DMCWIN &
C/C++ Programmers Tool Kit
Reference Manual

By Galil Motion Control, Inc.

Galil Motion Control, Inc.
3750 Atherton Road
Rocklin, California 95765
Phone: (916) 626-0101
Fax: (916) 626-0102
Internet Address: support@galilmc.com
URL: www.galilmc.com

Rev 2-01

Overview

This manual describes the contents of the DMCWIN16 and DMCWIN32 utilities disks. These disks include DLLs for developing programs on a PC to communicate with a Galil controller, a terminal program to send commands live to a controller, and many sample programs and utilities to help design a software interface.

Installation

To install the DMCWINxx program run the setup.exe program located on the installation disk. If installing from the Galil Software Products CD find the '\dmcwin' directory. Run 'DMCWIN16.exe' to install the 16 bit utilities and tools or 'DMCWIN32.exe' to install the 32 bit versions.

During installation a directory named '\DMCWIN' will be created on the hard drive. Most of the files will be copied there. A few files will be copied to the '\windows'directory. See the chart below for the files and locations. These files will also be needed for distribution of software created with the C++ Programmers Tool Kit:

	Windows 3.x	**Windows 95/98**	**Windows NT**
windows\system32			DMC32.DLL DMCBUS32.DLL DMCSER32.DLL
windows\system	VXD.386 GALIL.386 DMC16.DLL DMCBUS16.DLL DMCSER16.DLL	DMC32.DLL DMCBUS32.DLL DMCSER32.DLL	
windows\system\vmm32		GALIL.VXD GALILPCI.VXD GALILPNP.VXD WRTDEV0.VXD WRTDEV1.VXD WRTDEV2.VXD WRTDEV3.VXD	
windows\system32\drivers		GALILUSB.SYS (Win 98 and NT5 only)	GALIL.SYS GALILPCI.SYS WINRT.SYS

Note: Reboot the computer after installation.

DMC Terminal Program

DTerm is a live terminal program that communicates directly with the controller. It is used to send commands and programs to the controller. Special utility functions are also included.

Once the DMCWINxx disk is installed a 'Galil' program group is created. To start the terminal find the 'Galil Terminal' program under the 'Start - Programs - Galil' menu. Alternately, run the 'Dterm16.exe' or 'Dterm32.exe' program in the 'c:\dmcwin' directory.

Galil Registry

Before communicating with a Galil controller it must be entered in the Galil registry. This registry contains information on each controller in the system including the type of interface, address, and driver used. Some controllers will use Plug and Play to automatically enter information in the registry, others will have to be entered manually.

Note: *There are separate registries for 16 and 32 bit programs.*

Manually Registering a Controller

To manually register a new controller, first start the Dterm program. Once the program is running, click on the Registry menu item. This will bring up a window with a list of currently installed Galil controllers.

Figure 1: Accessing the Registry Menu

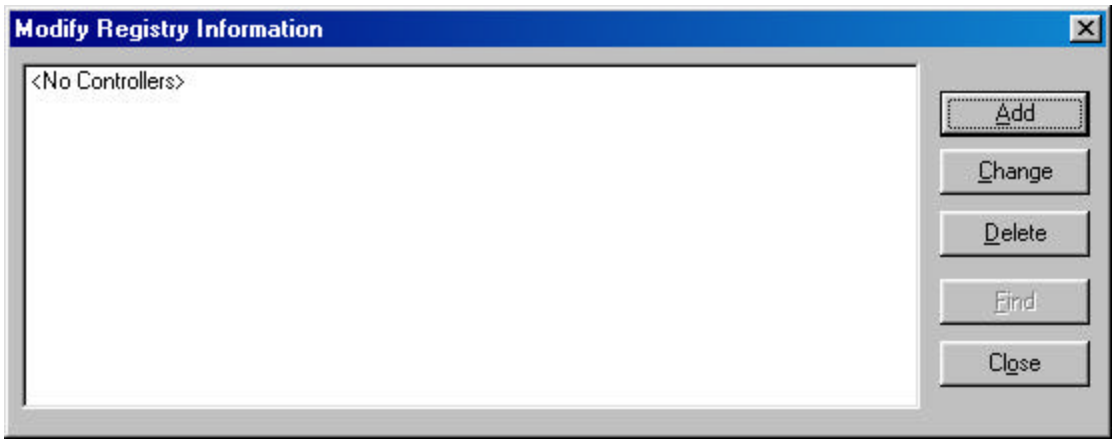


Figure 2: The Registry Menu

To add a new controller, click on the **Add** button, then fill in the appropriate data. Consult the installed controller's hardware manual for configuration settings. Click on OK to save the changes or Cancel to ignore the changes.

Note: *It is recommended that you use the Galil driver when registering a controller.*

To change an existing controller's settings, select the controller by clicking once on it, then clicking on the **Change** button. Change the data as necessary. Click on OK to save the changes, or Cancel to ignore the changes.

To locate a plug-and-play enabled controller (currently, the DMC-1600, DMC-1700, DMC-1800, and DMC-2000 in USB mode are plug-and-play enabled), click on the **Find** button. This function does not work for Windows 95/98. However, if you are running Windows 95/98, plug-and-play enabled controllers are automatically registered by the system. See the section entitled "Plug-and-Play".

To delete a controller, select the device by clicking once on it, then clicking on the **Delete** button.

When you have finished changing the Registry, click on the **Close** button.

Note: *changes to the configuration stored in the Windows Registry will not be recognized by the device drivers until you either re-boot the system or manually stop and start the device drivers.*

The Windows Registry may also be updated by using the Galil Registry API functions listed below.

Terminal

To establish communication with a controller select the 'Terminal' menu. Type commands in the upper box and press carriage return to send them to the controller. ALL COMMANDS MUST BE UPPERCASE. Multiple commands may be entered at the command line by separating them with semicolons. Command responses will appear in the lower box.

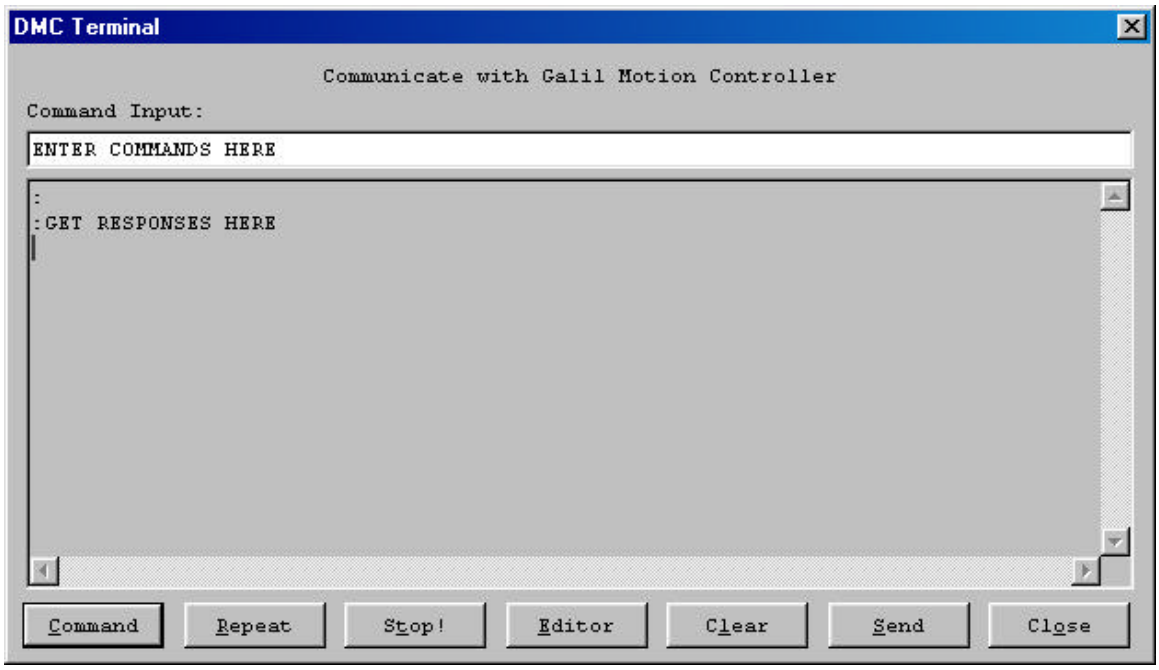


Figure 3: The Galil Terminal

Commands sent this way are executed immediately and are considered "live" commands. Certain commands, such as labels and program branching statements should not be sent in this manner. Examples of invalid live commands include:

#
JP
JS
IF

The commands above can only occur within programs downloaded to the controller. To download a program use the editor.

Other Terminal Features

There are a number of buttons located along the bottom of the Galil terminal. Here is a description of their functions:

Command: Executes the command entered at the command line. Pressing carriage return when the cursor is in the command line performs the same function.

Repeat: Repeats the last command(s) entered at the command line. Useful for repeating a series of commands when writing a full program is unnecessary.

Stop: Sends the ST command to the controller to stop motion on all axes and halt any program that is executing.

Editor: Sends the ED command to the controller to open the Editor window.

Clear: Clears the Terminal display.

Send: Sends a program to the Galil controller. When a file is sent, commands are executed as they are received and are not stored on the controller

Close: Closes the Terminal program, keeping the communications open.

Editor

The editor is used to enter and download programs into the controller. To start the editor type 'ED' in the terminal and hit return or click on the Editor button. A text editor will appear where the program can be entered. Consult the reference manuals for your controller for programming guidelines. To download the program select 'Download to Controller' from the 'File' menu. To upload a program currently in the controller select 'Upload from Controller' from the 'File' menu.

Other DMCWIN Features

File Menu

The file menu has a number a selections to manage files.

Start Up / Shut Down: Used to turn on and off communication to a controller. These functions can be used to stop communication with one controller and establish communication to another controller in the system without quitting the DMC Terminal program.

Upload File: Sends a program in the controller to a file on the hard disk.

Download File: Takes a file from the hard disk and loads it in the controller.

Send File: Sends a file from the hard disk to the controller. Sending is analogous to typing commands at the terminal prompt and carries the same limitations. The send function makes sure each command is accepted before the next is sent. 'Send' files can be of any size.

Download Array: Takes a delimited ASCII file of values and stores them in an array within the controller. Files may be either CR/LF or comma (.csv) delimited. A dialog box must be filled to define the array name, start and stop array index, and the file name. To use this feature, you must have declared an array on the controller in which to download the file.

Upload Array: Sends the contents of an array within the controller to a file on the hard drive. A dialog box must be filled to define the array name, start and stop array index, and the file name.

Convert ASCII to Binary: Converts a file of ASCII commands to their binary equivalent. Note that not all DMC commands have binary translations.

Convert Binary to ASCII: Converts a file of binary commands to their ASCII equivalent.

Send Binary File: Sends a binary file from the hard drive to the controller.

Options Menu

This menu contains a number utilities.

Test: Runs a communication test with a controller. When run, a few commands are sent and a small file is downloaded, uploaded and compared. 'Single' will run the test once, 'Continuous' will run the test until a key on the keyboard is pressed.

Version: Displays the type of controller and the version of the firmware.

Reset: Performs a standard reset on the controller.

Master Reset: Performs a master reset on the controller. If the time out is not set high enough a time out error may occur during the master reset. Clear the error message and continue.

Clear FIFO: Clears the controllers FIFO of all commands and responses. This can be useful if a communication problem causes responses to be misplaced.

Set Timeout: Sets the time the terminal program will wait for a response from the controller before a error message is generated. Normally the default time out value of 1000 milliseconds is sufficient. Larger time out values may be needed for the following conditions:

- A large program or array is uploaded or downloaded
- A master reset is performed
- Trip points, like AM, are used as commands or within send files

Start Device Driver/ Stop Device Driver: Used to load or unload the Galil device driver. This option can be used to have registry changes take effect without rebooting the computer.

Update Flash EEPROM: Used to update the firmware of certain Galil controllers like the DMC-1600, and DMC-1700 series. When selected, the program will ask for a firmware file to download to the EEPROM of the controller. The update process can take up to 5 minutes. Firmware files can be found on the Galil web site: www.galilmc.com

Display Data Record: Used to display a portion of the data record available from controllers with that option. The registry must be configured to use the data record access feature.

Diagnostics: When turned on, the terminal program creates a log file of all the communications with the controller. This can be used for diagnosing communication problems. The diagnostic file gets large quickly so be sure to turn the file on only when you are ready to run a test.

Plug-and-Play

The Galil DMC-1600, DMC-1700, and DMC-1800 are plug-and-play controllers, meaning that the controller's I/O bus address, IRQ, and DMA channel are configured through the PC's BIOS and provided software, not through dip switches or jumpers.

Note: it is strongly recommended that the plug-and-play feature of the DMC-1700 be used only with Windows 95/98. The DMC-1700 has a STD ISA jumper which allows for manual configuration.

To enable the plug-and-play feature of the DMC-1700 for Windows 3.x, you must install the Intel Plug-and-Play Utility provided with the DMC-1700 as a prerequisite to using the Galil Windows DLLs and Device Drivers. This utility can be used to either automatically or manually assign system resources to the Galil controller. No special drivers or utilities are required for the DMC-1600 or DMC-1800.

To enable the plug-and-play feature of the DMC-1700 for Windows NT (version 4.0 only), you must install the ISAPNP.SYS device driver as a prerequisite to using the Galil Windows DLLs and Device Drivers. This device driver is located on the Windows NT 4.0 installation CD-ROM in the \DRIVERS\ISAPNP sub-directory. You can use either the Galil Registry and Terminal program to manually assign system resources to the Galil controller. No special drivers or utilities are required for the DMC-1600 or DMC-1800.

Under Windows 95/98, plug-and-play installation and configuration is automatic. When you start Windows 95/98 after plugging in the controller, Windows 95/98 will prompt you for the "manufacturer's disk". This disk is provided with the DMC-1600, DMC-1700, and DMC-1800 controllers. If you need to, the Windows 95/98 Control Panel System application can be used to manually assign system resources to the Galil controller.

The DMC-2000 can use Universal Serial Bus to communicate with the PC. This device is also plug-and-play, although it is supported under Windows 98 and Windows NT 5.0 only.

For more comprehensive instructions regarding plug and play installation, please consult the 'Getting Started' section of the hardware reference for your controller.

