COMMUNICATIONS MANAGER FOR COOLMUSCLE MOTOR SYSTEMS

(COMMMANAGER)

MYOSTAT MOTION CONTROL

17817 LESLIE ST. UNIT 43

NEWMARKET ONTARIO

7/24/2006

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

This manual details how to develop PC side applications using the CommManager interface and Cool Muscle motor(s). In the following pages, it is assumed that the reader is already familiar with:

- The Cool Muscle Language (CML) (for readers yet unfamiliar with CML, please refer to the Cool Muscle User Manual before continuing with this document);
- Component Object Model (COM) Technology including ActiveX Controls;
- How to program using Object Oriented (OO) techniques.

Though not essential to understand the material, many concepts and terminology inside the document relate to Windows programming, and users with neither experience nor knowledge in this area will benefit from visiting http://www.msdn.microsoft.com before continuing to explore issues in COM, threads, and the Win32 API including handles, and conventions. Programming examples found inside this document have been written in either C++ or Visual Basic. Furthermore, the following discussion applies only to C-type Cool Muscle motors controlled via a PC.

2. TRANSFERABLE BENEFITS USING COMMMANAGER

2.1. A Brief Overview of Existing Cool Muscle Motor Network Topology

Traditional Cool Muscle network topology involves using a host device (a PC, PLC, embedded controller, etc.) with an RS232 Com port that communicates to a daisy chain network of Cool Muscle motors. A maximum of 16 Cool Muscle motors can be connected in a daisy chain network. In this daisy chain arrangement, there exists a master device on the daisy chain that handles all network communication between the chain and the host. Thereafter, communication is propagated down the chain from each link between the master motor and the first slave and then between each successive slave motor. It is important to note, that this topology does not require a permanent host device, rather the daisy chain can stand-alone and execute motion programs residing in memory that is pre-loaded by an initial connection to a host device.

2.2. Basic Requirements for a PC Application Controlling Cool Muscle Motors

When developing motion control applications involving the coordination of actions between subsystems there are certain actions and tasks that are prerequisite to the control aspect of the system. Applications involving Cool Muscle motors are also subject to these prerequisites. To control Cool Muscle motor networks and to coordinate motion for an overall system there are three critical actions:

- Polling or querying a motor for information
- Waiting for events from a motor
- Subsequently performing tasks based on return information from polling or event-driven motor info

Likewise when developing a PC application to act as a terminal or a soft-controller for a motion control system, there are basic requirements the application must fulfill to either convey information or to coordinate motor movement. The basic requirements for a PC application to communicate with the motor network is:

- To send a string of commands to the motor network
- To wait for some special return string from the motor network
- To acquire and analyze results returned from the motor network

- To ensure execution of CML commands does not freeze or deadlock operation of the GUI

Moreover, the programming of a PC application for controlling motors involves either iterating or recurring one or more of these requirements for a particular item or sequence of items.

Satisfying the above requirements with respect to application development is greatly simplified when dealing with a single motor system. In this instance the developer is responsible for GUI operation and a manageable degree of motor communication handling. The most difficult decision becomes timing of execution or rather waiting on motor events to trigger other system events (including more motor events). For example, suppose you are polling the motor and you send it the command “?96.1” then your application becomes suspended waiting for a return message. This presents a dangerous situation where the application is suspended on I/O that may never return. In the event a message does not return the application may crash and the system integrity compromised.

In large and complex systems involving many motors and external sub-systems, handling I/O communication becomes significant in magnitude. Combined with the development of stable GUI operation, system integrity becomes a major problem if the application is not robust in design. This is further complicated by the existence of *N* motors over a single RS232 Com port. The timing of motor response and query becomes increasingly critical to avoid, prevent, and handle the occurrence of collisions over network traffic. In this instance, a communication protocol must be established to transmit and receive messages in an efficient manner and deliver error messages or notification when contention or collision occurs.

2.3. Application Development with the CommManager

To release our users from complex and repetitive programming, MyoStat Motion Control engineers have developed an ATL ActiveX control, called CommManager, a software interface for the management of Cool Muscle motor network communication. This ActiveX control is a multi-threaded windows control that communicates with motors at the highest (real-time) priority allowed in Windows OS environments. Furthermore, the control can be inserted into Visual Basic

projects, Visual C++ projects, or any other language that supports COM (Component Object Model) technology.

The CommManager will free the application programmer from developing the communication code necessary to control Cool Muscle motor networks. Instead, a suite of functions is available to actively monitor network communication or passively wait for special return events. Furthermore, by using multiple CommManager objects in the user application the user can develop larger networks across multiple Com ports. The only limitation on network size becomes the amount of available Com ports on the PC.

2.3.1. CommManager Installation

The installation process for CommManager is the same as that of any other ActiveX control DLL file. Save the included DLL files, CommProj.dll and CMLScript1.dll, to a directory path that does not include the “space” character. Execute the following lines from a command prompt in that directory:

regsvr32 CommProj.dll
regsvr32 CMLScript1.dll

These commands will register each DLL in the system registry, and make CommManager available within the preferred development environment.

Note: To work with the Visual C++ examples distributed with the CommManager package, the DLL file "Diagram.dll" must also be registered in the manner described above.

2.3.2. CommManager Uninstallation

Before CommManager is removed from a system, references to the associated ActiveX control DLL files should be removed from the system registry. In the directory from which CommProj.dll and CMLScript1.dll were registered, execute the following from a command prompt:

regsvr32 CommProj.dll /u
regsvr32 CMLScript1.dll /u

These commands will unregister each DLL as a command component in the system registry. Once both files are unregistered, it is safe to manually remove all CommManager files from the system.

2.3.3. CommManager Password

A unique 12-character password is distributed with each licensed copy of CommManager to serve as a license key. This password must be included, via method call or value assignment, in any software client that is designed to be served by a CommManager COM object. For the CommManager object instance m_CommManager, the password assignment syntax for Visual C++ and Visual Basic is given below:

VB: m_CommManager.m_Password = "##########"

VC: m_CommManager.SetM_Password("##########");

Note: The password must be set for each CommManager object instance within the software client. Until the password is set, the client will not be able to open a Com Port using that instance of CommManager.

A detailed functional description for password assignment command is given in Section 4.3.8 of this manual, found on Page 46.

THE FEATURES OF COMMMANAGER

The end-user PC application and the CommManager interface interact in the following way: The application sends CommManager a CML command that is placed into a task queue. Items inside the task queue are executed by the Task Scheduler. Concurrently, CommManager receives transmitted results from the motor network and places these into indexed memory. Next, CommManager returns a message to the end-user application that indicates return information is available. Upon receipt of the return message, the end-user application can access the memory content. Below, Figure 1 shows an OSI model of interaction between the different layers of communication in the CommManager. Figure2 describes interaction between components within the CommManager Interface.

