gnd	Ground	Gnd	Ground
Ena+	Enable 3 Plus	Ena+	Enable 4 Plus
Ena-	Enable 3 Minus	Ena-	Enable 4 Minus
Cmd+	Command 3+	Cmd+	Command 4+
Cmd-	Command 3 -	Cmd-	Command 4 -

(Step 5)		(Step 6)	
Step+	Step 5 Plus	Step+	Step 6 Plus
Step-	Step 5 Minus	Step-	Step 6 Minus
Dir+	Direction 5 +	Dir+	Direction 6 +
Dir-	Direction 5 -	Dir-	Direction 6 -
Ena	Amp Enable 5	Ena	Amp Enable 6

(Step 7)		(Step 8)	
Step+	Step 7 Plus	Step+	Step 8 Plus
Step-	Step 7 Minus	Step-	Step 8 Minus
Dir+	Direction 7 +	Dir+	Direction 8 +
Dir-	Direction 7 -	Dir-	Direction 8 -
Ena	Amp Enable 7	Ena	Amp Enable 8
+5V	+5 Volts	+5V	+5 Volts
Gnd	Logic Ground	Gnd	Logic Ground

(Power/Outputs/Inputs)			
Output 1	Isolated Output 1	Input 1	Isolated Input 1
Output 2	Isolated Output 2	Input 2	Isolated Input 2
Output 3	Isolated Output 3	Input 3	Isolated Input 3
Output 4	Isolated Output 4	Input 4	Isolated Input 4
Output 5	Isolated Output 5	Input 5	Isolated Input 5
Output 6	Isolated Output 6	Input 6	Isolated Input 6
Output 7	Isolated Output 7	Input 7	Isolated Input 7
Output 8	Isolated Output 8	Input 8	Isolated Input 8
Input 17	Isolated Input 17	Input 9	Isolated Input 9
Input 18	Isolated Input 18	Input 10	Isolated Input 10
EStop	Emergency Stop	Input 11	Isolated Input 11
In Common	Input Common	Input 12	Isolated Input 12
Output Pwr	Output Power +	Input 13	Isolated Input 13
Output Rtn	Output Return	Input 14	Isolated Input 14
Logic Pwr	Logic Power +	Input 15	Isolated Input 15
Logic Rtn	Logic Ground	Input 16	Isolated Input 16

Power and Isolated I/O Connector

There (2) 16 point detachable plugs for power and isolated IO. Motion Server Block permits having 3 isolated power systems: Controller Logic Power, Isolated Outputs, and Isolated Inputs. In some systems a single supply might be used for two or more of these separated systems compromising isolation for the economy of a reduced number of supplies.

Controller logic power is required at the points labeled "Logic Pwr" and "Logic Rtn". It is extremely important that power is not reversed on MSB. Reversed power may cause unrepairable damage to the controller. A diode has been incorporated into the power system which will attempt to short the incoming power if it is reversed. If the power supply fuse trips, double check for the possibility of reversed power. Power must be in the range 7 volts to 35 volts DC, 17 watts. Current required can be found by dividing the required wattage, 17, by the voltage. For example, at 24 volts, approximately 700 ma is required to run the controller.

Isolated outputs are available to provide 1 amp of current to loads. The outputs are "sourcing" style outputs. Both positive and return connections must be made to MSB for output power. This power should be connected to points "Out Pwr" and "Out Rtn". It is extremely important that output power is not reversed on MSB. Reversed power may cause unrepairable damage to the controller. The outputs are designed to handle DC loads only. For AC loads, use an opto-22 style block through the TTL IO system.

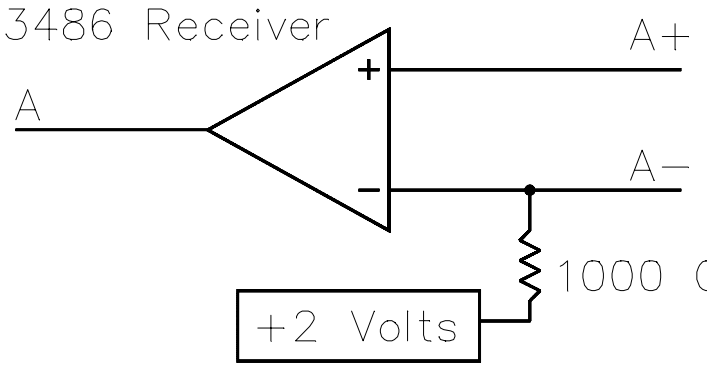
Isolated inputs are available which can be either sourcing or sinking. The sourcing or sinking style must be consistent across all of the inputs. The connection point "In Com" is used in conjunction with isolated inputs. If sinking style inputs are chosen, the "+" supply should connect to "In Com". If sourcing style inputs are to be used, then "In Com" should be connector to the return for the isolated input power supply.

