# _____NMCLIB04.DLL: A Windows DLL for NMC Modules_____

NMCLIB04.DLL is a Windows DLL that includes functions for communicating with ***PIC-SERVO*** *(v.4, v.5, v.10)* , ***PIC-STEP***, and ***PIC-I/O*** modules. It can be used with almost all Windows programming languages. This DLL was created using Borland's C++ Builder, and source code is included with the DLL.

## 1. Overview

This document describes the functions in NMCLIB04.DLL and compliments the data sheets for the ***PIC-SERVO***, ***PIC-STEP*** and ***PIC-I/O*** controller chips. Please refer to those data sheets for additional detailed descriptions of the parameters referenced by the DLL functions. The DLL library, controller chip data sheets, and example programs are available from *www.jrkerr.com/software.html*.

NMCLIB04.DLL is a library of functions for using NMC (Networked Modular Control) compatible modules. It is an upgrade from NMCLIB0X.DLL, and includes support for ***PICSERVO*** (v.4, v.5 and v.10) motor control modules, ***PIC-STEP*** stepper motor control modules, and ***PIC-IO*** multi-function I/O modules.

NMCLIB04.DLL consists of three levels of functions. At the lowest level is the group of serial I/O functions listed in SIO_UTIL.H. These functions provide basic (non-overlapped) COM port support independent of the NMC communication protocol. It includes functions for opening and closing COM ports, sending and receiving characters, etc. Typically, you will never need to call these functions directly.

The next level of functions, listed in NMCCOM.H, provide basic support of the NMC communication protocol. The functions at this level are independent of the types of modules used. They include initialization and reset functions for the entire network of modules, module control functions common to all module types such as reading or defining status data packets, and functions for retrieving data common to all module types.

The last level of functions are those specific to particular types of modules. These are described in PICSERVO.H, PIC-STEP.H and PICIO.H. They include functions for operations specific to each module type, and functions for retrieving module-type specific data.

In general, there are two types of module specific commands - ones for setting parameters or executing actions, and ones for retrieving data. Data for a module is located in one of two places: on the host PC, or in the module itself. Certain data sent to a module, like command velocities, for example, are not retrievable from the module. When such data are sent, however, the DLL stores the current value locally on the PC. This data can be retrieved at any time using the appropriate “Get” function such as `ServoGetCmdVel()`.

**••• CAUTION •••**

NMCLIB04.DLL is provided without charge for the convenience of developing stand-alone applications, and no warranty, implied or otherwise, is provided. It is up to the user to verify that any application using this library meets appropriate safety standards.

Data stored within the controller modules, such as position data, must first be read into the host PC. There are two ways to read in this data: you can use `NmcDefineStatus()` to have the data of interest returned to the host with every command sent to a module, or you can use `NmcReadStatus()` to read a particular bit of data just once. Once the current value of the data has been read from the controller, you can again use the appropriate "Get" function, such as `ServoGetPos()`, to retrieve the data from the DLL.

In addition to the function descriptions for each header file below, users should look at the example programs provided, and may even look at the source code provided for the DLL itself. The source code and make files are for Borland C++ Builder, although they should be portable to other C compilers.

## 2. Software Description

Figure 1 shows a typical application using the NMCLIB04.DLL Windows DLL. It consists of a Windows PC, a ***SSA-485*** Smart Serial Adapter, and several NMC modules (for example, two ***PIC-SERVO*** and one ***PIC-IO***). The PC is running the user application code and the NMCLIB04.DLL. The user application calls NMCLIB04.DLL functions to to initialize communications, send commands, and receive status from the NMC modules. The ***SSA-485*** is jumpered to operate in pass-through mode, and converts the RS232/USB serial data from the PC to RS485 (4 wire) serial data to the NMC modules.

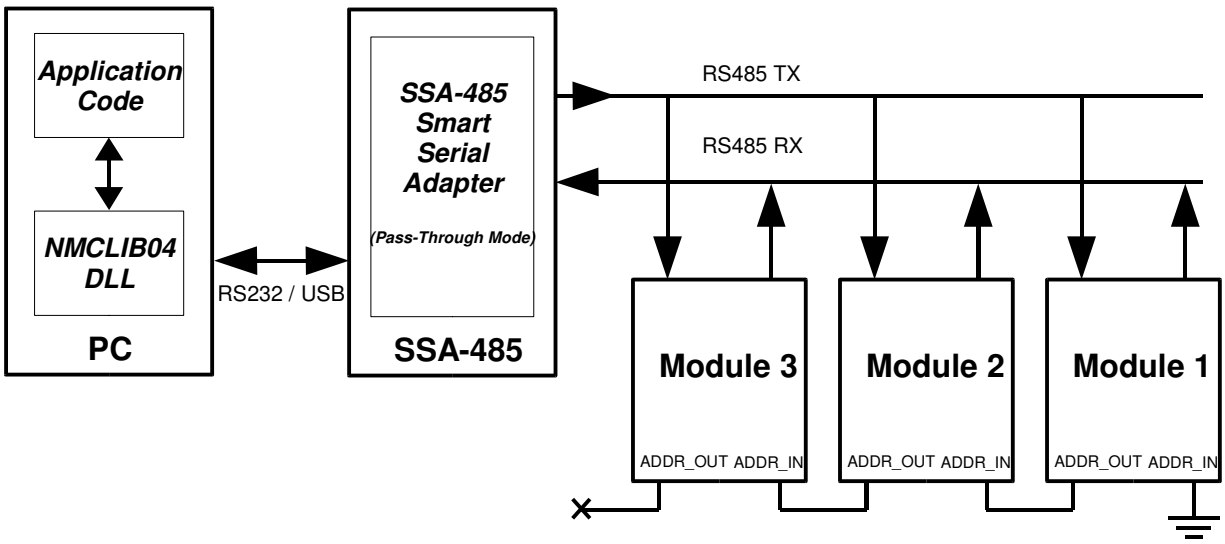


***Figure 1*** **- Typical Application Using NMCLIB04.DLL**

## 2.1 Initialization

Before any NMC module commands are sent, the NMCLIB04.DLL must be initialized by calling `NmcInit()`. `NmcInit()` performs the following tasks:

- Open UART
- Reset Modules
- Assign Addresses
- Initialize Structures

### Open UART

NMCLIB04.DLL communicates with the ***SSA-485*** Smart Serial Adapter (which communicates to the NMC modules) through a COM port on the PC. `NmcInit()` first opens the COM port at the default baud rate of 19200 using the port number specified in the command line argument, then starts performing NMC initialization. After the NMC initialization is complete, `NmcInit()` changes the baud rate to the rate specified in the command line argument.

After `NmcInit()` returns, the application may change the communications baud rate rate at any time by calling `NmcChangeBaud()`.

In order to prevent communications errors due to mismatched baud rates, the baud rate should be set back to the default rate of 19200 at the end of the application program (for example, if the rate was changed to 57600). This will ensure that when the application restarts at 19200, it will be able to talk to the modules. Otherwise, communications with the modules will fail because of a baud rate mismatch, and the modules will have to be power-cycled to set them back to 19200.

### Reset Modules

After the COM port is opened, a Reset command is sent to the universal reset address 0xFF. On receiving this command, modules will reset themselves to their power-up state. In this state they will be waiting to be assigned an address.

Because the universal reset address 0xFF is recognized only by new versions of ***PIC-SERVO*** and ***PIC-STEP*** modules, care must be taken when working with older versions of ***PIC-SERVO*** and ***PIC-STEP*** modules, and any ***PIC-IO*** modules. To ensure that these modules can be reset at initialization, the module's group address must be set to 0xFF using `NmcSetGroupAddress()` at the end of the application program (only if the group address was changed from the default 0xFF). Alternatively, power-cycling the module will restore the module group address to it's default value of 0xFF.

### Assign Addresses

After modules are reset, they are assigned an individual address between 1 and 32 inclusive. Addresses are assigned sequentially starting with address 1. Modules are also assigned group address 0xFF. After initialization, the application program cannot change a module's individual address, but may change a module's group address at any time with the `NmcSetGroupAddress()` command.

Individual addresses are assigned to modules based on their network topology, starting with the

daisy-chained module that is furthest from the host. For example, in Figure 1, module 1 is assigned address 1, module 2 is assigned address 2, etc. Note that if the network topology is changed such that the modules are daisy-chained if a different order, modules will be assigned different addresses.

### Initialize Internal Data Structures

NMCLIB04.DLL internal data structures are set during initialization. For each module detected, the module type and module version is read and stored, and the default status items is set to 0 (no additional status items are sent in the status packets in response to a command). The number of modules detected is stored, and returned by `NmcInit()`.

## 2.2 Command Addressing

The bulk of the functions in NMCLIB04.DLL are for sending commands to modules. Commands may be sent to modules using either their individual, group, or universal reset address.

### *Individual Address*

At power up, or after a reset, all modules have the default address of 0x00. The application program must run `NmcInit()` during initialization to assign each module a unique individual addresses between 1 and 32 inclusive. Once assigned, the application program cannot change the module's individual address. The modules keep their address until they are reinitialized by `NmcInit()` or power-cycled.

***PIC-SERVO*** modules may optionally save their configuration data to EEPROM (including the module's address) using the `ServoHardReset()` command. On power-up, the module will be configured with configuration data read from EEPROM. In general, we recommend the configuration data should not be saved in EEPROM, and the modules should be completely configured by the host on start-up. This will reduce addressing conflicts and problems associated with keeping track of the state of each module. See `ServoHardReset()` for examples of special circumstances when configuration data should be saved to EEPROM.

### *Group Address*

Each NMC controller module also has a group address. On power-up or reset, the group address defaults to 0xFF. Group address are set with the `NmcSetGroupAddress()` and are restricted to values between 128 and 255.

The purpose of the group address is to be able to send a single command (such as `NmcSynchOutput()`) to a several controllers at the same time. While the individual addresses of all controllers must be unique, a group of controllers can share a common group address. When a command packet is sent over the NMC network to a group address, all modules with a matching group address will execute the command.

The issue of which module will send a status packet in response to a group command is resolved with the distinction between group *members* and group *leaders*. When the group address for a module is set, the `NmcSetGroupAddress()` command will also specify if the module is to be the *leader* or a *member* of that group. If a module is a member of it's group and it receives a group command (*i.e.*, a command sent to it's group address), it will execute the command but *not*

send back a status packet. If a module is the leader of it's group and it receives group command, it *will* send back a status packet in addition to executing the command. (The status packet is just the same as one sent in response to an individually addressed command.)

For any group of modules sharing the same group address, only one module should be declared the group leader.

In certain instances (as when changing the Baud rate for all modules on the network), it is necessary to send a command to a group without a group leader. In this case, no status will be coming back from any controllers, and the host should wait for at least 0.51 milliseconds before sending another command to keep from overwriting the previous command. If you need to change the baud rate, it is best, when initially setting the addresses, to leave the group address for all modules at 0xFF with no group leaders. After changing the baud rate, you can then re-define the group addresses as needed.

### *Universal Reset Address*

For most applications, group commands are not needed and the group address for all modules is left at 0xFF. If, however, the modules are split up into several groups with different group addresses, sending a single Reset command to reset all controllers at once becomes problematic. To address this issue, newer versions of ***PIC-SERVO SC*** and ***PIC-STEP*** modules will always execute a Hard Reset command sent to the address 0xFF, independent of the value of the module's group address (note that when a Reset command is sent, no status packet will be returned). Older versions of ***PIC-SERVO*** and ***PIC-STEP*** modules, and all ***PIC-IO*** modules do not recognize the universal reset address. To reset these modules, the reset command can be sent to the module's individual address, it's group address (the default module group address is 0xFF unless it is changed by `NmcSetGroupAddress()`), or the module can be power cycled.

## 2.3Status Packets

Modules send status packets in response to commands. The status packet consists of status data followed by a single checksum byte. The default status packet contains a single status byte (plus the checksum byte) with basic information about the state of the module, including whether or not the previous command had a checksum error. The application program may optionally program the module to send additional status information using the `NmcDefineStatus()` and the `NmcReadStatus()` commands. Additional status information includes information such as position values, timeout/counter values, and input A/D values.

The conditions for when a module sends a status packet depends on the command sent and the command address. Modules never send a status packet is response to library functions `NmcHardReset()`, `NmcChangeBaud()`, and `ServoHardReset()`. Modules will send status packets in response to commands sent to their individual address, including when the module detects a command checksum error. For commands sent to a module's group address, the module will send a status packet if it is the group leader, otherwise no status packet will be sent.

Returned status information is stored internally by NMCLIB04.DLL. Application programs can retrieve the status information using a set of functions defined to retrieve individual status fields from ***PIC-SERVO***, ***PIC-STEP***, and ***PIC-I/O*** status packets.

Accessing status data within your program is a two step process. The first step is to retrieve the

status data from a module. If the status data of interest (e.g., position data) has been specified with an `NmcReadStatus()` command, any command sent to the module will cause that status data to be returned. You can also use `NmcReadStatus()` to have some item of status data sent back just once. The status data sent back is stored internally by NMCLIB04.DLL. To access the particular data fields stored within the DLL, you should use one of the functions listed in Section 4.

Note that the status data functions simply retrieve data from the module's data structure stored locally within the DLL. To insure that the status data is current, a `NmcReadStatus()` (with the appropriate bit in the 'statusitems' byte set) command should be issued just prior to calling the status data function. If `NmcDefineStatus()` has already been used to permanently include the relevant data item in the default status packet, any command sent to the module will update the local data structure.

## 3. Function Specification

The *NMCLIB04.DLL* functions are grouped into five categories: Serial I/O Functions, NMC functions, ***PIC-SERVO*** functions, ***PIC-STEP*** functions, and ***PIC-IO*** functions. The functions are summarized as follows:

### SIO Functions

| Function | Description |
|---|---|
| ErrorMsgBox | Display a simple message box for low-level error messages |
| ErrorPrinting | Controls whether low-level error messages are printed or suppressed |
| SimpleMsgBox | Display a simple message box |
| SioChangeBaud | Changes the baud rate of a COM port |
| SioClose | Closes a COM port |
| SioClrInbuf | Clears all the characters in a COM port's input buffer |
| SioGetChars | Read characters from a COM port |
| SioOpen | Opens a COM port at the specified baud rate. |
| SioPutChars | Puts characters out a COM port |
| SioTest | Returns the number of characters in a COM port's input buffer |

## NMC Functions

| Function | Description |
|---|---|
| FixSioError | Attempts to re-sync communications |
| InitVars | Initialize miscellaneous network variables |
| NmcChangeBaud | Changes the network baud rate of the COM port and modules |
| NmcDefineStatus | Defines what status data is returned after each command packet sent |
| NmcGetGroupAddr | Returns module's group address |
| NmcGetModType | Returns module's module type |
| NmcGetModVer | Returns module's firmware version |
| NmcGetStat | Returns module's current status byte |
| NmcGetStatItems | Returns the byte specifying which default status items are returned in a module's status data packet |
| NmcGroupLeader | Returns true is specified module is a group leader |
| NmcHardReset | Resets module to it's power up state |
| NmcInit | Opens the COM port and initializes the NMC network |
| NmcNoOp | No operation – module returns current status |
| NmcReadStatus | Reads the specified status items once |
| NmcSendCmd | Low level routine for sending a command packet |
| NmcSetGroupAddress | Sets a module's group address |
| NmcShutdown | Resets modules then closes COM port |
| NmcSynchInput | Causes a group of modules to synchronously capture input values |
| NmcSynchOutput | Causes a group of modules to synchronously execute a previously stored output command |

### *PIC-SERVO* Functions

| Function | Description |
|---|---|
| ServoAddPathpoints | Adds a set of path points for path mode operation |
| ServoClearBits | Clears the latched status bits |
| ServoGetAD | Returns the current A/D value |
| ServoGetAux | Returns the current auxiliary status byte |
| ServoGetCmdAcc | Returns the most recently issued command acceleration |
| ServoGetCmdPos | Returns the most recently issued command position |
| ServoGetCmdPwm | Returns the most recently issued command PWM value |
| ServoGetCmdVel | Returns the most recently issued command velocity |
| ServoGetGain | Returns the most recently issued servo gain values |
| ServoGetGain2 | Returns the most recently issued servo gain values (use this version for ***PIC-SERVO SC***) |
| ServoGetHome | Returns the current motor home position |
| ServoGetHomeCtrl | Returns the most recently issued home command control byte |
| ServoGetIoCtrl | Returns the most recently issued I/O command control byte |
| ServoGetMoveCtrl | Returns the most recently issued move command control byte |
| ServoGetNPoints | Returns the number of path points remaining |
| ServoGetPError | Returns the servo positioning error |
| ServoGetPos | Returns the current motor position |
| ServoGetStat | Low-level routine to process and store returned ***PIC-SERVO*** status data |
| ServoGetStopCtrl | Returns the most recently issued stop command control byte |
| ServoGetStopPos | Returns the most recently issued stop position |
| ServoGetVel | Returns the current motor velocity |
| ServoHardReset | Reset the controller to its power-up state and optionally store configuration data in EEPROM |
| ServoInitPath | Initializes the starting point of a path to the current motor position |
| ServoLoadTraj | Loads motion trajectory information |
| ServoNewMod | Creates and initializes a new SERVOMOD structure |

| Function | Description |
|---|---|
| ServoResetPos | Resets the position counter to zero |
| ServoResetRelHome | Resets the position relative to the home position register |
| ServoSetGain | Sets the servo gains |
| ServoSetGain2 | Sets the servo gains (use this version for new applications) |
| ServoSetHoming | Sets homing mode parameters for capturing the home position |
| ServoSetIoCtrl | Controls the configuration of the LIMIT1 and LIMIT2 I/O pins, as well as other miscellaneous functions |
| ServoSetPos | Set the servo position to a specific value |
| ServoStartMove | Synchronously starts preloaded motions |
| ServoStartPathMode | Starts execution of the path loaded into the internal path point buffer |
| ServoStopHere | Stop the motor at the specified position |
| ServoStopMotor | Stops the motor in the manner specified by mode |