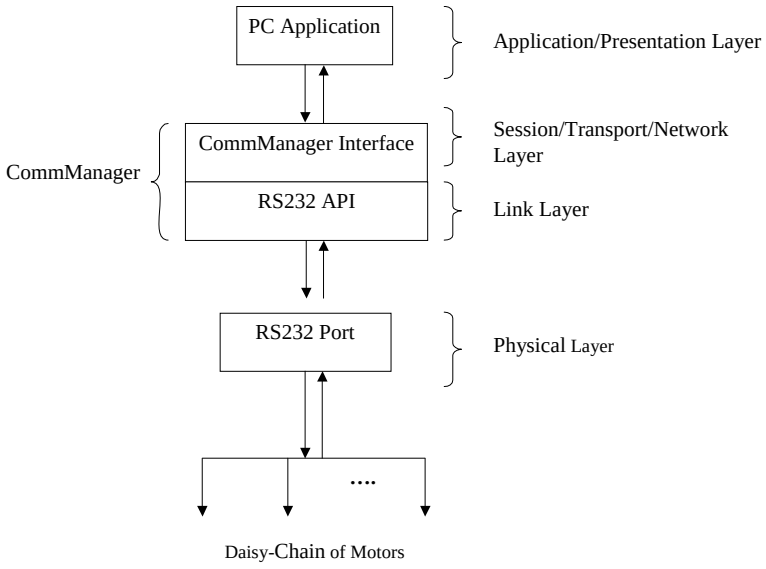

Figure 1: OSI Model of Interaction between Layers with the CommManager

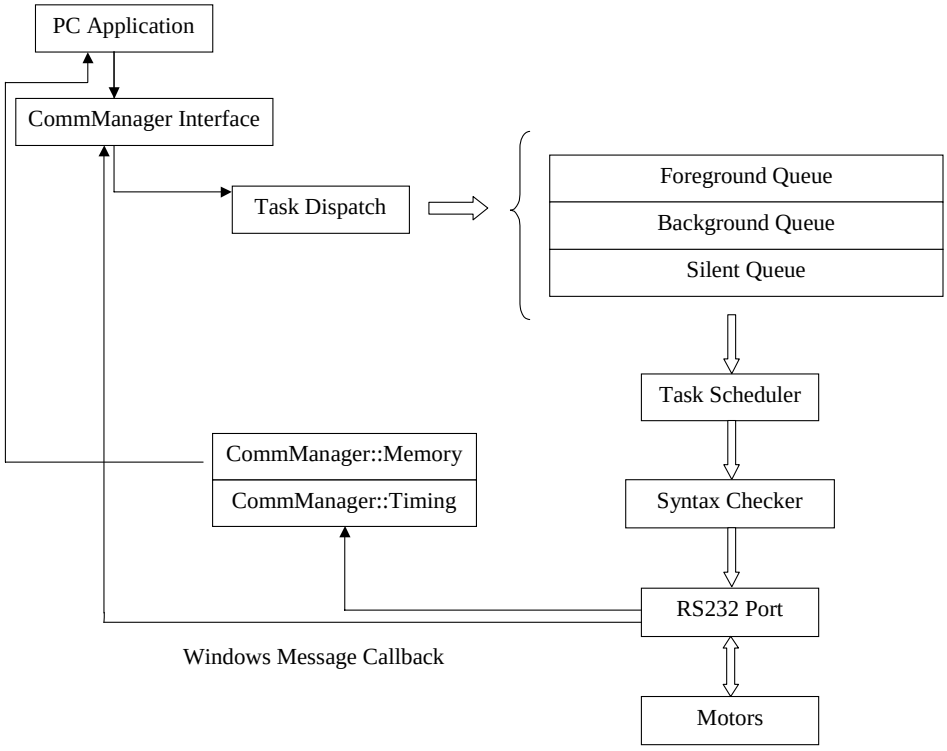

Figure 2: Components and Interaction of the CommManager

The CommManager consists of the following parts:

- COM interface: This interfaces to the user application. CML commands are passed from the user application to the interface. Return messages either come back to the application or a callback is issued to the calling function.
- Task Dispatcher: Places CML commands in one of three task queues according to the desired behaviour of queue items.
- Task Queues: Queues used to store commands from user. The three different task queues are the foreground queue, the background queue and the silent queue.
- Task Scheduler: Schedules the execution of tasks according to an internal multimedia timer.
- Syntax Checker: Checks to ensure semantics of CML observed. Wrong commands are dropped and are not sent to the motor network.
- RS232 API: Manages the low-level communication with the RS232 Com port.

- CommManager Memory: Memory used to store the return results of command execution.

2.4. Operational Properties of the Task Queues

The most significant data feature of the CommManager interface is the task queues. There exist three unique task queues each with a distinct set of features that operate on events placed inside each respective queue. Though each queue has a unique feature all queues are managed from a common thread and are subject to a user pre-defined allocation of execution time.

2.4.1. Timing and Scheduling of the Queues

Though the user cannot directly impose strict execution time in the sense of measured seconds, execution is adjusted according to a ratio of time units determined by the system clock, the OS task scheduler and the actual transmission rate. However, digital systems are very predictable and it has been found that the average CML command takes between 8 ~ 10 ms total time when using a transmission rate of 57.6Kbps and running a minimized set of applications on a WIN2K platform with Intel Pentium III architecture. It should be noted that in the Windows OS a priority scheduler is implemented to determine application processing and I/O handling; CommManager has real-time priority in the Windows OS. Though Windows is not a real-time OS, threads running on the real-time priority level are given highest priority on the task scheduler and these threads are given CPU time whenever I/O events are generated.

The CommManager itself contains a task scheduler to manage CML I/O from the three queues. Suppose some fundamental time unit α, which is the period allotted to threads executing in real-time priority. Next, suppose a time unit δ, which denotes the time required to execute a CML event in the silent queue. Also, suppose λ denotes the time given to execute a single CML event in the foreground queue or background queue. Last, η denotes the ratio of execution time allotted to the foreground/background queues. Then in the period of α, the CommManager task scheduler first executes δ and then executes $\eta\lambda$ foreground queue CML events and last executes λ. Refer to Figure 3 for an illustration of task scheduling by the OS and CommManager.

For any CML event still waiting for a motor response at the end of a time unit, the time unit is extended until a timeout period is reached. In the event timeout is reached in the foreground queue then the CML event is discarded and the calling application receives a message indicating a timeout occurred. If a timeout occurs to an event in the background queue, the calling application receives a timeout message and the CML event is placed at the back of the queue for the next execution. If a timeout occurs in the silent queue then "?99.X" (where X denotes the motor number) is immediately executed. If the response is "Ux.X=8" then the CommManager returns an in-position message to the calling application. Should no response return or "Ux.X=Y" return (where Y denotes any number other than 8) then an error message is sent to the calling application.

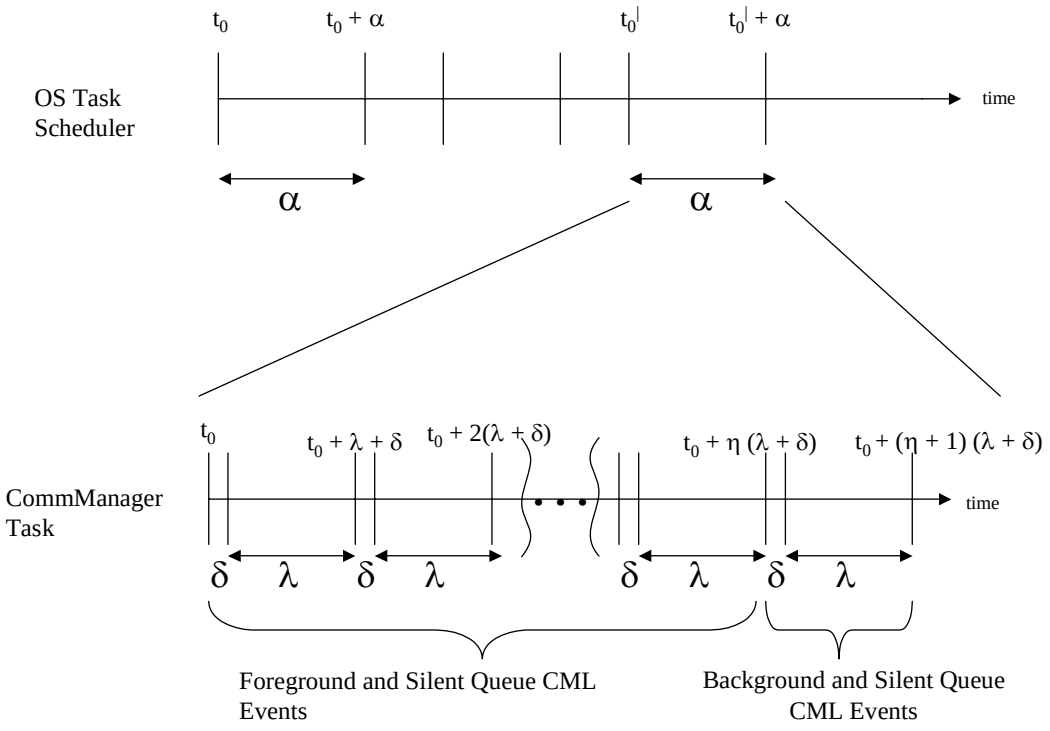

Figure 3: Snapshot of the Timing Required for CommManager Task Scheduling

2.4.2. Properties of the Foreground Queue

The foreground queue is a FIFO queue that executes CML commands one time and waits for a returning message before executing the next queued CML command. Once a return message is received the CML command exits the queue and will not be executed again. In the event an executed command in the foreground queue is waiting for a return message and the allotted time period of execution expires then the time period is extended for the remainder of a pre-defined time-out period. If time-out occurs then CommManager subsequently returns a queue enumerator argument outside the range of accepted values to the calling function. Though there is no restriction on what CML commands can enter the foreground queue, it is recommended that only those commands with return messages of relatively small wait duration be placed in the queue. This includes commands like “p1.1=500”, “k20.1=0”, “p1.1”, “~.1”, and “x5” but excludes commands like “^.1”, and “[.1” if the desired return message is “End!”. It is important to note that it is not necessary to wait for a return message for any items executed in the queue if so desired.