The Estop is handled like an input. The EStop signal must be conducting current, i.e. must be "on" for the system to be enabled. Disconnecting EStop produces the emegeancy shutdown event.

Servo Axis Connectors

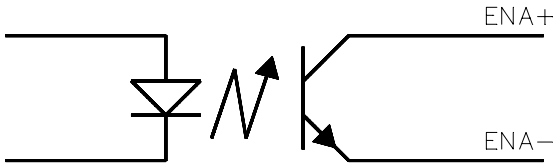
Servo signals are presented on a 12 point detachable screw terminal plug, one axis per plug. These connectors are the right-most connectors when looking at MSB and are labeled Servo 1, Servo 2, Servo 3, and Servo 4.

Encoder Signals go to a differential receiver as shown below.

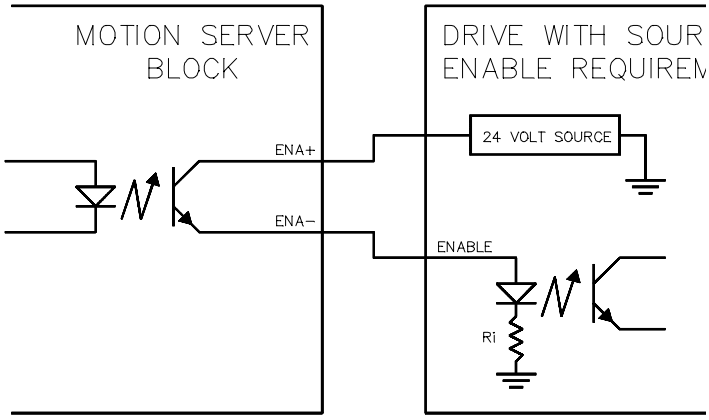


There is a voltage reference provided for the "-" side of the receiver in case only a single-ended encoder is provided. If using single ended encoders, leave the "-" disconnected so that the internal reference provides a threshold voltage suitable for the "+" voltage. Differential signals are recommended for improved noise immunity. Encoder signals go to "Enc A+", "Enc A-", "Enc B+", "Enc B-", "Enc I+" and "Enc I-". "+5V" and "Gnd" points are available to apply power to the encoder. The +5 volts is derived from the controller power.

Each servo axis has an optically isolated enable signal as shown in the following figure:

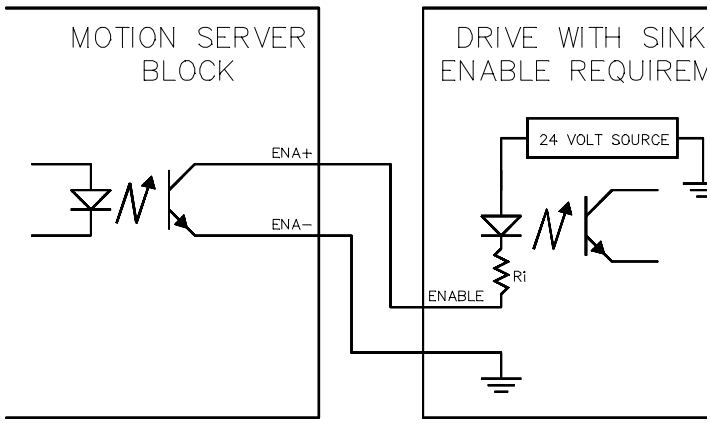


Both the positive and negative sides of the photo transistor are brought to the terminals. This permits using the enable signal in a sourcing or sinking manner. A drive requiring a sourcing enable signal could be hooked up in the following manner:



The "Ena +" signal goes to the supply voltage from the drive, and the "Ena -" signal goes to the enable signal of the drive. Current will flow when the controller indicates "enabled" for the drive.

