
SSA-485 Motion Control Library

The *SSA-485 Motion Control Library* is a library of C language functions for communicating with the **PIC-SERVO**, **PIC-STEP**, and **PIC-I/O** modules. This library is for use with Microchip's MCC18 C compiler. Included in the library are high-level routines for initializing, controlling, and managing the **PIC-SERVO**, **PIC-STEP**, and **PIC-I/O** modules. The library and example programs are available from www.jrkerr.com/software.html.

- For use with **SSA-485** Smart Serial Adapter board when configured as a stand-alone host using the PIC18F2620 microcontroller.
- Supports complete **PIC-SERVO**, **PIC-STEP**, or **PIC-I/O** command set.
- Uses Microchip's MCC18 C compiler, MPLAB development environment, and ICD2 In-Circuit Debugger.
- Up to 4 **PIC-SERVO**, **PIC-STEP**, or **PIC-I/O** modules can be controlled with *SSA-485 Motion Control Library*. (This number can be expanded if there is sufficient RAM available.)
- Communicates with modules at either 19,200 or 57,600 baud.

1. Software Description

The *SSA-485 Motion Control Library* is a library of high level functions for communicating with the **PIC-SERVO**, **PIC-STEP**, and **PIC-IO** modules. It is designed to run on the **SSA-485** Smart Serial Adapter in stand-alone mode with a PIC18F2620 microcontroller. Included are functions for initializing NMC (*Networked Modular Control*) communications, sending commands to modules, and retrieving status data from modules.

Figure 1 shows a typical application. It consists of a **SSA-485** and several NMC modules (for example, two **PIC-SERVO** and one **PIC-IO**). The **SSA-485** is running the *SSA-485 Motion Control Library* on the PIC18F2620 microcontroller, and is functioning as the host controller. It initializes communications with the NMC modules, assigns module addresses, sends commands to the modules, and evaluates status packets that are returned from the modules. The **SSA-485** communicates with the NMC modules using the full duplex, RS485 (4 wire) based NMC communications protocol. The **SSA-485** may optionally communicate with the outside world using the RS232 or USB interface.

... CAUTION ...

The **SSA-485** Motion Control Library 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.

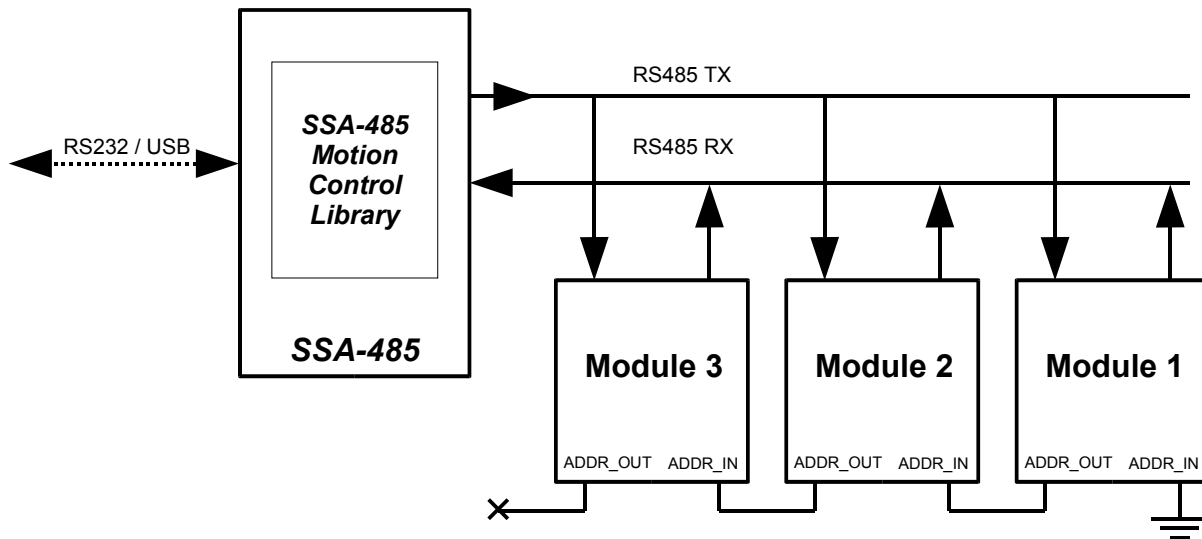


Figure 1 - Typical Application Using SSA Motion Control Library

1.1 Data Structures

The library is built around a set of global variables used for storing module information such as the number of modules, module types, and what module status information is returned.

```
#define MAXMOD      5           //room for 4 mods plus storage for mod 0
byte nummod;          //number of modules detected
byte moddata[MAXMOD][20]; //storage for module status data
byte modtype[MAXMOD];  //module types
byte statitems[MAXMOD]; //defines which status items will be sent
byte numstat[MAXMOD];  //number of addl status bytes sent
```