2.4.3. Properties of the Background Queue

The background queue is a circular FIFO queue that executes CML commands repeatedly. Similar to items in the foreground queue, items in the background queue are executed and do not exit the front of the queue until a return message is received. Likewise, in the event the allotted execution time expires, execution time is extended until the time-out period is reached. If the time-out period is reached then a queue enumerator argument returns an error value to the calling function. Regardless of time-out or successful message return the command is then placed at the back of the queue index for the next execution.

There is no restriction placed on what commands can execute in the queue but query type commands are best suited for the background queue for polling and status queries. This includes commands such as “?99.1”, “?1.1”, etc. Customers who need to make thousands of repetitive moves, for example customers using G-code, can send a sequence of motion type CML commands to the background queue for recursive execution.

2.4.4. Properties of the Silent Queue

Though the foreground and background queues are FIFO the silent queue executes all events inside the queue at once. The silent queue can be thought of as a FCFS queue where events arriving simultaneously are served simultaneously.

For each time period allocated to the silent queue, the queue routinely checks CommManager memory for return messages. The silent queue is ideal for watching return messages with long periods between return and execution. For example, we may place the item "[.1" on the silent queue and watch for the return "End!". Here we are watching to ensure the entire program executes and are not concerned whether or not the "[.1" echoed back to the application.

2.5. The Features of the CommManager

2.5.1. Management of Low-Level RS232C Communication

Inside CommManager is an internal data link layer to manage low-level RS232 communication to send and receive a stream of characters from a comm port. This object will find the baud rate setting of the Cool Muscle and set the PC baud rate automatically to match. There is a buffer in the object to manage the incoming stream of ASCII characters. This object works as an independent thread. Therefore, any operation of the GUI will not block the communication buffers. This object waits for comm port events from the Windows OS rather than polling the receive buffer.

2.5.2. An Accurate Timer

A timer using the Windows System "tick" (clock) is implemented and typically accurate within 1ms (this is provided the system is executing a minimized set of processes).

2.5.3. A Syntax Checker

Anything sent to the motor network will be checked for syntax with respect to CML. Wrong CML commands will be discarded and, consequently, not sent. There are two components to the Syntax Checker:

- Validation of CML commands, for instance "k75=1" is discarded since there is no k75 parameter.

- Parameter Value Range Check, for instance “k37=80” is discarded since k37 cannot be assigned the value 80.

2.5.4. Command Queues

There are three virtual command threads in CommManager. The first is the foreground thread, the second is the background thread and the third is the silent thread. The commands queued in each of the above three threads operate independently from one another. Execution of Foreground and Background CML commands will not block one another.

2.5.5. Call-back Functionality

In real-time control problems, timing is critical. The CommManager can callback a calling application according to an internal timer. This timer operates in an independent thread to ensure it will not be affected by GUI operation. Callback items can be placed in either the foreground or background queue and upon successful execution will directly return to the calling application rather than using OnComm or OnExecuteCML.

2.5.6. Management of CML Command Queues

User can send a set of CML commands and let the CommManager to execute them according to timing. For example, you can send 100 “?96.1” to CommManager in the foreground. The CommManager will execute these commands every 10msec. Therefore it is very easy to trace the position when motors are moving. This is useful in getting a step response of motor and so on.

2.5.7. Multi-Threaded Operation

CommManager is composed of 5 smaller threads whose separation guarantees excellent inter-operation and ensures elements inside one thread do not block the operation of other elements inside other threads. In particular, individual threads handle all GUI operation, queue management, RS-232 I/O handling, and multimedia timing.

3.2.8 CML Scripting Language for CML Devices

A CML Scripting language is introduced which is natural English based, easy to understand language. For example, you can draw a circle by the following macro command.

Circle(Motor1, Motor2, Starting angle, incremental angle, radius, speed)

To draw a rectangle card with four round corners, you only need to write the following script and execute the script with the help of CommManager:

```
|2.1,|2.2
Define px 15000
Define py 15000
Define R 4000
Define s 300
Define a 300
Define sCircle 50
Define Motor1 1
Define Motor2 2
Define DelAngle 900
ABSMove(Motor1,a,s,px)
Circle(Motor1,Motor2,900,-DelAngle,R,sCircle)
ABSMove(Motor2,a,s,-py)
Circle(Motor1,Motor2,0,-DelAngle,R,sCircle)
ABSMove(Motor1,a,s,-px)
Circle(Motor1,Motor2,2700,-DelAngle,R,sCircle)
ABSMove(Motor2,a,s,-R)
Circle(Motor1,Motor2,1800,-DelAngle,R,sCircle)
ABSMove(Motor1,a,s,0)
```

In addition, the CML scripting language includes nested-loop, branch and step execution commands. It is shown the Scripting language can save development time by reducing programming and debugging time.

For details, please refer to the document on the CML scripting language.

3.2.9 Polling and Event driven

CommManager is designed to deal with both polling and event driven programming. For the case of pulling programming, we the put the pulling on the background que and send CML commands to motor at foreground queue. For

Event driven, CommManager can capture and report 7 events from motors. There are 7 events happen when motors are running, which can shown as follows:

a) Motor Status:
   Ux.ID=x
b) Input changes
   IN.ID=x
c) Output1 changes
   OUT1.ID=x
d) Output2 changes
   OUT2.ID=x
e) Motor is powered on
   ID
f) The execution of a bank program finishes.
   End!
g) The search of origin finished.
   Origin

We will call Ux.ID=x event as Ux event, and so on later.

3. FUNCTIONS IN THE COMMMANAGER COM INTERFACE WITH PROGRAMMING EXAMPLES

3.1. General definitions for CommManager functions

3.1.1. Some common arguments to most CommManager functions

Most CommManager functions have the following arguments in their parameter list:

- *BSTR CML*, is a string or a string array of CML commands. I.e. the CML command that is to be sent to the motors. It can consist of a single command or and entire list of commands
- *BSTR Wait*, is a string or a string array of messages that CommManager waits to see return. When the string is returned the CommManager analyses the string and returns the appropriate result. If the string is not returned CommManager will time out. If the '*' character is used as the *Wait* string then CommManager will wait for the appropriate response for the command.
- *Long No*, is an index of CommManager memory where the results of executed CML code are saved. See *m_Result()* for details on retrieving the results.
- *Long ID*, is an ID used to identify callback messages sent from CommManager to the application. An ID is assigned to a CML command and the ID is returned when the event response associated with that command occurs.
- *Long Dim*, contains the number of array elements used in CommManager memory to store the results when multiple commands are executed.

If an index to CommManager memory is included in the parameter list of a function then a range check is carried out. If it goes beyond the upper bound of the memory then an error message will be issued and the function will not be executed, consequently, any CML string arguments will not be sent to the motor network. However, you can set the index argument as –1 to not use CommManager memory. It is important to note that CML commands intended to run in the background queue are assigned a positive index value.

3.1.2. Return arguments

Most of the functions of CommManager will return S_OK if the function executes successfully.

The above information is true for all CommManager functions unless documented otherwise in the following sections.

3.1.3. Events related with commands

CommManager can generate four events:

- *OnComm()*
- *OnExecuteCML()*
- *OnMotorEvent()*
- *OnRS232()*

A function beginning with “Send” will generate *OnComm()* and *OnExecuteCML()*. *ExecuteMCML()* will generate *OnExecuteCML()*. *OnMotorEvent()* will be generated when there is a event from motor. *OnRS232()* will be generated when there is an event from the RS-232 port. Refer to §4 for a complete description on the events generated.

3.1.4. Using ActiveX controls in your application

In this manual, we introduce the functions and their arguments according to their prototypes in the ActiveX control. They may take different forms in different applications depending on the programming language chosen. For example, in VB the m_Result property is used as follows:

```
x = m_CommManager.m_Result
```

However in VC, a default class generated by the IDE wizard will use m_Result as follows:

```
m_CommManager.GetM_Result(&x);
```

Use the Object Brower window included in VB or VC IDE to find information pertaining on how to manipulate methods and properties. This topic will not be expanded upon further in this manual since it is assumed the reader is already familiar with ActiveX Controls.