A sinking enable input can be handled with the following hook-up:



In this arrangement, the ground from the drive is brought to the "Ena -" signal, and the "Ena +" signal goes to the drive enable signal.

Stepper Axis Connectors

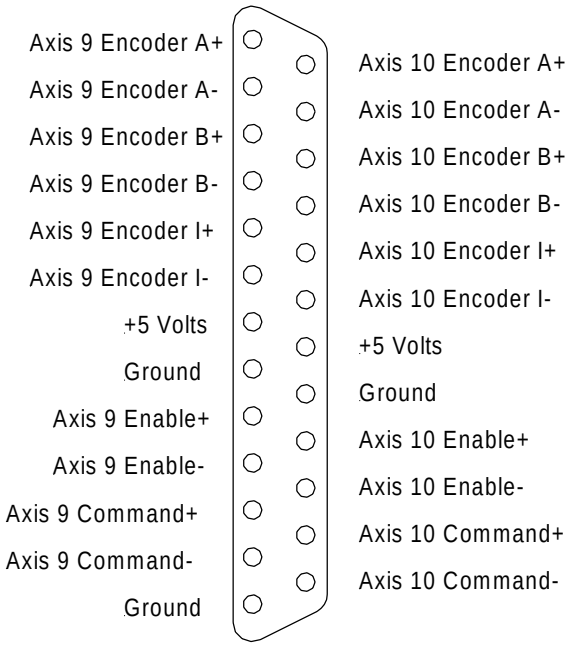
Stepper signals are presented on a 12 point detachable screw terminal plug, two axes per block. These plugs are in the middle of the connector group. Step and Direction signals are brought out on differential drivers. This supports greater noise immunity and frequency for drives which have differential receivers. Drives with optical isolators can also be supported which take a 5 volt signal. For drives with a common cathode (common minus), connect the "+" of the step and direction signals to the drive, and the common cathode to the "Gnd" signal. For drives with a common anode (common plus), connect the "-" step and direction signals to the drive and the common anode to "+5V". If the drive provides both plus and minus signals to the optoisolators, connect directly to the "+" and "-" signals from the drive.

TTL I/O Connector

The TTL IO connector is identical to the IO connector on Motion Server.

MSB V6 Connector

Connections to the additional servo axes available on MSBs having a V6 in the part number are made through the 25 pin female connector on the right side of the unit. The signals are electrically equivelent to the other servo axes. The axes are designated as 9 and 10.



MSB V6 Female Connector Definition

When running with a V6 option card the servo encoder counts rates are as follows:

Axis 1	4 Mhz
Axis 2	4 Mhz
Axis 3	2 Mhz
Axis 4	2 Mhz
Axis 5	2 Mhz
Axis 6	2 Mhz

Current Release Limitations

The initial release of motion server block does not have the following features which are intended to be incorporated into the product in future revisions:

- No high-speed position compare capability available

- No input capture for servos or steppers (index capture is available)

- "Cmd-" signal is common to GND. Isolated command signals are designed in the product but waiting for parts due 4th quarter 2000.

Chapter 9

9) DMS Connections

Description

Cabling to Motion Server is performed through flat ribbon cables terminated with IDC connectors.

Axis Group Connectors

There are (4) 60 pin connectors for axis information called "Axis Group" connectors. Each 60 pin ribbon cable supports (4) axis of signals. The 60 pin ribbon cable can be split apart into (4) identical 15 pin axis sub-cables. The signals have been chosen in a very regular pattern so that all of the 15 pin sub-cables are identical in layout.

I/O Connector

There is (1) 50 pin connector containing 48 bits of configurable I/O. Signals are configured as input or output in 4 bit groups.

EStop Connector

There is (1) 4 pin header used to configure E-Stop with a jumper or to cable to EStop. The jumper can be used to disable E-Stop, connect I/O signal 1 to be E-Stop, or can serve as a cable connector for an external E-Stop cable assembly.

External Bus Connector

There is (1) 26 pin connector which supports an external 8 bit bus allowing Motion Server to control additional hardware elements.