### *PIC-STEP* Functions

| Function | Description |
|---|---|
| StepGetAD | Returns the current A/D value |
| StepGetCmdAcc | Returns the most recently issued command acceleration |
| StepGetCmdPos | Returns the most recently issued command position |
| StepGetCmdSpeed | Returns the most recently issued command speed |
| StepGetCmdST | Returns the most recently command timer count |
| StepGetCtrlMode | Returns the control mode byte (set with StepSetParam) |
| StepGetHome | Returns the current motor home position |
| StepGetHomeCtrl | Returns the homing control byte |
| StepGetHoldCurrent | Returns the holding current (set with StepSetParam) |
| StepGetInbyte | Returns the current input byte |
| StepGetMinSpeed | Returns the minimum stepping speed |
| StepGetOutputs | Returns the most recently command output byte |
| StepGetPos | Returns the current motor position |
| StepGetRunCurrent | Returns the running current (set with StepSetParam) |
| StepGetStat | Low-level routine to process and store returned ***PIC-STEP*** status data |
| StepGetStepTime | Returns the current timer count |
| StepGetStopCtrl | Returns the stopping control byte |
| StepGetThermLimit | Returns the thermal limit (set with StepSetParam) |
| StepLoadTraj | Loads motion trajectory information |
| StepNewMod | Creates and initializes a new STEPMOD structure |
| StepResetPos | Resets position counter to zero |
| StepSetHoming | Set homing mode parameters for capturing the home position |
| StepSetOutputs | Sets or clears the general purpose output pins |
| StepSetParam | Set the ***PIC-STEP*** operating parameters |
| StepStopMotor | Stops a motor in the manner specified by mode |

### *PIC-I/O* Functions

| Function | Description |
|---|---|
| IoBitDirIn | Sets the direction of an I/O bit to be an input bit |
| IoBitDirOut | Set the direction of an I/O bit to be an output bit |
| IoClrOutBit | Clears the value of an output bit to 0 |
| IoGetADCVal | Returns the A/D value from channel 0, 1, or 2 |
| IoGetBitDir | Returns I/O bit direction |
| IoGetPWMVal | Returns the most recently set PWM value for channel 0 or 1 |
| IoGetStat | Processes and stores returned ***PIC_IO*** status data |
| IoGetTimerMode | Returns the most recently set timer control byte |
| IoGetTimerSVal | Returns the synchronously captured timer value |
| IoGetTimerVal | Returns the timer value |
| IoInBitSVal | Returns the value of a synchronously captured input bit |
| IoInBitVal | Returns the value of an input bit |
| IoNewMod | Creates and initializes a new IOMOD structure |
| IoOutBitVal | Returns the most recently set state of an output bit |
| IoSetOutBit | Set the value of an output bit to 1 |
| IoSetPWMVal | Set the PWM output values |
| IoSetSyncOutput | Sets the output bit values and the PWM values to be set synchronously when the NmcSynchStart() function is called. |
| IoSetTimerMode | Sets the mode of operation for the timer/counter |

| **ErrorMsgBox** |
| --- |
| Display a simple message box for low-level error messages |

Function Prototype

int ErrorMsgBox(char *msgstr);

File Name

sio_util.cpp

Include

sio_util.h

Return Value

Returns 0 if fails, or when error message printing is disabled (see ErrorPrinting()). Returns non-zero on success. See the Windows OS MessageBox() function description for success return values.

Arguments

msgstr – Error message string

Pointer to a null-terminated string that contains the message to be displayed.

Description

Displays a simple windows message box for low-level error messages under display control. ErrorMsgBox() message display is enabled by calling ErrorPrinting(1), and disabled by calling ErrorPrinting(0).

Example

To display the error message “Low Level Debugging Error”:

```
ErrorMsgBox("Low Level Debugging Error");
```

| **ErrorPrinting** |
| --- |
| Controls whether low-level error messages are printed or suppressed |

Function Prototype

void ErrorPrinting(int f);

File Name

sio_utils.cpp

Include

sio_utils.h

Return Value

None.

Arguments

f – Enable

Non-zero enables low-level error printing by ErrorMsgBox(), zero disables.

Description

Controls whether the low-level error messages are printed or suppressed by the ErrorMsgBox() function.

Example

To enable the printing of low-level error messages by ErrorMsgBox():

```
ErrorPrinting(1);
```

| **FixSioError** (Internal Library Function) |
| --- |
| Attempts to re-sync communications |

Function Prototype

void FixSioError(void);

File Name

nmccom.cpp

Include

nmccom.h

Return Value

None.

Arguments

None.

Description

Attempts to re-sync NMC communications. FixSioError() keeps track of sequential communication errors. If FixSioError() detects multiple sequential errors, it will display the message "Multiple communications errors – please reset the Network" and return.

**NOTE:** For normal operation, users do not need this command.

| **InitVars** (Internal Library Function) |
| --- |
| Initializes miscellaneous network variables |

Function Prototype

void InitVars(void);

File Name

nmccom.cpp

Include

nmccom.h

Return Value

None.

Arguments

None.

Description

Initializes miscellaneous network variables. Called during network initialization.

**NOTE:** For normal operation, users do not need this command.

| **IoBitDirIn** |
| --- |
| Sets the direction of an I/O bit to be an input bit |

Function Prototype

BOOL IoBitDirIn(byte addr, int bitnum);

File Name

picio.cpp

Include

picio.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

bitnum – Bit number

Bit number to set. Bit numbers 0-11 correspond to I/O pins 1-12. Bit numbers 12-15 are ignored.

Description

The ***PIC-IO*** uses a bit-field to control the direction of the I/O pins. Bit numbers 0-11 of the bit-field correspond to I/O pins 1-12. IoBitDirIn() sets the direction of I/O bit "bitnum" to be an input bit. On power-up, all bits are defined as inputs.

Example

To set I/O pin 4 (bit number 3) of module 1 to be an input bit:

```
IoBitDirIn(1, 3);
```

| **IoBitDirOut** |
| --- |
| Sets the direction of an I/O bit to be an output |

Function Prototype

BOOD IoBitDirOut(byte addr, int bitnum);

File Name

picio.cpp

Include

picio.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

bitnum – Bit number

Bit number to set. Bit numbers 0-11 correspond to I/O pins 1-12. Bit numbers 12-15 are ignored.

Description

The ***PIC-IO*** uses a bit-field to control the direction of the I/O pins. Bit numbers 0-11 of the bit-field correspond to I/O pins 1-12. IoBitDirOut() sets the direction of I/O bit "bitnum" to be an output bit. On power-up, all bits are defined as inputs. Make sure that any I/O pins defined as outputs are not connected to the output of another device, or else the ***PIC-I/O*** or the other device may be damaged.

Example

To set I/O pin 4 (bit number 3) of module 1 to be an output bit:

```
IoBitDirOut(1, 3);
```

| **IoClrOutBit** |
| --- |
| Clears the value of an output bit to 0 |

Function Prototype

BOOL IoClrOutBit(byte addr, int bitnum);

File Name

picio.cpp

Include

picio.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

bitnum – Bit number

Bit number to cleared. Bit numbers 0-11 correspond to I/O pins 1-12. Bit numbers 12-15 are ignored.

Description

The ***PIC-IO*** uses a bit-field to control the direction of the I/O pins. Bit numbers 0-11 of the bit-field correspond to I/O pins 1-12. IoClrOutBit() clears the value of output bit "bitnum" to 0. Make sure that any I/O pins defined as outputs are not connected to the output of another device, or else the ***PIC-I/O*** or the other device may be damaged. This has no effect if the bit is defined as an input.

Example

To clear I/O output pin 4 (bit number 3) of module 1 to zero:

```
IoClrOutBit(1, 3);
```

| **IoGetADCVal** |
| --- |
| Returns the A/D value from channel 0, 1, or 2 |

Function Prototype

byte IoGetACDVal(byte addr, int channel)*;

File Name

picio.cpp

Include

picio.h

Return Value

Returns ADC value (0 – 255) of specified channel, or 0 for invalid channel.

Arguments

addr – Module address

Module address (1 – 32)

channel – ADC channel

ADC channel (0, 1, or 2)

Description

Returns the A/D value from channel 0, 1, or 2 (stored locally) from a PIC-IO module. Note: this data is only valid if the SEND-ADn (n=1,2 or 3) bit has been set in the most recently issued NmcDefineStatus() command.

Example

To get the channel 2 A/D value from module 1:

```
adc_val = IoGetADCVal(1, 2);
```

*This function only retrieves data stored locally on the PC. To insure the data is current, NmcReadStatus should be called just prior to calling this function. Alternately, if NmcDefineStatus has been used to permanently include the relevant data item in the status packet, any command sent to a module will update the locally stored data.

| **IoGetBitDir** |
| --- |
| Returns I/O bit direction |

Function Prototype

BOOL IoGetBitDir(byte addr, int bitnum);

File Name

picio.cpp

Include

picio.h

Return Value

0 = Bit defined as an output
1 = Bit defined as an input

Arguments

addr – Module address

Module address (1 – 32)

bitnumber – Bit number

Bit number to be examined. Bit numbers 0-11 correspond to I/O pins 1-12. Bit numbers 12-15 are ignored.

Description

Returns 0 if a *PIC-IO* module I/O bit is defined as an output, 1 if defined as an input.

Example

To read the direction of bit 6 of module 1:

```
bit_dir = IoGetBitDir(1, 6);
```

| **IoGetPWMVal** |
| --- |
| Returns the most recently set PWM value for channel 0 or 1 |

Function Prototype

byte IoGetPWMVal(byte addr, int channel);

File Name

picio.cpp

Include

picio.h

Return Value

Returns the PWM value (0-255) of the specified channel.

Arguments

addr – Module address

Module address (1 – 32)

channel – Channel

PWM channel (0 or 1)

Description

Returns the most recently set PWM value for channel 0 or 1 of a ***PIC-IO*** module.

Example

To get the most recently set channel 0 PWM value of module 1:

```
pwm_val = IoGetPWMVal(1, 0);
```

| **IoGetStat** (Internal Library Function) |
| --- |
| Processes and stores returned ***PIC-IO*** status data |

Function Prototype

BOOL IoGetStat(byte addr);

File Name

picio.cpp

Include

picio.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

Description

IoGetStat() processes and stores the status data returned from a ***PIC-IO*** module. It takes the status data stored in the global array "inbuf", verifies the number of bytes received and checksums, then stores the status fields in the NMCMOD structure mod[addr].

**NOTE:** For normal operation, users do not need this command.

| **IoGetTimerMode** |
| --- |
| Returns the most recently set timer control byte |

Function Prototype

byte IoGetTimerMode(byte addr);

File Name

picio.cpp

Include

picio.h

Return Value

Returns the most recently set timer control byte.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the most recently set timer control byte for a ***PIC-IO*** module.

Example

To get the most recently set timer control byte for module 1:

```
tmr_mode = IoGetTimerMode(1);
```

| **IoGetTimerSVal** |
| --- |
| Returns the synchronously captured timer value |

Function Prototype

unsigned long IoGetTimerSVal(byte addr)*;

File Name

picio.cpp

Include

picio.h

Return Value

Returns the synchronously captured timer value.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the synchronously captured timer value (stored locally) from a ***PIC-IO*** module. Note: this data is only valid if the SEND_SYNC_TMR bit has be set in the most recently issued NmcDefineStatus() command.

Example

To return the synchronously captured timer value of module 1:

```
sval = IoGetTimerSVal(1);
```

*This function only retrieves data stored locally on the PC. To insure the data is current, NmcReadStatus should be called just prior to calling this function. Alternately, if NmcDefineStatus has been used to permanently include the relevant data item in the status packet, any command sent to a module will update the locally stored data.

| **IoGetTimerVal** |
| --- |
| Returns the timer value |

Function Prototype

unsigned long IoGetTimerVal(byte addr)*;

File Name

picio.cpp

Include

picio.h

Return Value

Returns the timer value.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the timer value (stored locally) from a ***PIC-IO*** module. Note: this data is only valid if the SEND_TIMER bit has been set in the most recently issued NmcDefineStatus () command.

Example

To return the timer value of module 1:

```
tmr_val = IoGetTimerVal(1);
```

*This function only retrieves data stored locally on the PC. To insure the data is current, NmcReadStatus should be called just prior to calling this function. Alternately, if NmcDefineStatus has been used to permanently include the relevant data item in the status packet, any command sent to a module will update the locally stored data.

| **IoInBitSVal** |
| --- |
| Returns the value of the synchronously captured input bit |

Function Prototype

BOOL IoInBitSVal(byte addr, int bitnum)*;

File Name

picio.cpp

Include

picio.h

Return Value

Value of the synchronously captured input bit.

Arguments

addr – Module address

Module address (1 – 32)

bitnum – Bit number

Bit number to be examined. Bit numbers 0-11 correspond to I/O pins 1-12. Bit numbers 12-15 are ignored.

Description

Returns the value of a synchronously captured input bit (stored locally) from a ***PIC-IO*** module. Note: this data is only valid if the SEND_SYNCH_IN bit has been set in the most recently issued NmcDefineStatus() command.

Example

To return the value of the synchronously capture input bit 2 of module 1:

```
synch_bit = IoInBitSVal(1, 2);
```

*This function only retrieves data stored locally on the PC. To insure the data is current, NmcReadStatus should be called just prior to calling this function. Alternately, if NmcDefineStatus has been used to permanently include the relevant data item in the status packet, any command sent to a module will update the locally stored data.

| **IoInBitVal** |
| --- |
| Returns the value of an input bit |

Function Prototype

BOOL IoInBitVal(byte addr, int bitnum)*;

File Name

picio.cpp

Include

picio.h

Return Value

Value of the input bit.

Arguments

addr – Module address

Module address (1 – 32)

bitnum – Bit number

Bit number to be examined. Bit numbers 0-11 correspond to I/O pins 1-12. Bit numbers 12-15 are ignored.

Description

Returns the value of an input bit (stored locally) from a ***PIC-IO*** module. Note: this data is only valid if the SEND_INPUTS bit has been set in the most recently issued NmcDefineStatus() command.

Example

To return the value of input bit 2 of module 1:

```
bit_val = IoInBitVal(1, 2);
```

*This function only retrieves data stored locally on the PC. To insure the data is current, NmcReadStatus should be called just prior to calling this function. Alternately, if NmcDefineStatus has been used to permanently include the relevant data item in the status packet, any command sent to a module will update the locally stored data.

| **IoNewMod** (Internal Library Function) |
| --- |
| Creates and initializes a new IOMOD structure |

Function Prototype

IOMOD* IoNewMod(void);

File Name

picio.cpp

Include

picio.h

Return Value

Pointer to the newly created IOMOD structure.

Arguments

None

Description

Creates, initializes, and returns a new IOMOD structure for storing ***PIC-IO*** data.

**NOTE:** For normal operation, users do not need this command.

| **IoOutBitVal** |
| --- |
| Returns the most recently set state of an output bit |

Function Prototype

BOOL IoOutBitVal(byte addr, int bitnum);

File Name

picio.cpp

Include

picio.h

Return Value

Returns the most recently set state (0 or 1) of an output bit.

Arguments

addr – Module address

Module address (1 – 32)

bitnum – Bit number

Bit number to be examined. Bit numbers 0-11 correspond to I/O pins 1-12. Bit numbers 12-15 are ignored.

Description

Returns the most recently set state of an output bit (bitnum = 1 – 11) of a ***PIC-IO*** module.

Example

To get the most recently set state of output bit 4 of module 1:

```
bit_val = IoOutBitVal(1, 4);
```

| **IoSetOutBit** |
| --- |
| Sets the value of an output bit to 1 |

Function Prototype

BOOL IoSetOutBit(byte addr, int bitnum);

File Name

picio.cpp

Include

picio.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

bitnum – Bit number

Bit number to be set. Bit numbers 0-11 correspond to I/O pins 1-12. Bit numbers 12-15 are ignored.

Description

Sets the value of a ***PIC-IO*** module output bit to 1. This has no effect if the bit is defined as an input.

Example

To set the value of output bit 4 to 1 on module 1:

```
IoSetOutBit(1, 4);
```

| **IoSetPWMVal** |
| --- |
| Set the PWM output values |

Function Prototype

BOOL IoSetPWM(byte addr, byte pwm1, byte pwm2);

File Name

picio.cpp

Include

picio.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

pwm1 – Output value on pin PWM1 (0 - 255)

Output value on pin PWM1. A value of 255 corresponds to 100% duty cycle (always HI) and 0 corresponds to a 0% duty cycle (always LO).

pwm2 – Output value on pin PWM2 (0 – 255)

Output value on pin PWM2. A value of 255 corresponds to 100% duty cycle (always HI) and 0 corresponds to a 0% duty cycle (always LO).