3.1.5. Constants defined in CommManager

There are several constants defined in CommManager that can be found in APPENDIX A. CommManager uses these constants when they return a value to the caller or when it sends a message to the caller. Constants are used in CommManager for the purpose of communicating with the user application. Constants are used to return messages to a calling function or send messages to a calling function. Furthermore, users can use the Object Browser window in Visual C and Visual Basic environments for definitions of the constants.

3.2. Function descriptions

In the following sections, it is assumed that m_CommManager is an instance of CommManager in the application project.

3.2.1. OpenCommPort()

HRESULT OpenCommPort(BSTR Setting)

Description:

Open a Com port. Each CommManager interface can only handle one real physical Com port. If you need to use multiple Com ports in your application then you must create a new instance of a CommManager for each Com port used.

Parameters:

- *BSTR Setting*: This string sets the properties for the Com port and contains the following:

  "COM# baud=# parity=% data=# stop=#"

  For example, to use Com Port 1 with a baud rate of 38.4kbps, no parity bit, 8 data bits and a stop bit the Setting string reads as:

  "COM1 baud=38400 parity=N data=8 stop=1"

Return value(s):

- *cmErrorOpenCommPortErr*: port is already open
- *cmCanNotOpenCommPort*: port cannot be opened.
- *S_OK*: successful

Example:

VB: *call m_CommManager.OpenCommPort("COM2 baud=38400 parity=N data=8 stop=1")*

VC: *m_CommManager.OpenCommPort((LPCTSTR)"COM2 baud=38400 parity=N data=8 stop=1");*

3.2.2. CloseCommPort()

HRESULT CloseCommPort()

Description:

Used to close the Com port. Once closed, the same Com port can be re-opened.

Parameters:

- None

Return Value(s):

- *S_OK:* successful

Example:

VB: *call m_CommManager.CloseCommPort ()*

VC: *m_CommManager.CloseCommPort();*

3.2.3. SendCommPort()

HRESULT SendCommPort(BSTR str)

Description:

Used to send a user-defined string from the comm port. Note that a carriage return must follow a CML command to be recognized by a motor. For example, to execute the "?99.1" command then send "?99.1" + "\r" to have the motor execute the command.

Parameters:

- *str*: A command string sent to motor

Return Value(s):

- *S_OK*: successful

Example:

VB: *call m_CommManager.SendCommPort("?99.1"+chr(13))*

VC: *m_CommManager.SendCommPort((LPCTSTR)"?99.1\r");*

Remark: This function will wait until all the data is fed to the RS232 buffer and does not run in the background. The port is ready to accept another string when this function returns.

3.2.4. ReadCommPort()

*HRESULT ReadCommPort(BSTR *str)*

Description:

This function is used to read the Com port receive buffer via CommManager memory.

Parameters:

- **str*: a pointer to a string that will receive data in the input buffer.

Return Value(s):

- *S_OK*: successful

Example:

VB: *call m_CommManager.ReadCommPort(str)*

VC: *m_CommManager.ReadCommPort((LPCTSTR*)&str);*

Remark: The CommManager will declare the memory for this string but the calling function must release the memory.

3.2.5. SendCML()

HRESULT SendCML(BSTR CML, BSTR Wait, long No, long ID)

Description:

This function is used to send a CML command to the foreground queue.

Parameters:

- Refer to §3.1.1

Return Value(s):

- *S_OK*: successful
- *CmGoesBeyondBoundary*: if user-defined "*long No*" is outside the permissible value range.

Example:

VB: call m_CommManager.SendCML("?99.1", "?99.1*", 0, 100)

VC: m_CommManager.SendCML((LPCTSTR)“?99.1”, (LPCTSTR)“?99.1*”, 0, 100);

Remark 1: If *Wait* is set to a null string then SendCML returns immediately and OnComm returns immediately. If the Wait string is identical to the CML string then the SendCML function returns immediately but *OnComm* triggers when the echo message returns to CommManager. If the *Wait* string is equal to the CML string plus an “*” then the function will not return until CommManager receives both the echo and the corresponding return message. The *Wait* string can either be null in value or contain any combination of ASCII character values that results in a complete or incomplete form of a CML command.

Remark 2: There are two communicating modes. 1.) with echo mode (K14.1=0) and 2.) without echo mode (K14.1=1111). E.g. when you send “?99.1” to motor, motor can respond in the following different ways depending on K14.

1.) K14.1=0

?99.1 (Sent to motor)

?99.1 (Returned from motor)

Uq.1=8 (Returned from motor)

2.) K14.1=1111

?99.1 (Sent to motor)

Uq.1=8 (Returned from motor)

If the correct setting of the *Wait* string is not clear, set *Wait = “*”*. In this case, CommManager will automatically set *Wait* properly.

3.2.6. SendmCML()

HRESULT SendmCML(BSTR CML, BSTR* Wait, int dim, long * No, long ID)*
Description:

This function is used to send several CML to the foreground queue once time. It is equal to multiple SendCML command.

Parameters:

- Refer to §3.1.1

Return Value(s):

- *S_OK*: successful

- *CmGoesBeyondBoundary*: if user-defined *"long No"* is outside the permissible value range.

Example:

Assume K14.1=0

VB: dim CML(2) as string, Wait(2) as string, No(2) as integer

CML(0) ="?99.1":CML(1)="?99.2"

Wait(0) ="?99.1":Wait(1)="?99.2*"*

No(0)=1:No(1)=2

Call m_CommManager.SendmCML(CML(0), Wait(0), 2, No(0), 101)

VC: CString CML[2], Wait[2]

int No[2];

CML[0]="?99.1"; CML[1]="?99.2"

Wait[0]="?99.1"; Wait[1]="?99.2*"*

No[0] = 1; No[1]=2;

m_CommManager.SendmCML((LPCTSTR)CML, (LPCTSTR*)Wait, 2, No, 101)*

3.2.7. SendbgCML()

HRESULT SendbgCML(BSTR CML, BSTR Wait, long No, long ID)

Description:

This function sends a CML command to the background queue.

Parameters:

- Refer to §3.1.1

Return Value(s):

- *S_OK*: successful
- *CmGoesBeyondBoundary*: if user-defined *"long No"* is outside the permissible value range.

Example:

VB: *call m_CommManager.SendbgCML("99.1", "?99.1*", 1,100)*

VC: *m_CommManager.SendbgCML((LPCTSTR)"?99.1",(LPCTSTR)"?99.1*", 1,100)*

3.2.8. SendbgmCML()

HRESULT SendbgmCML(BSTR CML, BSTR* Wait, int dim, long* No, long ID)*

Description:

Send multiple CML commands to the background queue.

Parameters:

- Refer to §3.1.1

Return Value(s):

- *S_OK*: successful
- *CmGoesBeyondBoundary*: if user-defined *"long No"* is outside the permissible value range.

Example:

Assume K14.1=0

VB: dim CML(2) as string, Wait(2) as string, No(2) as integer

CML(0) ="99.1":CML(1)="?99.2"

Wait(0) ="?99.1":Wait(1)="?99.2*"*

No(0)=1:No(1)=2

Call m_CommManager.SendbgmCML(CML(0), Wait(0), 2, No(0), 101)

VC: CString CML[2], Wait[2]

int No[2];

CML[0]="?99.1"; CML[1]="?99.2"

Wait[0]="?99.1"; Wait[1]="?99.2*"*

No[0] = 1; No[1]=2;

m_CommManager.SendbgmCML((LPCTSTR)CML,(LPCTSTR*)Wait,2,No, 101)*

3.2.9. SendFile()

HRESULT SendFile(BSTR file, int ID)

Description:

Send a CML file to the motor. I.e. a large amount of data that isn't necessarily looking for a response such as a program bank or K-parameters.

Parameters:

- *file*: a string containing CML data or settings
- *ID*: refer to §3.1.1

Return Value(s):

- *S_OK*: successful

Example:

VB: *call m_CommManager.SendFile(file, 100)*

VC: *m_CommManager.SendFile(file,100);*

Remark: This command sends a file to the motor line by line and waits for it to finish transferring. This command can prevent buffer overflow in both the Cool Muscle and PC.