Command Line or Batch Programming Utility

A console mode utility program named DMCBATCH.EXE is available for Win32 environments. This program may be used to send commands, send files, or download programs to the Galil controller. This utility may be useful if you want to send some commands or download a program whenever the host PC is re-booted. Executing DMCBATCH.EXE at the command prompt with the argument "/?" will give you a list of all the available options. For more information, please read the DMCBATCH.TXT file that was installed with DMCWIN.

C/C++ Programmers Tool Kit

The following section describes the basic requirements for programming in C, C++, and Visual Basic, along with simple programming examples. For additional examples, refer to the source code contained in the C, CPP and VB directories located under /DMCWIN.

Our DLL or Dynamic Link Library files are used within Windows development environments to create programs that communicate with Galil controllers. Two libraries are included with the tool kit:

1) **DMCCOM**: This library contains the basic communication routines. Most of these routines are also included in the class library.
2) **DMCMLIB**: This library contains special advanced motion-related functions.

Using these DLLs a program can send commands, upload and download programs and check the status of any controller in a system. Programming with the Galil DLLs is source code compatible across all Windows environments.

Simple Programming Model

No matter what platform is used to develop programs this simple model can be used as a reference. Although there are many functions available in the Galil communication routines only a small number are needed for most programs.

Declare Functions

Before any of the functions can used they must be declared somehow. Galil provides declaration files for Visual Basic, C, and C++. See the details below for each development environment.

Open Communication

Start a communication session with the controller using the DMCOpen routine. Here the controller registry number is passed to the routine and a handle is returned. Use this handle in all subsequent Galil routine calls.

Sending Commands

To send live commands to the controller use the DMCCommand routine. Most DMC commands can be passed to the controller in this way. The response from the controller is returned as a string. Care should be taken not to send commands that could cause a time out. These include trip points like AMX and AIX and DMC commands that do not produce a response from the controller like DL. These special cases can be handled with other routines like DMCFastCommand or DMCWriteData. See the API details for more information.

Downloading Programs

To download a program to the controller use the DMCDownloadFile routine. Here a DMC program that resides on the host computers hard disk is downloaded into the controller. Once downloaded , the program will run on the controller if the DMC command XQ is sent using the DMCCommand routine.

Close Communication

To end a communication session with the controller use the DMCClose routine. This should be done at the end of the program. If the DMCClose is not called Windows will not release the handle provided in the DMCOpen routine. When all the available handles are used Windows will produce an error during the DMCOpen call. During program development the DMCClose function may not be called due to crashes or aborts that cause the PC program to stop abnormally. It may be necessary to reboot the PC to release the handles under these conditions.

Galil Registry

Before communicating with a Galil controller it must be entered in the Galil registry. This registry contains information on each controller in the system including the type of interface, address, and driver used. Some controllers will use Plug and Play to automatically enter information in the registry, others will have to be entered manually.

Note: There are separate registries for 16 and 32 bit programs.

To add, edit, or delete an entry in the Galil registry use the DMCAddGalilRegistry, DMCModifyGalilRegistry, and DMCDeleteGalilRegistry.

DLLs with Visual Basic

Declare Functions

Before using the DLL routines add the module file included in the '\dmcwin\vb' directory named **DMCCOM40.BAS**. This module declares the routines making them available for the VB project. To add this file select 'Add Module' from the 'Project' menu in VB5/6. Use the **DMCCOM.BAS** module for 16 bit applications with VB3.

Sending Commands in VB

Most commands are sent to the controller with the DMCCommand routine. This routine allows any Galil command to be sent from VB to the controller. The DMCCommand routine will return the response from the controller in a string. Before sending any commands the DMCCOpen routine must be called. This routine establishes communication with the controller and needs to be called only once.

This example code illustrates the use of DMCOpen and DMCCommand. Here the controller is sent the command 'TPX' whenever a Command Button is pressed . The response is placed in a text box.

To use this example, start a new Visual Basic project, place a Text Box and a Command Button on a Form, add the appropriate VB module, and type the following code:

```
Dim Controller As Integer
Dim hDmc As Long
Dim RC As Long
Dim ResponseLength As Long
Dim Response As String * 256

Private Sub Command1_Click()
   RC = DMCCommand(hDmc, "TPX", Response, ResponseLength)
   Text1.Text = Val(Response)
End Sub

Private Sub Form_Load()
   ResponseLength = 256
   Controller = 1

   RC = DMCOpen(Controller, 0, hDmc)
End Sub

Private Sub Form_Unload(Cancel As Integer)
   RC = DMCClose(hDmc)
End Sub
```

Where : 'Controller' is the number for the controller in the registry
'hDmc' is the Windows handle used to identify the controller. It is returned by DMCOpen.
'RC' is the return code for the routine
'ResponseLength' is the response string length must be set to the size of the response string
'Response' is the string containing the controller response to the command

Downloading Programs in VB

To download a program to the controller use either the 'DMCDownloadFile' or the 'DMCDownloadFromBuffer' routine. Once the file is downloaded to the controller send the 'XQ' command to start the program. The following example creates a program in memory, downloads it to the controller, and executes it. The code is intended to be executed from within a Visual Basic Module and does not require a Form to execute.

```
Dim Controller As Integer
Dim hDmc As Long
Dim RC As Long
Dim ResponseLength As Long
Dim Response As String * 256
Dim Buffer As String

Private Sub Main()
        ResponseLength = 256
                Controller = 1
                Buffer = "#A;PR1000;BGX;AMX;EN"

                RC = DMCOpen(Controller, 0, hDmc)
                RC = DMCDownloadFromBuffer(hDmc, Buffer, "")
                RC = DMCCommand(hDmc, "XQ", Response, ResponseLength)
        RC = DMCClose(hDmc)
        End
End Sub
```

Note: See the DLL routine descriptions later in this manual for more functions.

DLLs with C/C++

DMCCOM

The DLL routines can be used as included functions or through a class library. All Galil communications programs written in C **must** include the **DMCCOM.H** file and access the DLLs through the declared routine calls.

C++ programs can use the DMCCOM.H routines or use the class library defined in **DMCWIN.H**. DMCWIN.H offers built-in thread safety, making multi-threaded applications easier to implement and more robust. When creating multi-threaded applications with DMCCOM.H, calls to Galil DLL's either should be limited to a single thread or protected within critical sections. Use of critical sections is demonstrated in DMCWIN.CPP.

The DMCCOM.H header file is located in the '\dmcwin\include' directory. C++ programs that use the class library need the files DMCWIN.H and DMCWIN.CPP which contain the class definitions and implementations respectively. These can be found in the '\dmcwin\cpp' directory.

DMCMLIB

These routines are available as functions only and are completely independent from the DMCCOM routines. To use, include the file **DMCMLIB.H** located in the '\dmcwin\include' directory.

Linking Your Application with the Galil DLLs

Because of differences between the Borland and Microsoft 32-bit compilers and linkers, unique library files are distributed for each compiler. Use **DMC32B.LIB** for Borland and **DMC32M.LIB** for Microsoft. The **DMC16.LIB** library file will work for either Borland or Microsoft for 16-bit projects. These library files can be found in the '\dmcwin\lib' directory.

Note: Certain compilers that are not strictly Borland or Microsoft (i.e. Labview CVI or basic ANSI C compiler) may not work with the above .LIB files. In that case, it may be necessary to assemble a new .lib file compatible with your compiler. Consult your software provider or Galil for details.

Use DMCMLIB.LIB for the DMCMLib functions. This library is 32 bit only.

Device Drivers for Windows NT and Windows 95/98

Two sets of device drivers are provided for low-level communication to Galil controllers: WinRT and Galil. The WinRT device drivers developed by Blue Water Systems, Inc. are commercial quality device drivers which provide more or less a generic interface to hardware. The Galil device drivers (developed by Galil) are optimized for use with Galil controllers and will yield significantly better performance. It is highly recommended that you use the Galil device drivers. The Galil Terminal program and registry API functions allow you to choose either one.

Note: if you have a DMC-1600, DMC-1700, or DMC-1800 you must use the Galil device drivers.

Sending Commands Using DMCCOM.H

To initiate communication, declare a variable of type HANDLEDMC (a long integer) and pass the address of that variable in the DMCOpen function. If the DMCOpen function is successful, the variable will contain the handle to the Galil controller which is required for all subsequent function calls. The following simple example program written as a Visual C++ console application tells the controller to move the X axis 1000 encoder counts. Remember to add either DMC32M.LIB or DMC32B.LIB to your project prior to compiling.

```
#include <windows.h>
#include <dmccom.h>

long rc;
HANDLEDMC hDmc;
HWND hWnd;

int main(void)
{
        // Connect to controller number 1
        rc = DMCOpen(1, hWnd, &hDmc);
        if (rc == DMCNOERROR)
        {
          char szBuffer[64];
          // Move the X axis 1000 counts
          rc = DMCCommand(hDmc, "PR1000;BGX;", szBuffer, sizeof(szBuffer));

          // Disconnect from controller number 1 as the last action
          rc = DMCClose(hDmc);
        }
        return 0;
}
```

Downloading Programs using DMCCOM.H

The following example downloads a file from the hard drive using DMCDownloadFile() and executes it by sending the command XQ.

Note: Files can also be downloaded from a buffer by using DMCDownloadFromBuffer().

```
long rc;
HANDLEDMC hDmc;
HWND hWnd;
char szBuffer[64];
char filename[] = ''c:\\dmcwin\\yourfile.dmc'';

int main(void)
{
        // Connect to controller number 1
        rc = DMCOpen(1, hWnd, &hDmc);
        //Download file
        rc = DMCDownloadFile(hDmc, filename, NULL);
        //Start program on controller at label 'A'
        rc = DMCCommand(hDmc, ''XQ#A'', szBuffer, sizeof(szBuffer));

        //Close communication
        rc = DMCClose(hDmc);

        return 0;
}
```

In this example the label was set to NULL, causing the existing program to be overwritten. If a label is used in place of NULL, the DMCDownloadFile will try to find that label within the program already present on the controller and start the download from there. A "#" will append the program to the existing program.

Note: All existing programs must be halted before an upload or download occurs.

Configuring the Galil Registry Using DMCCOM.H

The Galil registry API functions can be used to test if your controller is registered at the beginning of your application. The following code illustrates this:

```
GALILREGISTRY galilregistry;

// Is the controller registered?
if (DMCGetGalilRegistryInfo(1, &galilregistry) == DMCERROR_CONTROLLER)
{
  long rc;
  unsigned short controller;

  // Fill in your controller info, e.g., galilregistry.usAddress = 1000
  // Register the controller
  rc = DMCAddGalilRegistry(&galilregistry, &controller);
}
```

Reading DMA and the QR Record

Certain controllers offer a second channel of communication that can be used to provide information such as position and I/O status. This information is contained in a data record that is around 255 bytes in size. For PC based controllers the second channel can be accessed by DMA or a polled record. For controllers with only one communication port like the DMC-1802 and DMC-2000 the data record is requested by the QR command.

Reading DMA

DMA is configured to automatically update the data record in the memory of the PC at defined intervals (2^n milliseconds). Our DLLs can read that record in microseconds and provide the data to the program. There are two ways to read the DMA record:

1) Use the DMCGetDataRecordByItemId routine to return a single record item as in this Win32 console example.

```
#include "windows.h"
#include "Dmccom.h"
#include "stdio.h"

int main(void)
{
        long rc;
        HANDLEDMC hDmc;
        HWND hWnd=0;
        ULONG DMAlength=0;
        USHORT DataType;
        ULONG DMAdata;

        //Connect to controller #1
        rc = DMCOpen(1,hWnd, &hDmc);

        if (rc==DMCNOERROR)
        {
                //refresh local copy of DMA record
                rc = DMCRefreshDataRecord(hDmc, DMAlength);

                //get data record data item
                rc = DMCGetDataRecordByItemId(hDmc,DRIdAxisMotorPosition,
                DRIdAxis1, &DataType, &DMAdata);

                printf("Position of X is: %d%",(int)DMAdata);
        }

        rc = DMCClose(hDmc);

        return 0;
}
```

DMCGetRecordByItemId returns a single data record item. The item returned is defined by the second parameter passed to the function, in this case the constant DRIdAxisMotorPosition passed to get the motor position, and the third parameter DRIdAxis1 the axis number constant. All the constants used to define the record item are part of a header file 'dmcdrc.h' and are automatically included by the Dmccom.h file.

DMCRefreshDataRecord is used to copy the DMA record into a buffer in the DLL. All calls to read a record item are done on this local copy. If the data is time sensitive the buffer should be refreshed just before any call to DMCGetRecordByItemId.

Note: Previous versions of the DLLs used offsets to read the record item. The offsets can change from one firmware revision to the next making the DLL calls retrieve the wrong data. To prevent this the item id calls where created that automatically adjust to firmware changes.

2) Copy the DMA data into a local data structure as shown below:

```
#include "windows.h"
#include "Dmccom.h"
#include "stdio.h"

int main(void)
{
        long rc;
        HANDLEDMC hDmc;
        HWND hWnd=0;
        ULONG DMAlength=0;
        USHORT RecordSize;
        DMCDATARECORD MyDataRecord;

        //Connect to controller #1
        rc = DMCOpen(1,hWnd, &hDmc);

        if (rc==DMCNOERROR)
        {
                //refresh local copy of DMA record
                rc = DMCRefreshDataRecord(hDmc, DMAlength);

                //make sure the data record will fit in the structure
                rc = DMCGetDataRecordSize(hDmc, &RecordSize);

                if (sizeof(MyDataRecord) >= RecordSize)
                        //copy DMA record to the structure
                        rc = DMCCopyDataRecord(hDmc, (PVOID)&MyDataRecord);

                printf("Position of X is:%d%", (int)
                MyDataRecord.AxisInfo[0].MotorPosition);
        }

        rc=DMCClose(hDmc);

        return 0;
}
```

Here the DMA record is first copied to a local buffer with DMCRefreshDataRecord and then placed into our data record structure with DMCCopyDataRecord. As with the single item method the data must be refreshed before being read to make sure it is up to date. The structure used in this example is the same used for holding the data record received from the QR command.