Description

IoSetPWMVal() immediately sets the two PWM output values for a ***PIC-IO*** module. A value of 0 will turn off the PWM output driver, a value of 255 will turn it on with a 100% duty cycle.

Example

To have module 1 set PWM1 output to 25.1% duty cycle and PWM2 output to 50.2% duty cycle:

```
IoSetPWMVal(1, 64, 128);
```

**IoSetSyncOutput**

Sets the output bit values and the PWM values to be set synchronously when the NmcSynchStart() function is called.

Function Prototype

BOOL IoSetSynchOutput(byte addr, int outbits, byte pwm1, byte pwm2);

File Name

picio.cpp

Include

picio.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

outbits – Output bit values

Bits 0-11 of outbits corresponds to I/O pins 1-12. Setting a bit in outbits will cause the corresponding pin to go HI, clearing a bit will cause the pin to go LOW. If a pin is defined as an input, the bit value will be ignored.

pwm1 – Output value on pin PWM1 (0 – 255)

Output value on pin PWM1. A value of 255 corresponds to 100% duty cycle (always HI) and 0 corresponds to a 0% duty cycle (always LOW).

pwm2 – Output value on pin PWM2 (0 – 255)

Output value on pin PWM2. A value of 255 corresponds to 100% duty cycle (always HI) and 0 corresponds to a 0% duty cycle (always LOW).

Description

IoSetSyncOutput() stores output bit values and PWM values in ***PIC-I/O*** internal registers to be set synchronously when the NmcSynchOutput() function is called.

Example

To synchronously set module 1 output pins 2, 7, and 12 to HI and the remaining output pins to LOW, and set PWM1 output to 25.1% duty cycle and PWM2 output to 50.2% duty cycle (outputs are set when NmcSynchOutput() is called):

```
IoSetSynchOutput(1, 0x842, 64, 128);
```

| **IoSetTimerMode** |
| --- |
| Sets the mode of operation for the timer/counter |

Function Prototype

BOOL IoSetTimerMode(byte addr, byte tmrmode);

File Name

picio.cpp

Include

picio.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

tmrmode – Timer Mode

---- Logical OR of the following Timer Mode bits ----
OFFMODE - disable counter/timer
COUNTERMODE - select and enable counter mode
TIMERMODE - select and enable timer mode
RESx1 - no prescaler (count every event)
RESx2 - 2:1 prescaler (every other event counted)
RESx4 - 4:1 prescaler (every 4th event counted)
RESx8 - 8:1 prescaler (every 8th event counted)

Description

IoSetTimerMode() sets the operating mode for the ***PIC-IO*** counter/timer. If in counter mode, each rising edge of I/O bit 10 (JP10, pin 5) will be counted. This bit should be set as an input to count external events. In timer mode, the counter counts the ***PIC-I/O***'s 5.0 Mhz internal clock. The prescalar applies to both the counter and the timer modes. A call to this function will both set the mode and clear the counter/timer value to zero.

Example

To enable the module 1 timer/counter to counter mode with 2:1 prescaling:

```
IoSetTimerMode(1, COUNTERMODE|RESx2);
```

| **NmcChangeBaud** |
| --- |
| Changes the network baud rate of the COM port and modules |

Function Prototype

BOOL NmcChangeBaud(byte groupaddr, unsigned int baudrate);

File Name

nmccom.cpp

Include

nmccom.h

Return Value

0 = Fail
1 = Success

Arguments

groupaddr – Group address

Group address (0x80 – 0xFF)

baudrate – Baud rate

19200, 57600, 115200, or 230400 (valid for ***PIC-SERVO SC*** only)

Description

Sends a group command to all modules on the network to change their baud rate, and also changes the COM port's baud rate. 'groupaddr' should be 0xFF, unless the group address for all modules has been changed. All modules must have the same group address (with no group leader, so that no status packet is returned) before calling this function.

**NOTE**: If an application program uses NmcChangeBaud() to change the baud rate to other than the default of 19200 (or sets a different baud rate with NmcInit() ), it is recommended that the baud rate be changed back to 19200 before the application exits. If the application resets the controllers using NmcHardReset() or NmcShutdown(), then this is not necessary. But it the application is designed to leave the controllers active upon exit, then the baud rate should be reset to 19200.

Example

To change the network baud rate to 57600 using the default group address 0xFF:

```
NmcChangeBaud(0xFF, 57600);
```

| **NmcDefineStatus** |
| --- |
| Defines what status data is returned after each command packet sent |

Function Prototype

BOOL NmcDefineStat(byte addr, byte statusitems);

File Name

nmccom.cpp

Include

nmccom.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

statusitems – Status Items to be returned (use the logical OR of the predefined bits below)

For ***PIC-I/O*** modules, use the following defined bits:

SEND_INPUTS

Send input bit values (2 bytes – the first byte will contain the input values for I/O bits 1-8, the second byte will contain the input values for I/O bits 9-12 in the lower nibble)

SEND_AD1

Send A/D 1 value (1 byte)

SEND_AD2

Send A/D 2 value (1 byte)

SEND_AD3

Send A/D 3 value (1 byte)

SEND_TIMER

Send counter/timer value (4 bytes, least significant first)

SEND_ID

Send the device type and version number (2 bytes)

SEND_SYNCH_IN

Send input bit values captured with NmcSynchInput() (2 bytes)

SEND_SYNCH_TMR

Send counter/timer value captured with NmcSynchInput() (4 bytes)

For ***PIC-SERVO*** modules, use the following defined bits:

SEND_POS

Send position data (4 bytes - signed 32 bit integer)

SEND_AD

Send A/D value of voltage on CUR_SENSE pin (1 byte, 0 - 255)

SEND_VEL

Send actual velocity in encoder counts per servo cycle - the actual velocity has no integer component (2 bytes - signed 16 bit integer)

SEND_AUX
: Send auxiliary status byte (1 byte – see *Section 4* for bit field description)

SEND_HOME
: Send home position (4 bytes - signed 32 bit integer)

SEND_ID
: Send the device type and version number (2 bytes)

SEND_PERROR
: Send servo position error (2 bytes)

SEND_NPOINTS
: Send number of path points left in path buffer (1 byte)

For ***PIC-STEP*** modules, use the following defined bits:

SEND_POS
: Send position (4 bytes)

SEND_AD
: Send A/D value (1 byte)

SEND_ST
: Send current initial timer count (2 bytes)

SEND_INBYTE
: Send inputs byte (1 byte – see *Section 4* for bit field description)

SEND_HOME
: Send home position (4 bytes)

SEND_ID
: Send the device type and version number (2 bytes)

## Description

NmcDefineStat() defines what status data will be included in the status packet returned after a command packet is sent. The status data is selected by setting the "statusitems" argument to the logical OR of the desired status items. The selected status items will then be included in the status packet, along with the status byte which is always sent. The field size and format of each type of status data returned is listed above. Bit field descriptions for the ***PIC-SERVO*** status byte and auxiliary status byte, the ***PIC-STEP*** status byte and inputs byte, and the ***PIC-I/O*** status byte are described in *Section 4*.

For efficient communications, you should just select the data which you will always need access to. For data that only needs to be read periodically, use the NmcReadStatus() function instead. You can use the NmcDefineStatus() command at any time to change what status data is returned.

Note that this function is used for ***PIC-I/O***, ***PIC-SERVO***, and ***PIC-STEP*** modules, and that the "statusitems" argument defines must match the type of module that is being addressed.

## Example

To have ***PIC-SERVO*** module 1 send back position data and position error data with each status packet:

```
NmcDefineStatus(1, SEND_POS|SEND_PERROR);
```

| **NmcGetGroupAddr** |
| --- |
| Returns module's group address |

Function Prototype

byte NmcGetGroupAddr(byte addr);

File Name

nmccom.cpp

Include

nmccom.h

Return Value

Returns the modules group address (0x80 – 0xFF)

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the group address of the specified module.

Example

To get the group address of module 1:

```
group_addr = NmcGetGroupAddr(1);
```

| **NmcGetModType** |
| --- |
| Returns the module's module type |

Function Prototype

byte NmcGetModType(byte addr)*;

File Name

nmccom.cpp

Include

nmccom.h

Return Value

0 = ***PIC-SERVO*** module
2 = ***PIC-IO*** module
3 = ***PIC-STEP*** module

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the module type of the specified module.

Example

To get the module type of module 1:

```
mod_type = NmcGetModType(1);
```

*This function only retrieves data stored locally on the PC. To insure the data is current, NmcReadStatus should be called just prior to calling this function. Alternately, if NmcDefineStatus has been used to permanently include the relevant data item in the status packet, any command sent to a module will update the locally stored data.

| **NmcGetModVer** |
| --- |
| Returns the module's firmware version |

Function Prototype

byte NmcGetModVer(byte addr)*;

File Name

nmccom.cpp

Include

nmccom.h

Return Value

Returns the firmware version of the specified module. The firmware version is simply the integer value of the returned byte.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the firmware version of the specified module.

Example

To get the firmware version of module 1:

```
mod_ver = NmcGetModVer(1);
```

*This function only retrieves data stored locally on the PC. To insure the data is current, NmcReadStatus should be called just prior to calling this function. Alternately, if NmcDefineStatus has been used to permanently include the relevant data item in the status packet, any command sent to a module will update the locally stored data.

| **NmcGetStat** |
| --- |
| Returns the module's current status byte |

Function Prototype

byte NmcGetStat(byte addr)*;

File Name

nmccom.cpp

Include

nmccom.h

Return Value

Returns the current status byte (stored locally) of a controller.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the current status byte (stored locally) of a controller. Any command to a controller will update the locally stored value of the status byte. See *Section 4* for bit field descriptions of the ***PIC-SERVO, PIC-STEP,*** and ***PIC-I/O*** status byte.

Example

To get the status byte of module 1:

```
stat_byte = NmcGetStat(1);
```

*This function simply retrieves data from the module's data structure stored locally on the PC. To insure that the data is current, any command may be sent to the module just prior to calling this command.

| **NmcGetStatItems** |
| --- |
| Returns the byte specifying which default status items are returned in a module's status data packet |

Function Prototype

byte NmcGetStatItems(byte addr);

File Name

nmccom.cpp

Include

nmccom.h

Return Value

Returns the byte specifying the default status items to be returned in the status data packet.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the byte specifying the default status items to be returned in the status data packet. The bit field format of the returned byte specifying the default status items is the same as the "statitems" parameter of the NmcDefineStatus() function.

Example

To get the byte specifying the default status items for module 1 :

```
stat_items = NmcGetStatItems(1);
```

| **NmcGroupLeader** |
| --- |
| Returns true if the specified module is a group leader |

Function Prototype

BOOL NmcGroupLeader(byte addr);

File Name

nmccom.cpp

Include

nmccom.h

Return Value

0 = Module not a group leader
1 = Module is a group leader

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns true if the specified module is a group leader, otherwise false.

Example

To determine if module is a group leader:

```
is_leader = NmcGroupLeader(1);
```

| **NmcHardReset** |
| --- |
| Reset a module to it's power up state |

Function Prototype

BOOL NmcHardReset(byte addr);

File Name

nmccom.cpp

Include

nmccom.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32) or group address (0x80 – 0xFF)

Description

NmcReset() resets a controller to it's power up state. No status will be returned. The module's NMC address will be cleared. Typically, this command is issued to all the modules on the network using the default group address of 0xFF. (For special reset features of the ***PIC-SERVO SC*** modules, please see ServoHardReset().)

**NOTE:** Newer versions of ***PIC-SERVO*** and ***PIC-STEP*** modules have a universal reset address of 0xFF. For these types of modules, a reset command sent to 0xFF will always reset the modules, even if the their group address has been set to a different value.

Example

To reset all controllers on the network:

```
NmcHardReset(0xFF);
```

| **NmcInit** |
| --- |
| Opens the COM port and initializes the NMC network |

Function Prototype

int NmcInit(char *portname, unsigned int baudrate);

File Name

nmccom.cpp

Include

nmccom.h

Return Value

Returns the number of controllers found on the network.

Arguments

portname – Port name

Name of COM port ("COM*n*:", where *n* = 1 – 8)

baudrate – Baud rate

19200, 57600, 115200, or 230400 (valid for ***PIC-SERVO SC*** only)

Description

Opens the COM port specified by 'portname' at the specified baud rate, and initializes the network of motor controllers. Controller addresses are dynamically assigned, starting with the furthest controller with address 1. All group addresses are set to 0xFF. Returns the number of controllers found on in the network.

Example

To initialize NMC communications using COM port 1 at 115200 baud:

```
NmcInit(COM1, 115200);
```

| **NmcNoOp** |
| --- |
| No operation – module returns current status |

Function Prototype

BOOL NmcNoOp(byte addr);

File Name

nmccom.cpp

Include

nmccom.h

Return Value

0 = Fail

1 = Success

Arguments

addr – Module address

Module address (1 – 32)

Description

NmcNoOp() is used to force a module to send back it's current status data packet without taking any other action. For example, in a ***PIC-SERVO*** module, it is useful for polling the MOVE_DONE flag in the status byte to determine when a motion has finished.

Example

To force module 1 to return it's current status data:

```
NmcNoOp(1);
```

| **NmcReadStatus** |
| --- |
| Reads the specified status items once |

Function Prototype

BOOL NmcReadStatus(byte addr, byte statusitems);

File Name

nmccom.cpp

Include

nmccom.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

statusitems – Status Items to be returned (use the logical OR of the predefined bits below)

For ***PIC-I/O*** modules, use the following defined bits:

SEND_INPUTS

Send input bit values (2 bytes – the first byte will contain the input values for I/O bits 1-8, the second byte will contain the input values for I/O bits 9-12 in the lower nibble)

SEND_AD1

Send A/D 1 value (1 byte)

SEND_AD2

Send A/D 2 value (1 byte)

SEND_AD3

Send A/D 3 value (1 byte)

SEND_TIMER

Send counter/timer value (4 bytes, least significant first)

SEND_ID

Send the device type and version number (2 bytes)

SEND_SYNCH_IN

Send input bit values captured with NmcSynchInput() (2 bytes)

SEND_SYNCH_TMR

Send counter/timer value captured with NmcSynchInput() (4 bytes)

For ***PIC-SERVO*** modules, use the following defined bits:

SEND_POS

Send position data (4 bytes - signed 32 bit integer)

SEND_AD

Send A/D value of voltage on CUR_SENSE pin (1 byte, 0 - 255)

SEND_VEL

Send actual velocity in encoder counts per servo cycle - the actual velocity has no integer component (2 bytes - signed 16 bit integer)

SEND_AUX
Send auxiliary status byte (1 byte – see *Section 4* for bit field description)

SEND_HOME
Send home position (4 bytes - signed 32 bit integer)

SEND_ID
Send the device type and version number (2 bytes)

SEND_PERROR
Send servo position error (2 bytes)

SEND_NPOINTS
Send number of path points left in path buffer (1 byte)

For ***PIC-STEP*** modules, use the following defined bits:

SEND_POS
Send position (4 bytes)

SEND_AD
Send A/D value (1 byte)

SEND_ST
Send current initial timer count (2 bytes)

SEND_INBYTE
Send inputs byte (1 byte – see *Section 4* for bit field description)

SEND_HOME
Send home position (4 bytes)

SEND_ID
Send the device type and version number (2 bytes)

## Description

NmcReadStatus() is used to read specific status data from a module just one time; that is, the status packet returned in response to NmcReadStatus() will include the data specified in the "statusitems" argument, but the status packet returned with any subsequent commands will include the status data specified with the most recent NmcDefineStatus() call. For example, the NmcReadStatus() function is useful for retrieving a module's home position which needs to be read infrequently.

The status data is selected by setting the "statusitems" argument with the logical OR of the desired status items. The selected status items will be included in the status packet along with the status byte which is always sent. The field size and format of each type of status data returned is listed above. Bit field descriptions for the ***PIC-SERVO*** status byte and auxiliary status byte, the ***PIC-STEP*** status byte and inputs byte, and the ***PIC-I/O*** status byte are described in *Section 4*.

Note that this function is is used for ***PIC-I/O***, ***PIC-SERVO***, and ***PIC-STEP*** modules, and that the "statusitems" argument defines must match the type of module that is being addressed.