Axis Group Connector Definitions

The following Table defines the connectors for the axis groups. These connectors are designated "Axis 1-4", "Axis 5-8", "Axis 9-12", and "Axis 13-16" on the printed circuit board silk screen. The signal definitions is a regular pattern both along the connector, and from one connector to the next. For example, Pin 3 is always an Encoder B+ signal with the axis defined by which connector the pin is on. Each pin in any particular connector has 3 other counterparts spaced a multiple of 15 away. For example, pin 18 (pin 3 + 15) is also an Encoder B+ signal as well as pin 33 (pin 3 +30) and pin 48 (pin 3 + 45)

Pin Number	Description	Axis 1-4	Axis 5-8	Axis 9-12	Axis 13-16
1	Encoder A+	Axis 1	Axis 5	Axis 9	Axis 13
2	Encoder A-	Axis 1	Axis 5	Axis 9	Axis 13
3	Encoder B+	Axis 1	Axis 5	Axis 9	Axis 13
4	Encoder B-	Axis 1	Axis 5	Axis 9	Axis 13
5	Encoder I+	Axis 1	Axis 5	Axis 9	Axis 13
6	Encoder I-	Axis 1	Axis 5	Axis 9	Axis 13
7	Amp Enable High	Axis 1	Axis 5	Axis 9	Axis 13
8	Amp Enable Low	Axis 1	Axis 5	Axis 9	Axis 13
9	Position Capture	Axis 1	Axis 5	Axis 9	Axis 13
10	Position Compare	Axis 1	Axis 5	Axis 9	Axis 13
11	Motor Command	Axis 1	Axis 5	Axis 9	Axis 13
12	Step Pulse	Axis 1	Axis 5	Axis 9	Axis 13
13	Direction	Axis 1	Axis 5	Axis 9	Axis 13
14	+5 Volts	Axis 1	Axis 5	Axis 9	Axis 13
15	Ground	Axis 1	Axis 5	Axis 9	Axis 13
16	Encoder A+	Axis 2	Axis 6	Axis 10	Axis 14
17	Encoder A-	Axis 2	Axis 6	Axis 10	Axis 14
18	Encoder B+	Axis 2	Axis 6	Axis 10	Axis 14
19	Encoder B-	Axis 2	Axis 6	Axis 10	Axis 14
20	Encoder I+	Axis 2	Axis 6	Axis 10	Axis 14
21	Encoder I-	Axis 2	Axis 6	Axis 10	Axis 14
22	Amp Enable High	Axis 2	Axis 6	Axis 10	Axis 14
23	Amp Enable Low	Axis 2	Axis 6	Axis 10	Axis 14
24	Position Capture	Axis 2	Axis 6	Axis 10	Axis 14
25	Position Compare	Axis 2	Axis 6	Axis 10	Axis 14
26	Motor Command	Axis 2	Axis 6	Axis 10	Axis 14
27	Step Pulse	Axis 2	Axis 6	Axis 10	Axis 14
28	Direction	Axis 2	Axis 6	Axis 10	Axis 14
29	+5 Volts	Axis 2	Axis 6	Axis 10	Axis 14
30	Ground	Axis 2	Axis 6	Axis 10	Axis 14

Pin Number	Description	Axis 1-4	Axis 5-8	Axis 9-12	Axis 13-16
31	Encoder A+	Axis 3	Axis 7	Axis 11	Axis 15
32	Encoder A-	Axis 3	Axis 7	Axis 1	Axis 15
33	Encoder B+	Axis 3	Axis 7	Axis 11	Axis 15
34	Encoder B-	Axis 3	Axis 7	Axis 11	Axis 15
35	Encoder I+	Axis 3	Axis 7	Axis 11	Axis 15
36	Encoder I-	Axis 3	Axis 7	Axis 11	Axis 15
37	Amp Enable High	Axis 3	Axis 7	Axis 11	Axis 15
38	Amp Enable Low	Axis 3	Axis 7	Axis 11	Axis 15
39	Position Capture	Axis 3	Axis 7	Axis 11	Axis 15
40	Position Compare	Axis 3	Axis 7	Axis 11	Axis 15
41	Motor Command	Axis 3	Axis 7	Axis 11	Axis 15
42	Step Pulse	Axis 3	Axis 7	Axis 11	Axis 15
43	Direction	Axis 3	Axis 7	Axis 11	Axis 15
44	+5 Volts	Axis 3	Axis 7	Axis 11	Axis 15
45	Ground	Axis 3	Axis 7	Axis 11	Axis 15
46	Encoder A+	Axis 4	Axis 8	Axis 12	Axis 16
47	Encoder A-	Axis 4	Axis 8	Axis 12	Axis 16
48	Encoder B+	Axis 4	Axis 8	Axis 12	Axis 16
49	Encoder B-	Axis 4	Axis 8	Axis 12	Axis 16
50	Encoder I+	Axis 4	Axis 8	Axis 12	Axis 16
51	Encoder I-	Axis 4	Axis 8	Axis 12	Axis 16
52	Amp Enable High	Axis 4	Axis 8	Axis 12	Axis 16
53	Amp Enable Low	Axis 4	Axis 8	Axis 12	Axis 16
54	Position Capture	Axis 4	Axis 8	Axis 12	Axis 16
55	Position Compare	Axis 4	Axis 8	Axis 12	Axis 16
56	Motor Command	Axis 4	Axis 8	Axis 12	Axis 16
57	Step Pulse	Axis 4	Axis 8	Axis 12	Axis 16
58	Direction	Axis 4	Axis 8	Axis 12	Axis 16
59	+5 Volts	Axis 4	Axis 8	Axis 12	Axis 16
60	Ground	Axis 4	Axis 8	Axis 12	Axis 16

