Command Interface

Introduction

The Command Interface is a specification for a new way to communicate with Agile Systems servo controllers. The specification opens up the communication protocol so packets can be created independently of a library or platform. The Command Interface specification includes commands and registers for maximum control of the servo controller.

Agile Systems has moved to the Command Interface specification to give customers complete flexibility with their development choices. In the past, customers had to use Agile System's MAXLib motion control library to send a network packet that the controller would understand. Now, a software license is no longer necessary. Customers have the design flexibility to choose the level of software support required.

Features and Benefits

The release of Command Interface helps customers reduce servo system costs and increase portability. A rich instruction set, or commands set, is provided to give complete access to the servo controller. Key benefits of the Command Interface specification include:

- Packet Standardization across Agile's line of motion control products
- Increased portability for software developers
- Query-Response model with responses generated within one hundred microseconds
- Cyclic Redundancy Check (CRC) for error detection
- Rich command set including mathematical operations

With the introduction of the Command Interface, Agile System's is pleased to offer two powerful design tools to aid customers in system development: MAXCLib and Stand-alone operation.

MAXCLib

MAXCLib is a software translator that gives customers a universal, small size library for generating network packets on any hardware target for which an ANSI-C compiler exists. MAXCLib is a thread safe library that fully supports the Command Interface specification.

ANSI-C means many options. The possibilities for a host in a distributed control system have been expanded from the traditional PC with Windows NT/2000/XP to a wide range including PC's with Firewire and any operating system and 8-bit microcontrollers with bit-banged serial port. Even traditional PC hosted systems enjoy greater flexibility in Operating System selection.

Each MAXCLib function produces one packet that can be sent over a network to the controller. The code for interfacing with the network is written by the user. Examples of using MAXCLib are provided with the MAXCLib source files.

Stand-alone Operation

Agile Systems Inc. has responded to the need for increasing speed, precision and sophistication by introducing stand-alone operation for its line of Distributed Servo Motion Control products. Now, customers can develop custom applications and motion routines and store them locally to non-volatile memory. Plus, the stand-alone application can be executed without host interaction giving even more flexibility and ease-of-use for system developers.

Developing stand-alone applications is easy using the new Command Editor in DPWin - Agile's Windows based configuration and setup software tool. The Command Editor gives customers a user-friendly, graphical programming interface to configure and control all aspects of the motion hardware. Custom applications can be saved, loaded, downloaded, and executed for easy development and debugging. Customers can use a variety of instructions that include conditional jumps, event triggers and subroutines. User-defined registers are available along with a variety of arithmetic operations.

Utilizing Agile's Distributed Servo Controllers in stand-alone operation frees the host computer to handle other concurrent operations or independent tasks. Host commands can be issued to the controller at the same time as a program is being executed from non-volatile memory. Plus, a host controlled application can initiate stand-alone code whenever necessary. One of the biggest advantages of stand-alone operation is the ability to perform advanced motion control without the need of a host altogether.

Interaction with the stored routines would primarily be accomplished through the digital and analog IO. Another feature is the ability to group native MAX commands in a routine and store them on the controller for later execution. The host can issue a 'jump' packet to initiate the routine at any moment.

Applications Examples

Industrial Automation demands high reliability from all assembly systems. Executing programs embedded on the controller eliminates PC's with questionable reliability from assembly automation. A simple example consists of a PLC interfacing with the controller through digital inputs. When the PLC asserts one input, the controller moves the axes to position x, when another input is asserted the controller moves the axes to position y.

Another powerful feature enabled by storing programs locally is the ability to program OEM specific commands in the controller. Many OEM's have very specific requirements for 'Homing'. With Agile’s powerful command set, the homing routine can be stored on the controller and executed with a single 'jump' command issued from the host.

Generating Packets

How do customers generate Packets? There are three ways to create packets that are understood by the MAX controller.

1. You can use the ANSI C compliant MAXCLib library in your application to construct packets for transmission
2. You can use DPWin's Command Editor to generate and store packets on the controller.
3. You can construct the packets manually by filling in all the bytes according to the documentation showing the Command Interface network specification.

The first option should be used if you are developing custom software for your system. Further documentation should be consulted for further information regarding Agile's MAXCLib library.