## Example

To have ***PIC-STEP*** module 1 send back it's home position and A/D value:

```
NmcReadStatus(1, SEND_AD|SEND_HOMEPOS);
```

| **NmcSendCmd** (Internal Library Function) |
| --- |
| Low level routine for sending a command packet |

Function Prototype

byte NmcSendCmd(byte addr, byte cmd, char *datastr, byte n, byte stataddr);

File Name

nmccom.cpp

Include

nmccom.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32) or group address (0x80 – 0xFF)

cmd – Command type

-- commands for all modules --

SET_ADDR - set module address and group address
DEF_STAT - define status to be returned
READ_STAT - read specified status once
SET_BAUD - set baud rate
NOP - no operation but return defined status
HARD_RESET – reset module to its power up state

-- ***PIC-IO*** commands --

SET_IO_DIR – set I/O bit directions
SET_PWM – set PWM output values
SYNCH_OUT – output previously stored PWM and output bytes
SET_OUTPUT – set output bit values
SET_SYNCH_OUT – store output bit and PWM values for later output
SET_TMR_MODE – set timer/counter mode
SYNCH_INPUT – store the input bytes and timer value

-- ***PIC-Servo*** commands --

RESET_POS – reset position counter to 0
LOAD_TRAJ – load motion trajectory parameters
START_MOVE – start pre-loaded motion trajectory
SET_GAIN – set PID servo filter operating parameters
STOP_MOTOR – stop motor in one of four modes
IO_CTRL – set various I/O control options
SET_HOMING – set homing mode parameters
CLEAR_BITS – clear the latched status bits
SAVE_AS_HOME – saves current position in the home position register
ADD_PATHPOINT – add path points for path mode

-- ***PIC-Step*** commands --

RESET_POS – reset position counter to 0
LOAD_TRAJ – set motion parameters

START_MOVE – start pre-loaded trajectory
SET_PARAM – set control parameters
STOP_MOTOR – stop motor and enable/disable amplifier
SET_OUTPUTS – clear or set output pins
SET_HOMING – set homing parameters
SAVE_AS_HOME – saves current position in the home position register

datastr – Additional data

String containing additional command data

n – Number additional bytes

Number of additional bytes required for the additional command data

stataddr – Status data address

If command is sent to a group address, this is the individual address of the group leader (use 0 if no group leader). Otherwise, use the same value for addr and stataddr.

## Description

NmcSendCmd() is a low-level function for formatting and sending a command packet. It formats a NMC command packet from the argument data, then sends the command packet to the NMC network. If the status data address is set to a value other than 0, NmcSendCmd() will wait for and store the status packet returned by the module.

The module address argument (addr) should contain the address where the command packet should be sent. For example, it could contain the module address, the module group address, or the universal reset address 0xFF if a reset command is to be sent to all modules. The command type (cmd) contains one of the values listed above – note that some commands are shared by all module types, such as SETBAUD, while other commands are specific to a particular module type. The number of additional bytes (n) contains the number of additional data bytes that are to sent with the command. Additional data (datastr) contains the additional data bytes that store the command parameters – this is command data such as the baud rate for the SETBAUD command. Status data address (stataddr) contains the individual address where the defined status packet should be stored after the command is executed. If a command is sent to a group address with no group leader, stataddr should be set to 0.

For a complete description of the ***PIC-IO***, ***PIC-SERVO***, and ***PIC-STEP*** command set and usage, see the corresponding chip data sheets.

**NOTE:** For normal operation, users do not need this command. Use one of the high level command functions from the NMCLIB04.DLL library.

## Example

To have ***PIC-SERVO*** module 1 send back position data and position error data with each status packet, send the following DEFINESTAT command with additional data "datastr":

```
char datastr = SERVO_SEND_POS|SERVO_SEND_POSERROR;
NmcSendCmd(1, DEFSTAT, &datastr, 1, 1);
```

| **NmcSetGroupAddress** |
| --- |
| Sets a module's group address |

Function Prototype

BOOL NmcSetGroupAddress(byte addr, byte groupaddr, BOOL leader);

File Name

nmccom.cpp

Include

nmccom.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

groupaddr – Group address

Module's group address (0x80 – 0xFF)

leader – Group leader

Module's group leader status (1 = group leader, 0 = group member)

Description

NmcSetGroupAddress() is used to change the group address of a module from the default of 0xFF, and to set the module as either a group member or a group leader. Group addresses for modules are restricted to the range of 0x80 to 0xFF. Normally set just once during initialization, the group address can be reset to different values at any time. For any group address, there can be any number of members, but only one group leader. Note that you should set all group address back to the default of 0xFF (no group leaders) before the application terminates.

Example

To set module 1 to have a group address of 0x80 and to be the group leader, and set module 2 to have the same group address, but not be the group leader:

```
NmcSetGroupAddress(1, 0x80, 1);
NmcSetGroupAddress(2, 0x80, 0);
```

| **NmcShutdown** |
| --- |
| Resets modules then closes COM port |

Function Prototype

void NmcShutdown(void);

File Name

nmccom.cpp

Include

nmccom.h

Return Value

None

Arguments

None

Description

NmcShutdown() resets controllers assuming a default group address of 0xFF, then closes the COM port in use.

Example

To reset all modules with group address 0xFF and to close the currently used COM port:

```
NmcShutdown();
```

| **NmcSyncInput** |
| --- |
| Causes a group of modules to synchronously capture input values |

Function Prototype

BOOL NmcSynchInput(byte groupaddr, byte leaderaddr);

File Name

nmccom.cpp

Include

nmccom.h

Return Value

0 = Fail
1 = Success

Arguments

groupaddr – Module address

Module address (1 – 32) or group address (0x80 – 0xFF)

leaderaddr – Group leader address

If groupaddr is individual use: leaderaddr = addr
If groupaddr is group use: leaderaddr = group leader's individual address
(If no group leader use: leaderaddr = 0)

Description

NmcSyncInput() is used to synchronously save module data. The command is common to all NMC compatible modules, but the exact data saved is different for each type of module.

For ***PIC-SERVO*** and ***PIC-STEP*** modules, NmcSyncInput() is used to synchronously save the current position of the motor in the home position register. When issued to a group of modules, NmcSyncInput() will cause them to store their current positions in their corresponding home position registers. The home position register data can then be retrieved individually using the NmcReadStatus() function for each module. This allows the host to take a snapshot of the configuration of a multi-axis system.

For ***PIC-IO*** modules, NmcSyncInput() causes the current input bit values and the counter/timer value to be synchronously stored in the ***PIC-I/O***'s internal registers. These values can be retrieved using the NmcReadStatus() function.

This command may be sent to either an individual module or to a group of modules by setting module address argument (groupaddr) to the appropriate individual or group address. The group leader address argument (leaderaddr) should contain the address of the module that will send the status packet response. If this command is sent to a group address, then the group leader address argument should be set to the group leader, or to 0 if there is no group leader. If this command is sent to an individual address, then the group leader address argument should be set to the individual address.

Note that by using the group address this command can be used to simultaneously save module data on modules of different types. For example, when issuing this command to a group containing ***PIC-STEP***, ***PIC-SERVO*** and ***PIC-IO*** modules, ***PIC-IO*** input bit and time/counter values can be stored synchronously with ***PIC-STEP*** and ***PIC-SERVO*** motor position values.

## Example

To synchronously save the current position in the home position registers of the ***PIC-SERVO*** modules with group address 0x80 and group leader address 0x01:

```
NmcSyncInput(0x80, 0x01);
```

| **NmcSyncOutput** |
| --- |
| Causes a group of modules to synchronously execute a previously stored output command |

Function Prototype

BOOL NmcSynchOutput(byte groupaddr, byte leaderaddr);

File Name

nmccom.cpp

Include

nmccom.h

Return Value

0 = Fail
1 = Success

Arguments

groupaddr – Module address

Module address (1 – 32) or group address (0x80 – 0xFF)

leaderaddr – Group leader address

If groupaddr is individual use: leaderaddr = groupaddr
If groupaddr is group use: leaderaddr = group leader address
(If no group leader use: leaderaddr = 0)

Description

The NmcSyncOutput() function is used to synchronously execute a command using preloaded command data. The command is common to all NMC compatible modules, but the exact command executed is different for each type of module.

For ***PIC-SERVO*** and ***PIC-STEP*** modules, the NmcSyncOutput() function starts a motion. It is intended to be sent to a group of modules which have been preloaded (using the ServoLoadTraj() or StepLoadTraj() commands) with motion profile data. When sent to the group address for these controllers, it will cause all controllers to start their motions simultaneously. Note that the data loaded into each controller with the Load Trajectory command merely sits unused in a buffer until the NmcSyncOutput() command is received. If another Load Trajectory command is received before the NmcSyncOutput() command, it will overwrite all of the previous trajectory data.

For ***PIC-I/O*** modules, the NmcSyncOutput() function synchronously sets the output bit values and PWM values previously stored with the SetSynchOutput() command.

This command may be sent to either an individual module or a group of modules by setting module address argument (addr) to the appropriate individual or group address. The group leader address argument (leaderaddr) should contain the individual address of the group leader. If there is no group leader, leaderaddr should be set to 0. If this command is sent to an individual address, then the group leader address argument should be set to the individual address.

Note that by using the group address this command can be used to synchronize modules of different types. For example, when issuing this command to a group containing ***PIC-STEP***, ***PIC-SERVO*** and ***PIC-I/O*** modules, ***PIC-I/O*** output bit and PWM values can be set synchronously with starting ***PIC-STEP*** and ***PIC-SERVO*** motions.

## Example

To synchronously start the preloaded motions on the ***PIC-SERVO*** modules with group address 0x80 and group leader address 0x01:

```
NmcSyncOutput(0x80, 0x01);
```

| **ServoAddPathPoints** |
| --- |
| Adds a set of path points for path mode operation |

Function Prototype

BOOL ServoAddPathpoints(byte addr, int npoints, long *path, int freq);

File Name

picservo.cpp

Include

picservo.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

npoints – Number of points

Number of points in the list (1 – 7).

path – Path points list

Path points list formatted as absolute position data.

freq – Path point frequency

P_30HZ, P_60HZ, or P_120HZ

Description

Adds between 1 and 7 path points to the ***PIC-SERVO's*** internal path point buffer. 'npoints' is the number of points to be added, and the *absolute* path point positions should be loaded into the array 'path'. The variable 'freq' should be set to one of the constant values P_30HZ, P_60HZ or P_120HZ. This function will convert the absolute path data into incremental path data (as required by the ***PIC-SERVO*** chip) and perform the proper bit formatting prior to downloading.

When starting a new path mode motion, ServoInitPath() must be called first to initialize the starting point of the path to the current motor position before adding any points with ServoAddPathPoints(). After path points have been added with ServoAddPathPoints(), motion along the stored path is started by calling ServoStartPathMode(). Once path mode operation is started, calls to ServoLoadTraj() are not permitted until the path has completed or the motor is stopped with a ServoStopMotor() command.

The path points added to the path point buffer should be closely spaced and form a smooth path for the motor to follow. Note that you can get the exact initial command position of the motor by reading the motor position and the position error with the same NmcReadStatus() command and then adding them together. You will then specify your path points starting from there.

The number of path points currently residing in the buffer can be read using the

NmcReadStatus() command. The buffer on the ***PIC-SERVO SC**** can hold a maximum of 128 path points. After the path mode motion has started and as the path point buffer is depleted, you can dynamically add additional path points to the buffers as they empty (using ServoAddPathPoints() ) to create motions of any length.

Note: When using ***PIC-SERVO CMC*** (version 5), the advanced features bit of the ServoStopMotor() command must first be enabled before using this function.

## Example

For module address 1 which is currently at position -200, add four path points -100, 0, 100, 200 at 60 Hz:

```
long plist[8] = { -100, 0, 100, 200 };
ServoAddPathPoints(1, 4, plist, P_60HZ);
```

*The ***PIC-SERVO CMC*** path point buffer can only hold 96 points. Therefore, you will want to limit the number of points you add to 96 if using a mix of ***PIC-SERVO SC*** and ***PIC-SERVO CMC*** controllers.

| **ServoClearBits** |
| --- |
| Clears the latched status bits |

Function Prototype

BOOL ServoClearBits(byte addr);

File Name

picservo.cpp

Include

picservo.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

Description

The OVERCURRENT and POS_ERROR bits in the status byte and the POS_WRAP and SERVO_OVERRUN in the auxiliary status byte are latched flags which remain set until explicitly cleared with the ServoClearBits() command. All of these latched bits are cleared with a single command.

Example

To clear the latched status bits on module 1:

```
ServoClearBits(1);
```

| **ServoGetAD** |
| --- |
| Returns the current A/D value |

Function Prototype

byte ServoGetAD(byte addr)*;

File Name

picservo.cpp

Include

picservo.h

Return Value

Returns the current A/D value.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the current A/D value (stored locally) of a ***PIC-SERVO*** module. Use `NmcReadStatus()` or `NmcDefineStatus()` to update the locally stored data.

Example

To get the A/D value from module 1:

```
ad_val = ServoGetAD(1);
```

*This function only retrieves data stored locally on the PC. To insure the data is current, NmcReadStatus should be called just prior to calling this function. Alternately, if NmcDefineStatus has been used to permanently include the relevant data item in the status packet, any command sent to a module will update the locally stored data.

| **ServoGetAux** |
| --- |
| Returns the current auxiliary status byte |

Function Prototype

byte ServoGetAux(byte addr)*;

File Name

picservo.cpp

Include

picservo.h

Return Value

Returns the current auxiliary status byte.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the current auxiliary status byte (stored locally) of a ***PIC-SERVO*** module. Use `NmcReadStatus()` or `NmcDefineStatus()` to update the locally stored data.

Example

To get the auxiliary status byte from module 1:

```
aux_stat = ServoGetAux(1);
```

*This function only retrieves data stored locally on the PC. To insure the data is current, NmcReadStatus should be called just prior to calling this function. Alternately, if NmcDefineStatus has been used to permanently include the relevant data item in the status packet, any command sent to a module will update the locally stored data.

| **ServoGetCmdAcc** |
| --- |
| Returns the most recently issued command acceleration |

Function Prototype

long ServoGetCmdAcc(byte addr);

File Name

picservo.cpp

Include

picservo.h

Return Value

Returns the most recently issued command acceleration.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the most recently issued command acceleration for a ***PIC-SERVO*** module.

Example

To get the most recently issued command acceleration from module 1:

```
cmd_accel = ServoGetCmdAcc(1);
```

| **ServoGetCmdPos** |
| --- |
| Returns the most recently issued command position |

Function Prototype

long ServoGetCmdPos(byte addr);

File Name

picservo.cpp

Include

picservo.h

Return Value

Returns the most recently issued command position.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the most recently issued command position for a ***PIC-SERVO*** module.

Example

To get the most recently issued command position from module 1:

```
cmd_pos = ServoGetCmdPos(1);
```

| **ServoGetCmdPwm** |
| --- |
| Returns the most recently issued command PWM value |

Function Prototype

byte ServoGetCmdPwm(byte addr);

File Name

picservo.cpp

Include

picservo.h

Return Value

Returns the most recently issued command PWM.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the most recently issued command PWM value for a ***PIC-SERVO*** module.

Example

To get the most recently issued command PWM from module 1:

```
cmd_pwm = ServoGetCmdPwm(1);
```

| **ServoGetCmdVel** |
| --- |
| Returns the most recently issued command velocity |

Function Prototype

long ServoGetCmdVel(byte addr);

File Name

picservo.cpp

Include

picservo.h

Return Value

Returns the most recently issued command velocity.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the most recently issued command velocity for a ***PIC-SERVO*** module.

Example

To get the most recently issued command velocity from module 1:

```
cmd_vel = ServoGetCmdVel(1);
```

| **ServoGetGain** |
| --- |
| Returns the most recently issued servo gain values |

Function Prototype

void ServoGetGain(byte addr, short int *kp, short int *kd, short int * ki, short int * il, byte *ol, byte *cl, short int * el, byte * sr, byte *dc);

File Name

picservo.cpp

Include

picservo.h

Return Value

None

Arguments

addr – Module address

Module address (1 – 32)

kp - Position gain Kp

kd - Derivative gain Kd

ki - Integral gain Ki

il - Integration limit IL

ol - Output limit OL

cl - Current limit CL

el - Position error limit EL

sr - Servo rate divisor SR

dc - Amplifier deadband compensation DB

Description

Returns the most recently issued servo gain values for a ***PIC-SERVO*** module. Use ServoGetGain2() for ***PIC-SERVO SC*** servo modules.

Example

To get the most recently issued servo gain values for module 1:

```
short int kp, kd, ki, il, el;
byte ol, cl,  sr, dc;
ServoGetGain(1,&kp,&kd,&ki,&il,&ol,&cl,&el,&sr,&dc);
```

| **ServoGetGain2** |
| --- |
| Returns the most recently issued servo gain values (for use with ***PIC-SERVO SC)*** |

Function Prototype

void ServoGetGain2(byte addr, short int *kp, short int *kd, short int * ki,
short int * il, byte *ol, byte *cl, short int * el, byte * sr, byte *dc, byte *sm);

File Name

picservo.cpp

Include

picservo.h

Return Value

None

Arguments

addr – Module address
Module address (1 – 32)
kp - Position gain Kp
kd - Derivative gain Kd
ki - Integral gain Ki
il - Integration limit IL
ol - Output limit OL
cl - Current limit CL
el - Position error limit EL
sr - Servo rate divisor SR
dc - Amplifier deadband compensation DB
sm - Step rate multiplier SM

Description

Returns the most recently issued servo gain values for a ***PIC-SERVO SC*** module. For non-***PIC-SERVO SC*** modules, use ServoGetGain().

Example

To get the most recently issued servo gain values for ***PIC-SERVO SC*** module 1:

```
short int kp, kd, ki, il, el;
byte ol, cl,  sr, dc, sm;
ServoGetGain2(1, &kp, &kd, &ki, &il, &ol,
              &cl, &el, &sr, &dc, &sm);
```

| **ServoGetHome** |
| --- |
| Returns the current motor home position |

Function Prototype

long ServoGetHome(byte addr)*;

File Name

picservo.cpp

Include

picservo.h

Return Value

Returns the current motor home position.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the current motor home position (stored locally) of a ***PIC-SERVO*** module. Use `NmcReadStatus()` or `NmcDefineStatus()` to update the locally stored data.