Using the QR Command

For controllers that do not have a second communication channel the QR command can be used instead of DMA to retrieve a data record. When the QR command is sent the controller will respond with a binary record about 255 bytes in size. To place this response in the structure use the DMCCommand routine as shown below:

```
#include "windows.h"
#include "Dmccom.h"
#include "stdio.h"

int main(void)
{
        long rc;
        HANDLEDMC hDmc;
        HWND hWnd=0;
        DMCDATARECORDQR MyDataRecordQR;

        //Connect to controller #1
        rc = DMCOpen(1,hWnd, &hDmc);

        if (rc==DMCNOERROR)
        {

                rc = DMCCommand(hDmc, "QR\r", (LPCHAR)&MyDataRecordQR,
                sizeof(MyDataRecordQR));

                printf("X Pos: %d",
                MyDataRecordQR.DataRecord.AxisInfo[0].MotorPosition);

        }

        rc=DMCClose(hDmc);

        return 0;
}
```

The QR is sent using the DMCCommand routine and a pointer to the response is returned. This pointer has been cast as a LPCHAR so that the DMCCommand would except it.

The data record structure is located in the “dmcdrc.h” file which is automatically included by the “Dmccom.h” file.

Sending Commands Using the Class Library

Most of the API functions are available in the class library. An object of type CDMCWin is declared which allows access to the class functions. When the class constructor is called the communication channel is opened. To send commands use the .command function. Unlike the routine calls using DMCCOM.H the handle does not need to be passed to all subsequent class calls. Make sure to call the .close routine before ending the program so Windows can release the handle.

This example tells the controller to start jogging the Y axis at 10000 counts per second:

```
CDMCWin controller(1, hWnd, 0);
controller.Command(“JG ,10000; BGY’’, szBuffer, sizeof(sxBuffer));
```

DMCMLIB Functions

These routines perform advanced functions that can extend the capabilities of the motion controller.

DMCEllipse

This function allows a controller to perform elliptical motion. When it is called it will create a text file containing 'VP' commands that define an ellipse. This file can then be sent to the card to perform the motion using DMCSendFile or inserted into a larger sequence of 'VP' and 'CR' commands.

LONG GALILCALL DMCEllipse(char *filename, int FirstOffset, int SecondOffset, double a, double b, double theta, double delta_theta, double increment, double rotation);

filename
Location used to save the calculated VP commands

FirstOffset
Starting location for the first axis in encoder counts

SecondOffset
Starting location for the second axis in encoder counts

a and *b*
Major and minor axis lengths in encoder counts

theta
Starting angle in degrees

delta_theta
Total angular distance to travel in degrees

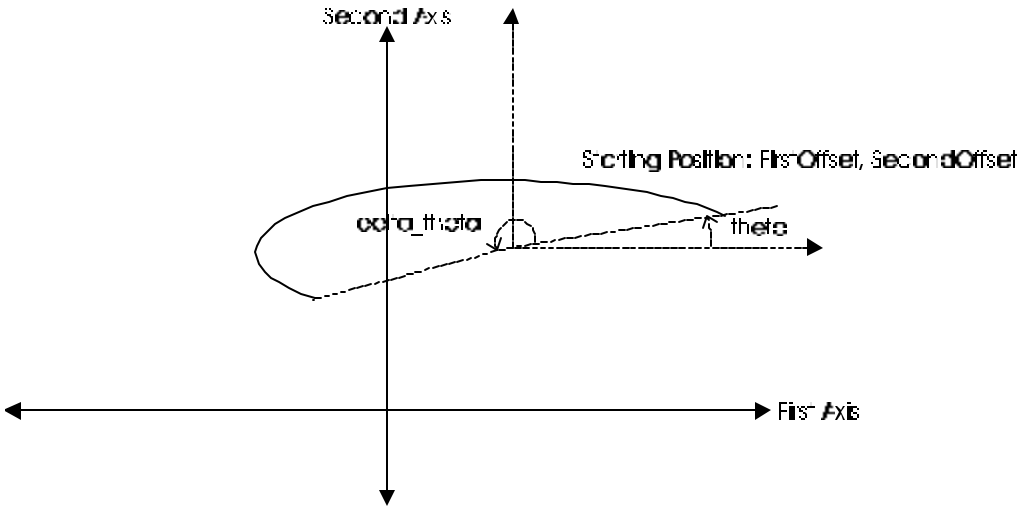
increment
Increment advancement of theta used to calculate a new set of points in degrees

rotation
Angular translation of the major axis in degrees

Description
The parameters sent to the DMCEllipse routine are similar to those for the 'CR' command. An ellipse is defined by the starting angle, ending angle, and the major and minor axis. This routine takes those values and creates an ellipse using 'VP' commands. The end result is a sequence of line segments that produce the curve.

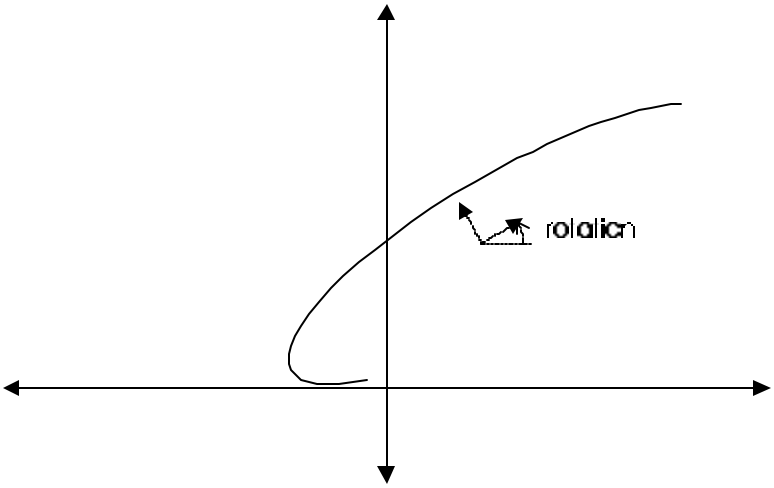
The first point calculated is at the angular position *theta*. The next is at *theta+increment* and so on until the final angle (*theta+delta_theta*) is reached. Obviously, the length of the computed segments will determine how smooth the curve will be. To control the coarseness of the curve use a small increment value. The smaller the increment the more points are created.

FirstOffset and *SecondOffset* determine the position of the first point of the curve.



Ellipse with starting angle of 30 degrees, a delta angle of 190 and an offset starting position.

Using *rotation* the entire ellipse can be translated a number of degrees.



Although it is not clear by the drawing, the starting position is unchanged by the rotation.

Note: The DMCEllipse function can be used to create circles with a very large radius. The Galil 'CR' command is limited to 6000000. To make larger circles make a=b=radius of circle.

Error Codes

If there are no errors the DMCEllipse routine returns a 0. Here are the conditions that can generate errors and the code that is returned.

Condition	Code
Bad arguments: delta_theta positive but increment negative and visa versa; delta_theta = 0; increment=0; a or b <=0	-12
Could not open output file	-4

Example

This C program uses the DMCEllipse routine to make an ellipse. The ellipse is calculated and put into a file. Then the file is sent to the controller to perform the motion.

```
#include "DMCMLIB.H"

int main(void)
{
        long rc;
        char filename[] = "c:\\junk\\ellout.sen";
        HANDLEDMC hDmc;
        HWND hWnd;
        char szBuffer[80];

        //make the output file
        rc = DMCEllipse(filename, 1000, 1500, 2000.0, 800.0, 30.0, 190.0, 5.0, 30.0);

        //send the output file
        rc = DMCOpen(1, hWnd, &hDmc);
        rc = DMCSendFile(hDmc, filename);

        //start motion
        rc = DMCCommand(hDmc, "VE", szBuffer, sizeof(szBuffer));
        rc = DMCCommand(hDmc, "BGS", szBuffer, sizeof(szBuffer));

        rc = DMCClose(hDmc);

        return 0;
}
```

Here the ellipse started with offsets 1000 and 1500, with a major axis of 2000 , minor axis of 800, starting angle of 30, delta angle of 190 and a rotation of 30. To make the motors move the DMCSendFile is used to take the VP commands from the file and put them in the coordinated motion segment buffer inside the controller. The 'LE' sent by DMCCommand tells the controller the sequence is ended and the 'BGS' command starts the motion.

It is not necessary to include DMCCOM.H in the above example because it is already included in the DMCMLIB.H library header file.

DMCSpline

Use DMCSpline to fit a curve through the points that define a 1-8 axis move to remove infinite accelerations around corners.

LONG GALILCALL DMCSpline(char *source_filename, char *output_filename, double delta_vector)

source_filename
Input file

output_filename
Output file

delta_vector
Vector distance between spline fit points

Description

DMCSpline fits a curve through the points in the input file using the equation:

$$f(v) = A + B \cdot v + C \cdot v^2 + D \cdot v^3$$

where v = vector distance

For each segment in the input file coefficients are calculated for each axis then the points are created as a function of the vector distance. A point is created for every v where v = 0 to the total vector distance with increments of *delta_vector.*

The input file must be in the following form:

LI nnnnnnnn,nnnnnnnn,nn…. nn, nnnnnnnn < sssssss
LI nnnnnnnn,nnnnnnnn,nn…. nn, nnnnnnnn < sssssss
LI nnnnnnnn,nnnnnnnn,nn…. nn, nnnnnnnn < sssssss
…

where nnnnnnnn = number in the range +/-8,000,000
sssssssssss = speed of the vector in the range 2-8,000,000

Each segment must be at least 2 counts long or an error will occur. If a speed value is not specified do not use the less than sign. A negative speed is ignored. The number of axis is defined by the number of commas in the first line. Make sure the number of commas match on each and every line. The 'LI' (or any non-numeric characters) at the beginning of the line is ignored and can be removed.

Error Codes

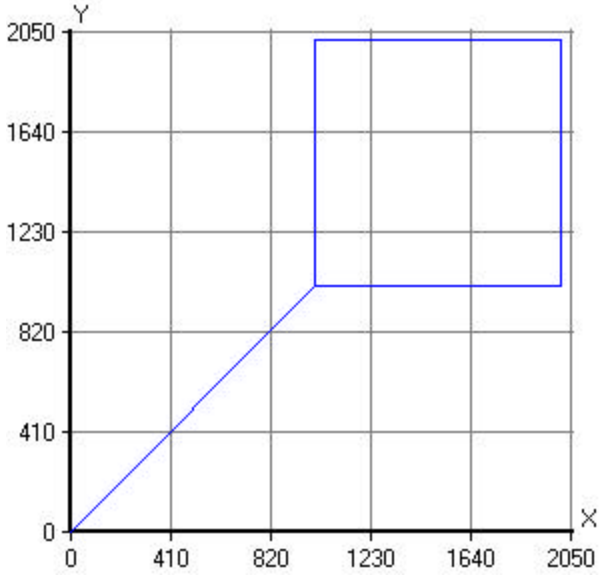
If there are no errors the DMCSpline routine returns a 0. Here are the conditions that can generate errors and the code that is returned.

Condition	Code
Axis distance greater than 8 characters long or speed over 12 characters long.	-12
More than 8 axis are specified in input file	-12
Could not open output or input file or input file is empty	-4

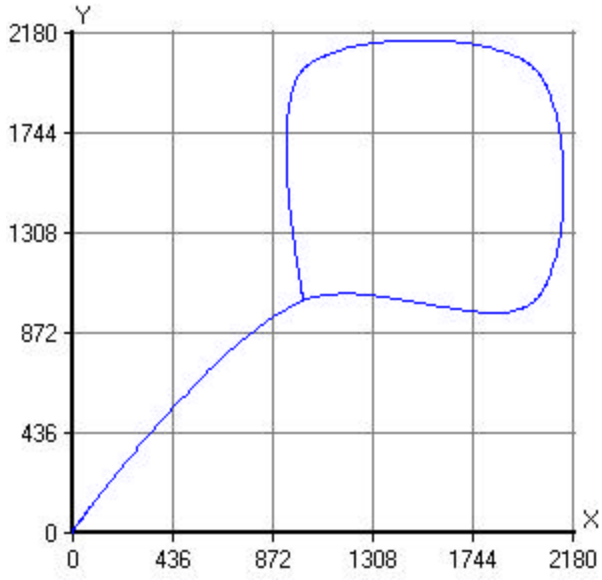
Example
This input file defining a two axis move is used:

```
LI 1000,1000
LI 1000,0
LI 0,1000
LI -1000,0
LI 0,-1000
```

If plotted this is what the original file looks like:



After the spline fit has been run the path looks like this:



Note: A smaller delta_vector will result in a smoother curve but it will also create more LI points in the output file. If the delta_vector is larger than the segment then the no smoothing will occur on that segment.

Here is a sample C program to create a spline motion. This program uses splinein.txt for the input file, splinout.txt for the output and 10 counts for the vector increment.

```
#include "DMCMLIB.H"
int main(void)
{
        long rc;
        char outfile[] = "c:\\junk\\splinout.txt";
        char infile[]= ‘‘c:\\junk\\splinein.txt”;
        HANDLEDMC hDmc;
        HWND hWnd;
        char szBuffer[80];

        //make the spline fit
        rc = DMCSpline(infile, outfile, 10.0);

        //send the output file
        rc = DMCOpen(1, hWnd, &hDmc);
        rc = DMCSendFile(hDmc, filename);

        //start motion
        rc = DMCCommand(hDmc, "LE", szBuffer, sizeof(szBuffer));
        rc = DMCCommand(hDmc, "BGS", szBuffer, sizeof(szBuffer));

        rc = DMCClose(hDmc);

        return 0;
}
```

DMCSCurve

This function generates an S-curve motion profile for a given point to point move.

ULONG GALILCALL DMCSCurve(char Axis, unsigned long Distance, unsigned long Speed, unsigned long Acceleration, unsigned long Jerk, PSZ FileName)

Axis
Axis on which S-curve is to be performed

Distance
Length of move, expressed in encoder counts

Speed
Desired slew speed (encoder counts/sec)

Acceleration
Desired acceleration rate (encoder counts/sec$^{2)}$. The deceleration rate is set to the same value so that the motion profile is symmetrical.