The second option is available for those wishing to download custom routines or stand-alone programs to the controller. DPWin provides a user-friendly environment for developing custom routines in a Windows based environment. Read more about the DPWin Command Editor later in this manual.

The third option is akin to programming in the days before assemblers were invented. You must find the instruction you want, determine its op-code or command index, properly construct the arguments, and calculate a Cyclic Redundancy Check to transmit.

DPWin Command Editor

Overview

The DPWin Command Editor is a powerful, easy-to-use integrated development environment for programming stand-alone applications. The Editor provides an intuitive point-and-click approach to creating, debugging and executing custom applications. Many features of the Editor are described later in this section.

The Editor has the look and feel of a spreadsheet application and consists cells divided into four columns. Column one is an automatically generated line number. Column two contains a space to declare any subroutine labels. Column three contains the commands of the program. Column four contains optional comments. The keyboard cursor keys and/or mouse can be used to navigate through the workspace.

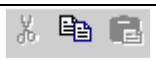
The following highlights some important details of the using the Command Editor:

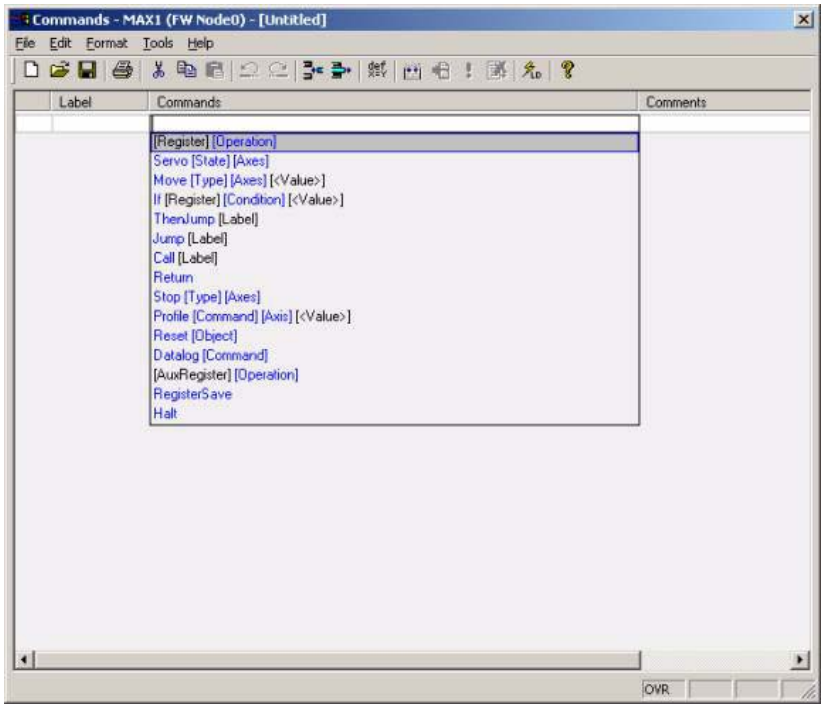
- Every program must be one row in length to a maximum of 2000 rows.
- The program model is a one-to-one approach. Each row of the workspace is equivalent to one command, or similarly one packet. And each row consists of an instruction time of fifty microseconds

The following highlights important details of the programming workspace:

- Labels cannot begin with numbers
- Text for labels, aliases, commands and arguments is case sensitive

Toolbar

	Create a new, open an existing, or save the open programming workspace
	Print a program workspace listing
	Cut, copy, or paste selected area
	Undo, Redo last action
	Insert or delete a line
	Alias definition table. Declare constants or rename objects.
	Build, Download, Run, and stop execution of a program from the workspace
	'Execute on boot' option that runs a program when controller is powered.