Example

To get the current home position from module 1:

```
home_pos = ServoGetHome(1);
```

*This function only retrieves data stored locally on the PC. To insure the data is current, NmcReadStatus should be called just prior to calling this function. Alternately, if NmcDefineStatus has been used to permanently include the relevant data item in the status packet, any command sent to a module will update the locally stored data.

| **ServoGetHomeCtrl** |
| --- |
| Returns the most recently issued home command control byte |

Function Prototype

byte ServoGetHomeCtrl(byte addr);

File Name

picservo.cpp

Include

picservo.h

Return Value

Returns the most recently issued home command control byte.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the most recently issued home command control byte for a ***PIC-SERVO*** module.

Example

To get the most recently issued home command control byte for module 1:

```
home_ctrl = ServoGetHomeCtrl(1);
```

| **ServoGetIoCtrl** |
| --- |
| Returns the most recently issued I/O command control byte |

Function Prototype

byte ServoGetIoCtrl(byte addr);

File Name

picservo.cpp

Include

picservo.h

Return Value

Returns the most recently issued I/O command control byte.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the most recently issued I/O command control byte for a ***PIC-SERVO*** module.

Example

To get the most recently issued I/O command control byte for module 1:

```
io_ctrl = ServoGetIoCtrl(1);
```

| **ServoGetMoveCtrl** |
| --- |
| Returns the most recently issued move command control byte |

Function Prototype

byte ServoGetMoveCtrl(byte addr);

File Name

picservo.cpp

Include

picservo.h

Return Value

Returns the most recently issued move command control byte.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the most recently issued move command control byte for a ***PIC-SERVO*** module.

Example

To get the most recently issued move command control byte for module 1:

```
move_ctrl = ServoGetMoveCtrl(1);
```

| **ServoGetNPoints** |
| --- |
| Returns the number of path points remaining |

Function Prototype

byte ServoGetNPoints(byte addr)*;

File Name

picservo.cpp

Include

picservo.h

Return Value

Returns the current number of path points remaining.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the number of path points (stored locally) remaining in the ***PIC-SERVO*** (***CMC*** or ***SC***) path-point buffer. Use `NmcReadStatus()` or `NmcDefineStatus()` to update the locally stored data.

Example

To get the current number of path points left from module 1:

```
npoints = ServoGetNPoints(1);
```

*This function only retrieves data stored locally on the PC. To insure the data is current, NmcReadStatus should be called just prior to calling this function. Alternately, if NmcDefineStatus has been used to permanently include the relevant data item in the status packet, any command sent to a module will update the locally stored data.

| **ServoGetPError** |
| --- |
| Returns the servo positioning error |

Function Prototype

short int ServoGetPError(byte addr)*;

File Name

picservo.cpp

Include

picservo.h

Return Value

Returns the servo positioning error.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the servo positioning error (stored locally) of a ***PIC-SERVO*** (***CMC*** or ***SC***) module. Use `NmcReadStatus()` or `NmcDefineStatus()` to update the locally stored data.

Example

To get the servo positioning error from module 1:

```
pos_error = ServoGetPError(1);
```

*This function only retrieves data stored locally on the PC. To insure the data is current, NmcReadStatus should be called just prior to calling this function. Alternately, if NmcDefineStatus has been used to permanently include the relevant data item in the status packet, any command sent to a module will update the locally stored data.

| **ServoGetPos** |
| --- |
| Returns the current motor position |

Function Prototype

long ServoGetPos(byte addr)*;

File Name

picservo.cpp

Include

picservo.h

Return Value

Returns the current motor position.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the current motor position (stored locally) of a ***PIC-SERVO*** module. Use `NmcReadStatus()` or `NmcDefineStatus()` to update the locally stored data.

Example

To get the current motor position from module 1:

```
pos = ServoGetPos(1);
```

*This function only retrieves data stored locally on the PC. To insure the data is current, NmcReadStatus should be called just prior to calling this function. Alternately, if NmcDefineStatus has been used to permanently include the relevant data item in the status packet, any command sent to a module will update the locally stored data.

| **ServoGetStat** (Internal Library Function) |
| --- |
| Low-level routine to process and store returned ***PIC-SERVO*** status data |

Function Prototype

BOOL ServoGetStat(byte addr);

File Name

picservo.cpp

Include

picservo.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

Description

ServoGetStat() is a low-level routine to process and store returned ***PIC-SERVO*** status data. It first verifies the number of status bytes received and the check sum of the status bytes, then stores the received status data internally.

**NOTE:** For normal operation, users do not need this command.

| **ServoGetStopCtrl** |
| --- |
| Returns the most recently issued stop command control byte |

Function Prototype

byte ServoGetStopCtrl(byte addr);

File Name

picservo.cpp

Include

picservo.h

Return Value

Returns the most recently issued stop command control byte.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the most recently issued stop command control byte for a ***PIC-SERVO*** module.

Example

To get the most recently issued stop command control byte for module 1:

```
stop_ctrl = ServoGetStopCtrl(1);
```

| **ServoGetStopPos** |
| --- |
| Returns the most recently issued stop position |

Function Prototype

long ServoGetStopPos(byte addr);

File Name

picservo.cpp

Include

picservo.h

Return Value

Returns the most recently issued stop position.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the most recently issued stop position ( by a 'stop here' command) for a ***PIC-SERVO*** module.

Example

To get the most recently issued stop position for module 1:

```
stop_pos = ServoGetStopPos(1);
```

| **ServoGetVel** |
| --- |
| Returns the current motor velocity |

Function Prototype

short int ServoGetVel(byte addr)*;

File Name

picservo.cpp

Include

picservo.h

Return Value

Returns the current motor velocity.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the current motor velocity (stored locally) of a ***PIC-SERVO*** module. Use `NmcReadStatus()` or `NmcDefineStatus()` to update the locally stored data.

Example

To get the current motor velocity from module 1:

```
vel = ServoGetVel(1);
```

*This function only retrieves data stored locally on the PC. To insure the data is current, NmcReadStatus should be called just prior to calling this function. Alternately, if NmcDefineStatus has been used to permanently include the relevant data item in the status packet, any command sent to a module will update the locally stored data.

**ServoHardReset** (Only valid for ***PIC-SERVO SC*** – v.10 and greater)

Reset the controller to its power-up state and optionally store configuration data in EEPROM

Function Prototype

BOOL ServoHardReset(byte addr, byte mode);

File Name

picservo.cpp

Include

picservo.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

mode – Reset mode control byte

---- Logical OR of the following reset control bits ----
SAVE_DATA - save config. data in EPROM
RESTORE_ADDR - restore address on power-up
EPU_AMP - enable amplifier on power-up
EPU_SERVO - enable servo on power up
EPU_STEP - enable step & direction mode on power up
EPU_LIMITS - enable limit switch protection on power up
EPU_3PH - enable 3-phase commutation on power up
EPU_ANTIPHASE - enable antiphase PWM on power up

Description

ServoHardReset() resets the ***PIC-SERVO SC*** to its power-up state, but it does not restore any configuration data stored in EEPROM, except for the Antiphase or 3-Phase options. Only an actual power-cycle or reset via the MCLR pin will cause the rest of the EEPROM data to be restored. The ServoHardReset() command performs only the configuration reset described below. For a simple reset (no data written to or erased from the EEPROM), use the NmcHardReset() command.

Configuration Reset – In a configuration reset, data will be written to or erased from the EEPROM. If the SAVE_DATA bit of the reset mode control byte is set, the reset mode control byte itself will be saved in EEPROM, along with the individual and group addresses, the current velocity and acceleration values, and all of the parameters set with the Set Gain command. If the SAVE_DATA bit of the reset mode byte is cleared, a value of 0 will be stored in the EEPROM for the reset mode control byte, and all other EEPROM data will be erased. Normally, a configuration reset will be sent to an individual controller.

On a hardware reset (power-up or reset via MCLR), the ***PIC-SERVO SC*** will read the reset mode control byte and restore the saved data if the SAVE_DATA bit is set. It will also look at bits 1 - 7 of the control byte to see what other operating options should be restored.

Note that if bit RESTORE_ADDR of the control byte is not set, the individual and group addresses will not be restored, and the address of the module will have to be initialized by NmcInit(). (Note: the servo and amplifier will not be enabled until the address is initialized.) This is useful for when you want to save operating parameters, but are using the ***PIC-SERVO SC*** with other NMC modules which do not have the EEPROM configuration feature. If bit RESTORE_ADDR is set, the addresses will be restored and the ADDR_OUT pin will be lowered immediately on powerup.

If you set the EPU_LIMITS bit of the reset mode control byte, the currently selected option for limit switch protection will be saved in EEPROM and then restored on power-up.

**NOTE:** In general the configuration data should not be saved in EEPROM, and the modules should be completely configured by the host on start-up. This will reduce problems associated with keeping track of the state of each module. Examples of systems where configuration data should be saved in EEPROM are:

- If the module is running stand-alone in Step-and-Direction mode, then configuration should be saved to EEPROM.
- If the amplifier uses 3-phase or Antiphase mode, then the EPU_3PH or EPU_ANTIPHASE bits should be saved in EEPROM but no other bit should be set.

## Example

To save operating parameters for module 1 and have it power-up ready to receive Step & Direction signals and in 3-Phase commutation output mode:

```
ServoHardReset(0x01, SAVE_DATA|RESTORE_ADDR|
                      EPU_AMP|EPU_SERVO|
                      EPU_STEP|EPU_3PH);
```

To erase the EEPROM configuration data for module address 1:

```
ServoHardReset(0x01, 0);
```

| **ServoInitPath** |
| --- |
| Initializes the starting point of a path to the current motor position |

Function Prototype

void ServoInitPath(byte addr);

File Name

picservo.cpp

Include

picservo.h

Return Value

None

Arguments

addr – Module address

Module address (1 – 32)

Description

Initializes the starting point of a new path motion to the current motor command position. This function should be called just prior to adding path points to new path mode motion using ServoAddPathPoints(), and should not be called again until path point operation has exited (either after the complete path point motion has been executed, or after the path point motion is stopped with ServoStopMotor()).

When using ***PIC-SERVO CMC*** version 5, the advanced features bit of the ServoStopMotor() command must first be enabled before using this function.

Example

To initialize the starting point of a path to the current motor position for module 1:

```
ServoInitPath(1);
```

| **ServoLoadTraj** |
| --- |
| Loads motion trajectory information |

Function Prototype

BOOL ServoLoadTraj(byte addr, byte mode, long pos, long vel,
long acc, byte pwm);

File Name

picservo.cpp

Include

picservo.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address
Module address (1 – 32)
mode – Trajectory mode
---- Logical OR of the following load trajectory mode bits ----
LOAD_POS – load position data
LOAD_VEL – load velocity data
LOAD_ACC – load acceleration data
LOAD_PWM – load PWM data
ENABLE_SERVO – enable PID servo
VEL_MODE – enable velocity profile
REVERSE – reverse (use to specify reverse PWM or Velocity)
MOVE_REL – move relative
START_NOW – start Now
pos – Position data
Position data if LOAD_POS bit of Trajectory Mode is set
(signed 32 bit integer: -2,147,483,648 to +2,147,483,647)
vel – Velocity data
Velocity data if LOAD_VEL bit of Trajectory Mode is set
(positive 32 bit integer: 0 to +83,886,080)
acc – Acceleration data
Acceleration data if LOAD_ACC bit of Trajectory Mode is set
(positive 32 bit integer: 0 - +2,147,483,647)
pwm – PWM data
PWM data if LOAD_PWM bit of Trajectory Mode is set
(positive 8 bit integer: 0 to +255)

Description

ServoLoadTraj() is used to send motion trajectory and PWM information to a ***PIC-SERVO*** module. It is flexible in that it lets you send only the data needed for a particular motion. For example, suppose you have already loaded acceleration and velocity parameters, and

you only need to send commands with updated position data. In this case, you would set the LOAD_POS bit of the Trajectory Mode control byte (along with other trajectory mode bits as needed), and then only the position data bytes would be sent to the controller. If any of the bits LOAD_POS, LOAD_VEL or LOAD_ACC are not set, then the corresponding pos, vel or acc data will be ignored.

The position, velocity and acceleration parameters are programmed as 32 bit quantities in units of encoder counts and servo ticks. The lower 16 bits of the velocity and acceleration parameters represent a fractional component. Please refer to the ***PIC-SERVO SC*** *Chip Data Sheet* section *4.4.7 Specifying Positions, Velocities and Accelerations* for more detail on how to specify these parameters.

The ENABLE_SERVO, VEL_MODE, and REVERSE Trajectory Mode control bits govern the mode of operation. The ENABLE_SERVO bit should be set to enable the PID servo - this is the normal mode of operation. If the ENABLE_SERVO bit is not set, you will default to PWM mode operation and the PWM value provided will be used. (If no PWM value is sent, the current output value will be used.)

The VEL_MODE bit is used to select velocity profile mode (VEL_MODE bit is set) or trapezoidal profile mode (VEL_MODE bit is cleared).

Velocity, acceleration and PWM parameters should all be positive. If you need to specify a reverse velocity or PWM value, you should set the REVERSE bit of the Trajectory Mode control byte.

In trapezoidal position mode, however, the position data may be positive or negative, and a reverse direction bit *does not* control the direction. In trapezoidal position mode, the this bit is instead interpreted as a MOVE_REL bit for specifying relative motions. If this bit is set, the position data will be interpreted as being relative rather than absolute.

Lastly, the START_NOW bit of the Trajectory Mode control byte is used to specify if you want the command data to take effect immediately, or if you want to wait for a NmcSyncOutput() command to be sent. For individual axis control, it is usually easiest to set the START_NOW bit and eliminate the need for a separate NmcSyncOutput() command. However, if you need to start several controllers moving at exactly the same time, you can send them individual Load Trajectory commands without the START_NOW bit set. This will cause the data to simply sit in a temporary buffer. You can then issue a NmcSyncOutput() command to the group address containing several controllers, thereby starting all motions at the same time.

You should note that if the START_NOW bit is not set, the motion parameters will be ignored until a NmcSyncOutput() command is issued. If you send a new Load Trajectory before sending a NmcSyncOutput() command, the temporary buffer will be over-written, erasing your previous command data.

There are three status bits associated with velocity and trapezoidal profiling: MOVE_DONE (in the status byte), SLEW and ACCEL (in the auxiliary status byte). See Section 4.1 for descriptions of these status bits.

Example

Move module 1 to an absolute position of -1500, velocity of 100,000, acceleration of 100 in trapezoidal profile mode, starting now:

```
ServoLoadTraj(1, LOAD_POS|LOAD_VEL|LOAD_ACC|
                 ENABLE_SERVO|START_NOW,
                 -1500, 100000, 100, 0);
```

Move module address 1 with a velocity of -100,000 in velocity profile mode starting now (assume the acceleration parameter has already been loaded):

```
ServoLoadTraj(1, LOAD_VEL|ENABLE_SERVO|VEL_MODE,
                 REVERSE|START_NOW,
                 0, 100000, 0, 0);
```

Load velocity (100,000) and acceleration (100) data for subsequent motions, and leave the ***PIC-SERVO*** in PWM mode with an PWM value of 0:

```
ServoLoadTraj(1, LOAD_VEL|LOAD_ACC|LOAD_PWM|START_NOW,
                  0, 100000, 100, 0);
```

| **ServoNewMod** (Internal Library Function) |
| --- |
| Creates and initializes a new SERVOMOD structure |

Function Prototype

SERVOMOD *ServoNewMod(void);

File Name

picservo.cpp

Include

picservo.h

Return Value

Pointer to the new SERVOMOD structure.

Arguments

None

Description

Creates, initializes, and returns a new SERVOMOD structure for storing ***PIC-SERVO*** data.

**NOTE:** For normal operation, users do not need this command.

| **ServoResetPos** |
| --- |
| Resets the position counter to zero |

Function Prototype

BOOL ServoResetPos(byte addr);

File Name

picservo.cpp

Include

picservo.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

Description

ServoResetPos() resets the position counter to a value of zero. The current command position used by the PID position servo will also be set to zero to prevent the motor from jumping. This function should not be used while the motor is moving.

Example

To reset the position counter to zero for ***PIC-SERVO*** module 1:

```
ServoResetPos(1);
```

| **ServoResetRelHome** |
| --- |
| Resets the position relative to the home position register |

Function Prototype

BOOL ServoResetRelPos(byte addr);

File Name

picservo.cpp

Include

picservo.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

Description

Resets the position for a ***PIC-SERVO*** (***CMC*** or ***SC***) module relative to the home position register (*i.e.*, the home position is now the zero position and all absolute motor positions will be commanded and reported relative to the home position).