3.2.10. SendcbbgCML()

HRESULT SendcbbgCML(BSTR CML, BSTR Wait, long No, long func, long ID)

Description:

This command puts a task on the background queue with callback to a specific function via the allocation of memory.

Parameters:

- *func*: the return address of the calling application
- others: refer to §3.1.1

Return Value(s):

- *S_OK*: successful
- *cmAllocateMemoryError*: If the allocation of memory failed.

Example:

VB: See sample program

VC: See sample program

Remark: This function is used for real-time control. Two motors can cooperate with each other to draw an ellipse or do torque control. The function will be called back after the querying command finishes.

3.2.11. SendcbCML()

HRESULT SendcbCML(BSTR CML, BSTR Wait, long No, long func, long ID)

Description:

This function sends a CML command and calls a specific function via the allocation of memory when it returns.

Parameters:

- *func*: the return address of the calling application
- others: refer to §3.1.1

Return Value(s):

- *S_OK*: successful
- *cmAllocateMemoryError*: If the allocation of memory failed.

Example:

VB: See sample program

VC: See sample program

Remark: This function is used for real-time control. Two motors can cooperate with each other to draw an ellipse or do torque control. The function will be called back after the querying command finishes.

3.2.12. SendSilentCML()

*HRESULT SendSilentCML(BSTR*CML,BSTR*Wait,long TimeOut,long Dim,long ID)*

Description:

This function puts a task into the silent queue.

Parameters:

- *TimeOut*: is defined in seconds
- *Dim*: the number of elements in the CML string.
- others: refer to §3.1.1

Return Value(s):

- *S_OK*: if successful
- *CmGoesBeyondBoundary*: if index of the memory goes beyond the boundary

Example:

VB:Dim cml(3) As String, Wait(3) As String

cml(0) = "|2.1", cml(1) = "|2.2", cml(2) = "^.1", cml(3) = "^.2"

wait(0) = "", wait(1) = "", wait(2) = "Ux.1=8", wait(3) = "Ux.2=8"

call m_CommManager.SendbgCML("?96.1", "?96.1", 1, 103)*

call m_CommManager.SendSlientCML(cml(0), wait(0), 10, 2, 102)

call m_CommManager.SendSlientCML(cml(2), wait(2), 10, 1, 100)

call m_CommManager.SendSlientCML(cml(3), wait(3), 10, 1, 101)

VC:CString CML[3], Wait[3]

CML[0] = "|2.1"; CML[1] = "|2.2"; CML[2] = "^.1"; CML[3] = "^.2";

Wait[0] = ""; Wait[1] = ""; Wait[2] = "Ux.1=8"; Wait[3] = "Ux.2=8";

m_CommManager.SendbgCML("?96.1", "?96.1", 1,103)*

m_CommManager.SendSilentCML(&CML[0], &Wait[0], 10, 2, 102)

m_CommManager.SendSilentCML(&CML[2], &Wait[2], 10, 1, 102)

m_CommManager.SendSilentCML(&CML[3], &Wait[3], 10, 1, 102)

Remark: This function is used to put a task in the silent queue. The tasks in the silent queue will not actively query the motor, instead it will wait until all waiting messages specified by the *Wait* array come back. If the messages from the motor cannot come back within the timeout interval then a message will be issued to indicate the timeout.

3.2.13. SendCMLBlock()

HRESULT SendCMLBlock(BSTR CML, BSTR Wait, long ErrorCode, long* Val, BSTR* MotorEcho)*

Description:

This function is used to send a CML command to motor and get the querying result when it returns.

Parameters:

- Refer to §4.1.1
- *ErrorCode*: error code that is defined in the function *SendCML*.
- *Val*: the result of the query
- *MotorEcho*: A string that contains the log of communication during the querying

Return Value(s):

- *S_OK*: if executed successfully
- *CmGoesBeyondBoundary*: if user-defined No is outside the permissible value range.

Example:

VB:Dim ErrCode as long, Val as long, echo1 as string

call m_CommManager.SendCMLBlock("?99.1", "?99.1", ErrCode, Val, echo1)*

In this you will get the results as follows:

ErrCode = 0

Val=8

Echo1="?99.1"

?99.1

Ux.1=8

VC:Long ErrCode, Val;

BSTR echo1;

m_CommManager.SendCMLBlock((LPCTSTR) "?99.1", (LPCTSTR) "?99.1", &ErrCode, &Val, & echo1)*

Remark: This function and *SendCML()* are quite similar but differ in that the function does not use the Comm event to return the value. The value is returned by the function immediately. It also does not use the task queue and will wait until the foreground queue is empty before executing.

3.2.14. GetM_Result()

long m_Result(long index)

long GetM_Result(long index)*

Description:

GetM_Result() is a read only CommManager memory-search function. A calling application can use it to retrieve results from the CommManager memory.

Parameters:

- *index*: The memory index from 0 to 1023.

Return Value(s):

- A long type value.

Example:

VB: dim x as long 'result

Dim y as long 'index

x = m_CommManager.m_Result(y)

VC: long x,y;

X = m_CommManager.GetM_Result(x);

3.2.15. ExecuteCML

HRESULT ExecuteCML(BSTR CML, long No, long ID)

Description:

Execute a CML motion Command, such as "^.1", "[.1" and so on.

Parameters:

- Refer to §4.1.1

Return Value(s):

- *S_OK:* if executed successfully
- *CmGoesBeyondBoundary:* if user-defined *No* is outside the permissible value range

Example:

VB: call m_CommManager.ExecuteCML("|2.1,p.1=1000,^.1", 1, 100)

VC:m_CommManager.ExecuteCML((LPCTSTR) "|2.1,p.1=1000,^.1", 1, 100)

Remark: This function will send the *CML* command to the foreground queue and will automatically send a "?99.1" command to the background queue. You will receive a message in the event function *OnExecuteCML()* for every iteration of the background queue. The value in m_Result will display the state of the task executed in *ExecuteCML()*.

3.2.16. DeletebgCML()

HRESULT DeletebgCML()

Description:

This function deletes all the CML commands in the background queue.

Parameters:

- None

Return Value(s):

- *S_OK:* if executed successfully

Example:

VB: call m_CommManager.DeletebgCML()

VC: m_CommManager.DeletebgCML();

3.2.17. ExecutemCML

HRESULT ExecutemCML(BSTR CML, long* No, int dim, int* ID)*

Description:

A multiple CML command execute.

Parameters:

- Refer to §4.1.1

Return Value(s):

- *S_OK:* if executed successfully
- *CmGoesBeyondBoundary*: if user-defined *No* is outside the permissible value range

Example:

VB: call m_CommManager.ExecutemCML(CML(0), No(0), 2, ID(0))

VC: m_CommManager.ExecutemCML((LPCTSTR) CML, No, 2, ID)*

3.2.18. TimerInterval()

*HRESULT TimerInterval(short *pVal)*

HRESULT TimerInterval(short newVal)

Description:

The timer for the foreground queue in milliseconds. A task in the queue will be executed every *newVal* milliseconds.

Parameters:

- **pVal* and *newVal* are used to get and to set the timer interval respectively

Return Value(s):

- None

Example:

VB: m_CommManager.TimerInterval = 10

VC: m_CommManager.SetTimerInterval(10);

Remark: If the *TimerInterval* is too short, it is possible that a command will not finish during the interval. In this case, the task will be executed continuously without interruption while future tasks are queued.

3.2.19. TimerStart()

HRESULT TimerStart()

Description:

Start the multimedia timer

Parameter:

- None

Return Value(s):

- *S_OK* if executed successfully

Example:

VB: call m_CommManager.TimerStart

VC: m_CommManager.TimerStart();

Remark: *TimerStart()* does not start a timer but records the timestamp at the execution.

3.2.20. TimerStop()

HRESULT TimerStop(double time1)*

Description:

TimerStop() returns the difference between the timestamps of the start and stop timer.

Parameter:

- *time1*: pointer to a double type variable.

Return Value(s):

- *S_OK*: if executed successfully

Example:

VB: call m_CommManager.TimerStop(time1)

VC: m_CommManager.TimeStop(&time1);

3.2.21. m_BackForeRatio()

*HRESULT m_BackForeRatio(short *pVal)*

HRESULT m_BackForeRatio(short newVal)

Description:

The foreground and background execution times are set according to this ratio. For a positive value the ratio is set newVal:1 (forground:background). For a negative value the ratio is set |newVal|:1 (background:foreground). See the remarks below the example for a more complete description of the functionality of the method.