Jerk
Maximum acceptable jerk (encoder counts/sec^3)

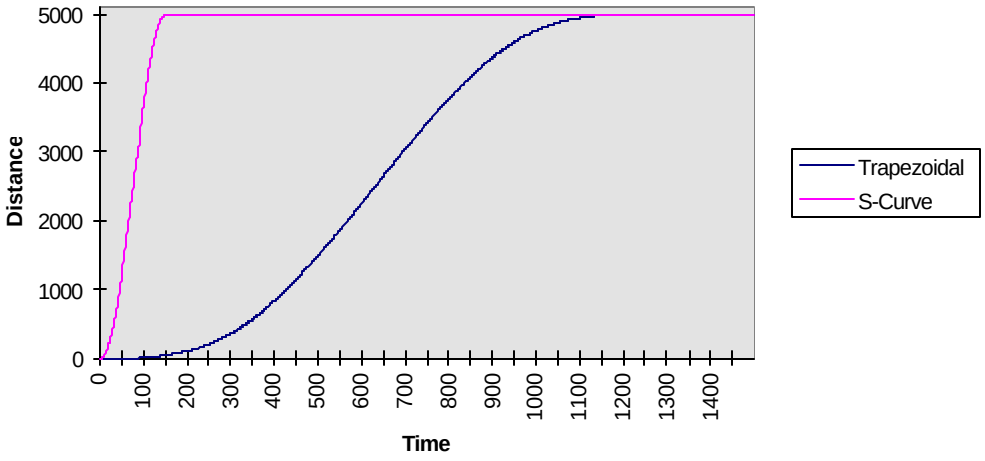
FileName
Destination path for DMC output file

Description

This function generates an S-curve motion profile based on a set of motion constraints. If the specified constraints would result in a discontinuous profile (i.e. triangular acceleration or velocity) the function optimizes the parameters so that a true S-curve is generated. The motion profile is generated through a series of contour data (CD) moves.

Note: there is a small amount of rounding error in this function which may cause the actual length of a move to vary from the specified length by a few counts (usually less than 10).

The following figure shows an exaggerated example of an S-curve. The trapezoidal graph is a 5000 count Position Relative move with a slew speed of 25000 and an acceleration of 256000. The S-curve graph is the same move generated by the DMCSCurve function with a limitation on jerk of 10000 counts/sec^3.



Error Codes
If there are no errors, DMCHelix returns a 0. The following codes are returned in the event of an error:

Condition	Code
Could not open output file	-4
Out of memory	-8
Bad arguments: One or more arguments to a function was NULL or invalid	-12
Invalid Distance parameter	-97
Invalid Speed parameter	-98
Invalid Acceleration parameter	-99

Example
The following C program uses the DMCSCurve function to generate an S-curve, send the resulting output file to the controller and execute it.

```
#include “dmcmlib.h”

void main(void)
{
        long rc;
        char filename[] = "c:\\windows\\desktop\\scurve.dmc";
        HANDLEDMC hDmc;
        HWND hWnd;

// make output file
rc = DMCSCurve('X',10000,50000,256000,15625000,filename);
// DMCSCurve(char Axis, unsigned long Distance, unsigned long Speed,
// unsigned long Acceleration, unsigned long Jerk, PSZ FileName)

// send output file
        rc = DMCOpen(1, hWnd, &hDmc);
        rc = DMCDownloadFile(hDmc, filename);

// start motion
        rc = DMCCommand(hDmc, “XQ”, szBuffer,sizeof(szBuffer));
}
```

In the above example, the X axis is used to generate an S-curve profile. The length of the move is 10000 counts, speed is 50000 counts/sec, acceleration is 256000 counts/sec^2 and jerk is 15625000 counts/sec^3.

After the output file has been generated, it is downloaded to the controller and executed using routines from DMCCOM.H, which has already been included in the DMCMLIB.H header file.

DMCHelix

This function generates a helical motion profile. Two axes follow a coordinated circular path while a third axis generates a linear motion perpendicular to the circle.
The function also compensates for offset error (Desired Distance - Actual Distance) which results from round-off error in the GR command.

ULONG GALILCALL DMCHelix(char PlaneAxis1, char PlaneAxis2, char TraverseAxis, short Pitch, ulong Radius, short StartAngle, long Dist, ulong Speed, ulong Accel, ulong Decel, PSZ FileName)

PlaneAxis1, PlaneAxis2
Coordinated axes for circular motion

TraverseAxis
Perpendicular axis about which the helix is generated

Pitch
Pitch of helix, expressed in encoder counts traversed per circle. The sign of the pitch determines the direction of the circle: Positive pitch= counter-clockwise, Negative pitch=clockwise

Radius
Radius of circle, expressed in encoder counts

StartAngle
Beginning angle of helix, expressed in degrees

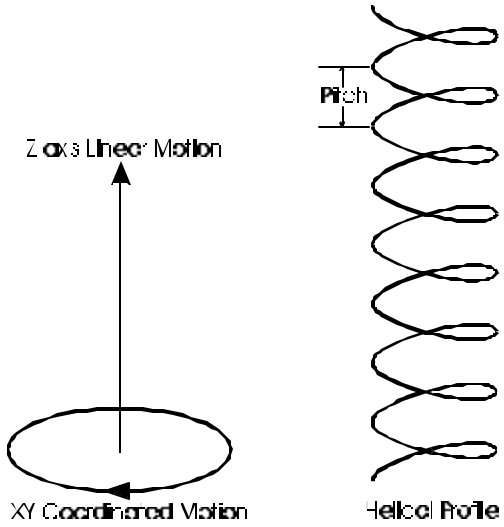
Dist
Desired traverse distance of geared axis

Speed, Accel, Decel
Helical motion parameters. These parameters apply directly to the circular motion, not the traverse motion.

FileName
Destination path for DMC output file

Description

Helical motion is a combination of two types of motion: a circular move in one plane and a linear move perpendicular to the plane of the circle. *PlaneAxis1* and *PlaneAxis2* define the plane of the circle while *TraverseAxis* defines the axis used for the linear move. The *Pitch* parameter relates the circular motion to the linear motion: it is the number of encoder counts traversed by the *TraverseAxis* per circle. The sign of *Pitch* determines the direction of the circle. For example, in the figure below *Pitch* is a negative number since the circle is drawn in the clockwise direction.



Error Codes
If there are no errors, DMCHelix returns a 0. The following codes are returned in the event of an error:

Condition	Code
Bad arguments: One or more arguments to a function was NULL or invalid	-12
Could not open output file	-4

Example
The following C program uses the DMCHelix function to generate a helix, send the resulting output file to the controller and execute it.

```
#include “dmcmlib.h”

void main(void)
{
        long rc;
        char filename[] = "c:\\windows\\desktop\\helix.dmc";
        HANDLEDMC hDmc;
        HWND hWnd;

// make output file
        rc = DMCHelix('X', 'Y', 'Z', -360, 5000, 0, 100000, 50000, 256000, 256000, filename);
// DMCHelix(PlaneAxis1, PlaneAxis2, TraverseAxis, Pitch, Radius, StartAngle, Dist, Speed,
// Accel, Decel, FileName)

// send output file
        rc = DMCOpen(1, hWnd, &hDmc);
        rc = DMCDownloadFile(hDmc, filename);

// start motion
        rc = DMCCommand(hDmc, “XQ”, szBuffer,sizeof(szBuffer));

}
```

In the above example, the X and Y axes are used for circular motion while the Z axis is used for linear motion. The pitch of the helix is -360, which means that the Z axis will move forward 360 counts for every XY circle. Since the sign of the

pitch is negative, the helix will be generated in the clockwise direction. The radius is 5000, start angle is 0, traversed distance is 100000, speed is 50000, and both acceleration and deceleration are 256000.

DMCGeneralTuning

Use this function to create a DMC program file that can be downloaded and run to automatically tune the servo motor.

LONG GALILCALL DMCGeneralTuning(CHAR chAxis, USHORT usStepSize, PSZ pszFileName)

chAxis
The axis to be tuned.

UsStepSize
The step size used during the tuning process.

PszFileName
File name use to save the auto tune program.

Description

General tuning is a method used by Galil to automatically tune servo motors. The DMCGeneralTuning function creates a Galil program and stores it in a file. The file must be downloaded and then executed on a Galil controller to perform the tuning.

Once the Galil program is run the servo motor will be stepped back and forth and its position error monitored. Gradually the PID values are increased until the motor becomes unstable. The values are then backed off. The final values should be appropriate for most servo systems. Manual fine tuning may be needed to produce the best system response.

While general tuning is the most flexible tuning method from Galil it may not work on all systems. If this is the case we recommend using WSDK where many other tuning methods are available.

Error Codes

If there are no errors the DMCGeneralTuning routine returns a 0. A non zero return value indicates an error occurred. See the header file DMCCOM.H for a definition of the error codes.

Example

This C program creates the automatic tuning file. This file is then downloaded and run on the controller.

```
#include "windows.h"
#include "dmccom.h"
#include "dmcmlib.h"

int main(void)
{
        int rc;
        char filename[] = "c:\\junk\\gtune.dmc";
        USHORT stepsize = 300;
        HANDLEDMC hDmc;
        HWND hWnd=0;
        char szBuffer[64];
        char axis='X';

        //Create tuning file
        rc = DMCGeneralTuning(axis, stepsize, filename);

        //Open communications with the controller
```

```
        rc = DMCOpen(1, hWnd, &hDmc);

        //Download the file
        rc = DMCDownloadFile(hDmc, filename, NULL);

        //Execute the file
        rc = DMCCommand(hDmc, "XQ", szBuffer, sizeof(szBuffer));

        return 0;
}
```

DMCAutoTuning

Use this function to create a DMC program file that can be downloaded and run to automatically tune the servo motor.

LONG GALILCALL DMCAutoTuning(CHAR chAxis, double dPulseMagnitudeVolts, USHORT usPulseDurationMS, LPSTR lpstrFileName)

chAxis
The axis to be tuned.

dPulseMagnitudeVolts
Pulse magnitude in volts to be sent to the amplifier.

usPulseDurationMS
Pulse duration in milliseconds.

lpstrFileName
File name used to save the auto tune program.

Description

DMCAutoTuning is an alternative to the DMCGeneralTuning function to automatically tune servo motors. The DMCAutoTuning function creates a Galil program and stores it in a file. The file must be downloaded and then executed on a Galil controller to perform the tuning.

The program generated by DMCAutoTuning will send a series of pulses to the amplifier of the specified voltage and duration. These disturbances are used to determine the optimum crossover frequency of the system. After the best crossover frequency is found, the PID values are adjusted for best response at the selected frequency. The final values should be appropriate for most servo systems. Manual fine tuning may be needed to produce the best system response.

While auto tuning is the most flexible tuning method from Galil it may not work on all systems. If this is the case we recommend using WSDK where many other tuning methods are available.

Error Codes

If there are no errors the DMCAutoTuning routine returns a 0. A non zero return value indicates an error occurred. See the header file DMCCOM.H for a definition of the error codes.

Example

This C program creates the automatic tuning file. This file is then downloaded and run on the controller.

```
#include "windows.h"
#include "dmccom.h"
#include "dmcmlib.h"

int main(void)
{
        int rc;
        char filename[] = "c:\\junk\\atune.dmc";
        double pulsesize = 5.5;
        USHORT pulseduration 20;

        HANDLEDMC hDmc;
        HWND hWnd=0;
```

```
        char szBuffer[64];
        char axis='X';

        //Create tuning file
        rc =  DMCAutoTuning(axis, pulsesize, pulseduration, filename)

        //Open communications with the controller
        rc = DMCOpen(1, hWnd, &hDmc);

        //Download the file
        rc = DMCDownloadFile(hDmc, filename, NULL);

        //Execute the file
        rc = DMCCommand(hDmc, "XQ", szBuffer, sizeof(szBuffer));

        return 0;
}
```

DLL API List for DMCCOM

The following DLL routines are available with the DMCCOM.H header file. For more information see the detailed descriptions later in this manual.

DMCOpen	Open communications w/interrupt support
DMCOpen2	Open communications w/interrupt support
DMCGetHandle	Return the controller handle
DMCChangeInterruptNotification	Notify for interrupt by handle or thread
DMCClose	Close communications
DMCCommand	Send a command
DMCFastCommand	Send special commands
DMCBinaryCommand	Send binary command
DMCGetUnsolicitedResponse	Read card messages
DMCGetAdditionalResponseLen	Read long response length
DMCGetAdditionalResponse	Read long response
DMCError	Read error message
DMCClear	Clear FIFO
DMCReset	Reset controller
DMCMasterReset	Master reset controller
DMCVersion	Get firmware version
DMCDownloadFile	Download a file from hard disk
DMCDownloadFromBuffer	Download file from buffer
DMCUploadFile	Upload file to hard disk
DMCUploadToBuffer	Upload file to buffer
DMCSendFile	Send a file
DMCSendBinaryFile	Send a binary file
DMCArrayDownload	Download an array
DMCArrayUpload	Upload an array
DMCCommand_AsciiToBinary	Convert ASCII command to binary
DMCCommand_BinaryToAscii	Convert a binary command to ASCII
DMCFile_AsciiToBinary	Convert file of ASCII to binary
DMCFile_BinaryToAscii	Convert file of binary to ASCII
DMCReadSpecialConversionFile	Special conversion file
DMCRefreshDataRecord	Request a new data record
DMCGetDataRecord	Return part of the data record
DMCGetDataRecordByItemId	Return part of the data record preferred method.
DMCGetDataRecordSize	Returns number of bytes in data record
DMCCopyDataRecord	Copy DMA record into a structure
DMCGetDataRecordRevision	Get version of record
DMCDiagnosticsOn	Start diagnostic file
DMCDiagnosticsOff	Stop diagnostics

DMCGetTimeout	Return the current timeout
DMCSetTimeout	Set timeout
DMCWriteData	Low level write routine
DMCReadData	Low level read routine
DMCAddGalilRegistry and DMCAddGalilRegistry2	Add a controller to the registry
DMCModifyGalilRegistry and DMCModifyGalilRegistry2	Modify a registry entry
DMCGetGalilRegistryInfo and DMCGetGalilRegistryInfo2	Read registry for one controller
DMCEnumGalilRegistry and DMCEnumGalilRegistry2	Read entire registry
DMCDeleteGalilRegistry	Delete a controller from registry
DMCRegisterPnpControllers	Update registry for PNP controllers
DMCWaitForMotionComplete	Wait for motion complete
DMCStartDeviceDriver	Starts Galil device driver
DMCStopDeviceDriver	Stops Galil device driver
DMCDownloadFirmwareFile	Download Firmware
DMCSelectController	Selection Dialog
DMCEditRegistry	Registry Dialog

DLL API Details

Error Codes

All functions return an error code. If the function returned DMCNOERROR (0), the function completed successfully. An error code less than 0 is a local error (see the error codes above). An error code greater than 0 is an Win32 API error (32-bit DLLs). These are documented in the Win32 Programming Reference.