Example

To reset the position counter relative to the home position for ***PIC-SERVO*** module 1:

```
ServoResetRelHome(1);
```

| **ServoSetGain** |
| --- |
| Sets the servo gains |

Function Prototype

BOOL ServoSetGain(byte addr, short int kp, short int kd, short int ki, short int il, byte ol, byte cl, short int el, byte sr, byte dc);

File Name

picservo.cpp

Include

picservo.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address
Module address (1 – 32)
kp - Position gain Kp
positive 16 bit integer: 0 - +32,767
kd - Derivative gain Kd
positive 16 bit integer: 0 - +32,767
ki - Integral gain Ki
positive 16 bit integer: 0 - +32,767
il - Integration limit IL
positive 16 bit integer: 0 - +32,767
ol - Output limit OL
unsigned 8 bit integer: 0 - 255
cl - Current limit CL
unsigned 8 bit integer: 0 - 255
odd values: CUR_SENSE proportional to motor current
even values: CUR_SENSE inversely proportional to motor current
el - Position error limit EL
positive 16 bit integer: 0 - +32,767
sr - Servo rate divisor SR
unsigned 8 bit integer: 1 - 255
dc - Amplifier deadband compensation DB
unsigned 8 bit integer: 0 - 255

Description

ServoSetGain() is used to set most of the non-motion profile related operating parameters of the ***PIC-SERVO***. The PID gain value, the integration limit, output limit, position error limit, servo rate divisor and amplifier deadband are all described in detail in Section *4.3* of the ***PIC-SERVO SC*** *Chip Data Sheet,* and the step rate multiplier is described in Section *4.4.6 Step and Direction Input Mode*.

The current limit value CL is used to set the allowable current level as described in Section *4.7* of the ***PIC-SERVO SC*** *Chip Data Sheet*. The parameter itself has a different meaning based on whether the value is odd or even (*i.e.*, bit 0 is set or cleared). If the CL value used is odd, the ***PIC-SERVO SC*** assumes that the voltage on the CUR_SENSE pin increases proportionally as the current increases. If the analog reading at the CUR_SENSE pin is greater than CL, an overcurrent condition will be triggered.

Some amplifiers, however, may produce a voltage signal inversely proportional to the motor current, or may only have a digital signal which is lowered when some current threshold is reached. (*i.e.*, the voltage on CUR_SENSE goes down as the current goes up.) Therefore, the ***PIC-SERVO SC*** will interpret an *even* values of CL such that if the analog reading at the CUR_SENSE pin is *less than* CL, and overcurrent condition will be triggered.

Note that a CL value of 0 (the minimum even value) or a CL value of 255 (the maximum odd value) will effectively disable the current limiting feature.

New applications should use ServoSetGain2() for all version of the ***PIC-SERVO***.

## Example

To set the gains of module address 1 to the following values:

Kp = 100, Kd = 1000, Ki = 50, IL = 200,
OL = 255, CL = 53 (directly proportional), EL = 4000
SR = 1, DB = 0

use the command string:

```
ServoSetGain(1, 100, 1000, 50, 200,
                255, 53, 4000, 1, 0);
```

| **ServoSetGain2** |
| --- |
| Set the servo gains (use this version for new applications) |

Function Prototype

byte ServoSetGain(byte addr, short int kp, short int kd, short int ki, short int il, byte ol, byte cl, short int el, byte sr, byte dc, byte sm);

File Name

picservo.cpp

Include

picservo.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address
Module address (1 – 32)
kp - Position gain Kp
positive 16 bit integer: 0 - +32,767
kd - Derivative gain Kd
positive 16 bit integer: 0 - +32,767
ki - Integral gain Ki
positive 16 bit integer: 0 - +32,767
il - Integration limit IL
positive 16 bit integer: 0 - +32,767
ol - Output limit OL
unsigned 8 bit integer: 0 - 255
cl - Current limit CL
unsigned 8 bit integer: 0 - 255
odd values: CUR_SENSE proportional to motor current
even values: CUR_SENSE inversely proportional to motor current
el - Position error limit EL
positive 16 bit integer: 0 - +32,767
sr - Servo rate divisor SR
unsigned 8 bit integer: 1 - 255
dc - Amplifier deadband compensation DB
unsigned 8 bit integer: 0 - 255
sm - Step rate multiplier SM
unsigned 8 bit integer: 1 – 255

Description

ServoSetGain() is used to set most of the non-motion profile related operating parameters of the ***PIC-SERVO***. The PID gain value, the integration limit, output limit, position error limit, servo rate divisor and amplifier deadband are all described in detail in Section *4.3* of the ***PIC-SERVO SC*** *Chip Data Sheet,* and the step rate multiplier is described in

Section *4.4.6 Step and Direction Input Mode*.

The current limit value CL is used to set the allowable current level as described in Section *4.7* of the ***PIC-SERVO SC*** *Chip Data Sheet*. The parameter itself has a different meaning based on whether the value is odd or even (*i.e.*, bit 0 is set or cleared). If the CL value used is odd, the ***PIC-SERVO SC*** assumes that the voltage on the CUR_SENSE pin increases proportionally as the current increases. If the analog reading at the CUR_SENSE pin is greater than CL, an overcurrent condition will be triggered.

Some amplifiers, however, may produce a voltage signal inversely proportional to the motor current, or may only have a digital signal which is lowered when some current threshold is reached. (*i.e.*, the voltage on CUR_SENSE goes down as the current goes up.) Therefore, the ***PIC-SERVO SC*** will interpret an *even* values of CL such that if the analog reading at the CUR_SENSE pin is *less than* CL, and overcurrent condition will be triggered.

Note that a CL value of 0 (the minimum even value) or a CL value of 255 (the maximum odd value) will effectively disable the current limiting feature.

New applications should use this version of the Set Gain command (rather than ServoSetGain()) for all versions of the ***PIC-SERVO***.

### Example

To set the gains of module address 1 to the following values:

Kp = 100, Kd = 1000, Ki = 50, IL = 200,
OL = 255, CL = 53 (directly proportional), EL = 4000
SR = 1, DB = 0, SM = 5

use the command string:

```
ServoSetGain(1, 100, 1000, 50, 200,
             255, 53, 4000, 1, 0, 5);
```

| **ServoSetHoming** |
| --- |
| Sets homing mode parameters for capturing the home position |

Function Prototype

BOOL ServoSetHoming(byte addr, byte mode);

File Name

picservo.cpp

Include

picservo.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

mode – Homing mode

---- Logical OR of the following load homing mode bits ----
ON_LIMIT1 - home on change in limit 1
ON_LIMIT2 - home on change in limit 2
HOME_MOTOR_OFF - turn motor off when homed
ON_INDEX - home on change in index
HOME_STOP_ABRUPT - stop abruptly when homed
HOME_STOP_SMOOTH - stop smoothly when homed
ON_POS_ERR - home on excessive position error
ON_CUR_ERR - home on overcurrent error

Description

Set Homing is used for specifying conditions for capturing the home position of a motor, and also for specifying any desired automatic stopping mode for when the home position is found.

Bits ON_LIMIT1, ON_LIMIT2, ON_INDEX, ON_POS_ERR, and ON_CUR_ERR specify the conditions for capturing the home position. The home position will be captured when *any* of the specified conditions occurs. Bits ON_LIMIT1, ON_LIMIT2, and ON_INDEX specify that the homing function look for *changes* in the states of the limit and index input pins from when the homing command is issued. It does not matter if the pin voltages start off HI or LO.

Bits ON_POS_ERR, and ON_CURR_ERR allow you to also capture home on the occurrence of a position error or on current limiting. Note that you should use the ServoClearBits() command to clear the value of these status bits before issuing the ServoSetHoming() command.

Bits HOME_MOTOR_OFF, HOME_STOP_ABRUPT, and HOME_STOP_SMOOTH are used to

specify an automatic stopping mode for the motor once the home position has been captured. Only one (or none) of these bits should be set.

When the ServoSetHoming() command is issued, the HOME_IN_PROG bit of the status byte will be set. You should then issue a motion command to move towards one of the triggers used for homing. The HOME_IN_PROG bit will then be cleared when any of the selected homing conditions occur, and the current motor position is stored in the home position register. The home position register can be read using the NmcReadStatus() command.

Once the homing process is complete, you can re-issue the ServoSetHoming() command if desired to capture a different home position. Sending a ServoSetHoming() command with the control byte equal to zero will cancel any homing in progress and clear the HOME_IN_PROG bit.

Note that ServoSetHoming() does not start any homing motion. After calling ServoSetHoming(), you should use ServoLoadTraj() to start moving the motor towards a limit or index switch.

### Example

To have module 1 capture the home position on a change of LIMIT1 or LIMIT2 and then stop abruptly:

```
ServoSetHoming(1,ON_LIMIT1|ON_LIMIT2|HOME_STOP_ABRUPT);
```

| **ServoSetIoCtrl** |
| --- |
| Controls the configuration of the LIMIT1 and LIMIT2 I/O pins, as well as other miscellaneous functions |

Function Prototype

BOOL ServoSetIoCtrl(byte addr, byte mode);

File Name

picservo.cpp

Include

picservo.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

mode – IO Control Mode

---- Logical OR of the following I/O control bits ----
SET_OUT1: 1 = set limit 1 output, 0 = clear limit 1 output
SET_OUT2: 1 = set limit 2 output, 0 = clear limit 1 output
IO1_IN: 1 = limit 1 is an input, 0 = limit 1 is an output
IO2_IN: 1 = limit 2 is an input, 0 = limit 1 is an output
LIMSTOP_OFF - turn off motor on limit
LIMSTOP_ABRUPT - stop abruptly on limit
THREE_PHASE: set for 3-phase mode
ANTIPHASE: set for antiphase mode
FAST_PATH: 0 = 30 or 60 Hz path execution, 1 = 60 or 120 Hz
STEP_MODE: 0 = normal operation, 1 = Step & Direction enabled

Description

**CAUTION: Use extreme care in setting the parameters for this command – incorrect settings could damage your amplifier or the *PIC-SERVO* chip.**

There are significant differences in the operation of ServIoCtrl() between different versions of the ***PIC-SERVO***. Different versions of ***PIC-SERVO*** support different sets of I/O control bits as shown below in Table 1.

In the ***PIC-SERVO SC,*** ServoIoCtrl() is used to set a number of operating options. The ServoIoCtrl() command should be issued prior to enabling the amplifier to make sure that the output options are set to be compatible with the amplifier type.

In earlier versions of the ***PIC-SERVO*** this command is primarily used to optionally redefine the limit switch inputs as outputs.

**Table 1 - Allowed I/O Control Bits by *PIC-SERVO* Version**

| ***PIC-SERVO V.4 and earlier*** | ***PIC-SERVO V.5*** | ***PIC-SERVO SC*** |
|---|---|---|
| SET_OUT1 | SET_OUT1 | LIMSTOP_OFF |
| SET_OUT2 | SET_OUT2 | LIMSTOP_ABRUPT |
| IO1_IN | IO1_IN | THREE_PHASE |
| IO2_IN | IO2_IN | ANTIPHASE |
| . | FAST_PATH | FAST_PATH |
| . | . | STEP_MODE |

Bits IO1_IN, IO2_IN, SET_OUT1 and SET_OUT2 are used in ***PIC-SERVO*** version 5 and earlier to to set the I/O direction and output value of pins LIMIT1 and LIMIT2.

Bits LIMSTOP_OFF and LIMSTOP_ABRUPT are used to enable the limit switch protection described in Section 4.7 of the ***PIC-SERVO SC*** Chip Data Sheet, automatically stopping the motor abruptly or turning the motor off when a limit switch is hit. Only one of these bits should be set. If Step and Direction mode is enabled, neither of these bit should be set.

Bit THREE_PHASE is used to enable 3-Phase commutation mode and Bit ANTIPHASE is used to enable Antiphase PWM mode. No more than one of these bits should be set. If you want to use PWM & Direction mode (the default), neither bit should be set. See Section 4.5.2 and Section 4.5.3 of the ***PIC-SERVO SC*** Chip Data Sheet for a description of Antiphase PWM and 3-Phase commutation mode.

Bit FAST_PATH, used in ***PIC-SERVO*** V.5 and later, is used to set the fast path option for path control mode described in Section 4.4.5 of the ***PIC-SERVO SC*** Chip Data Sheet.

Bit STEP_MODE is used to enable the Step & Direction input mode described in Section 4.4.6 of the ***PIC-SERVO SC*** Chip Data Sheet. If Step & Direction mode is selected, Bits LIMSTOP_OFF and LIMSTOP_ABRUPT should both be clear.

Note that each time an I/O control command is issued, every one of the mode options will be enabled or disabled according the corresponding bit of the control byte.

Example

For ***PIC-SERVO SC*** module 1, disable the limit switch protection, enable 3-phase commutation, and enable Step & Direction input mode:

```
ServoIoCtrl(1, THREE_PHASE|STEP_MODE);
```

| **ServoSetPos** |
| --- |
| Set the servo position to a specific value |

Function Prototype

BOOL ServoSetPos(byte addr, long pos);

File Name

picservo.cpp

Include

picservo.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

pos – Position

32 bit position data used for setting position counter
(signed 32 bit integer: -2, 147, 483, 648 to +2, 147, 483, 647)

Description

Sets the position of a ***PIC-SERVO SC*** ( v.10) module to a specified value.

Example

To set servo module 1 to position 100:

```
ServoSetPos(1, 100);
```

| **ServoStartMove** |
| --- |
| Synchronously starts preloaded motions |

Function Prototype

BOOL ServoStartMove(byte grouppaddr, byte groupleader);

File Name

picservo.cpp

Include

picservo.h

Return Value

0 = Fail
1 = Success

Arguments

groupaddr – Module address

Module address (1 – 32) or group address (0x80 – 0xFF)

groupleader – Group leader address

If groupaddr is individual use: leaderaddr = addr
If groupaddr is group use: leaderaddr = group leader address
(If no group leader use: leaderaddr = 0)

Description

The ServoStartMove() command is intended to be sent to a group of servos which have been preloaded (using the ServoLoadTraj() command) with motion profile data. When sent to the group address for these servos it will cause all servos to start their motions simultaneously. Note that the data loaded into each servo with the ServoLoadTraj() command merely sits unused in a buffer until a ServoStartMove() command is received. If another ServoLoadTraj() command is received before the ServoStartMove() command, it will overwrite all the previous trajectory data.

This command may be sent to either an individual servo or a group of servos by setting the module address argument (groupaddr) to the appropriate individual or group address. The group leader address argument (leaderaddr) should contain the individual address of the group leader. If there is no group leader, leaderaddr should be set to 0. If this command is sent to an individual address, then the group leader address argument should be set to the individual address.

Note: A similar command, NmcSyncOutput(), is used to synchronously execute commands using preloaded data for modules of different types.

Example

To synchronously start the preloaded motions on ***PIC-SERVO*** modules with group address 0x80 and group leader address 0x01:

```
ServoStartMove(0x80, 1);
```

| **ServoStartPathMode** |
| --- |
| Starts execution of the path loaded into the internal path point buffer |

Function Prototype

BOOL ServoStartPathMode(byte groupaddr, byte groupleader);

File Name

picservo.cpp

Include

picservo.h

Return Value

0 = Fail
1 = Success

Arguments

groupaddr – Module address

Module address (1 – 32) or group address (0x80 – 0xFF)

groupleader – Group leader address

If groupaddr is individual use: groupleader = groupaddr
If groupaddr is group use: groupleader = group leader addresses
(If no group leader use: groupleader = 0)

Description

ServoStartPathMode() starts the execution of a the path loaded into the ***PIC-SERVO***'s internal path point buffer. 'groupaddr' is the group address for all controllers to be started and 'leaderaddr' is the individual address for the group's leader (use leaderaddr = 0 if there is no leader).

After the ServoStartPathPoints() command is sent and the path motion starts executing, the PATH_MODE bit in the auxiliary status byte will be set. Usually you will want to send this command to the entire group of controllers involved in a multi-axis motion to retain coordination. As the path motion executes, the old path points will be removed from the path point buffer.

The number of path points currently residing in the buffer can be read using the NmcReadStatus() command. The buffer on the ***PIC-SERVO SC****** can hold a maximum of 128 path points. Even after the path mode motion has started, you can use ServoAddPathpoints() to dynamically add additional path points to the buffers as they empty to create motions of any length.

When the path point buffer runs out, the motor will stop at the last specified path point. The ServoStopMotor() command (any mode) can also be used to terminate a path mode motion. When a path mode motion is terminated, the PATH_MODE bit in the auxiliary

*The ***PIC-SERVO CMC*** path point buffer can only hold 96 points. Therefore, you will want to limit the number of points you add to 96 if using a mix of ***PIC-SERVO SC*** and ***PIC-SERVO CMC*** controllers.

status byte will be cleared.

When using ***PIC-SERVO CMC*** version 5, the advanced features bit of the ServoStopMotor() command must first be enabled before using this function.

## Example

To start path motion for group address 0x81 with group leader address = 1:

```
ServoStartPathMode(0x81, 1);
```

| **ServoStopHere** |
| --- |
| Stop the motor at the specified position |

Function Prototype

BOOL ServoStopHere(byte addr, byte mode, long pos);

File Name

picservo.cpp

Include

picservo.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

mode – Stop mode

Should be set to the value: (AMP_ENABLE | STOP_HERE)

pos – Stop position

Unprofiled command position (signed 32 bit integer)

Description

The ServoStopHere() command is a special version of the Stop Motor command and is used to stop the motor at a specified position. If the STOP_HERE bit is set, the PID command position will be set to the position specified with the "pos" argument, and the motor will move abruptly at that position and stop. There is no profiling to smooth the motion. If using this mode, the distance between the specified goal position and the current motor position should be less than the position error limit specified with the ServoSetGain() command, otherwise, a position error will be generated. This stopping mode will cause the MOVE_DONE bit of the status byte to be set.