Parameter:

- **pVal* and *newVal* are the parameters for the foreground ratio.

Return Value(s):

- *S_OK*: if executed successfully

Example:

VB: m_CommManager.m_BackForeRatio = 4
VC: m_CommManager.SetM_BackForeRatio(4)
Remark: An important parameter for the CommManager, m_BackForeRatio sets the ratio of foreground events versus background events for a single period of queue task scheduling. m_BackForeRatio > 0 means foreground tasks will execute m_BackForeRatio queue time units for every 1 background queue time unit for a single period. m_BackForeRatio < 0 means foreground tasks will execute 1 queue time unit for every |m_BackForeRatio| background queue time units executed per period. Special considerations include:

- m_BackForeRatio = 0 results in a single pre-emptive queue whereby events are executed on a FCFS basis. This version of queuing pre-empts empty time-units incurred when executing tasks based on a ratio of |m_BackForeRatio| > 0. However, tasks are still loaded into either the foreground queue or background queue and thus will inherit the properties of that queue when executed;
- m_BackForeRatio = 32767 results in a foreground queue only;
- m_BackForeRatio = -32767 results in a background queue only.

Refer to §3.1 for more in-depth consideration of queue operation and functionality.

3.2.22. DeleteCML()

HRESULT DeleteCML()

Description:

Deletes all the tasks in the foreground queue.

Parameters:

- None

Return Value(s):

- *S_OK*: if executed successfully

Example:

VB: call m_CommManager.DeleteCML
VC: m_CommManager.DeleteCML();

3.2.23. DeleteCMLOnQues()

HRESULT DeleteCMLOnQues(EventConstant QueID, long TaskID)

Description:

Delete a CML command in the queue assigned by *QueID* with the *TaskID*.

Parameters:

- *QueID*: Queue ID is defined as follows:

  ForeGroundQue = vbForeGnd = 1

  BackGroundQue = vbBackGnd = 2

  Command Sent by ExecuteCML (ExecutemCML) = vbExecute = 3

  SilentQue = vbSilent = 4

Return Value(s):

- *S_OK*: if executed successfully

Example:

VB: call m_CommManager.DeleteCMLOnQues(1, 100)
VC: m_CommManager.DeleteCMLOnQues(1,100);

3.2.24. MotorResponse()

HRESULT MotorResponse(int MotorID, long Altitude, long MaxSpeed, long MaxAcc, int dim, MotorExecuteBankConstant Type, long ID)

Description:

This function is used to do a motor test.

Parameters:

- *MotorID* is the id of motor in the daisy chain.
- *Altitude* is the altitude of motion in the unit of pulses
- *MaxSpeed* is the speed of the motion in the unit of pulses per second
- *MaxAcc* is the maximum acceleration of the motion in the unit of Kilopulses per second squared
- *dim* is number of sampling points during the motor test
- *Type* is the query command.
- *ID* refer to the general definition in §4.1.1

Return Value(s):

The CommManager memory from index range 0 to *dim*-1 will be used to record the experiment Results. The data is in *m_Result* and the time that the data is sampled is in *m_Timing*.

Example:

VB: call m_CommManager.MotorResponse(1, 1000, 100, 100, 200, 96, 101)
VC: CString t, t1;
t = "K37.1=3"; t1 = t + "";*
m_CommManager.SendCML(t, t1, 0, 0);
t = "K23.1=0"; t1 = t + ""; m_P = 1000; m_S = 100; m_A = 100;*
m_CommManager.SendCML(t, t1, 0, 0);
m_CommManager.MotorResponse(1, m_P, m_S, m_A, 200, 96, 101);
Remark: The units of Altitude, MaxSpeed and MaxAcc depend on the motor's resolution parameter (K37). Be sure to understand and set this command first.

3.2.25. m_Timing()

*HRESULT m_Timing(int index, double *pVal)*

Description:

CommManager memory used together with the m_Result.

Parameters:

- *index* is the index of CommManager memory.
- **pVal* a pointer to a double type variable

Return Value(s):

- *S_OK*: if executed successfully

Example:

VB: x = m_CommManager.m_Timing(1)
VC: m_CommManager.GetM_Timing(&x);

3.2.26. SetMemoryLength()

HRESULT SetMemoryLength(long len)
Description:

Release the CommManager memory and reallocate the CommManager memory with dimension *len*.

Parameter:

- *len* is the dimension of the CommManager memory.

Return Value(s):

- *S_OK*: if successful
- *cmAllocateMemoryError*: If the allocation of memory failed.

Example:

VB: call m_CommManager.SetMemoryLength(1024)
VC: m_CommManager.SetMemoryLength(1024);
Remark: The default size of CommManager Memory is 1024 (0-1023)

3.2.27. m_QueueBackCount()

*HRESULT m_QueueBackCount(long *pVal)*
Description:

This function retrieves the number of tasks in the background queue.

Parameter:

- **pVal*: a point to a double type variable.

Return Value(s):

- *S_OK*: if executed successfully

Example:

VB: x = m_CommManager.m_QueueBackCount

VC: m_CommManager.GetM_QueueBackCount(&I)

3.2.28. m_QueueForeCount()

*HRESULT m_ QueueForeCount (long *pVal)*

Description:

This function retrieves the number of tasks in the foreground queue.

Parameter:

- **pVal:* is a point to a double type variable.

Return Value(s):

- *S_OK*: if executed successfully

Example:

VB: x = m_CommManager.m_ QueueForeCount

VC: m_CommManager.GetM_ QueueForeCount(&i)

3.2.29. m_QueueSilentCount()

*HRESULT m_QueueSilentCount(long *pVal)*

Description:

This function retrieves the number of tasks in the silent queue.

Parameter:

- **pVal*: is a point to a double type variable.

Return Value(s):

- *S_OK* if executed successfully

Example:

VB: x = m_CommManager.m_QueueSilent

VC: m_CommManager.GetM_QueueSilent (&i)

Remark: Useful for determining if the queues are overloaded during execution.

3.2.30. m_CMLCheck()

*HRESULT m_CMLCheck(long *pVal)*

HRESULT m_CMLCheck(long newVal)

Description:

Boolean type function to turn CML syntax checking on or off.

Parameters:

- **pVal* and *newVal* are values from and to CommManager respectively. To enable CML syntax checking then set *newVal* equal to 1. All other values will disable syntax checking.

Return Value(s):

- *S_OK*: if executed successfully

Example:

VB: m_CommManager.m_CMLCheck = 1
VC: m_CommManager.SetM_CMLCheck (1)

3.2.31. m_WriteToLogFile()

*HRESULT m_WriteToLogFile(long *pVal)*
HRESULT m_WriteToLogFile(long newVal)
Description:

This function is used to turn on/off a log file of all motor network and Host PC communication.

Parameters:

- **pVal* and *newVal* are values from and to CommManager respectively. To enable data logging set *newVal* equal to 1. All other values will disable data logging.

Return Value(s):

- *S_OK*: if successful

Example:

VB: m_CommManager.m_WriteToLogFile = 1
VC: m_CommManager.SetM_WriteToLogFile(1)
Remark: The log file is saved as log.dat and is placed in the root directory of the CommManager object. The log.dat has no limit in size and is intended for debugging purposes. It should be noted that continuous daily use of the log file

could result in PC system performance degradation and may result in a system crash. The log.dat will be cleared when CommManager is loaded.

3.2.32. TestCommDelay()

HRESULT TestCommDelay(int Motor1, int Motor2, double TimeDelay)*

Description:

This function will send "?99.1" to *Motor1* and *Motor2* and return the propagation time difference between them.

Parameters:

- *Motor1*: ID of the 1st motor in the daisy chain
- *Motor2*: ID of the 2nd motor in thedaisy chain

Return Value(s):

- *S_OK:* if executed successfully

Example:

VB: call m_CommManager.TestCommDelay(1,2 TimeDelay)
VC: m_CommManager.TestCommDelay(1,2,&TimeDelay);

3.2.33. Functions for Wrapping Basic CML Commands

The following functions are simple wrapper functions of CML commands to free the user from CML

In the following, 'MotorID' is the ID of the motor in the daisy chain. 'No' is the index of CommManager memory. 'ID' is a message ID. 'Type' is the type of query command during the execution. It takes the following values:

- vbPError = 95 query position error
- vbPos = 96 query position
- vbVel = 97 query current speed
- vbTorq = 98 query current torque
- vbStatus = 99 query current motor status

These values are defined in *enum* of *MotorExecuteBankConstant*.