DMCNOERROR 0
No error occurred.

DMCERROR_TIMEOUT -1
A time-out occurred while waiting for a response from the Galil controller.

DMCERROR_COMMAND -2
There was an error with the command sent to the Galil controller. The full message depends on the error code returned from the Galil controller.

DMCERROR_CONTROLLER -3
The Galil controller could not be found in Windows registry.

DMCERROR_FILE -4
File could not be opened. This error usually occurs when the file name is invalid or the file can not be found.

DMCERROR_DRIVER -5
Device driver could not be opened, or a read or write error occured.

DMCERROR_HANDLE -6
Invalid Galil controller handle. This error will occur if you try to communicate with the Galil controller without first calling DMCOpen.

DMCERROR_HMODULE -7
Support dynamic link library could not be loaded. This error will occur if DMCBUS16.DLL or DMCBUS32.DLL can not be found for bus controllers and if DMCSER16.DLL or DMCSER32.DLL can not be found for serial controllers.

DMCERROR_MEMORY -8
Out of memory.

DMCERROR_BUFFERFULL -9
Response from the controller was larger than the response buffer supplied. The user-supplied buffer for DMCCommand was too small to fit the complete response from the Galil controller. You can call DMCGetAdditionalResponseLen to find out how much data is left to retrieve and then call DMCGetAdditionalResponse to retrieve the additional response data.

DMCERROR_RESPONSEDATA -10
Response from the controller overflowed the internal additional response buffer. The response from the Galil controller was so large that it overflowed both the user-supplied buffer and the internal additional response buffer. You must increase the size of the user-supplied buffer.

DMCERROR_DMA -11
Could not communicate with DMA channel.

DMCERROR_ARGUMENT -12
One or more required arguments to a DMC API function call was NULL.

DMCERROR_DATARECORD -13
Could not access DMA data record.

DMCERROR_DOWNLOAD -14
File download failed. The problem is most likely a file that has too many lines or one or more lines which exceed the line length restriction.

DMCERROR_FIRMWARE -15
Could not update the controller's firmware.

DMCERROR_CONVERSION -16
Could not convert the DMC command (ASCII to binary or binary to ASCII).
DMCERROR_REGISTRY -17
Could not access or modify the controller's registry information. You need to log-on to Windows (Windows NT that is) with Administrator privileges.
DMCERROR_RESOURCE -18
Windows reports a resource conflict with the current hardware configuration. This error could occur if you tried to manually assign resources to a Galil plug-and-play controller.
DMCERROR_BUSY -19
Could not write to the controller because it is currently busy. You may need to set the time-out value higher because the previous command is taking longer to process by the controller than expected.
DMCERROR_DEVICE_DISCONNECTED -20
A Windows plug-and-play controller, such as one which communicates through USB, was disconnected from the system. You must close the handle to the controller (using DMCClose) as soon as possible.

API Routine Details

The following section provides detailed descriptions of each DMCCOM command. For each command listed a C function prototype is provided, along with descriptions of each parameter.

Syntax varies for Visual Basic and C++; you can compare the syntax differences by looking at the DMCCOM40.BAS for Visual Basic and DMCWIN.CPP for C++.

DMCOpen
LONG FAR GALILCALL DMCOpen(USHORT usController, HWND hwnd, PHANDLEDMC phdmc);

Open communications with the Galil controller. The handle to the Galil controller is returned in the argument phdmc. For every DMCOpen, you must issue a DMCClose.

Note: hwnd is not used for controllers which do not support bus interrupts.

usController
A number between 1 and 16. Up to 16 Galil controllers may be addressed per process.
hwnd
The window handle to use for notifying the application program of an interrupt via PostMessage.
phdmc
Buffer to receive the handle to the Galil controller to be used for all subsequent API calls. Users should declare a variable of type HANDLEDMC and pass the address of the variable to the function.

DMCOpen2
LONG FAR GALILCALL DMCOpen2(USHORT usController, LONG lThreadID, PHANDLEDMC phdmc);

Open communications with the Galil controller with interrupt handling. The handle to the Galil controller is returned in the argument phdmc. For every DMCOpen2, you must issue a DMCClose.

Note: lThreadIDis not used for controllers which do not support bus interrupts.

usController
A number between 1 and 16. Up to 16 Galil controllers may be addressed per process.
lThreadID
The thread ID to use for notifying the application program of an interrupt via PostThreadMessage.
phdmc

Buffer to receive the handle to the Galil controller to be used for all subsequent API calls. Users should declare a variable of type HANDLEDMC and pass the address of the variable to the function.

DMCGetHandle
LONG FAR GALILCALL DMCGetHandle(USHORT usController, PHANDLEDMC phdmc);

Get the handle for a Galil controller which was already opened using DMCOpen or DMCOpen2. The handle to the Galil controller is returned in the argument phdmc.

usController
A number between 1 and 16. Up to 16 Galil controllers may be addressed per process.
phdmc
Buffer to receive the handle to the Galil controller to be used for all subsequent API calls. Users should declare a variable of type HANDLEDMC and pass the address of the variable to the function.

DMCChangeInterruptNotification
LONG FAR GALILCALL DMCChangeInterruptNotification(HANDLEDMC hdmc, LONG lHandle);

Change the window handle used in DMCOpen or the thread ID used in DMCOpen2. This value is for notification of interrupts.

Note: This function is not used for controllers which do not support bus interrupts.

hdmc
Handle to the Galil controller.
lHandle
New window handle or thread ID.

DMCClose
LONG FAR GALILCALL DMCClose(HANDLEDMC hdmc);

Close communications with the Galil controller.

hdmc
Handle to the Galil controller.

DMCCommand
LONG FAR GALILCALL DMCCommand(HANDLEDMC hdmc, PSZ pszCommand, PCHAR chResponse, ULONG cbResponse);

Send a DMC command in ASCII format to the Galil controller.

Note: This function can only send commands or groups of commands up to 1024 bytes long.

hdmc
Handle to the Galil controller.
pszCommand
The command to send to the Galil controller.
pchResponse
Buffer to receive the response data. If the buffer is too small to receive all the response data from the controller, the error code DMCERROR_BUFFERFULL will be returned. The user may get additional response data by calling the function DMCGetAdditionalResponse. The length of the additional response data may ascertained by call the function DMCGetAdditionalResponseLen. If the response data from the controller is too large for the internal additional response buffer, the error code DMCERROR_RESPONSEDATA will be returned.

cbResponse
Length of the buffer.

DMCFastCommand
LONG FAR GALILCALL DMCFastCommand(HANDLEDMC hdmc, PSZ pszCommand);

Send a DMC command in ASCII format to the Galil controller and do not wait for a response. Use this function with extreme caution as command errors will not be reported and the out-going FIFO or communications buffer may fill up. In some applications it may be necessary to first send the Galil command 'CW,1' to allow the controller to continue program execution when the output FIFO is full.

Use this function for Galil commands which do not return an acknowledgment from the controller such as providing data for the Galil DL and QD commands. Other uses may be to send contour data.

Note: This function can only send commands or groups of commands up to 1024 bytes long.

hdmc
Handle to the Galil controller.
pszCommand
The command to send to the Galil controller.

DMCBinaryCommand
LONG FAR GALILCALL DMCBinaryCommand(HANDLEDMC hdmc, PBYTE pbCommand, ULONG ulCommandLength, PCHAR pchResponse, ULONG cbResponse);

Send a DMC command in binary format to the Galil controller. Most commands have a binary equivalent that will be processed faster by the controller.

Note: This function is for the DMC-1600 and DMC-1700 only.

hdmc
Handle to the Galil controller.
pbCommand
The command to send to the Galil controller in binary.
ulCommandLength
The length of the command (binary commands are not null-terminated).
pchResponse
Buffer to receive the response data. If the buffer is too small to receive all the response data from the controller, the error code DMCERROR_BUFFERFULL will be returned. The user may get additional response data by calling the function DMCGetAdditionalResponse. The length of the additional response data may ascertained by call the function DMCGetAdditionalResponseLen. If the response data from the controller is too large for the internal additional response buffer, the error code DMCERROR_RESPONSEDATA will be returned.
cbResponse
Length of the buffer.

DMCGetUnsolicitedResponse
LONG FAR GALILCALL DMCGetUnsolicitedResponse(HANDLEDMC hdmc, PCHAR chResponse, ULONG cbResponse);

Query the Galil controller for unsolicited responses. These are messages output from programs running in the background in the Galil controller. The most common command to produce messages is 'MG'. In most cases 'MG' should not be used. Instead set a flag variable in the Galil program and poll for that variable using DMCCommand.

hdmc
Handle to the Galil controller.
pchResponse
Buffer to receive the response data.
cbResponse
Length of the buffer.

DMCGetAdditionalResponseLen
LONG FAR GALILCALL DMCGetAdditionalResponseLen(HANDLEDMC hdmc, PULONG pulResponseLen);

Query the Galil controller for the length of additional response data. There will be more response data available if the DMCCommand function returned DMCERROR_BUFFERFULL.

hdmc
Handle to the Galil controller.
pulResponseLen
Buffer to receive the additional response data length.

DMCGetAdditionalResponse
LONG FAR GALILCALL DMCGetAdditionalResponse(HANDLEDMC hdmc, PCHAR pchResponse, ULONG cbResponse);

Query the Galil controller for more response data. There will be more response data available if the DMCCommand function returned DMCERROR_BUFFERFULL. Once this function is called, the internal additional response buffer is cleared.

hdmc
Handle to the Galil controller.
pchResponse
Buffer to receive the response data.
cbResponse
Length of the buffer.

DMCError
LONG FAR GALILCALL DMCError(HANDLEDMC hdmc, LONG lError, PCHAR pchMessage, LONG cbMessage);

Retrieve the error message text from a DMCERROR_COMMAND error.

hdmc
Handle to the Galil controller.
pchMessage
Buffer to receive the error message text.
cbMessage
Length of the buffer.
DMCClear

LONG FAR GALILCALL DMCClear(HANDLEDMC hdmc);

Clear the Galil controller FIFO.

hdmc
Handle to the Galil controller.

DMCReset
LONG FAR GALILCALL DMCReset(HANDLEDMC hdmc);

Reset the Galil controller.

hdmc
Handle to the Galil controller.

DMCMasterReset
LONG FAR GALILCALL DMCMasterReset(HANDLEDMC hdmc);

Master reset the Galil controller. This may take up to 5 seconds. Make sure the timeout is large enough before performing the master reset.

hdmc
Handle to the Galil controller.

DMCVersion
LONG FAR GALILCALL DMCVersion(HANDLEDMC hdmc, PCHAR pchVersion, ULONG cbVersion);

Get the firmware version of the Galil controller.

hdmc
Handle to the Galil controller.
pchVersion
Buffer to receive the version information.
cbVersion
Length of the buffer.

DMCDownloadFile
LONG FAR GALILCALL DMCDownloadFile(HANDLEDMC hdmc, PSZ pszFileName, PSZ pszLabel);

Download a file to the Galil controller from a file on the hard drive.

hdmc
Handle to the Galil controller.
pszFileName
File name to download to the Galil controller.
pszLabel
Program label to download to. This argument is ignored if NULL.

DMCDownloadFromBuffer
LONG FAR GALILCALL DMCDownloadFromBuffer(HANDLEDMC hdmc, PSZ pszBuffer, PSZ pszLabel);

Download from a memory buffer to the Galil controller.

hdmc

Handle to the Galil controller.
pszBuffer
Buffer of DMC commands to download to the Galil controller.
pszLabel
Program label to download to. This argument is ignored if NULL.

DMCUploadFile
LONG FAR GALILCALL DMCUploadFile(HANDLEDMC hdmc, PSZ pszFileName);

Upload a file from the Galil controller to the hard disk.

hdmc
Handle to the Galil controller.
pszFileName
File name to upload from the Galil controller.

DMCUploadToBuffer
LONG FAR GALILCALL DMCUploadToBuffer(HANDLEDMC hdmc, PCHAR pchBuffer, ULONG cbBuffer);

Upload to a memory buffer from the Galil controller.

hdmc
Handle to the Galil controller.
pchBuffer
Buffer of DMC commands to upload from the Galil controller.
cbBuffer
Length of the buffer.

DMCSendFile
LONG FAR GALILCALL DMCSendFile(HANDLEDMC hdmc, PSZ pszFileName);

Send a file consisting of DMC commands in ASCII format to the Galil controller. Commands are executed immediately.

hdmc
Handle to the Galil controller.
pszFileName
File name to send to the Galil controller.

DMCSendBinaryFile
LONG FAR GALILCALL DMCSendBinaryFile(HANDLEDMC hdmc, PSZ pszFileName);

Send a file consisting of DMC commands in binary format to the Galil controller. Commands are executed immediately.

Note: This function is for the DMC-1600 and DMC-1700 only.

hdmc
Handle to the Galil controller.
pszFileName
File name to send to the Galil controller.

DMCArrayDownload
LONG FAR GALILCALL DMCArrayDownload(HANDLEDMC hdmc, PSZ pszArrayName, USHORT usFirstElement, USHORT usLastElement, PCHAR pchData, ULONG cbData, PULONG cbBytesWritten);

Download an array to the Galil controller. The array must already exist in the controller. Array data can be delimited by a comma or CR (0x0D) or CR/LF (0x0D0A).

Note: The firmware on the controller must be recent enough to support the QD command.

hdmc
Handle to the Galil controller.
pszArrayName
Array name to download to the Galil controller.
usFirstElement
First array element.
usLastElement
Last array element.
pchData
Buffer to write the array data from. Data does not need to be NULL terminated.
vbData
Length of the array data in the buffer.
cbBytesWritten
Number of bytes written.

DMCArrayUpload
LONG FAR GALILCALL DMCArrayUpload(HANDLEDMC hdmc, PSZ pszArrayName, USHORT usFirstElement, USHORT usLastElement, PCHAR pchData, ULONG cbData, PULONG pulBytesRead, SHORT fComma);

Upload an array from the Galil controller. The array must exist on the controller. Array data will be delimited by a comma or CR (0x0D) depending of the value of fComma.

Note: The firmware on the controller must be recent enough to support the QU command.

hdmc
Handle to the Galil controller.
pszArrayName
Array name to upload from the Galil controller.
usFirstElement
First array element.
usLastElement
Last array element.
pchData
Buffer to read the array data into. Array data will not be NULL terminated.
cbData
Length of the buffer.
pulBytesRead
Number of bytes read.