Example

Use the following command string to cause servo module 1 to stop at position 100:

```
ServoStopHere(1, AMP_ENABLE | STOP_HERE, 100);
```

| **ServoStopMotor** |
| --- |
| Stop the motor in the manner specified by mode |

Function Prototype

BOOL ServoStopMotor(byte addr, byte mode);

File Name

picservo.cpp

Include

picservo.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

mode – Stop mode

---- Logical OR of the following stop mode bits ----
ENABLE_AMP – enable amplifier
MOTOR_OFF – turn motor off
STOP_ABRUPT – stop motor abruptly
STOP_SMOOTH – stop motor smoothly
ADV_FEATURE - enable features in ver.5 CMC

Description

The Stop Motor command is used to both stop the motor in one of three ways, and also to control whether the amplifier is enabled or not.

If bit ENABLE_AMP of the control byte is set, the amplifier enable output will be set if and when the motor supply voltage is within the proper range. If in PWM & Direction or Antiphase PWM modes, this will simply result in raising the ENABLE0 output. If in 3-Phase commutation mode, this will enable the commutation logic to raise the proper combination of ENABLE pins. If Bit ENABLE_AMP is cleared to 0, all ENABLE pins will be lowered.

If bit MOTOR_OFF of the control byte is set, the motor will be turned off by disabling the PID servo and setting the PWM output to 0. This stopping mode will cause the MOVE_DONE bit and the POS_ERROR bit of the status byte to be set. The ENABLE_AMP bit may be set or cleared with this stopping mode.

If bit STOP_ABRUPT of the control byte is set, the motor will immediately attempt to servo to its current position, causing the motor to stop abruptly. If this mode is selected, the amplifier enable bit should be set as well. This stopping mode will cause the MOVE_DONE bit of the status byte will be set.

If bit STOP_SMOOTH is set, the motor will decelerate to a stop smoothly at the current

programmed acceleration value. If this mode is selected, the amplifier enable bit should be set as well. This stopping mode will cause the MOVE_DONE bit of the status byte to be cleared while decelerating and then set again once the motor has stopped.

If bit ADV_FEATURE is set, it enables the advanced features on the ***PIC-SERVO CMC*** (version 5) -- this bit applies only to the ***PIC-SERVO*** version 5. Once the advanced features are enabled, they remain enabled until the ***PIC-SERVO*** chip is reset..

Only one of the stopping mode bits STOP_MOTOR, STOP_ABRUPT, and STOP_SMOOTH should be selected. If none of these bit are set, the amplifier will be enabled or disabled as specified, but no other action will be taken.

## Example

For module 1, use the following command string to turn off the motor and disable the amplifier:

```
ServoStopMotor(1, MOTOR_OFF);
```

For module address 1, use the following command string to smoothly decelerate to a stop:

```
ServoStopMotor (1,  ENABLE_AMP|STOP_SMOOTH);
```

| **SimpleMsgBox** |
| --- |
| Display a simple message box |

Function Prototype

int SimpleMsgBox(char *msgstr);

File Name

sio_util.cpp

Include

sio_util.h

Return Value

Returns 0 if fails, and non-zero on success. See the Windows OS MessageBox() function description for success return values.

Arguments

msgstr – Message string

Pointer to a null-terminated string that contains the message to be displayed.

Description

Displays a simple windows message box.

Example

To display the message “Hello World”:

```
SimpleMsgBox(“Hello World”);
```

| **SioChangeBaud** (Internal Library Function) |
| --- |
| Changes the baud rate of a COM port |

Function Prototype

BOOL SioChangeBaud(HANDLE ComPort, unsigned int baudrate);

File Name

sio_util.cpp

Include

sio_util.h

Return Value

0 = Fail
1 = Success

Arguments

ComPort – COM port handle
baudrate – Baud rate
9600, 19200, 38400, 57600, 115200, or 230400

Description

Changes the baud rate of the specified COM port.

**NOTE:** For normal operation, users do not need this command.

Example

To change the baud rate of: port 'ComPort' to 115200 baud:

```
HANDLE ComPort;
SioChangeBaud(ComPort, 115200);
```

| **SioClose** (Internal Library Function) |
| --- |
| Closes a COM port |

Function Prototype

BOOL SioClose(HANDLE ComPort);

File Name

sio_util.cpp

Include

sio_util.h

Return Value

0 = Fail
1 = Success

Arguments

ComPort – COM port handle

Description

Closes a COM port.

**NOTE:** For normal operation, users do not need this command.

Example

To close the COM port specified by 'ComPort':

```
HANDLE ComPort;
SioClose(ComPort);
```

| **SioClrInbuf** (Internal Library Function) |
| --- |
| Clears all the characters in a COM port's input buffer |

Function Prototype

BOOL SioClrIbuf(HANDLE ComPort);

File Name

sio_util.cpp

Include

sio_util.h

Return Value

0 = Fail
1 = Success

Arguments

ComPort – COM port handle

Description

Clears all the characters in a COM port's input buffer.

**NOTE:** For normal operation, users do not need this command.

Example

To clear all the characters in the COM port specified by 'ComPort':

```
HANDLE ComPort;
SioClrInbuf(ComPort);
```

| **SioGetChars** (Internal Library Function) |
| --- |
| Read characters from a COM port |

Function Prototype

DWORD SioGetChars(HANDLE ComPort, char *stuff, int n);

File Name

sio_util.cpp

Include

sio_util.h

Return Value

Returns the number of characters actually read.

Arguments

ComPort – COM port handle

stuff – Character array

Store characters read – must be large enough to hold up to 'n' characters.

n – Number of characters to read

Description

SioGetChars() reads in 'n' characters from the specified COM port and puts them in the array pointed to by 'stuff'. This function has a timeout value of approximately 100 millisec. It returns the number of characters actually read.

**NOTE:** For normal operation, users do not need this command.

Example

To read 4 characters from the COM port specified by 'ComPort' into the array 'buffer':

```
char buffer[100];
HANDLE ComPort;
SioGetChars(ComPort, buffer, 4);
```

| **SioOpen** (Internal Library Function) |
| --- |
| Opens a COM port at the specified baud rate. |

Function Prototype

HANDLE SioOpen(char *name, unsigned int baudrate);

File Name

sio_util.cpp

Include

sio_util.h

Return Value

Returns a handle for the COM port stream.

Arguments

portname – Port name

Name of COM port ("COM*n*:", where $n = 1 - 8$)

baudrate – Baud rate

9600, 19200, 38400, 57600, 115200, or 230400

Description

Opens a COM port at the specified baud rate.

**NOTE:** For normal operation, users do not need this command.

Example

To open COM port 1 at 115200 baud:

```
SioOpen(COM1, 115200);
```

| **SioPutChars** |
| --- |
| Puts characters out a COM port |

Function Prototype

BOOL SioPutChars(HANDLE ComPort, char *stuff, int n);

File Name

sio_util.cpp

Include

sio_util.h

Return Value

0 = Fail
1 = Success

Arguments

ComPort – COM port handle

Handle of COM port used to send characters.

stuff – Character data

Array containing character data to send

n – Send number

Number of characters to send

Description

Shoves n characters out the COM port specified by handle ComPort.

Example

To send 12 characters from array 'buffer' out the COM port with handle 'ComPort':

```
HANDLE ComPort;
char buffer[100];
SioPutChars(ComPort, buffer, 12);
```

| **SioTest** |
| --- |
| Returns the number of characters in a COM port's input buffer |

Function Prototype

DWORD SioTest(HANDLE ComPort);

File Name

sio_util.cpp

Include

sio_util.h

Return Value

Returns the number of characters in the ComPort's input buffer.

Arguments

ComPort – COM port handle

Handle of COM port to be tested.

Description

Returns the number of characters in the ComPort's input buffer.

Example

To return the number of characters in the COM port with handle 'ComPort':

```
HANDLE ComPort;
SioTest(ComPort);
```

| **StepGetAD** |
| --- |
| Returns the current A/D value |

Function Prototype

byte StepGetAD(byte addr)*;

File Name

picstep.cpp

Include

picstep.h

Return Value

Returns the current A/D value.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the current A/D value (stored locally) for a ***PIC-STEP*** module.

Example

To get the current A/D value of module 1:

```
ad_val = StepGetAD(1);
```

*This function only retrieves data stored locally on the PC. To insure the data is current, NmcReadStatus should be called just prior to calling this function. Alternately, if NmcDefineStatus has been used to permanently include the relevant data item in the status packet, any command sent to a module will update the locally stored data.

| **StepGetCmdAcc** |
| --- |
| Returns the most recently issued command acceleration |

Function Prototype

byte StepGetCmdAcc(byte addr);

File Name

picstep.cpp

Include

picstep.h

Return Value

Returns the most recently issued command acceleration.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the most recently issued command acceleration time for a ***PIC-STEP*** module.

Example

To get the most recently issued command acceleration time for module 1:

```
cmd_accel = StepGetCmdAcc(1);
```

| **StepGetCmdPos** |
| --- |
| Returns the most recently issued command position |

Function Prototype

long StepGetCmdPos(byte addr);

File Name

picstep.cpp

Include

picstep.h

Return Value

Returns the most recently issued command position.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the most recently issued command position for a ***PIC-STEP*** module.

Example

To get the most recently issued command position for module 1:

```
cmd_pos = StepGetCmdPos(1);
```

| **StepGetCmdSpeed** |
| --- |
| Returns the most recently issued command speed |

Function Prototype

byte StepGetCmdSpeed(byte addr);

File Name

picstep.cpp

Include

picstep.h

Return Value

Returns the most recently issued command speed.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the most recently issued command speed for a ***PIC-STEP*** module.

Example

To get the most recently issued command speed for module 1:

```
cmd_speed = StepGetCmdSpeed(1);
```

| **StepGetCmdST** |
| --- |
| Returns the most recently issued command timer count |

Function Prototype

unsigned short int StepGetCmdST(byte addr);

File Name

picstep.cpp

Include

picstep.h

Return Value

Returns the most recently issued command timer count.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the most recently issued command timer count for a ***PIC-STEP*** module.

Example

To get the most recently issued command timer count for module 1:

```
cmd_st = StepGetCmdST(1);
```

| **StepGetCtrlMode** |
| --- |
| Returns the control mode byte (set with StepSetParam) |

Function Prototype

byte StepGetCtrlMode(byte addr);

File Name

picstep.cpp

Include

picstep.h

Return Value

Returns the control mode byte.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the control mode byte (set with StepSetParam) for a ***PIC-STEP*** module.

Example

To get the control mode byte for module 1:

```
ctrl_mode = StepGetCtrlMode(1);
```

| **StepGetHome** |
| --- |
| Returns the current motor home position |

Function Prototype

long StepGetHome(byte addr)*;

File Name

picstep.cpp

Include

picstep.h

Return Value

Returns the current motor home position.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the current motor home position (stored locally) of a ***PIC-STEP*** module.

Example

To get the current motor home position of module 1:

```
home_pos = StepGetHome(1);
```

*This function only retrieves data stored locally on the PC. To insure the data is current, NmcReadStatus should be called just prior to calling this function. Alternately, if NmcDefineStatus has been used to permanently include the relevant data item in the status packet, any command sent to a module will update the locally stored data.

| **StepGetHomeCtrl** |
| --- |
| Returns the homing control byte |

Function Prototype

byte StepGetHomeCtrl(byte addr);

File Name

picstep.cpp

Include

picstep.h

Return Value

Returns the homing control byte.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the homing control byte for a ***PIC-STEP*** module.

Example

To get the homing control byte of module 1:

```
home_ctrl = StepGetHomeCtrl(1);
```

| **StepGetHoldCurrent** |
| --- |
| Returns the holding current (set with StepSetParam) |

Function Prototype

byte StepGetHoldCurrent(byte addr);

File Name

picstep.cpp

Include

picstep.h

Return Value

Returns the holding current.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the holding current (set with StepSetParam) for a ***PIC-STEP*** module.

Example

To get the holding current for module 1:

```
hold_cur = StepGetHoldCurrent(1);
```

| **StepGetInbyte** |
| --- |
| Returns the current input byte |

Function Prototype

byte StepGetInbyte(byte addr)*;

File Name

picstep.cpp

Include

picstep.h

Return Value

Returns the current input byte.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the current input byte (stored locally) of a ***PIC-STEP*** module. This includes the E-stop, limit switch, homing switch and auxiliary input bits.

Example

To get the current input byte of module 1:

```
inbyte = StepGetInbyte(1);
```

*This function only retrieves data stored locally on the PC. To insure the data is current, NmcReadStatus should be called just prior to calling this function. Alternately, if NmcDefineStatus has been used to permanently include the relevant data item in the status packet, any command sent to a module will update the locally stored data.

| **StepGetMinSpeed** |
| --- |
| Returns the minimum stepping speed |

Function Prototype

byte StepGetMinSpeed(byte addr);

File Name

picstep.cpp

Include

picstep.h

Return Value

Returns the minimum stepping speed.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the minimum stepping speed for a ***PIC-STEP*** module.

Example

To get the minimum stepping speed of module 1:

```
min_speed = StepGetMinSpeed(1);
```

| **StepGetOutputs** |
| --- |
| Returns the most recently issued command output byte |

Function Prototype

byte StepGetOutputs(byte addr);

File Name

picstep.cpp

Include

picstep.h

Return Value

Returns the most recently issued command output byte.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the most recently issued command output byte for a ***PIC-STEP*** module.

Example

To get the most recently issued command output byte of module 1:

```
outputs = StepGetOutputs(1);
```

| **StepGetPos** |
| --- |
| Returns the current motor position |

Function Prototype

long StepGetPos(byte addr)*;

File Name

picstep.cpp

Include

picstep.h

Return Value

Returns the current motor position.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the current motor position (stored locally) of a ***PIC-STEP*** module.

Example

To get the current motor position of module 1:

```
pos = StepGetPos(1);
```

*This function only retrieves data stored locally on the PC. To insure the data is current, NmcReadStatus should be called just prior to calling this function. Alternately, if NmcDefineStatus has been used to permanently include the relevant data item in the status packet, any command sent to a module will update the locally stored data.

| **StepGetRunCurrent** |
| --- |
| Returns the running current (set with StepSetParam) |

Function Prototype

byte StepGetRunCurrent(byte addr);

File Name

picstep.cpp

Include

picstep.h

Return Value

Returns the running current.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the running current (set with StepSetParam) of a ***PIC-STEP*** module.

Example

To get the running current of module 1:

```
run_cur = StepGetRunCurrent(1);
```

| **StepGetStat** (Internal Library Function) |
| --- |
| Low-level routine to process and store returned ***PIC-STEP*** status data |

Function Prototype

BOOL StepGetStat(byte addr);

File Name

picstep.cpp

Include

picstep.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

Description

StepGetStat() processes and stores the status data returned for a ***PIC-STEP*** module. It takes the status data stored in the global array "inbuf", verifies the number of bytes received and checksums, then stores the status fields in the NMCMOD structure mod [addr].

**NOTE:** For normal operation, users do not need this command.

| **StepGetStepTime** |
| --- |
| Returns the current timer count |

Function Prototype

unsigned short int StepGetStepTime(byte addr)*;

File Name

picstep.cpp

Include

picstep.h

Return Value

Returns the current timer count.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the current timer count (stored locally) for a ***PIC-STEP*** module.

Example

To get the current timer count of module 1:

```
step_time = StepGetStepTime(1);
```

*This function only retrieves data stored locally on the PC. To insure the data is current, NmcReadStatus should be called just prior to calling this function. Alternately, if NmcDefineStatus has been used to permanently include the relevant data item in the status packet, any command sent to a module will update the locally stored data.

| **StepGetStopCtrl** |
| --- |
| Returns the stopping control byte |

Function Prototype

byte StepGetStopControl(byte addr);

File Name

picstep.cpp

Include

picstep.h

Return Value

Returns the stopping control byte.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the stopping control byte for a ***PIC-STEP*** module.

Example

To get the stopping control byte of module 1:

```
stop_ctrl = StepGetStopCtrl(1);
```

| **StepGetThermLimit** |
| --- |
| Returns the thermal limit (set with StepSetParam) |

Function Prototype

byte StepGetThermLimit(byte addr);

File Name

picstep.cpp

Include

picstep.h

Return Value

Returns the thermal limit.

Arguments

addr – Module address

Module address (1 – 32)

Description

Returns the thermal limit (set with StepSetParam) for a ***PIC-STEP*** module.

Example

To get the thermal limit of module 1:

```
therm_lim = StepGetThermLimit(1);
```

| **StepLoadTraj** |
| --- |
| Loads motion trajectory information |

Function Prototype

BOOL StepLoadTraj(byte addr, byte mode, long pos, byte speed, byte acc, float raw_speed);

File Name

picstep.cpp

Include

picsstep.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address
Module address (1 – 32)
mode - Trajectory mode
---- Logical OR of the following load trajectory mode bits ----
LOAD_POS – load position data
LOAD_SPEED – load velocity data
LOAD_ACC – load acceleration data
LOAD_ST – load initial time count
STEP_REV – use reverse direction
START_NOW – start now
pos – Position data (4 bytes)
Position data if LOAD_POS bit of Trajectory Mode is set
(signed 32 bit integer: -2,147,483,648 to +2,147,483,647)
speed – Speed data (1 byte)
Speed data if LOAD_SPEED bit of Trajectory Mode is set
(8 bit integer 1 to 250)
acc – Acceleration data (1 byte)
Acceleration data if LOAD_ACC bit of Trajectory Mode is set
(8 bit integer: 1 – 255; larger values = slower accel)
raw_speed – Raw speed
Speed data if LOAD_ST bit of Trajectory Mode is set
(float: 0.4 – 250.0)

Description

All motion parameters are set with this command. Setting one of the LOAD_POS, LOAD_VEL, LOAD_ACC, or LOAD_ST bits in the mode will cause the corresponding data (pos, speed, acc, or raw_speed respectively) to be sent to the ***PIC-STEP***. The selection of which data is loaded will also determine which operating mode (trap. profile, profiled velocity, unprofiled velocity) will be used. In addition, there are restrictions on which modes can be entered from any given current mode. Table 2 below details which goal

modes can be entered from any starting mode, and which data needs to be loaded in order to enter a specific goal mode.