Command Pop Up Menu

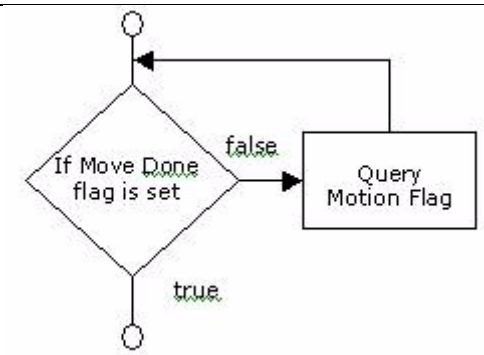
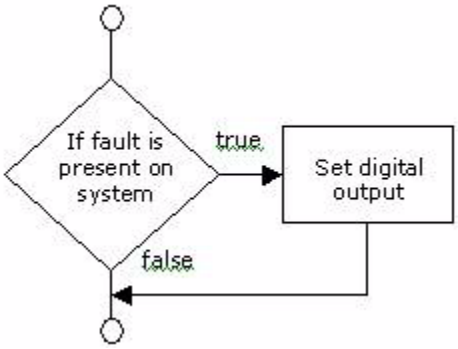
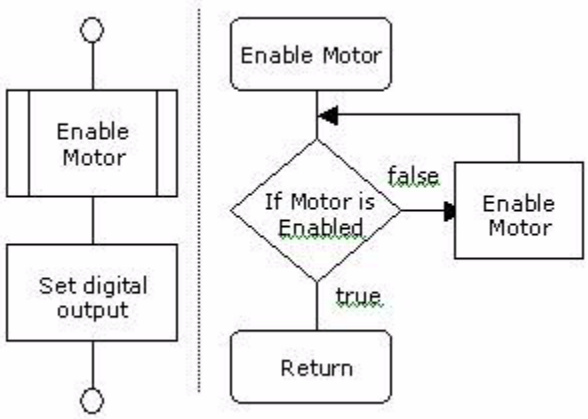
The Command Editor provides an interactive Command Pop Up menu to help with the successful creation of commands. The pop up menu will appear anytime a cell is activated under the 'command' column in the workspace. To activate a cell, simply single click on the cell with the left mouse button or start to type with the keyboard.

The interactive pop up menu is handy for creating commands since every command contains a number of arguments. Once a command is selected, the pop up menu will list the first argument selections. Once all the arguments for the command have been satisfied, the interactive menu will

disappear. If at any time you make a mistake, simply click on the previous selection in the cell. Or you may press the 'esc' key to hide the menu. Sometimes arguments require that a value be entered. Use the keypad to enter values.

Program Control

Program flow is sequential, top-to-bottom, left-to-right. Structured program development is accomplished through repetition, subroutines and conditional statements. Otherwise, tasks are accomplished issuing Agile commands and accessing Agile registers. Each program is formed by combining as many of each type of control as is appropriate.

Repetition	*While motion incomplete* *Query motion flag*	
Conditional Branch	*If fault present* *Then set digital output*	
Subroutine	*Enable motor* *Set digital output*	

Example: Cut-to-length routine

For this example, we are going to mimic a cut-to-length motion application. The mechanical system will move a saw, or some form of a cutting device, to a desired distance at which point, an output will engage the saw and a cut will be made.