3.2.33.1. MotorGoOrigin()

HRESULT MotorGoOrigin(int MotorID, long No, long ID)

Description: Returns the motor to its origin

Remark: The returning result is in m_Result[No]

3.2.33.2. MotorExecBank()

HRESULT MotorExecBank(int MotorID, int BankNo, int dim, MotorExecuteBankConstant Type, long ID)

Description: Executes a bank program

Remark: The returning result is in m_Result[From *0* to *dim-1*]

3.2.33.3. MotorStop()

HRESULT MotorStop(int MotorID, long ID)

Description: Stops the motor

3.2.33.4. MotorEnable()

HRESULT MotorEnable(int MotorID, long ID)

Description: Enables the motor.

3.2.33.5. MotorDisable()

HRESULT MotorDisable(int MotorID, long ID)

Description: Disables the motor.

3.2.33.6. MotorDynaExec()

HRESULT MotorDynaExec(int MotorID, int dim, MotorExecuteBankConstant Type, long ID)

Description: Executes the dynamic bank contents – i.e. motor moves to p0.1 at speed, s0.1, and acceleration, a0.1.

Remark: The returning result is in m_Result[From 0 to dim-1]

3.2.33.7. MotorCP()

HRESULT MotorCP(int MotorID, int dim, MotorExecuteBankConstant Type, long ID)

Description: Motor moves using the continuous point feature

Remark: The returning result is in m_Result[From 0 to dim-1]

3.2.33.8. MotorQuery()

HRESULT MotorQuery(int MotorID, int No, int Type, long ID)

Description: Queries motor status

Remark: The returning result is in m_Result[No]

3.3. Functions For Running CML Script

3.3.1. GetVersion()

HRESULT GetVersion(BSTR version);*

Description: This function is used to query the version of CommManager.

Parameters:

- Version: to receive the version of CommManager

Return Value(s):

- *S_OK*: if executed successfully

Example:

VB: Dim ver as string
call m_CommManager.GetVersion(ver)
VC: BSTR ver;
m_CommManager.GetVersion(&ver)

3.3.2. m_SchedulerStyle()

A property of CommManager which is under development. At present, you can use m_BackForeRatio to define the style of the Scheduler of the CommManager.

3.3.3. RegisterEvents()

RegisterEvents(long EventType, long MotorID, double TimeOut);

Description:

This function will register an event to CommManager so that when an event happens, the calling application will get notified from Event (OnMotorEvent) of CommManager. See more on the event types in the remarks below the example.

Parameters:

- *EventType*:
    - 1 - "Ux"
    - 2 - " IN"
    - 3 - "OUT1"
    - 4 - "OUT2"
    - 5 - "ID"

- 6 - "End!"
- 7 - "Origin"

- *MotorID*: A number between 1 to 16.
- *TimeOut*: A time period for timeout time whose unit is second.

Return:

- *S_OK*: if executed successfully

Example:

VB: call m_CommManager.RegisterEvents(1, 1, 10.0)
VC: m_CommManager.RegisterEvents(1, 1, 10.0);

Remark 1: This function is an early function in CommManager development. Now all the events from the motor are registered as system events when CommManager is initialised. This function is different from the system events as you can assign a timeout.

Events during operation depend on the setting of K23. The following are events when K23=7:

1. Motor Status: *Ux.ID=x*
2. Input changes: *IN.ID=x*
3. Output1 changes: *OUT1.ID=x*
4. Output2 changes: *OUT2.ID=x*
5. Power on: *ID*
6. Completion of bank execution: *End!*
7. Origin search completion: *Origin*

Note: Ux.ID=x event will now be referred to as Ux event, and so on.

Remark 2: If we register the following event to CommManager

VB: *call m_CommManager.RegisterEvents(1, 1, 10.0)*

Then if an event of Ux.1=x happens within 10.0sec then *OnMotorEvent* will be issused with the following parameters:

MotorEvent = 1
MotorID = 1

Value = x

If it times out then *OnMotorEvent* will be issused with the following parameters:

MotorEvent = 1

MotorID = 1

Value = -77777777

3.3.4. DetectMotorCommunication()

DetectMotorCommunication(long MemoryAddress)

Description:

This function scans how many motors are there in the network. Not implemented as of this time.

3.3.5. BlockExecute()

BlockExecute(long flag)

Description:

This function blocks the execution of the application with Keyboard and mouse activated. This function is an auxiliary function to simplify the programming of sequential actions. It will be released when *Ux*, *End* and *Origin* events happen.

Parameters:

- Flag:
  - *True* - activate blocking
  - *False* – deactivate blocking

Return:

- *S_OK*: if executed successfully

Examples:

VB: call m_CommManager.BlockExecution(1)

VC: m_CommManager.BlockExecution(1);

Remark: To provent this function hangs up your program, it is recommdated to call BlockExecution(0) before exit the program. For VB program, call it at query_unload.

For VC program, call it in the deconstructor of a class that calls the BlockExecution.

3.3.6. DisableBlock()

HRESULT DisableBlock(No)

Description: This function will terminate the execution of the current script(s).

Parameter:

- *No*

Return value:

- *No*

Examples:

VB: m_CommManager.DisableBlock = 0

VC: m_CommManager.Put_DisableBlock(0);

Remark: Set DisableBlock() to a value that is not equal to 1 will terminate the execution of the current script(s). It is different from StopMScript function. StopMScript only terminates the script with BankNo.

3.3.7. RunScript()

HRESULT RunScript(BSTR bank)

Description: Run a CML script for CML devices. This function will not block your application and will issuse a message to your application when the script finishes. The definition can be found in the document of CML script language. It contains a set of natural English based commands, such as Circle(1, 2,0, 3600, 1000,100) draws a circle with radius 1000, speed 100 by means of motor 1 and motor 2.

Parameters:

- *bank*: A string contains the script

Return value:

- *S_OK*: if executed successfully

Examples:

VB:dim bank

bank = "Loop(3)"+ vbcrlf

bank = bank + "p.1=10000,^.1"+vbcrlf

bank = bank + "p.1=-10000,^.1"+vbcrlf

bank = bank + "LoopEnd"+vbcrlf

call commport1.RunScript(bank)

This will cause motor1 go forward and backward three times.

VC: CString bank;

bank = "Loop(3)\r\n";

bank = bank + "p.1=10000,^.1\r\n";

bank = bank + "p.1=-10000,^.1\r\n";

bank = bank + "LoopEnd\r\n";

m_CommManager.RunScript((LPCTSTR)bank);

Remark: To run RunScript successfully, set K14=1111. This turns off the echo message from motors.

3.3.8. m_Password()

Description: A password to be able to use CommManager

Parameter(s):

- *No*

Return value(s)

- S_OK: if executed successfully

Examples:

VB: m_CommManager.m_Password = "1234567890"

VC: m_CommManager.SetM_Password("1234567890");

3.3.9. RunMScript()

RunMScript(BSTR mBank, int Dim)*

Description: Run several scripts at the same time. For some advanced application, there is a need to group motors on the network into different groups and execute scripts at the same time.

Parameters:

- *mBank*: a string array contains scripts
- *Dim*: the dimension of mBank

Return Value(s):

- *S_OK*: if executed successfully

Examples:

Refer to 4.3.8

3.3.10. StopMScript()

StopMScript(int BankNo)

Description: stop a script launched by RunScript() or RunMScript()

Parameters:

- *BankNo*: the No of the script that you want to stop

Return Values:

- *S_OK*: if executed successfully

Examples:

VB: call m_CommManager.StopMScript(1)

VC: m_CommManager.StopMScript(1);

Remark: When the script is lauched by RunScript, set BankNo = 1

3.3.11. m_ScriptVar()

Description: An array with the dimension of 1024 used by script.

Please refer to the document of CML scripting language.

3.3.12. m_TranslatedCMLScript()

*m_TranslatedCMLScript(long Index, BSTR *pVal)*

Description: Before a script execution, it is be translated to CML language. The translated results will be saved in the m_TranslatedCMLScript(). By looking at this, you can understand what is happening in the CommManager.