MAXMOD

MAXMOD is a defined constant that contains one more than the maximum number of modules allowed in the system. The default value of MAXMOD is 5, which means the library will detect and assign addresses to the first 4 modules and ignore the rest. In general, the value of MAXMOD should not be changed without a careful evaluation of the system memory: if MAXMOD is increased, more system memory will be required to store module data, and less memory will be available for the application.

nummod

The variable nummod stores the number of modules detected during system initialization. It is set during system initialization by the function Nmclnit() and will have a value ranging from 0 to MAXMOD-1. After system initialization, applications can read nummod to determine the number of modules in the system. The value of nummod should not be changed directly.

modtype

The array modtype stores the module type of each module. The defined module types are:

```
#define SERVOMODTYPE 0 // PIC-SERVO module type
#define IOMODTYPE    2 // PIC-IO module type
#define STEPMODTYPE  3 // PIC-STEP module type
```

modtype is a byte array of size MAXMOD. modtype[1] stores the module 1 type, modtype[2] stores the module 2 type, etc. modtype[0] is not used. Users can use the macro ModType() to determine the type of module. See *Section 3 Status Packet Description*.

statitems

The array statitems stores which additional status items have been selected to be returned in a status packet for each module. For example, statitems[1] could store that **PIC-SERVO** 1 will return a status packet that includes position data and velocity data. This information is used internally by the library to process status packets. statitems is a byte array of size MAXMOD. statitems[0] is unused, statitems[1] stores the module 1 status items, statitems[2] stores the module 2 status items, etc. The value of statitems is set when NmcDefineStat() is called. The values in statitems should not be changed directly.

numstat

The array numstat stores how many additional status bytes will be returned in each status packet for each module. For example, a **PIC-SERVO** returning position and velocity data will send an additional 6 status bytes of information. This information is used internally by the library to process status packets. numstat is a byte array of size MAXMOD. numstat[0] is unused, numstat[1] stores the additional status bytes needed for module 1, numstat[2] stores the additional status bytes needed for module 2, etc. numstat calculated and set when NmcDefineStat() is called. The values in numstat should never be changed directly.

moddata

The array moddata is used to store the status information returned by each module. moddata is a two dimensional byte array of size [MAXMOD][20] – large enough to store the longest status packet for each module. moddata[0] is used internally by the library to temporarily store each received status packet. After the complete status packet is received, the status data is moved to the moddata[] corresponding to the module. moddata[1] stores status data for module 1, moddata[2] stores status data for module 2, etc. To retrieve status data, use the retrieve status macros defined for each module type. For example, ServoPos() and ServoVel() are used to retrieve **PIC-SERVO** position and velocity status bytes respectively. See *Section 3 Status Packet Description* for information on retrieving status packet data.

1.2 Initialization

Before any NMC module commands are sent, the *SSA-485 Motion Control library* must be initialized by calling NmcInit(). NmcInit() performs the following tasks:

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

Open UART

The *SSA-485 Motion Control Library* communicates with the modules using a UART implemented in software. NmcInit() initializes the UART to the default rate of 19200 baud. After initialization, the application may change the communications rate to 57,600 baud by calling NmcSetBaud().

In order to prevent communications errors due to mismatched baud rates, the baud rate should be set back to 19,200 at the end of the application program (if the rate was changed to 57,600). This will ensure that when the application restarts at 19,200, 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 19,200.

Reset Modules

After the communications UART 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 the `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 its default value of 0xFF.

Assign Addresses

After modules are reset, they are assigned an individual address between 1 and MAXMOD-1 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 first daisy-chained module. 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 in a different order, modules will be assigned different addresses.

Initialize Data Structures

The *SSA-485 Motion Control Library* global data structures are set during initialization. `nummod` is set to the number of modules detected, `modtype` is filled with the type of each module, `statitems` and `numstat` is set to 0 for all modules (no additional status items are sent in the status packets).

1.3 Command Addressing

The bulk of the functions in the *SSA-485 Motion Control Library* 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 MAXMOD-1 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 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 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 Start Motion) 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 its group and it receives a group command (*i.e.*, a command sent to its group address), it will execute the command but *not* send back a status packet. If a module is the leader of its 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.

1.4 Status Packets

Modules send status packets in response to commands. The default status packet contains 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 byte information using the `NmcDefineStat()` and the `NmcReadStat()` 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 in response to library functions `NmcReset()`, `NmcSetBaud()`, 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 in a `moddata[]` array. Application programs can retrieve the `moddata[]` status information using a set of macros 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. The the status data of interest (e.g., position data) has been specified with an `NmcReadStat()` command, any command sent to the module will cause the that status data to be returned. You can also use `NmcReadStat()` to have some item of status data sent back just once. The status data sent back is stored in the array `moddata`. To access the particular data fields stored in this array, you should use one of the macros listed in Section 3.