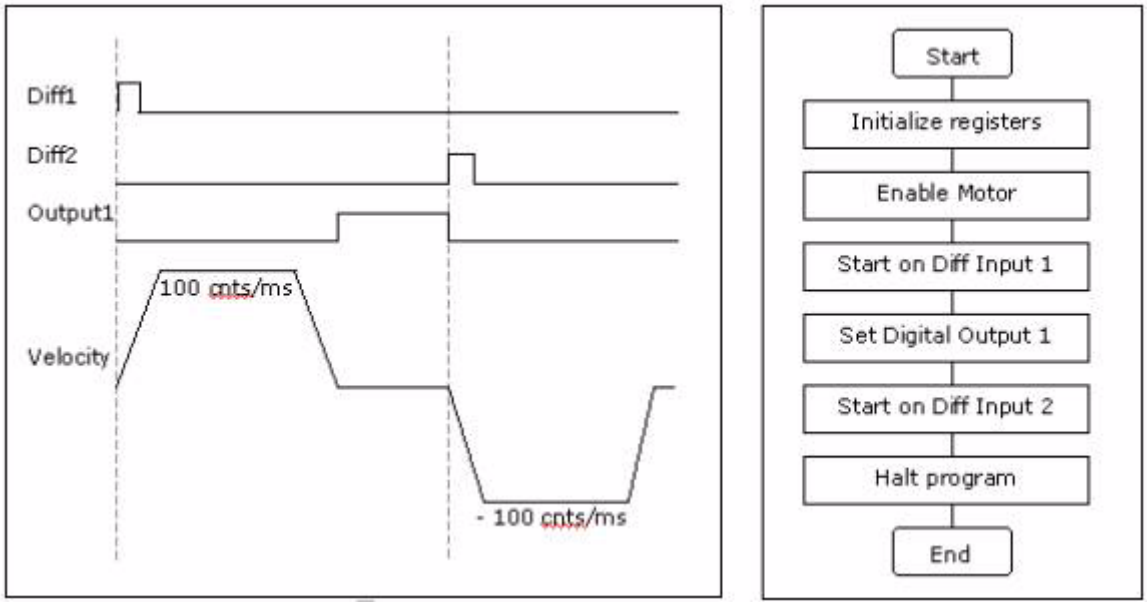


Figure 2Cut-to-length motion profile and flowchart

In order to accomplish the cut-to-length application, we will use the simple case where the motion of the motor must be delayed until a start pulse is given (applied to differential input 1). When the initial motion is complete, an output signal (applied to Optical Output 1) must be set active until a start pulse is given (applied to differential input 2). The output signal will be cleared and the motor will return to the starting position.

Consider the motion illustrated in Cut-to-length motion profile and flowchart. A forward motion will start on a pulse on the differential input 1. The velocity, acceleration, jerk, rates equal 50, 0.5, 0.1 respectively. After the motion has completed, the controller will assert Digital Output 1. Once a pulse on the differential input 2 is found, the controller will de-assert Digital Output 1 and start a backward motion until the original starting location is achieved. The distance forward will equal the distance backward.

The application assumes a system where the resolution of the encoder is 6000 counts per turn and that one revolution of the motor corresponds to mechanical travel distance of 4 inches. So, 2.4 feet of travel is equivalent to 43200 counts (2.4 feet * 18000 counts/feet). A user register will contain the distance in units of feet. The user register will be set to 2.0 feet initially. The flowchart in Cut-to-length motion profile and flowchart describes the program flow.

The following masks will be used for determining the differential input state and, therefore, must be defined as aliases:

```
Diff1           = 0x1000
Diff2           = 0x2000
```

Note: input and outputs are hardware specific. Refer to the Register Reference documentation for a listing of controller I/O and associated bit mask values.

Begin:	Velocity_Maximum[Axis1] = 50	//Set Velocity
	Acceleration_Positive_Maximum[Axis1] = 0.5	//Set Acceleration
	Jerk_Accelerating_Start_Maximum[Axis1] = 0.1	//Set Jerk
	User_0[Axis1] = 2.0	//Distance = 2.0 feet (Default)
	Call EnableMotor	//Enable Motor
	Reset Origin Axis1 0	//Reset position to 0
CheckDiff1:	If Digital_Input_Register_2 & Diff1 IsTrue	//Wait for Diff Input 1
	ThenJump CheckDiff1	//Check Diff 1 again
HostBegin:	Call MoveAbsolute	//Move Motor
	Digital_Output_Register_0 = 0x1001	//Assert Output 1
CheckDiff2:	If Digital_Input_Register_2 & Diff2 IsTrue	//Wait for Diff Input 2
	ThenJump CheckDiff2	//Check Diff 2 again
	Digital_Output_Register_0 = 0x1000	//Clear Output 1
	User_0[Axis1] = 0	//Distance = 0.0 feet
	Call MoveAbsolute	//Move Motor
	User_0[Axis1] = 2.0	//Distance = 2.0 feet (Default)
	Halt	//Halt program
EnableMotor:	Alignment_Type[Axis1] = 3	//Set alignment to 180 degrees
	Servo Enable Axis1	//Servo Motor
isMotorEnabled:	If Axis_Servo_Status[Axis1] & 1 IsTrue	//Wait for Motor to be Enabled
	ThenJump isMotorEnabled	//Check again
	Return	//Return from EnableMotor subroutine
MoveAbsolute:	User_1[Axis1] = User_0[Axis1]	//Copy distance to another user register
	User_1[Axis1] *= 18000	//Convert distance to encoder position
	Move Absolute Axis1 User_1[Axis1]	//Move motor to the distance required
	Move Go Axis1	//Begin Motion
WaitForMotion:	If Axis_Status[Axis1] & Axis_Status_Move_Done IsFalse	//Wait for Move to complete
	ThenJump WaitForMotion	//Check again
	Return	//Return from MoveAbsolute subroutine