I/O Connector Definition

The 50 pin connector provides inputs and outputs. The pin number is the I/O number with the exception of 49 (+5) and 50 (ground). Input or output sense is configured in 4 bit groups. The groups are defined by "splitting" the connector into (2) 1x50 strips, and then slicing those strips into (12) groups of (4) bits each. This partitioning was chosen so that the even-pin strip could be configured as inputs allowing a standard OPTO-22 cable to plug into the connector without contention between the cable grounds (located on all the even pins) and signals normally available on those pins.

	Description	**Pin**	**Pin**	**Description**	
Group 1	I/O 1	1	2	I/O 2	Group 2
Group 1	I/O 3	3	4	I/O 4	Group 2
Group 1	I/O 5	5	6	I/O 6	Group 2
Group 1	I/O 7	7	8	I/O 8	Group 2
Group 3	I/O 9	9	10	I/O 10	Group 4
Group 3	I/O 11	11	12	I/O 12	Group 4
Group 3	I/O 13	13	14	I/O 14	Group 4
Group 3	I/O 15	15	16	I/O 16	Group 4
Group 5	I/O 17	17	18	I/O 18	Group 6
Group 5	I/O 19	19	20	I/O 20	Group 6
Group 5	I/O 21	21	22	I/O 22	Group 6
Group 5	I/O 23	23	24	I/O 24	Group 6
Group 7	I/O 25	25	26	I/O 26	Group 8
Group 7	I/O 27	27	28	I/O 28	Group 8
Group 7	I/O 29	29	30	I/O 30	Group 8
Group 7	I/O 31	31	32	I/O 32	Group 8
Group 9	I/O 33	33	34	I/O 34	Group 10
Group 9	I/O 35	35	36	I/O 36	Group 10
Group 9	I/O 37	37	38	I/O 38	Group 10
Group 9	I/O 39	39	40	I/O 40	Group 10
Group 11	I/O 41	41	42	I/O 42	Group 12
Group 11	I/O 43	42	44	I/O 44	Group 12
Group 11	I/O 45	45	46	I/O 46	Group 12
Group 11	I/O 47	47	48	I/O 48	Group 12
	+5 Volts	49	50	Ground	

EStop Connector Definition

The EStop connector has 4 "pins" defined as follows

pin 1	Trimmed off for key
pin 2	Ground
pin 3	E-Stop input
pin 4	I/O 1 from 50 pin connector

Placing a jumper between pins 2 and 3 enables the E-Stop (which must be maintained at ground against its 4.7k pullup). This is not recommended if doing anything besides bench testing free spinning motors.

Placing the jumper between pins 3 and 4 redirects the EStop to be from the general I/O connector where an OPTO-22 module rack may be hooked in, or some other IO interconnect that has been chosen for general purpose I/O

A third option is to put a 4 x 1 plug into this header with a cable for pins 2 and 3. A normally closed switch would serve as an E-Stop switch. If the switch disconnected, or the cable was missing, the controller will not enable power to the amplifiers.