DMCCommand_AsciiToBinary
LONG FAR GALILCALL DMCCommand_AsciiToBinary(HANDLEDMC hdmc, PSZ pszAsciiCommand, ULONG ulAsciiCommandLength, PBYTE pbBinaryResult, ULONG cbBinaryResult, ULONG FAR *pulBinaryResultLength);

Convert an ASCII DMC command to a binary DMC command.

Note: This function is for the DMC-1600 and DMC-1700 only.

hdmc
Handle to the Galil controller.
pszAsciiCommand
ASCII DMC command(s) to be converted.
ulAsciiCommandLength
Length of DMC command(s).
pbBinaryResult
Buffer to receive the translated DMC command.
cbBinaryResult
Length of the buffer.
pulBinaryResultLength
Length of the translated DMC command.

DMCCommand_BinaryToAscii
LONG FAR GALILCALL DMCCommand_BinaryToAscii(HANDLEDMC hdmc, PBYTE pbBinaryCommand, LONG ulBinaryCommandLength, PSZ pszAsciiResult, ULONG cbAsciiResult, ULONG FAR *pulAsciiResultLength);

Convert a binary DMC command to an ASCII DMC command.

Note: This function is for the DMC-1600 and DMC-1700 only.

hdmc
Handle to the Galil controller.
pbBinaryCommand
Binary DMC command(s) to be converted.
ulBinaryCommandLength
Length of DMC command(s).
pszAsciiResult
Buffer to receive the translated DMC command.
cbAsciiResult
Length of the buffer.
pulAsciiResultLength
Length of the translated DMC command.

DMCFile_AsciiToBinary
LONG FAR GALILCALL DMCFile_AsciiToBinary(HANDLEDMC hdmc, PSZ pszInputFileName, PSZ pszOutputFileName);

Convert a file consisting of ASCII commands to a file consisting of binary commands.

Note: This function is for the DMC-1600 and DMC-1700 only.

hdmc
Handle to the Galil controller.
pszInputFileName
File name for the input ASCII file.
pszOutputFileName
File name for the output binary file.

DMCFile_BinaryToAscii
LONG FAR GALILCALL DMCFile_BinaryToAscii(HANDLEDMC hdmc, PSZ pszInputFileName, PSZ pszOutputFileName);

Convert a file consisting of binary commands to a file consisting of ASCII commands.

Note: This function is for the DMC-1600 and DMC-1700 only.

hdmc
Handle to the Galil controller.
pszInputFileName
File name for the input binary file.
pszOutputFileName
File name for the output ASCII file.

DMCReadSpecialConversionFile
LONG FAR GALILCALL DMCReadSpecialConversionFile(HANDLEDMC hdmc, PSZ pszFileName);

Read into memory a special BinaryToAscii/AsciiToBinary conversion table.

Note: This function is for the DMC-1600 and DMC-1700 only.

hdmc
Handle to the Galil controller.
pszFileName
File name for the special conversion file.

DMCRefreshDataRecord
LONG FAR GALILCALL DMCRefreshDataRecord(HANDLEDMC hdmc, ULONG ulLength);

Refresh the data record used for fast polling.

Note: This function is for the DMC-1600 and DMC-1700 only.

hdmc
Handle to the Galil controller.
ulLength
Refresh size in bytes. Set to 0 unless you do not want a full-buffer refresh.

DMCGetDataRecord
LONG FAR GALILCALL DMCGetDataRecord(HANDLEDMC hdmc, USHORT usGeneralOffset, USHORT usAxisInfoOffset, PUSHORT pusDataType, PLONG plData);

Get a data item from the data record used for fast polling.

Note: This function is for the DMC-1600 and DMC-1700 only.

hdmc
Handle to the Galil controller.
usGeneralOffset
Data record offset for general data item.
usAxisInfoOffset
Additional data record offset for axis data item.
pusDataType

Data type of the data item. If you are using the standard, pre-defined offsets, set this argument to zero before calling this function. The actual data type of the data item is returned on output.
plData
Buffer to receive the data record data. Output only.

DMCGetDataRecordByItemId
LONG FAR GALILCALL DMCGetDataRecordByItemId(HANDLEDMC hdmc, USHORT usItemId, USHORT usAxisId, PUSHORT pusDataType, PLONG plData);

Get a data item from the data record used for fast polling. Gets one item from the data record by using Id (see data record Ids defined in DMCDRC.H). To retrieve data record items by offset instead of Id, use the function DMCGetDataRecord.

Note: this function is for the DMC-1600, DMC-1700, DMC-1800 only.

hdmc
Handle to the Galil controller.
usItemId
Data record item Id.
usAxisId
Axis Id used for axis data items.
pusDataType
Data type of the data item. The data type of the data item is returned on output. Output Only.
plData
Buffer to receive the data record data. Output only.

DMCGetDataRecordSize
LONG FAR GALILCALL DMCGetDataRecordSize(HANDLEDMC hdmc, PUSHORT pusRecordSize);

Get the size of the data record used for fast polling.

Note: this function is for the DMC-1600, DMC-1700, and DMC-1800 only.

hdmc
Handle to the Galil controller.
pusRecordSize
The size of the data record is returned on output. Output Only.

DMCCopyDataRecord
LONG FAR GALILCALL DMCCopyDataRecord(HANDLEDMC hdmc, PVOID pDataRecord);

Get a copy of the data record used for fast polling. The data record is only as recent as the last call made to DMCRefreshDataRecord.

Note: this function is for the DMC-1600, DMC-1700, and DMC-1800 only.

hdmc
Handle to the Galil controller.
pDataRecord
A copy of the data record is returned on output. Output Only.

DMCGetDataRecordRevision
LONG FAR GALILCALL DMCGetDataRecordRevision(HANDLEDMC hdmc, PUSHORT pusRevision);

Get the revision of the data record structure used for fast polling.

Note: this function is for the DMC-1600, DMC-1700, and DMC-1800 only.

hdmc
Handle to the Galil controller.
pusRevision
The revision of the data record structure is returned on output. Output Only.

DMCDiagnosticsOn
LONG FAR GALILCALL DMCDiagnosticsOn(HANDLEDMC hdmc, PSZ pszFileName, BOOL fAppend);

Turn on diagnostics to log all communications to the controller.

hdmc
Handle to the Galil controller.
pszFileName
File name for the diagnostic file.
fAppend
TRUE if the file will open for append, otherwise FALSE.

DMCDiagnosticsOff
LONG FAR GALILCALL DMCDiagnosticsOff(HANDLEDMC hdmc);

Turn off diagnostics log.

hdmc
Handle to the Galil controller.

DMCGetTimeout
LONG FAR GALILCALL DMCGetTimeout(HANDLEDMC hdmc, LONG FAR* pTimeout);

Get current time-out value.

hdmc
Handle to the Galil controller.
pTimeout
Buffer to receive the current time-out value in milliseconds.

DMCSetTimeout
LONG FAR GALILCALL DMCSetTimeout(HANDLEDMC hdmc, LONG lTimeout);
Set time-out value. If the time-out value is set to zero, the DLLs will ignore time-out errors. This is useful for sending Galil commands which do not return a response, such as providing records to the DL or QD commands.

hdmc
Handle to the Galil controller.
lTimeout
Time-out value in milliseconds.

DMCGetDelay
LONG FAR GALILCALL DMCGetDelay(HANDLEDMC hdmc, LONG FAR* pDelay);

Get communications delay value. The delay is a short pause between a write to and read from the Galil controller.

***** THIS FUNCTION IS OBSOLETE. DELAY IS NO LONGER USED *****

hdmc
Handle to the Galil controller.
pDelay
Buffer to receive the current delay value in milliseconds.

DMCSetDelay
LONG FAR GALILCALL DMCSetDelay(HANDLEDMC hdmc, LONG lDelay);

Set communications delay value. The delay is a short pause between a write to and read from the Galil controller.

***** THIS FUNCTION IS OBSOLETE. DELAY IS NO LONGER USED *****

hdmc
Handle to the Galil controller.
lDelay
Delay value in milliseconds.

DMCWriteData
LONG FAR GALILCALL DMCWriteData(HANDLEDMC hdmc, PCHAR pchBuffer, ULONG cbBuffer, PULONG pulBytesWritten);

Low-level I/O routine to write data to the Galil controller. Data is written to the Galil controller only if it is "ready" to receive it. The function will attempt to write exactly cbBuffer characters to the controller.

Note: For Win32 and WinRT driver the maximum number of bytes which can be written each time is 64. There are no restrictions with the Galil driver.

hdmc
Handle to the Galil controller.
pchBuffer
Buffer to write the data from. Data does not need to be NULL terminated.
cbBuffer
Length of the data in the buffer.
pulBytesWritten
Number of bytes written.

DMCReadData
LONG FAR GALILCALL DMCReadData(HANDLEDMC hdmc, PCHAR pchBuffer, ULONG cbBuffer, PULONG pulBytesRead);

Low-level I/O routine to read data from the Galil controller. The routine will read what ever is currently in the FIFO (bus controller) or communications port input queue (serial controller). The function will read up to cbBuffer characters from the controller. The data placed in the user buffer (pchBuffer) is NOT NULL terminated. The data returned is not guaranteed to be a complete response - you may have to call this function repeatedly to get a complete response.

Note: For Win32 and WinRT driver the maximum number of bytes which can be read each time is 64. There are no restrictions with the Galil driver.

hdmc
Handle to the Galil controller.
pchBuffer
Buffer to read the data into. Data will not be NULL terminated.
cbBuffer
Length of the buffer.
pulBytesRead
Number of bytes read.

DMCAddGalilRegistry and DMCAddGalilRegistry2
LONG FAR GALILCALL DMCAddGalilRegistry(PGALILREGISTRY pgalilregistry, PUSHORT pusController);
LONG FAR GALILCALL DMCAddGalilRegistry2(PGALILREGISTRY2 pgalilregistry2, PUSHORT pusController);

Add a Galil controller to the Windows registry. The controller number is returned in the argument pusController. The DMCAddGalilRegistry2 function is a replacement for DMCAddGalilRegistry. Both functions are compatible with each other.

pgalilregistry/pgalilregistry2
Pointer to a GALILREGISTRY or GALILREGISTRY2 struct.
pusController
Buffer to receive the Galil controller number.

DMCModifyGalilRegistry and DMCModifyGalilRegistry2
LONG FAR GALILCALL DMCModifyGalilRegistry(USHORT usController, PGALILREGISTRY pgalilregistry);
LONG FAR GALILCALL DMCModifyGalilRegistry2(USHORT usController, PGALILREGISTRY2 pgalilregistry2);

Change a Galil controller in the Windows registry. The DMCModifyGalilRegistry2 function is a replacement for DMCModifyGalilRegistry. Both functions are compatible with each other.

usController
Galil controller number.
pgalilregistry/pgalilregistry2
Pointer to a GALILREGISTRY or GALILREGISTRY2 struct.

DMCGetGalilRegistryInfo and DMCGetGalilRegistryInfo2
LONG FAR GALILCALL DMCGetGalilRegistryInfo(USHORT usController, PGALILREGISTRY pgalilregistry);
LONG FAR GALILCALL DMCGetGalilRegistryInfo2(USHORT usController, PGALILREGISTRY2 pgalilregistry2);

Get Windows registry information for a given Galil controller. The DMCGetGalilRegistryInfo2 function is a replacement for DMCGetGalilRegistryInfo. Both functions are compatible with each other.

usController
Galil controller number.
pgalilregistry/pgalilregistry2
Pointer to a GALILREGISTRY or GALILREGISTRY2 struct.

DMCEnumGalilRegistry and DMCEnumGalilRegistry2
LONG FAR GALILCALL DMCEnumGalilRegistry(USHORT FAR* pusCount, PGALILREGISTRY pgalilregistry);

LONG FAR GALILCALL DMCEnumGalilRegistry2(USHORT FAR* pusCount, PGALILREGISTRY2 pgalilregistry2);

Enumerate or list all the Galil controllers in the Windows registry. The user needs to make two calls to this function. The first call should have a NULL for the argument pgalilregistry. The number of GALILREGISTRY structs (number of Galil controllers in the Windows registry) will be returned in the argument pusCount. The second call should have sufficient memory allocated for all the GALILREGISTRY structs to be returned and pass the pointer to that memory as the argument pgalilregistry. It is the users responsibility to allocate and free memory to hold the GALILREGISTRY structs. The DMCEnumGalilRegistry2 function is a replacement for DMCEnumGalilRegistry. Both functions are compatible with each other.

pusCount
Pointer to the number of GALILREGISTRY structs returned.
pgalilregistry/pgalilregistry2
Pointer to a GALILREGISTRY or GALILREGISTRY2 struct.

DMCDeleteGalilRegistry
LONG FAR GALILCALL DMCDeleteGalilRegistry(SHORT sController);

Delete a Galil controller in the Windows registry.

sController
Galil controller number. Use -1 to delete all Galil controllers.

DMCRegisterPnpControllers
LONG FAR GALILCALL DMCRegisterPnpControllers(USHORT* pusCount);

Update the Windows registry for all Galil plug-and-play controllers. This function may add new controllers to the registry or update existing ones.

Note: This function is for the DMC-1600 and DMC-1700 operating under Win16 and Windows NT only.

pusCount
Pointer to the number of Galil plug-and-play controllers registered (and/or updated).

DMCWaitForMotionComplete
LONG FAR GALILCALL DMCWaitForMotionComplete(HANDLEDMC hdmc, PSZ pszAxes, SHORT fDispatchMsgs);

Wait for motion complete by creating a thread to query the controller. The function returns when motion is comp lete.

hdmc
Handle to the Galil controller.
pszAxes
Which axes to wait for: X, Y, Z, W, E, F, G, H, or S for coordinated motion. To wait for more than one axis (other than coordinated motion), simply concatenate the axis letters in the string.
fDispatchMsgs
Set to TRUE if you want to get and dispatch Windows messages while waiting for motion complete. This flag is always TRUE for Win16.

DMCStartDeviceDriver
LONG FAR GALILCALL DMCStartDeviceDriver(USHORT usController);

Start the device driver associated with the given controller. All controller handles must be closed. Use this function to recycle the device driver after making a configuration change through the Windows registry. For bus controllers only. For Win32 only.

usController
Galil controller number.