Table: Cut-to-length example

Remarks for Cut-to-length program:

- It is assumed that a motor has been tuned prior to this exercise and is connected to axis 1
- The differential inputs have been wired in such a way that the ‘active’ state is a logic ‘off’ or ‘low’.
- The example has been written in such a way that a host application could initiate motion separately. An application using MAXCLib could construct the “Jump” command and execute the program at any label defined in the program. For instance, the host application could set a value for the User_0 register and jump to the “HostBegin” label, bypassing the differential input 1 pulse.

Example: Executing multiple programs

Only one main program can be downloaded and stored in controller non-volatile memory. However, multiple routines can make up the main program. Simple or complex routines can be placed end to end with the option of running one routine at a time.

Each row in the Command Editor could be a possible starting point of a routine. Normal execution will always start from the first line in the Command Editor, but a host application has the option of executing from any starting point.

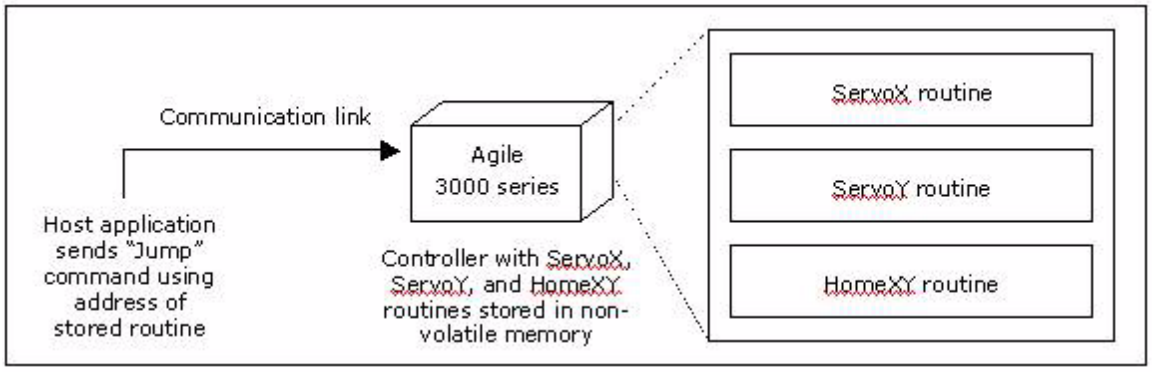
A single program should be made to have a start and finish. In the Command Editor, any row can be the start of a program and a "Halt" command indicates the end of the program. When combining multiple routines, make sure to put a "Halt" command at the end of the routine to stop execution. It should be noted that multiple routines cannot be executed simultaneously.

The point of storing multiple routines can only be realized when using a host application. For example, look at the group of routines in Table 6 Multiple program example. There are three routines listed: ServoX, ServoY, and HomeXY. Please note that the routines are incomplete and are only shown for demonstration purposes.

0	ServoX:	Call EnableX	//Start ServoX routine
1		Halt	//End ServoX routine
2	ServoY:	Call EnableY	//Start ServoY routine
3		Halt	//End ServoY routine
4	HomeXY:	Call HomeX	//Start HomeXY routine
5		Call HomeY	
6		Halt	//End HomeXY routine

Table: Multiple program example

Using the "Jump" command, a host application could execute any of the three routines. The "Jump" command requires an address as one of the parameters. The address for the ServoX, ServoY, and HomeXY routines are 0, 2, and 4 respectively.



Example: Input polling and Jogging

Programs can be run indefinitely when constantly polling for state changes. In this example, we want a 3-pole input switch to jog a motor at a specified speed. The 3-pole switch is connected to the controller through differential inputs where the upper position of the switch activates differential input one and the lower position of the switch activates differential input two. The middle position of the pole ensures that both differential inputs are inactive.