Parameter:

- *Index*: the number of the script interested
- *pVal*: a pointer to a string.

Return Values:

- *S_OK*: if executed successfully

3.3.13. ScriptStepExecute()

ScriptStepExecute()

Description: Execute the script step by step.

Parameters:

- *No*

Return Values:

- *S_OK:* if executed successfully

Examples:

VB: Call m_CommManager.ScriptStepExecute()

VC: m_CommManager.ScriptStepExecute();

Remark: Stepping execution will start when "*S_Stepping*" appears in the script. When "*E_Stepping"* is encountered the stepping execution terminates. For example:

Loop(3)
p.1=1000,^.1
p.1=-1000,^.1
LoopEnd
S_Stepping
Loop(3)
p.1=1000,^.1
p.1=-1000,^.1
LoopEnd
E_Stepping
Loop(3)
p.1=1000,^.1
p.1=-1000,^.1
LoopEnd

Then only the loop marked with yellow will be executed step by step. When you call ScriptStepExecute. S_Stepping and E_Stepping is not valid for the RunScript and RunMScript.

4. EVENTS OF COMMMANAGER

4.1. OnComm and OnExecuteCML

HRESULT OnComm(long CommEvent, long MotorID, long MsgID)

HRESULT OnExecuteCML(long CommEvent, long MotorID, long MsgID)

Description:

If you insert the CommManager as an ActiveX control into your project then you can use these events. OnComm event will be issused when you use the commands beginning with “Send” to send commands to motors. OnExecuteCML will be issused when you execute a command by ExecuteCML or ExecutemCML.

Parameters:

- CommEvent, CommManager uses these constants to distinguish by what kind of command the event is triggered. The EventConstant is defined as follows:
  - vbTimeout = -1 Triggered when timeout occurs
  - vbDirect = 0 Triggered by SendCommPort function
  - vbForeGnd = 1 Triggered by a foreground command
  - vbBackGnd = 2 Triggered by a background command
  - vbExecute = 3 Triggered by a ExecuteCML function
  - vbSilent = 4 Triggered by a silent command Info, CommManager uses this number to indicate by which command the event is triggered
  - vbBlock = 5 Triggered by a SendCMLBlock command.

Return Value(s):

- None

4.2. OnMotorEvent()

HRESULT OnMotorEvent(long MotorEvent, long MotorID, long Value)

Description:

Capture the events from motor. Seven events will be generated, shows as follows:

1) "Ux": Generated when Ux.Id=* (*= 0, 1, 2, 4, 8) returned from motor.

2) "IN": Generated when Inputs of motors change

3) "OUT1": Generated when Outputs 1 of motors change

4) "OUT2": Generated when Outputs 2 of motors change

5) "ID": Generated when motors are powered on.

6) "End!": Generated when a bank program finishes.

7) "Origin": Generated when motor arrives its origin.

In additional to those, when a script finishes, OnMotorEvent will be issused with *MotorEvent = 8, MotorID = 0, Value = 0*.

Parameters:

- *MotorEvent*: belongs to 1-8. The value means a kind of event occurs.
- *MotorID*: the motor that generates the event
- *Value*: The event value.

For example: Ux.4=8 occurs then OnMotorEvent will be generated with the following parameters:

MotorEvent = 1
MotorID = 4
Value = 8

4.3. OnRS232()

OnRS232(CommPortEventConstant EventNo)

This event tells you a CommPort hardware event has happened.

Parameter:

- *EventNo* takes the following value:
  - EV_RXCHAR = 0x0001, // Any Character received
  - EV_RXFLAG = 0x0002, // Received certain character
  - EV_TXEMPTY = 0x0004, // Transmitt Queue Empty
  - EV_CTS = 0x0008, // CTS changed state
  - EV_DSR = 0x0010, // DSR changed state
  - EV_RLSD = 0x0020, // RLSD changed state
  - EV_BREAK = 0x0040, // BREAK received
  - EV_ERR = 0x0080, // Line status error occurred

- EV_RING = 0x0100, // Ring signal detected
- EV_PERR = 0x0200, // Printer error occured
- EV_RX80FULL = 0x0400, // Receive buffer is 80 percent full
- EV_EVENT1 = 0x0800, // Provider specific event 1
- EV_EVENT2 = 0x1000, // Provider specific event 2

Return value:

- No

5. CALLBACK OF COMMMANAGER

CommManager can callback a function defined by the user application. This is important when we do real-time control. The callback function can be set to CommManager by SendcbbgCML() function. The prototype of the callback function is defined as follows:

*long _stdcall func(long x, BSTR*u)*

Where x is measurement data from the CommManager, and u is a point of a string that will be sent to a motor. This function usually can be used to implement a controller. For example, by VB, we can implement a torque control by the following command.

Call m_CommManager.SendcbbgCML("?98.1", "?98.1", 0, AddressOf controllerX, 101)*

Where the controllerX is defined a module as public function as follows:

```
Public Function controllerX(ByVal x As Long, ByRef u As String) As Long
    sp2 = CInt(Main.TS.Text)
    err = sp2 - x
    s1 = err * 1
            Main.TR.Text = x
            Debug.Print s1, x
            If (s1 > 25000) Then
               s1 = 25000
    ElseIf (s1 < -25000) Then
         s1 = -25000
    End If
    u = "s0.1=" + CStr(Int(s1))
    Print #1, x, Main.ComPort1.m_Timing(0)
End Function
```

6. EXAMPLES USING COMMMANAGER

There are five projects sent together with the CommManager control. They are projects of

a) BasicFunctions

   Examples of SendCML, SendbgCML, ExecuteCML and so on are given. This project shows that we can execution of "^.1" on the foreground queue then qeury the motor position on the background queue. We can monitor the position during the motion. This project also shows that we can execute a set of CML Commands serially as the same to the exection of Bank program in the motor.

b) DrawACircle

   Examples of using SendcbbgCML and callback. This project shows how to implement an external position controller by CommManager so that we can make two motors work cooperatively. However, be very cautious using this program, because windows OS is not a real-time OS. Therefore, DO NOT EXECUTE ANY OTHER PROGRAM WHEN RUNNING THIS APPLICATION.

c) StepResponse

   Examples of using MotorResponse. This project shows how to get a step response of motor

d) TorqueControl

   Examples of using SendcbbgCML. This project shows how to implement an external torque controller by CommManager

e) SilentCML.

   Examples of using SendSilentCML. This project shows how to use SendSilentCML to execute CML Command silently.

Each project has VB and VC version.

7. CONCLUSION

CommManager was developed to help users manage complex and repetitive programming in their developing of control software for Cool Muscle motor system. Several other functions will be added to the CommManager such as G-code commands.

APPENDIX A Constants Defined in the CommManager

There are three sets of constants defined in the CommManager. They are type information or error message. The error message is somewhat strange because it is COM message. You can find out its real value by the Object Browser.

```
enum EventConstant {
    vbTimeout = -1,
    vbDirect = 0,
    vbForeGnd = 1,
    vbBackGnd = 2,
    vbExecute = 3,
    vbSilent = 4,
} EventConstant;

enum MotorExecuteBankConstant{
    vbPError = 95,
    vbPos = 96,
    vbVel  = 97,
    vbTorq = 98,
    vbStatus = 99,
} MotorExecuteBankConstant;

enum CmError{
    cmErrorOpenComPortErr = MAKE_HRESULT(SEVERITY_ERROR,FACILITY_ITF,0xFFFF),
     cmSearchingBaud rateError = MAKE_HRESULT(SEVERITY_ERROR, FACILITY_ITF,0xFFFE),
     cmCanNotOpenComPort = MAKE_HRESULT(SEVERITY_ERROR, FACILITY_ITF,0xFFFD),
     cmSendCMLError = MAKE_HRESULT(SEVERITY_ERROR, FACILITY_ITF,0xFFFC),
     cmGoesBeyondBoundary = MAKE_HRESULT(SEVERITY_ERROR, FACILITY_ITF,0xFFFB),
     cmComPortNotOpen = MAKE_HRESULT(SEVERITY_ERROR, FACILITY_ITF,0xFFFA),
     cmWrongMotorID = MAKE_HRESULT(SEVERITY_ERROR, FACILITY_ITF,0xFFF9),
     cmAllocateMemoryError = MAKE_HRESULT(SEVERITY_ERROR, FACILITY_ITF,0xFFF8)
} CmError
```