Function Specification

The *SSA-485 Motions Control Library* functions are grouped into four categories: NMC functions, **PIC-SERVO** functions, **PIC-STEP** functions, and **PIC-IO** functions. The functions are summarized as follows:

NMC Functions

Function	Description
NmcDefineStat	Defines what status data is returned after each command packet sent
NmcInit	Initializes the NMC controller and software serial port
NmcNoOp	No operation – return current status data only
NmcReadStat	Reads the specified status items once
NmcReset	Resets a group of controllers
NmcSendCmd	Low level routine for sending a command packet
NmcSetBaud	Changes baud rate
NmcSetGroupAddress	Sets the group address for a module
NmcSyncSave	Synchronously saves data for multiple modules
NmcSyncStart	Synchronously starts preloaded motions

PIC-SERVO Functions

Function	Description
ServoAddPathPoints	Adds a set of path points for path mode operation
ServoClearBits	Clears the latched status bits
ServoHardReset	Resets module and saves config data in EEPROM
ServoIoCtrl	Sets I/O control mode
ServoLoadTraj	Loads trajectory information
ServoMoveData	Low level routine that moves data from mod-0 buffer into the appropriate mod buffer
ServoNumStat	Low level routine that returns the number of additional status packet bytes
ServoResetPos	Resets position to 0
ServoSetGain	Set the PID gains and other operating parameters
ServoSetHoming	Sets homing mode
ServoStopMotor	Stops motor and enables/disables amplifier

PIC-STEP Functions

Function	Description
StepLoadTraj	Sets load trajectory information
StepMoveData	Low level routine that moves data from the mod-0 buffer into the appropriate mod buffer
StepNumStat	Low level routine that returns the number of additional status packet bytes
StepSetOutputs	Sets the value of the auxiliary output bits
StepResetPos	Resets position to 0
StepSetHoming	Sets the homing mode
StepSetParam	Sets PIC-STEP operating parameters
StepStopMotor	Stops the motor

PIC-I/O Functions

Function	Description
IoMoveData	Low level routine that moves data from the temporary status buffer to the specified module status buffer
IoNumStat	Low level routine that returns the number of additional status packet bytes
IoSetDir	Sets the direction of the I/O pins
IoSetOutputs	Set the value of the output I/O pins
IoSetPWM	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

IoMoveData (Internal Library Function)

Moves PIC-I/O status data from the temporary status buffer to the specified module status buffer.
--

Function Prototype

```
void IoMoveData(byte addr);
```

File Name

```
picio.c
```

Include

```
picio.h
```

Return Value

```
None
```

Arguments

```
addr – Module address  
Module address (1 – (MAXMOD-1))
```

Description

When a module returns status data in response to a command, the *SSA-485 Motion Control Library* stores the status data in the temporary status buffer `moddata[0]`. After being stored in the temporary status buffer, the status data must be moved to the appropriate module status buffer. `IoMoveData()` is used to move **PIC-I/O** status item data from the temporary status buffer `moddata[0]` to the specified module status buffer `moddata[addr]`.

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

Example

To move the **PIC-I/O** status data stored in temporary status buffer to the module 1 status buffer:

```
IoMoveData(1);
```

IoNumStat (Internal Library Function)
--

Returns the number of additional status packet bytes for a PIC-I/O module
--

Function Prototype

```
byte IoNumStat(byte addr);
```

File Name

```
picio.c
```

Include

```
picio.h
```

Return Value

Returns the number of additional status packet bytes for a **PIC-I/O** module.

Arguments

addr – Module address
Module address (1 – (MAXMOD-1))

Description

Modules may be programmed to return additional status information using the `NmcDefineStat()` and `NmcReadStat()` commands. `IoNumStat()` is used to calculate and return the number bytes required to send the additional status packet information for the **PIC-I/O** module with address “addr”.

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

Example

To return the number of additional status packet bytes for **PIC-I/O** module 1:

```
IoNumStat(1);
```

IoSetDir
Sets the direction of the I/O pins

Function Prototype

```
byte IoSetDir(byte addr, int bitdir);
```

File Name

```
picio.c
```

Include

```
picio.h
```

Return Value

```
0 = Successful return  
-1 = Receive status timeout error  
-2 = Status checksum error
```

Arguments

addr – Module address

Module address (1 – (MAXMOD-1))

bitdir – Bit direction

Bits 0-11 of bitdir correspond to I/O pins 1-12. Setting a bit in bitdir to 1 will make the I/O pin an input, clearing a bit in bitdir will make the pin an output (i.e., 1=input, 0=output). Bits 12-15 are ignored.