Upper	Positive	Differential input one active, two inactive
Middle	Stopped	Differential inputs one and two inactive
Lower	Negative	Differential input two active, one inactive

Consider a system where the resolution of the encoder is 6000 counts per turn and the motor is required to run at a speed of 600 rpm (10 revolutions per second) and must accelerate to that speed over 100 milliseconds. The acceleration will be 0.6 counts per millisecond squared and the velocity will be set to 64 counts per millisecond.

The following masks will be used for determining the differential input state and, therefore, must be defined as aliases:

```
isDiff1         = 0x1000
isDiff2         = 0x2000
IsNoDiff        = 0x0000
```

Note: input and outputs are hardware specific. Refer to the Register Reference documentation for a listing of controller I/O and associated bit mask values.

```
Begin:          Acceleration_Positive_Maximum[Axis1] = 0.6       // Acceleration rate
                Acceleration_Negative_Maximum[Axis1] = 0.6       // Deceleration rate
CheckIO:        If Digital_Input_Register_2 & isDiff1 IsFalse    // Check Diff1 is in active state
                ThenJump JogPositive                             // Start Positive Velocity
                If Digital_Input_Register_2 & isDiff2 IsFalse    // Check Diff2 is in active state
                ThenJump JogNegative                             // Start Negative Velocity
                If Digital_Input_Register_2 & isNoDiff IsFalse   // Check for no active Diff inputs
                ThenJump StopJog                                 // Start Zero Velocity
                Jump CheckIO                                     // Check IO again
JogPositive:    Move Jog Axis1 64.0                              // Positive Velocity
                Call isAxis1Done                                 // Wait for Velocity Setpoint
                Jump CheckIO                                     // Check IO again
JogNegative:    Move Jog Axis1 -64.0                             // Negative Velocity
                Call isAxis1Done                                 // Wait for Velocity Setpoint
                Jump CheckIO                                     // Check IO again
StopJog:        Move Jog Axis1 0                                 // Zero Velocity
                Call isAxis1Done                                 // Wait for Velocity Setpoint
                Jump CheckIO                                     // Check IO again
isAxis1Done:    If Axis_Status[Axis1] & Axis_Status_Move_Done IsFalse  // Check for Done flag on Axis1
                ThenJump isAxis1Done                             // Check Done flag again
```

	Return	// Return from subroutine

Table: Input polling and jogging example

Remarks for Input Polling and Jogging program:

- It is assumed that a motor has been tuned prior to this exercise and is connected to axis 1
- The motor must be successfully enabled prior to execution and no faults present on axis 1
- The differential inputs have been wired in such a way that the ‘active’ state is a logic ‘off’ or ‘low’.

Example: Point-to-point motion

In this example, we want to perform a relative point-to-point motion with the objective of rotating the motor one revolution in a positive direction and one revolution in a negative direction along a trapezoidal velocity, with a total time of 0.3 seconds per revolution. Assuming an encoder resolution of 6000 counts per turn and acceleration/deceleration times of 0.1 sec each, we can express the following motion parameters for each revolution:

- Distance of 6000 counts
- Maximum velocity 23 counts/ms
- Acceleration 0.6 counts/ms2
- Deceleration 0.6 counts/ms2

Begin:	Velocity_Maximum[Axis1] = 23	//Set Speed
	Acceleration_Positive_Maximum[Axis1] = 0.6	//Set Acceleration Rate
	Acceleration_Negative_Maximum[Axis1] = 0.6	//Set Deceleration Rate
	Move Relative Axis1 6000	// Positive Distance
	Move Go Axis1	// Issue move
	Call isMoveDone	//Wait for motion done
	Move Relative Axis1 -6000	//Negative Distance
	Move Go Axis1	//Issue move
	Call isMoveDone	//Wait for motion done
	Halt	//End program
isMoveDone:	If Axis_Status[Axis1] & Axis_Status_Move_Done IsFalse	//Check for done flag set
	ThenJump isMoveDone	//Keep checking
	Return	//Return from subroutine

Table: Point-to point motion example

Remarks for Point-to-point motion program:

- It is assumed that a motor has been tuned prior to this exercise and is connected to axis 1
- The motor must be successfully enabled prior to execution and no faults are present on axis 1

Feel free to verify that the move is performing as calculated by using DPWin's 4-channel oscilloscope. Set channel one to 'Velocity Setpoint' and channel two to 'Position Setpoint'. Set the Datalog rate to 0.75ms with 400 sample points and set the trigger settings to '+ve motion go' on 'Axis1_1'. Lastly, press the 'Auto' button and execute the program.