The position data is used as the goal position in trapezoidal profile mode and in the unprofiled position mode. The speed data is used as the goal speed in velocity profile mode or as the maximum speed in trapezoidal profile mode. (If the goal speed is less than the minimum profile speed, the minimum profile speed will be used instead.) The acceleration data is used in both trapezoidal and velocity profile mode. The initial timer count data is used in unprofiled velocity and unprofiled position mode. It is also used as the starting point for ramping if you choose to enter velocity mode and decelerate or accelerate to a new velocity.

Position values can be either positive or negative and represent an absolute motor position, but the speed and acceleration should always be positive. Please see the ***PIC-STEP*** chip data sheet for details on specifying values for the position, speed and acceleration.

Bit STEP_REVERSE is used with the velocity profile mode or unprofiled velocity mode to set the direction of motion. Note that if the motor is moving, the direction of motion cannot be changed without first stopping the motor using the StepStopMotor() command.

If bit START_NOW is set, the motion will be executed immediately. If it is not set, the command will have no effect whatsoever (and may be overwritten by another StepLoadTraj() command) until a NmcSyncOutput() command is called with the module's address or group address.

### Example

Move module 1 to an absolute position of -1500 steps, speed of 100 (2500 steps per sec.), acceleration of 40, in trapezoidal profile mode, starting now (assume ***PIC-STEP*** is in 1x speed mode):

```
StepLoadTraj(1, LOAD_POS|LOAD_SPEED|LOAD_ACC|START_NOW,
                 -1500, 100, 40);
```

Move module address 1 with a velocity of -100 in velocity profile mode starting now (assume the acceleration parameter has already been loaded, and ***PIC-STEP*** is in 1x speed mode):

```
StepLoadTraj(1, LOAD_SPEED|REVERSE|START_NOW,
                0, 100, 0);
```

**Table 2 - *PIC-STEP* Operating Mode Transition Table**

| | *Starting Mode* | | | | |
|---|---|---|---|---|---|
| *Goal Mode* | **Stopped** | **Trap. Profile** | **Vel. Profile** | **Unprofiled Velocity, With Stop** | **Unprofiled Velocity, No Stop** |
| **Stopped** | --- | *Use Stop Command* | *Use Stop Command* | *Use Stop Command* | *Use Stop Command* |
| **Trap. Profile** | ✔ Position<br>○ Velocity<br>○ Acceleration<br>✘ Tmr. Count | *Not Allowed* | *Not Allowed* | *Not Allowed* | *Not Allowed* |
| **Vel. Profile** | ✘ Position<br>○ Velocity<br>○ Acceleration<br>✘ Tmr. Count | ✘ Position<br>○ Velocity<br>○ Acceleration<br>✘ Tmr. Count | ✘ Position<br>○ Velocity<br>○ Acceleration<br>✘ Tmr. Count | ✘ Position<br>○ Velocity<br>○ Acceleration<br>✘ Tmr. Count | ✘ Position<br>○ Velocity<br>○ Acceleration<br>✘ Tmr. Count |
| **Unprofiled Velocity, With Stop** | ✔ Position<br>✘ Velocity<br>✘ Acceleration<br>✔ Tmr. Count | ✘ Position<br>✘ Velocity<br>✘ Acceleration<br>✔ Tmr. Count | *Not Allowed* | ✘ Position<br>✘ Velocity<br>✘ Acceleration<br>✔ Tmr. Count | *Not Allowed* |
| **Unprofiled Velocity, No Stop** | ✘ Position<br>✘ Velocity<br>✘ Acceleration<br>✔ Tmr. Count | *Not Allowed* | ✘ Position<br>✘ Velocity<br>✘ Acceleration<br>✔ Tmr. Count | *Not Allowed* | ✘ Position<br>✘ Velocity<br>✘ Acceleration<br>✔ Tmr. Count |

✔ Load this parameter

✘ Do not load this parameter

○ Optionally load this parameter

*If a parameter is not loaded, the previously loaded value will be used.*

| **StepNewMod** (Internal Library Function) |
| --- |
| Creates and initializes a new STEPMOD structure |

Function Prototype

STEPMOD *StepNewMod(void);

File Name

picstep.cpp

Include

picstep.h

Return Value

Pointer to the new STEPMOD structure.

Arguments

None

Description

Creates, initializes, and returns a new STEPMOD structure for storing ***PIC-STEP*** data.

**NOTE:** For normal operation, users do not need this command.

| **StepResetPos** |
| --- |
| Resets position counter to zero |

Function Prototype

BOOL StepResetPos(byte addr);

File Name

picstep.cpp

Include

picstep.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

Description

StepResetPos() resets the position counter to a value of zero. Do *not* issue this command when the motor is in motion.

Example

To have module 1 reset the position counter to zero:

```
StepResetPos(1);
```

| **StepSetHoming** |
| --- |
| Set homing mode parameters for capturing the home position |

Function Prototype

BOOL StepSetHoming(byte addr, byte mode);

File Name

picstep.cpp

Include

picstep.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

mode – Homing mode

---- Logical OR of the following load homing mode bits ----
ON_LIMIT1 - home on change in limit 1
ON_LIMIT2 - home on change in limit 2
HOME_MOTOR_OFF - turn motor off when homed
ON_HOMESW - home on change in index
HOME_STOP_ABRUPT - stop abruptly when homed
HOME_STOP_SMOOTH - stop smoothly when homed

Description

StepSetHoming() causes the controller to monitor the specified conditions and capture the home position when *any* of the flagged homing conditions occur. The HOME_IN_PROG bit in the Status byte is set when this command is issued and it is lowered when the home position has been found. Setting one (and only one) of bits HOME_MOTOR_OFF, HOME_STOP_ABRUPT, or HOME_STOP_SMOOTH will cause the motor to stop automatically in the specified manner once the home condition has been triggered.

Example

To have module 1 capture the home position on a change of LIMIT1 or LIMIT2 and then stop abruptly:

```
StepSetHoming(1, ON_LIMIT1|ON_LIMIT2|HOME_STOP_ABRUPT);
```

| **StepSetOutputs** |
| --- |
| Sets or clears the general purpose output pins |

Function Prototype

BOOL StepSetOutputs(byte addr, byte outbyte);

File Name

picstep.cpp

Include

picstep.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

outbyte – Output values

Bits 0 thru 4 of outbyte correspond to output pins OUT1 – OUT5. Setting a bit in outbyte will cause the corresponding pin to go HI, clearing a bit will cause the pin to go LOW.

Description

StepSetOutput() sets or clears the general purpose output pins OUT1 – OUT5.

Example

To have module 1 set the OUT1 and OUT2 output pins HIGH, and the OUT3, OUT4, and OUT5 output pins LOW:

```
StepSetOutputs(1, 0x03);
```

| **StepSetParam** |
| --- |
| Set the ***PIC-STEP*** operating parameters |

Function Prototype

BOOL StepSetParam(byte addr, byte mode, byte minspeed, byte runcur, byte holdcur, byte thermlim);

File Name

picstep.cpp

Include

picstep.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address
Module address (1 – (MAXMOD-1))
mode – Operating mode
---- Logical OR of the following set parameter mode bits ----
SPEED_8X - use speed units of 200 step pulses/sec.
SPEED_4X - use speed units of 100 step pulses/sec.
SPEED_2X - use speed units of 50 step pulses/sec.
SPEED_1X - use speed units of 25 step pulses/sec.
IGNORE_LIMITS - do not stop automatically on limit switches
IGNORE_ESTOP - do not stop automatically on e-stop
ESTOP_OFF – turn amplifier off on estop or limit switch
minspeed – minimum stepping speed (1 – 250)
runcur – running current limit (0 – 255)
holdcur – holding current limit (0 – 255)
thermlim – thermal limit (0 – 255)

Description

Sets control parameters governing the operation of the ***PIC-STEP***. This command must be issued before any motions can be executed. If this command is issued while the motor is in motion, any changes to the speed mode bits and minimum profile speed will be ignored. Please see the ***PIC-STEP*** chip data sheet for details on specifying these parameter values.

Example

To set ***PIC-STEP*** module 1 to use 1x speed, minimum speed of 10, running current 100, hold current 50, and thermal limit 0 (disables thermal shutdown feature):

```
StepSetParam(1, SPEED_1X, 10, 100, 50, 0);
```

| **StepStopMotor** |
| --- |
| Stops a motor in the manner specified by mode |

Function Prototype

BOOL StepStopMotor(byte addr, byte mode);

File Name

picstep.cpp

Include

picstep.h

Return Value

0 = Fail
1 = Success

Arguments

addr – Module address

Module address (1 – 32)

mode – Stop mode

---- Logical OR of the following stop motor control bits ----
ENABLE_AMP – enable amplifier
STOP_ABRUPT – stop motor abruptly
STOP_SMOOTH – stop motor smoothly

Description

Stops the motor in the specified manner. If bit ENABLE_AMP of the Stop Control Byte is set, the ***PIC-STEP*** AMP_EN pin will be set; if bit ENABLE_AMP is cleared, the ***PIC-STEP*** AMP_EN pin will be cleared, regardless of the state of the other bits. If bit STOP_ABRUPT is set, the motor will stop abruptly at its current position. If bit STOP_SMOOTH is set, the motor will decelerate to a stop using the current acceleration time for the deceleration ramp. Only one of bits STOP_ABRUPT or STOP_SMOOTH should be set at one time.

When you stop smoothly, you are effectively setting the goal speed to zero, and when the minimum profile speed is reached, the motor will stop. Note that if, after stopping smoothly, you want to enter the trapezoidal profile mode, you will have to load a new goal velocity (along with the position) because StepStopMotor() command will have set the goal velocity to zero.

Note that the StepStopMotor() command must be issued in order to initially enable the amplifier. The amplifier can be enabled without setting any of the other stop control bits.

Example

To make ***PIC-STEP*** module 1 stop smoothly:

```
StepStopMotor(1, ENABLE_AMP|STOP_SMOOTH);
```

## 4. Status Packet Description

NMCLIB04.DLL includes a set of functions for accessing various types of status data which has been returned to the host in status packets. Note that these functions do not query the controller modules for data; they simple return the values stored locally on the PC. NmcReadStatus() (or and other command which causes a status packet to be sent) must be called first to ensure that the data stored locally on the PC is current.

### 4.1 *PIC-SERVO* Status Packet

### Status Packet Functions

| Function | Type | Status Field Description |
|---|---|---|
| NmcGetStat(addr) | byte | Status byte. See status byte bit field definitions. |
| ServoGetPos(addr) | long | Motor position. Signed 32 bit integer. |
| ServoGetAD(addr) | byte | A/D value of voltage on CUR_SENSE pin. Range: 0 - 255 |
| ServoGetVel(addr) | short int | Motor velocity in encoder counts per servo cycle. Signed 16 bit integer. |
| ServoGetAux(addr) | byte | Auxiliary status byte. See auxiliary status byte bit field definitions. |
| ServoGetHome(addr) | long | Motor home position. Signed 32 bit integer. |
| NmcGetModType(addr) | byte | Module Type. 0=PIC-SERVO, 2=PIC-I/O, 3= PIC-STEP. |
| NmcGetModVer(addr) | byte | Module Version. |
| ServoGetPError(addr) | short int | Servo positioning Error. Signed 16 bit integer. |
| ServoGetNPoints(addr) | byte | Number of path points left in path buffer. |

### Status Byte Bit Fields

| Bit | Name | Definition |
|---|---|---|
| 0 | MOVE_DONE | Clear when in the middle of a trapezoidal profile move, or in velocity mode, when accelerating from one velocity to the next. This bit is set otherwise, including while the position servo is disabled. |
| 1 | CKSUM_ERROR | Set if there was a checksum error in the most recently received command packet. |
| 2 | OVERCURRENT | Set if current limiting occurred. Must be cleared by user with ServoClearBits() command. |
| 3 | POWER_ON | Set if motor power is the voltage on the VOLT_SENSE pin is between 0.9v and 4.5v. Clear otherwise. |
| 4 | POS_ERR | Set if the position error exceeds the position error limit. It is also set whenever the position servo is disabled. Must be cleared by user with ServoClearBits() command. |
| 5 | LIMIT1 | Value of limit switch 1 input. |
| 6 | LIMIT2 | Value of limit switch 2 input. |
| 7 | HOME_IN_PROG | Set while searching for a home position. Reset to zero once the home position has been captured. |

### Auxiliary Status Byte Bit Fields

| Bit | Name | Definition |
|---|---|---|
| 0 | INDEX | Encoder index input value. |
| 1 | POS_WRAP | Set if the 32 bit position counter overflows or underflows. Must be cleared with the ServoClearBits() command. |
| 2 | SERVO_ON | Set if the position servo is enabled, clear otherwise. |
| 3 | ACCEL_DONE | Set when the motor is accelerating, clear when decelerating. This bit has no meaning when stopped or at a constant velocity. |
| 4 | SLEW_DONE | Set when moving at a constant velocity or when stopped. Clear when accelerating or decelerating. |
| 5 | SERVO_OVERRUN | This bit is set only if the calculations required for one servo cycle take longer than 0.51 milliseconds. This can happen if Step inputs exceed the allowable step input rate. Cleared with the ServoClearBits() command. |
| 6 | PATH_MODE | This bit is set when a path mode motion is in progress. It is cleared when the path point buffer is emptied or if a ServoStopMotor() command is issued. |
| 7 | not used | |

## 4.2 *PIC-STEP* Status Packet

### Status Packet Functions

| Function | Type | Status Field Description |
|---|---|---|
| NmcGetStat(addr) | byte | Status byte. See status byte bit field definitions. |
| StepGetPos(addr) | long | Motor position. Signed 32 bit integer. |
| StepGetAD(addr) | byte | A/D value of voltage on CUR_SENSE pin. Range: 0 – 255. |
| StepGetStepTime(addr) | unsigned short int | Current initial timer count. |
| StepGetInbyte(addr) | byte | Inputs byte. See inputs byte bit field definitions. |
| StepGetHome(addr) | long | Motor home position. Signed 32 bit integer. |
| NmcGetModType(addr) | byte | Module Type. 0=PIC-SERVO, 2=PIC-I/O, 3= PIC-STEP. |
| NmcGetModVer(addr) | byte | Module Version. |

### Status Byte Bit Fields

| Bit | Name | Definition |
|---|---|---|
| 0 | MOTOR_MOVING | Set when motor is moving. |
| 1 | CKSUM_ERROR | Set if there was a checksum error in the most recently received command packet. |
| 2 | AMP_ENABLED | Set when amplifier enable output signal is HIGH (amplifier is enabled). |
| 3 | POWER_ON | Set when the power sense input signal is HIGH (motor power is on). |
| 4 | AT_SPEED | Set when at the commanded speed . |
| 5 | VEL_MODE | Set when in velocity profile mode. |
| 6 | TRAP_MODE | Set when in trapezoidal profile mode. |
| 7 | HOME_IN_PROG | Set while homing in progress, cleared when home found . |

### Inputs Byte Bit Fields

| Bit | Name | Definition |
|---|---|---|
| 0 | ESTOP | Value of the emergency stop input. |
| 1 | AUX_IN1 | Value of IN1 general purpose input. |
| 2 | AUX_IN2 | Value of IN2 general purpose input. |
| 3 | FWD_LIMIT | Value of limit switch 1 input. |
| 4 | REV_LIMIT | Value of limit switch 2 input. |
| 5 | HOME_SWITCH | Value of home switch input. |
| 6 | not used | |
| 7 | not used | |

## 4.3 *PIC-I/O* Status Packet

### Status Packet Functions

| Function | Type | Status Field Description |
|---|---|---|
| NmcGetStat(addr) | byte | Status byte. See status byte bit field definitions. |
| IoInBitVal(addr, bitnum) | BOOL | Value of specified input bit (bitnum = 0-11). |
| IoGetADCVal(addr, channel) | byte | Value of specified A/D channel input. (channel = 0, 1, 2). |
| IoGetTimerVal(addr) | unsigned long | Counter/timer value. |
| NmcGetModType(addr) | byte | Module Type. 0=PIC-SERVO, 2=PIC-I/O, 3= PIC-STEP. |
| NmcGetModVer(addr) | byte | Module Version. |
| IoInBitSVal(addr, bitnum) | BOOL | Value of specified input bit captured with NmcSychInput() command (bitnum = 0-11). |
| IoGetTimerSVal | unsigned long | Value of counter/timer captured with NmcSyncInput() command. |

### Status Byte Bit Fields

| Bit | Name | Definition |
|---|---|---|
| 0 | not used | |
| 1 | CKSUM_ERROR | Set if there was a checksum error in the most recently received command packet. |
| 2 | not used | |
| 3 | not used | |
| 4 | not used | |
| 5 | not used | |
| 6 | not used | |
| 7 | not used | |