DMCStopDeviceDriver
LONG FAR GALILCALL DMCStopDeviceDriver(USHORT usController);

Stop the device driver associated with the given controller. All controller handles must be closed. Use this function to recycle the device driver after making a configuration change through the Windows registry. For bus controllers only. For Win32 only.

usController
Galil controller number.

DMCDownloadFirmwareFile
LONG FAR GALILCALL DMCDownloadFirmwareFile(HANDLEDMC hdmc, PSZ pszFileName, SHORT fDisplayDialog);

Update the controller's firmware. This function will open a binary firmware file and refresh the flash EEPROM of the controller.

Note: This function is for the DMC-1200, DMC-1600, DMC-1700, DMC-1800, and DMC-2000 only.

hdmc
Handle to the Galil controller.
pszFileName
File name to download to the Galil controller.
fDisplayDialog
Display a progress dialog while the firmware file is being downloaded.

DMCSelectController
SHORT FAR GALILCALL DMCSelectController(HWND hwnd);

Select a Galil motion controller from a list of registered controllers. Returns the selected controller number or -1 if no controller was selected.

Note: This function invokes a dialog window.

hwnd
The window handle of the calling application. If NULL, the window with the input focus is used.

DMCEditRegistry
VOID FAR GALILCALL DMCEditRegistry(HWND hwnd);

Edit the Windows registry: add, change, or delete Galil motion controllers.

Note: This function invokes a dialog window.

hwnd
The window handle of the calling application. If NULL, the window with the input focus is used.

Application Programming Topics

Downloading Programs to the Controller

Two API functions are provided to download Galil language programs to the controller: DMCDownloadFile and DMCDownloadFromBuffer. DMCDownloadFromBuffer assumes that each line of one or more commands is separated by a CR/LF, just as if your buffer was a mirror image of a text file. The program label argument can be used to download the Galil language program to a specific label or program line, or may be set to NULL to replace any existing Galil language program.

You may also use the Galil "DL" command to build your own download function. If you do, make sure that you set the time-out (using the API function DMCSetTimeout) to zero as the DL command will not return an acknowledgment from the controller (except in the case of an error) until the end of file character ('\' or Cntl-Z) is sent. Remember to restore the time-out value before exiting your function.

Interrupt Handling

Application programs can receive notification of interrupts from the controller by handling the WM_DMCINTERRUPT message which is defined in DMCCOM.H. The wParam or WORD parameter of the message contains the interrupt status from the controller. The lParam or DWORD parameter is the HANDLEDMC (which was returned from DMCOpen or DMCOpen2), which is useful if you have more than one controller installed which may issue interrupts. The WM_DMCINTERRUPT message is posted to your application using either PostMessage or PostThreadMessage depending on if you set the interrupt notification to a window handle (HWND) or a thread Id (see the text for API functions DMCOpen and DMCOpen2). Interrupt notification can be changed by using the API function DMCChangeInterruptNotification. For standard Windows SDK programming, the WM_DMCINTERRUPT message is retrieved by using GetMessage for HWND or GetThreadMessage for thread Id.

For MFC applications, you will need the following code:

In your .H file, manually add the function prototype for the interrupt handling function to your message map, **after** the Class Wizard comments.

```
//{{AFX_MSG(CMyApp)
afx_msg void OnTimer(UINT nIDEvent);
afx_msg void OnCommand1
afx_msg void OnCommand2
//}}AFX_MSG
afx_msg LONG OnDMCInterrupt(UINT nWP, LONG nLP);
DECLARE_MESSAGE_MAP()
```

In your .CPP file, manually add the ON_MESSAGE macro to your message map, **after** the Class Wizard comments.

```
BEGIN_MESSAGE_MAP(CMyApp, CWinApp)
  //{{AFX_MSG_MAP(CMyApp)
  ON_WM_TIMER()
  ON_COMMAND(IDC_COMMAND1, OnCommand1)
  ON_COMMAND(IDC_COMMAND2, OnCommand2)
  //}}AFX_MSG_MAP
  ON_MESSAGE(WM_DMCINTERRUPT, OnDMCInterrupt)
END_MESSAGE_MAP()
```

In your .CPP file, manually add the interrupt handling function:

```
LONG CMyApp::OnDMCInterrupt(UINT nWP, LONG nLP)
{
  short nStatus = (short)nWP;

  return 0L;
}
```

For OWL applications, you will need the following code:

In your .H file, manually add the function prototype for the interrupt handling function in the response table.

```
//{{MyApplicationRSP_TBL_BEGIN}}
protected:
  void Command1 ();
  void Command2 ();
  LRESULT DmcInterrupt (unsigned int, long);
//{{MyApplicationRSP_TBL_END}}
DECLARE_RESPONSE_TABLE(MyApplication);
};   //{{MyApplication}}
```

In your .CPP file, manually add the EV_MESSAGE macro to your response table.

```
DEFINE_RESPONSE_TABLE1(MyApplication, TApplication)
//{{MyApplicationRSP_TBL_BEGIN}}
  EV_COMMAND(IDC_COMMAND1, Command1),
  EV_COMMAND(IDC_COMMAND1, Command2),
  EV_MESSAGE(WM_DMCINTERRUPT, DMCInterrupt),
//{{MyApplicationRSP_TBL_END}}
END_RESPONSE_TABLE;
```

In your .CPP file, manually add the interrupt handling function:

```
LRESULT MyApplication::DMCInterrupt (unsigned int message, long messagel)
{
  short nStatus = (short)message;

  return 0L;
}
```

Diagnostics

There are two API functions DMCDiagnosticsOn and DMCDiagnosticsOff which can be used to provide a detailed trace of communications between your application and the controller. The trace output gets very large very quickly, but it does provide a lot of detail, especially when error conditions arise.

To cut down on the amount of trace being output, call DMCDiagnosticsOn as close to the part of your program you wish to debug as possible. In addition, DMCDiagnosticsOn can be called with a flag to append trace output rather than replace trace output.

Managing the Time-out and Delay Values

The time-out value used by the API functions is often misinterpreted. The time-out value is used by API functions such as DMCCommand to control how long the function will wait before being able to send a command to the controller (controller status set to ready-to-receive) or how long to wait for a complete response (usually a ":" or "?"). Lowering the time-out value does not mean that the DMCommand or other API functions will respond faster, as most of the time the API functions return long before the time-out has expired. In fact, setting the time-out value too low will cause some functions, such as DMCDownloadFile to fail. For most Galil commands and API functions, the default value of 1000 is usually adequate. Some Galil commands such as master reset (^R^S or the API function DMCMasterReset) may take up to five seconds to complete, so the time-out value may need to be temporarily raised. If you do not care about the response from the controller and/or about time-out conditions, you can set the time-out value to zero.

The time-out value is initialized to the value stored in the Windows Registry database each time you call DMCOpen. You can get the current time-out value with the API function DMCGetTimeout. You can set the current time-out value with the API function DMCSetTimeout.

Note: the lowest you can set the time-out value in the Windows Registry database is 1 ms. The default time-out value is 1000 ms or 1 second.

Delay is no longer used by DMCWIN. Changing its value will have no effect on any API functions.

Receiving Large Amounts of Data from the Controller

Most Galil language commands return less than 256 characters of data. There are, however, some exceptions such as "LS" to list a program and "QU" to upload an array. Since it is inefficient to allocate a static buffer to handle the largest possible amount of data returned from the controller, how do we handle this case? Call the API function DMCCommand with your "normal" size buffer and inspect the return code. If it is DMCERROR_RESPONSEDATA, your "normal" size buffer is much too small and must be increased. The return code DMCERROR_RESPONSEDATA means that an internal buffer used to hold response data overflow has been exceeded and data has been lost. The internal buffer is approximately 8K in size to give you an idea how large your "normal" buffer needs to be.

If the return code is DMCERROR_BUFFERFULL, you can retrieve the rest of your response data by calling the API function DMCGetAdditionalResponse. The API function DMCGetAdditionalResponseLen will provide the length of buffer needed to retrieve the balance of the data from DMCCommand. For example:

```
long rc;
long lDataLength;
char szBuffer[256];

rc = DMCCommand(hdmc, "QU ARRAY1[]", szBuffer, sizeof(szBuffer));
if (rc == DMCNOEROR)
  ; /* return success */
else if (rc == DMCERROR_BUFFERFULL)
{
  rc = DMCGetAdditionalResponseLen(hdmc, &lDataLength);
  if (rc == DMCNOERROR)
  {
    char* pBuffer = (char*)malloc(lDataLength);
    if (pBuffer)
    {
      rc = DMCGetAdditionalResponse(hdmc, pBuffer, lDataLength);
      if (rc != DMCNOERROR)
        ; /* return an error */
      free(pBuffer);
    }
    else
      ; /* return an error */
  }
}
else
  ; /* return an error */
```

Low-Level I/O

There are two functions available for performing low-level I/O: DMCWriteData and DMCReadData. They are ideal if you need to send data to the controller which is time critical, such as contour data or linear interpolation segments. The most important point to remember if you use DMCWriteData to send commands to the controller, is that you need to periodically call DMCReadData or DMCClear to clear the outbound FIFO or communications buffer. Failure to do so will cause communications and application programs to halt if the outbound FIFO or communications buffer fills up. On Galil bus controllers, the outbound FIFO is configured to hold 256 bytes.

Multiple Thread Applications

If you are using multiple threads in your application program and more than one thread makes calls to the Galil DLLs, you should wrap each Galil API call with a critical section. This will ensure the integrity of each transaction with the controller. For example:

```
EnterCriticalSection(&CritSec);
rc = DMCCommand(hDmc, szCMD, szResponse, sizeof(szResponse));
LeaveCriticalSection(&CritSec);
```

Do not forget to initialize the critical section in your application startup and delete the critical section in your application shutdown.

Waiting for Motion to Complete

A common programming task in communicating with Galil controllers is having to write a wait for motion to complete routine. This is because sending a Galil command such as AMX (wait for the motion on the X axis to complete before continuing) from the PC to the controller will cause communications to/from the controller to stop until the motion is complete. There is an API named DMCWaitForMotionComplete which does most of the work for you. For Win32

environments, the function is multi-threaded so as to query the controller in the most efficient manner. The function has an argument named *fDispatchMsgs* which is used to determine whether or not to keep getting and dispatching Windows messages while waiting. For Win16 environments, this argument is ignored; that is, it is always TRUE. For Win32 console programs or program threads which have no windows, the argument should be set to FALSE since there is no message loop. For Win32 programs or program threads which have windows, the argument should be set to TRUE so your program can continue to process messages while waiting.

If you are using multiple threads in your application program and more than one thread makes calls to the Galil DLLs, you should not use the DMCWaitForMotionComplete function. Instead, you should build your WaitForMotionComplete function using the sample code below. Note that for multithreaded applications, it is assumed that you will wrap calls to the Galil DLLs with critical sections. The sample code below assumes that you have already allocated and initialed a critical section as a global variable.

```
typedef struct _MOTIONTHREAD
{
   HANDLEDMC  hDmc;
   char       szAxes[16];
   long       rc;
   DWORD      ThreadId;
} MOTIONTHREAD;

DWORD WINAPI MotionCompleteThread(LPVOID pArgs);

LONG WaitForMotionComplete(LONG hDmc, PSZ pszAxes, SHORT fDispatchMsgs)
{
   long          rc = WAIT_OBJECT_0 + 1;
   DWORD         ThreadId;
   HANDLE        hThread;
   MOTIONTHREAD  mt;

   if (!pszAxes)
      return DMCERROR_ARGUMENT;

   mt.hDmc = hDmc;
   strcpy(mt.szAxes, pszAxes);
   mt.rc = 0;
   mt.ThreadId = GetCurrentThreadId();

   hThread = CreateThread(NULL, 0, MotionCompleteThread, (LPVOID)&mt, 0,
      &ThreadId);
   if (!hThread)
      return DMCERROR_DRIVER;

   if (fDispatchMsgs)
   {
      while (rc != WAIT_OBJECT_0)
      {
         rc = MsgWaitForMultipleObjects(1, &hThread, FALSE, INFINITE,
            QS_ALLINPUT);
         if (rc == WAIT_OBJECT_0 + 1)
         {
            MSG   msg;
            while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
            {
```

```
               if (msg.message == WM_MOTIONCOMPLETE)
               {
                  rc = WAIT_OBJECT_0;
                  break;
               }
               TranslateMessage(&msg);
               DispatchMessage(&msg);
            }
         }
      }
   }
   else
      WaitForSingleObject(hThread, INFINITE);

   CloseHandle(hThread);

   return mt.rc;
}

DWORD WINAPI MotionCompleteThread(LPVOID pArgs)
{
   int            i;
   int            bLoop;
   int            iAxes;
   long           rc = 0L;
   char           szCMD[8];
   char           szResponse[32];
   MOTIONTHREAD*  pmt = (MOTIONTHREAD*)pArgs;

   if (!pmt)
   {
      ExitThread((DWORD)DMCERROR_DRIVER);
      return (DWORD)DMCERROR_DRIVER;
   }

   // How many axes?
   iAxes = strlen(pmt->szAxes);

   for (i = 0; i < iAxes; i++)
   {
      // Check each axis in turn
      bLoop = TRUE;
      sprintf(szCMD, "MG_BG%c\r", pmt->szAxes[i]);
      // Loop while motion is not complete
      while (bLoop)
      {
         EnterCriticalSection(&m_CritSec);
         rc = DMCCommand(pmt->hDmc, szCMD, szResponse,
            sizeof(szResponse));
         LeaveCriticalSection(&m_CritSec);
         if (rc == DMCNOERROR)
         {
            if (!atoi(szResponse))
               bLoop = FALSE;
```

```
            else
               Sleep(5);
         }
         else if (rc == DMCERROR_TIMEOUT)
            ; // Do nothing
         else
            bLoop = FALSE;
      }
   }

   if (pmt->ThreadId)
      PostThreadMessage(pmt->ThreadId, WM_MOTIONCOMPLETE, 0, 0);

   pmt->rc = rc;

   ExitThread(0);

   return 0;
}
```

Binary Communications (DMC-1200, DMC-1600, DMC-1700, DMC-1800, and DMC-2000 Only)

All Galil controllers are communicated with in ASCII format. That is, you send commands in ASCII and the controller responds in ASCII. The Galil DMC-1200, DMC-1600, DMC-1700, and DMC-2000 add a binary form of communication: you can send commands in binary format. The controller responds to binary commands the same as ASCII commands: by ASCII. The advantage of sending commands in binary is speed of execution. If the command is already in binary form, it does not need to be decoded by the controller resulting in higher throughput. Sending contour data is one area where speed is critical and may gain from sending commands in binary.