Example: Position capture and unsolicited packets to host

At some point, it may be desirable to send a notification packet to a host. Unsolicited packets can provide helpful information and could be sent at any point in a program by using the "RegisterSend" command. This feature provides flexibility to the developer to have the stand-alone program notify the host when necessary.

For example, a homing routine could send a packet indicating that an axis of a mechanical system has reached a known starting point. An unsolicited packet can be used as a troubleshooting tool when designed complicated routines.

For this example, we will mimic a simple homing routine where a sensor attached to differential input 1 will indicate the home position. The axis will be jogged in a negative direction at a constant velocity until the input is triggered. To end the routine, an unsolicited packet will be sent to the host.

This example will also show how to use the high-speed position capture feature. The captured position will indicate the exact encoder position when the homing sensor connected to the differential input was reached. This position will be sent to the host as part of the unsolicited packet.

Home:	Position_Capture_Input_Register[Axis1] = 2322	// Capture on Diff inputs register #
	Position_Capture_Input_Bitmask[Axis1] = 0x1000	// Capture on Diff 1 only
	Position_Capture_Input_Level_Sense_Mask[Axis1] = 0	// 1->0 transition
	Position_Capture_Input_Transition[Axis1] = 0	// Enable Single transition
	Position_Capture_Start[Axis1] = 1	// Enable position capture
	Move Jog Axis1 -10.0	// Jog at 10 cnts/ms
	Call isCapture	// Call position capture subroutine
	User_C0 = Position_Capture_Position[Axis1]	// Save captured position
	User_C0 Send	// Send captured position to host
	Halt	// End program
IsCapture:	If Position_Capture_Complete[Axis1] != 1	// Check for capture flag being set
	ThenJump isCapture	// If not, check again
	Stop Soft Axis1	// Stop motion on Axis1
	Return	// Return from subroutine

Table: Position capture and unsolicited packet to host example

Remarks for Position capture example program:

- It is assumed that a motor has been tuned prior to this exercise and is connected to axis 1
- The motor must be successfully enabled prior to execution and no faults are present on axis 1
- The position capture feature requires proper register configuration. Consult the Register Reference documentation for further explanation.

Example: Servo motor and Fault Detection

The following example illustrates how to servo a motor connected to the MAX Controller. The program involves a single subroutine that will, first, check for fault conditions and, second, servo the motor using a predefined alignment type. Digital output 1 will be set in the event of an error present on axis 1. Otherwise, the subroutine will not return until the servo flag is set on axis 1 indicating an 'enabled' state.

Begin:	Call EnableMotor	// Enable motor
	Halt	// End program
EnableMotor:	If Axis_Status[Axis1] & Axis_Status_Error IsTrue	// Check for Axis1 faults
	ThenJump SetFaultOutput	// Fault present, set output
	Jump ServoMotor	// No faults, servo motor
SetFaultOutput:	Digital_Output_Register_0 = 0x0001	// Set Digital Output 1
	Jump EndSubroutine	// Skip to the end
ServoMotors:	Alignment_Type[Axis1] = 3	// 180 degree alignment
	Servo Enable Axis1	// Servo Axis1
isAxis1Enabled:	If Axis_Servo_Status[Axis1] & Servo_Disabled IsTrue	// Check if Axis1 is Enabled
	ThenJump isAxis1Enabled	// Still Disabled, keep checking
EndSubroutine:	Return	// End subroutine

Table: Servo motor and fault detection example

Remarks for Servo Motor and Fault Detection program:

- It is assumed that a motor has been tuned prior to this exercise and is connected to axis 1.
- 180 degree alignment should only be used for sinusoidal commutation.