External Bus Connector

The remaining 26 pin connector provides a simplified 8-bit bus that can be used to connect to additional hardware. Note that Douloi provides a PC/104 "bridge" accessory that is driven by this connector. The PC/104 format allows the use of many third part cards

Power signals from this connector should only be for signal-level power. If you need any significant current, use a disk-drive connector. Additional details about the use of this bus are available from Douloi Automation on request.

Pin	Description
1	Data 0
2	Data 1
3	Data 2
4	Data 3
5	Data 4
6	Data 5
7	Data 6
8	Data 7
9	Addr 0
10	Addr 1
11	Addr 2
12	Addr 3
13	Addr 4
14	Addr 5
15	Addr 6
16	Select
17	Write/Read
18	Comm_Capture_1
19	Comm_Capture_2
20	Comm_Capture_3
21	Comm_Capture_4
22	Reset
23	+12 Volts
24	-12 Volts
25	+5 Volts
26	Ground

10) Configuring Motion Server for Binary Commands

Overview

Motion Server contains on-board a 486 processor and real-time operating system that can perform application programs independently from the host computer. Programs which run on Motion Server are 32 bit, compiled programs which execute very quickly. Because high speed application programs are available, many motion controller functions normally implemented in "EProm" firmware on other controller products are implemented in Motion Server as application programs. This provides the most flexibility and insures that development work will never "hit the wall" because of a controller limitation.

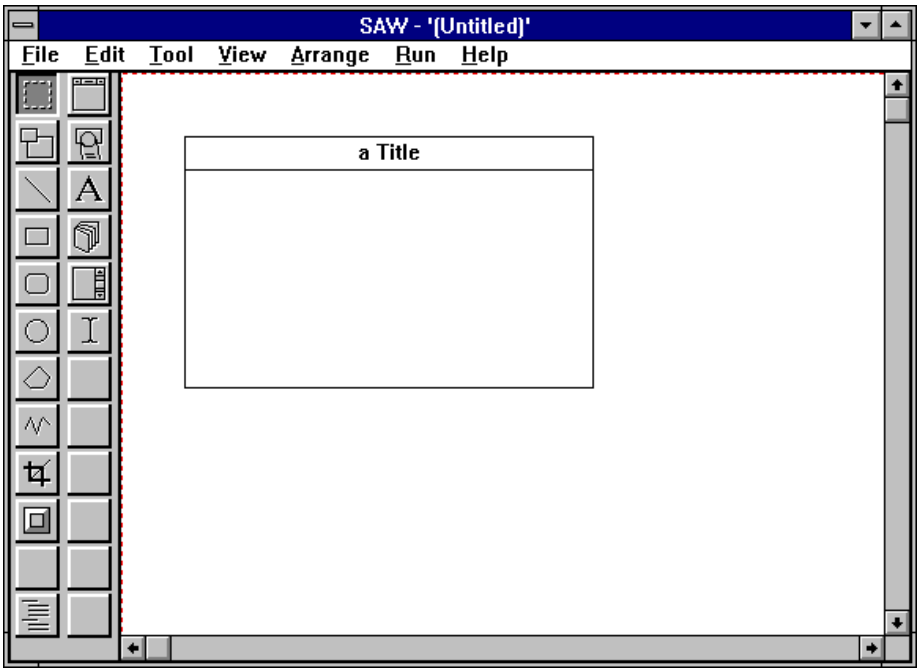
The Binary Command Interpreter is a "stock" application provided by Douloi Automation which works in conjunction with the DMS_BC32.DLL. The binary command interpreter runs on Motion Server and responds to communication packets coming from the host as constructed by the DMS_BC32 library. As far as Motion Server is concerned, interpreting binary commands is just one of many possible applications that might be provided for Motion Server to run.

The following instructions explain how to configure Motion Server to contain the binary commands program on-board, and how to have Motion Server "autostart" the application as it's default application.