Use the API function DMCBinaryCommand to send a command to the controller in binary. Use the API function DMCSendBinaryFile to send a file of binary commands to the controller. In addition, there are API functions to convert commands and files to/from ASCII/binary. Note that converting a command from ASCII to binary in-line or on-the-fly will may still result in improved throughput. An example follows of converting a command from ASCII to binary on-the-fly then sending it to the controller.

```
LONG  rc;
CHAR  szCommand = "PR1000,2000"
CHAR  szResponse[256];
BYTE  BinaryCommand[32];
ULONG BinaryCommandLength;

// Convert a command from ASCII to binary
rc = AsciiCommandToBinaryCommand(szCommand, strlen(szCommand),
BinaryCommand,
   sizeof(BinaryCommand), &BinaryCommandLength);

// Send the binary command to the controller and get the response
rc = BinaryCommand(BinaryCommand, BinaryCommandLength, szResponse,
   sizeof(szResponse));
```

Data Record Access (DMC-1600, DMC-1700, and DMC-1800 Only)

The Galil DMC-1600, DMC-1700, and DMC-1800 add a new feature called Data Record Access. The Galil DMC-1600, DMC-1700, and DMC-1800 can provide, at specified intervals, a data record containing status information for

the controller in general and for each axis. The data record is sent from the controller to the Windows Drivers/DLLs using either DMA (DMC-1700 only) or a secondary FIFO channel. The Galil command "DR" is used to set the Data Record Access method (DMA or FIFO) and the refresh rate. The data record is always available, even if the primary FIFO channel is blocked because a trip-point (such as after motion) is pending. The standard data record structure is as follows:

General Information

unsigned short int	SampleNumber
unsigned char	Input0 (Inputs 0 - 7)
unsigned char	Input1 (Inputs 8 - 15)
unsigned char	Input2 (Inputs 16 - 23)
unsigned char	Input3 (Inputs 24 - 31)
unsigned char	Input4 (Inputs 32 - 39)
unsigned char	Input5 (Inputs 40 - 47)
unsigned char	Input6 (Inputs 48 - 55)
unsigned char	Input7 (Inputs 56 - 63)
unsigned char	Input8 (Inputs 64 - 71)
unsigned char	Input9 (Inputs 72 - 79)
unsigned char	Output0 (Outputs 0 - 7)
unsigned char	Output1 (Outputs 8 - 15)
unsigned char	Output2 (Outputs 16 - 23)
unsigned char	Output3 (Outputs 24 - 31)
unsigned char	Output4 (Outputs 32 - 39)
unsigned char	Output5 (Outputs 40 - 47)
unsigned char	Output6 (Outputs 48 - 55)
unsigned char	Output7 (Outputs 56 - 63)
unsigned char	Output8 (Outputs 64 - 71)
unsigned char	Output9 (Outputs 72 - 79)
unsigned char	ErrorCode
unsigned char	Status
unsigned short int	SegmentCount (S)
unsigned short int	CoordinatedMoveStatus (S)
long	CoordinatedMoveDistance (S)

Some firmware revisions also include information for the coordinated motion T axis, which repeats the last three items:

unsigned short int	SegmentCount (T)
unsigned short int	CoordinatedMoveStatus (T)
long	CoordinatedMoveDistance (T)

Axis Information (Repeated for each Axis)

unsigned short int	Status
unsigned char	Switches
unsigned char	StopCode
long	ReferencePosition
long	MotorPosition
long	PositionError
long	AuxillaryPosition
long	Velocity
short int	Torque
short int	Analog Input*

* Since the Analog Input data is contained in the Axis Information section, you can only retrieve data for the

analog inputs up to the number of axes on your controller. This is because Analog Input 1 is with the X axis information, Analog Input 2 is with the Y axis information, and so on.

Six API functions are provided to use the Data Record Access feature. DMCRefreshDataRecord gets a new copy of the data record and places it in system memory. This function must be called each time you wish to update the data record. DMCGetDataRecord and DMCGetDataRecordByItemId get a specific data record item from the copy of the data record already in memory. The header file DMCDRC.H defines several enums which list all the data record item by offsets and by Ids. DMCCopyDataRecord is used to copy the data record from system memory to a local buffer in your application program. In order to determine how large the local buffer must be, you must call DMCGetDataRecordSize.
The following sample code demonstrates how to retrieve the motor position for the X, Y, and Z axes:

```
long   rc;
long   xPos, yPos, zPos;
USHORT usDataType;

/* Refresh the data record - we want comparable information (same sample
period or time interval) for all three axes */
rc = DMCRefreshDataRecord(hdmc, 0);
/* Note: always set the data type argument to 0 before calling
DMCGetDataRecord, unless you know the data type. Acceptable data types
are listed in the enum DMCDataRecordTypes in DMCDRC.H */

/* Get the X axis motor position */
usDataType = 0;
rc = DMCGetDataRecord(hdmc, REV4GenOffAxis1,
   REV4AxisOffAxisMotorPosition, &usDataType, &xPos);

/* Get the Y axis motor position */
usDataType = 0;
rc = DMCGetDataRecord(hdmc, REV4GenOffAxis2,
   REV4AxisOffAxisMotorPosition, &usDataType, &yPos);

/* Get the Z axis motor position */
usDataType = 0;
rc = DMCGetDataRecord(hdmc, REV4GenOffAxis3,
   REV4AxisOffAxisMotorPosition, &usDataType, &zPos);
```

You will notice that the offset constants are prefixed with a revision number. This is because the data record structure is periodically updated. In order to determine which revision of the Data Record Access feature is available on your controller, call the API function DMCGetDataRecordRevision. In order to insulate your program from changes in offset values, you could use the API function DMCGetDataRecordByItemId instead of the API function DMCGetDataRecord. Because DMCGetDataRecordByItemId uses Ids which never change over time, you will never have to modify your application program if new firmware on your controller changes the data record structure. There is a very slight performance penalty. The above example using DMCGetDataRecordByItemId is repeated below:

```
/* Refresh the data record - we want comparable information (same sample
period or time interval) for all three axes */
rc = DMCRefreshDataRecord(hdmc, 0);

/* Get the X axis motor position */
rc = DMCGetDataRecordByItemId(hdmc, DRIdAxisMotorPosition, DRIdAxis1,
   &usDataType, &xPos);
```

```
/* Get the Y axis motor position */
rc = DMCGetDataRecordByItemId(hdmc, DRIdAxisMotorPosition, DRIdAxis2,
   &usDataType, &xPos);

/* Get the Z axis motor position */
rc = DMCGetDataRecordByItemId(hdmc, DRIdAxisMotorPosition, DRIdAxis3,
   &usDataType, &xPos);
```

Data Record Access Using DMCCopyDataRecord or the QR Command

Some controllers such as the DMC-1802 and the DMC-2000 do not implement the Data Record Access feature with a secondary communications channel. They do, however, support a form of Data Record Access via the QR command. This command returns, in binary format, a copy of the current data record on demand. There is no need to call the function DMCRefreshDataRecord. Use the DMCDATARECORDQR structure (defined in DMCDRC.H) to overlay the binary data returned. It can be used with the function DMCCommand in the following way:

```
LONG rc;
DMCDATARECORDQR MyDataRecordQR;

rc = DMCCommand(hdmc, "QR\r", &MyDataRecordQR, sizeof(MyDataRecordQR));
```

The QR command can return a full copy of the data record or a subset depending on the arguments used with it. For more information, consult the Galil Command Reference for your controller.

Note: you may wish to alter the DMCDATARECORDQR structure depending on your controller's configuration. See the header file DMCDRC.H for more details.
For controllers which implement the Data Record Access feature with a secondary communications channel, the function DMCCopyDataRecord can be used to retrieve a copy of the current data record (current as of the last call to the function DMCRefreshDataRecord). The structure DMCDATARECORD can be used as template for the data returned. The actual length of the data returned can be determined by using the function DMCGetDataRecordSize. You MUST allocate sufficient storage for the entire data record before calling the function DMCCopyDataRecord. An example follows:

```
LONG rc;
USHORT RecordSize;
DMCDATARECORD MyDataRecord;

rc = DMCGetDataRecordSize(hdmc, &RecordSize);
if (sizeof(MyDataRecord) >= RecordSize)
   // Safe to make call
   rc = DMCCopyDataRecord(hdmc, (PVOID)&MyDataRecord);
else
   // Must allocate more storage by adjusting DMCDATARECORD structure
   ;
```

Note: you may wish to alter the DMCDATARECORD structure depending on your controller's configuration. See the header file DMCDRC.H for more details.

Sample Programs

A number of sample programs are included in the DMCWIN directory under \C\Samples, \CPP\Samples and \VB\Samples.

Visual Basic Sample Program

There is a DLL routine declaration module named DMCCOM.BAS that should be included in every VB program that uses our DLLs. This declaration file comes in 16 bit format (DMCCOM.BAS) and 32 bit format (DMCCOM40.BAS).

If you are using VB version 3.0, use the 'vbtest.mak' sample program. For VB version 4,5 and 6 use 'vbtest40.vbp'.

C/C++ Sample Programs

There are a number of sample applications written for both C and C++. The C programs are intended for those interested in using the standard DMCCOM.H functions while the C++ programs are intended for those interested in using the DMCWIN.H class library.

Distributing Your Application

The following shows how to install the appropriate device drivers for each Windows environment. See the File List section below for a complete distribution list. For additional information, please consult **DISTNOTE.TXT** located within the DMCWIN directory.

For Windows 3.x

Copy DMC16.DLL to the WINDOWS\SYSTEM directory.
For bus controllers, copy DMCBUS16.DLL to the WINDOWS\SYSTEM directory.
For serial controllers, copy DMCSER16.DLL to the WINDOWS\SYSTEM directory.

For all bus controllers, copy VXD.386 to the WINDOWS\SYSTEM directory.
For all PCI bus controllers, copy GALIL.386 to the WINDOWS\SYSTEM directory.

You must also modify the SYSTEM.INI file located in the WINDOWS directory. In the section named 386Enh, add the line device=vxd.386. For example:

```
[386Enh]
device=vxd.386
```

If you are using a PCI bus controller such as the DMC-1800, you must also modify the SYSTEM.INI to include the following line:

```
[386Enh]
device=galil.386
```

Note: The device driver file GALIL.386 is only required if you are using a PCI bus controller such as the DMC-1800 and you are also using interrupts. This file is not required for other controllers or if you are not using interrupts.

For Windows NT and Windows 95/98

Copy DMC32.DLL to the WINDOWS\SYSTEM (or SYSTEM32 for NT) directory.

For bus controllers, copy DMCBUS32.DLL to the WINDOWS\SYSTEM (or SYSTEM32 for NT) directory.

For serial controllers, copy DMCSER32.DLL to the WINDOWS\SYSTEM (or SYSTEM32 for NT) directory.

For universal serial bus (USB) controllers, copy GALILUSB.SYS to the WINDOWS\SYSTEM32\DRIVERS directory.

For Windows NT

If you are using the WinRT drivers:
Copy WINRT.SYS to the WINDOWS\SYSTEM32\DRIVERS directory.

If you are using the Galil drivers:

For ISA bus controllers, copy GALIL.SYS to the WINDOWS\SYSTEM32\DRIVERS directory.

For PCI or CompactPCI bus controllers, copy GALILPCI.SYS to the WINDOWS\SYSTEM32\DRIVERS directory.

For Windows 95/98

If you are using the WinRT drivers:

Copy WRTDEV0.VXD to the WINDOWS\SYSTEM\VMM32 directory
Copy WRTDEV1.VXD to the WINDOWS\SYSTEM\VMM32 directory
Copy WRTDEV2.VXD to the WINDOWS\SYSTEM\VMM32 directory
Copy WRTDEV3.VXD to the WINDOWS\SYSTEM\VMM32 directory

If you are using the Galil drivers:

For ISA bus controllers, copy GALIL.VXD to the WINDOWS\SYSTEM\VMM32 directory.

For PCI or CompactPCI bus controllers, copy GALILPCI.VXD to the WINDOWS\SYSTEM\VMM32 directory.

For plug-and-play controllers (DMC-1600, DMC-1700, and DMC-1800), copy GALILPNP.VXD to the WINDOWS\SYSTEM directory.

Partial File List

C/C++ Programming Header Files

DMCCOM.H
DMCDRC.H
DMCDRCO.H

C++ Programming Files

DMCWIN.H
DMCWIN.CPP

32-Bit Files

DTERM32.EXE
DMCREG32.EXE
DMCBATCH.EXE
GALILUSB.SYS (in \Windows\System32\Drivers)
For Windows NT
DMC32.DLL (in \Windows\System32)
DMCBUS32.DLL (in \Windows\System32)
DMCSER32.DLL (in \Windows\System32)
WINRT.SYS (in \Windows\System32\Drivers)
GALIL.SYS (in \Windows\System32\Drivers)
GALILPCI.SYS (in \Windows\System32\Drivers)

For Windows 95/98

DMC32.DLL (in \Windows\System)
DMCBUS32.DLL (in \Windows\System)
DMCSER32.DLL (in \Windows\System)
WRTDEV0.VXD (in \Windows\System\Vmm32)
WRTDEV1.VXD (in \Windows\System\Vmm32)
WRTDEV2.VXD (in \Windows\System\Vmm32)
WRTDEV3.VXD (in \Windows\System\Vmm32)
GALIL.VXD (in \Windows\System\Vmm32)
GALILPCI.VXD (in \Windows\System\Vmm32)
GALILPNP.VXD (in \Windows\System\Vmm32)

16-Bit Files

DTERM16.EXE
DMCREG16.EXE
DMC16.DLL (in \Windows\System)
DMCBUS16.DLL (in \Windows\System)
DMCSER16.DLL (in \Windows\System)
VXD.386 (in \Windows\System)

For Visual Basic 3.0

DMCCOM.BAS
Test Project
VBTEST.FRM
VBTEST.MAK

For Visual Basic 4.0 and Higher

DMCCOM40.BAS
Test Project
VBTEST40.FRM
VBTEST40.VBP