Configuration

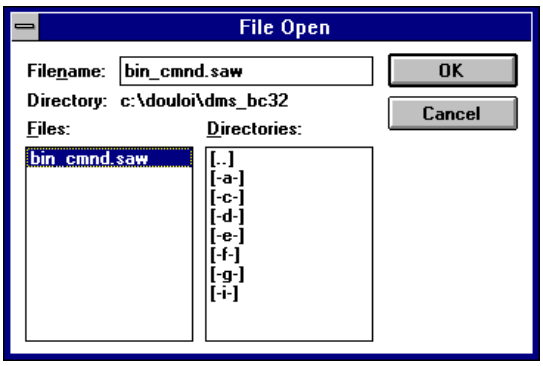
Configuration is performed with Servo Application Workbench (SAW). Follow the setup instructions to install SAW if it has not been installed already. Also install the Binary Command Interpreter (BCI) disk.

Start SAW. You should see the default "blank page" as shown in the following figure:

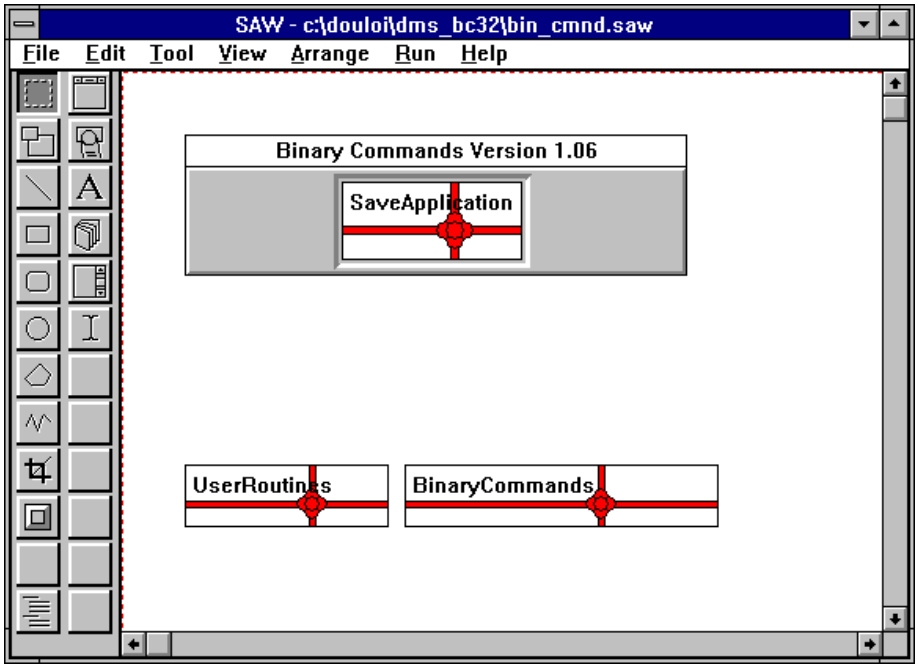


Use the "File|Open" menu selection and go to the Binary Command installation directory. The default value for this directory is c:\Douloi\dms_bc32.

From this directory select the BIN_CMND application:



You should see an application similar to the following:



From the SAW main menu select "Run|Start App". A message box should appear indicating that the application is compiling. When the application is finished, it will start and show a display similar to the following:

Push the "Save" button showing in the Binary Commands window. A series of messages will be shown indicating download progress. When programming is complete, a "done" message will be shown in the display. The Binary Commands dialog is the "console" for the on-board software application and represents the application. If the Binary Commands dialog is explicitly closed (by clicking on the system menu and closing) the on-board software closes also and the command intepreter stops running. To keep the software operating and still close the SAW software, leave the Binary Commands dialog alone and close the SAW development software instead. This closes the development environment and also removes the Binary Commands dialog (because that dialog is hosted by the SAW environment) but leaves the on-board software running.

Turn Switch 3 on Motion Server to the "on" position. Switch 3 means "autostart the application resident in memory". This will cause the interpreter to automatically start when the computer is turned on again in the future.

The autostart switch only has influence when the Motion Server resets. Note that control-alt-delete software resets do not reset the Motion Server card.

If during start-up the green "heartbeat" light stops blinking for a prolonged period (i.e. greater than 10 seconds) set Switch 3 to "off" and cycle power again. If the blinking does not stop then there is a problem with the autostart program. Please consult Douloi Automation if this occurs.

Starting SAW ends the currently running autostart program. To resume the autostart program after having used SAW, cycle power, perform a hardware reset of the host to reset the Motion Server card, or load SAW to run the Binary Command program and exit by closing the SAW develoment environment (leaving the Binary Command dialog alone).