Description

IoSetDir() sets the directions of the individual I/O bits. 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 module 1 pins 1, 6, and 11 to inputs and the remaining pins to outputs:

```
IoSetDir(1, 0x421);
```

IoSetOutputs
Sets the value of the output I/O pins

Function Prototype

```
byte IoSetOutputs(byte addr, int outval);
```

File Name

```
picio.c
```

Include

```
picio.h
```

Return Value

```
0 = Successful return  
-1 = Receive status timeout error  
-2 = Status checksum error
```

Arguments

addr – Module address

Module address (1 – (MAXMOD-1))

outval – Output bit values

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

Description

IoSetOutputs() immediately sets the values for the I/O bits defined as outputs. If an I/O bit is defined as an input, the corresponding data byte bit value is ignored.

Example

To set module 1 output pins 2, 7, and 12 HI and the remaining output pins LO:

```
IoSetOutputs(1, 0x842);
```

IoSetPWM

Set the PWM output values

Function Prototype

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

File Name

picio.c

Include

picio.h

Return Value

0 = Successful return
-1 = Receive status timeout error
-2 = Status checksum error

Arguments

addr – Module address

Module address (1 – (MAXMOD-1))

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

IoSetPWM() immediately sets the two PWM output values. 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:

```
IoSetOutputs(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

```
byte IoSetSyncOutput(byte addr, int outbits, byte pwm1, byte pwm2);
```

File Name

picio.c

Include

picio.h

Return Value

0 = Successful return
-1 = Receive status timeout error
-2 = Status checksum error

Arguments

addr – Module address

Module address (1 – (MAXMOD-1))

outbits – Output bit values

Bits 0-11 of outval corresponds to I/O pins 1-12. Setting a bit in outval 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 NmcSynchStart() 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 NmcSynchStart() is called):

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

IoSetTimerMode

Sets the timer/counter mode of operation

Function Prototype

```
byte IoSetTimerMode(byte addr, byte tmrmode);
```

File Name

picio.c

Include

picio.h

Return Value

0 = Successful return
-1 = Receive status timeout error
-2 = Status checksum error

Arguments

addr – Module address

Module address (1 – (MAXMOD-1))

tmrmode – Timer Mode

---- Logical OR of the following Timer Mode bits ----

COUNTTMRDISABLE - disable counter/timer

COUNTTMRENABLE - enable counter/timer

TMRMODE - select timer mode

COUNTMODE - select counter mode

PRESCALE_1 - no prescaler (count every event)

PRESCALE_2 - 2:1 prescaler (every other event counted)

PRESCALE_4 - 4:1 prescaler (every 4th event counted)

PRESCALE_8 - 8:1 prescaler (every 8th event counted)

Description

IoSetTimerMode() sets the operating mode of the 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 prescaler 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, COUNTTMRENABLE | COUNTMODE | PRESCALE_2);
```

NmcDefineStat

Defines what status data is returned after each command packet sent

Function Prototype

```
byte NmcDefineStat(byte addr, byte statusitems);
```

File Name

nmccom.c

Include

nmccom.h

Return Value

0 = Successful return
-1 = Receive status timeout error
-2 = Status checksum error

Arguments

addr – Module address

Module address (1 – (MAXMOD-1))

statusitems – Status Items to be returned

---- Logical OR of the following status item bits ----

---- Optional status items common to all modules

SEND_ID

Send the device type and version number (2 bytes)

---- Optional **PIC-IO** status items

IO_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)

IO_SEND_AD1

Send A/D 1 value (1 byte)

IO_SEND_AD2

Send A/D 2 value (1 byte)

IO_SEND_AD3

Send A/D 3 value (1 byte)

IO_SEND_TMR

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

IO_SEND_SYNCH_INPUTS

Send input bit values captured with the NmcSynchSave() command (2 bytes)

IO_SEND_SYNCH_TMR

Send counter/timer value captured with the NmcSynchSave() command (4 bytes)

---- Optional **PIC-SERVO** status items

SERVO_SEND_POS

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

SERVO_SEND_AD

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

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

SERVO_SEND_AUX
Send auxiliary status byte (1 byte – see *Section 3* for bit field description)

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

SERVO_SEND_POSEERROR
Send servo position error (2 bytes)

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

---- Optional **PIC-STEP** status items

STEP_SEND_POS
Send position (4 bytes)

STEP_SEND_AD
Send A/D value (1 byte)

STEP_SEND_TC
Send current initial timer count (2 bytes)

STEP_SEND_INPUTS
Send inputs byte (1 byte – see *Section 3* for bit field description)

STEP_SEND_HOMEPOS
Send home position (4 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 with the logical OR of the desired status items. The selected status items will be sent in the order listed above (**not** the order they are listed in the function call) and will follow the default status byte which is always sent with each packet. 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 3*.

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 NmcReadStat() function instead. You can use the NmcDefineStat() 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 (for example, the SERVO_ defines must be used when NmcDefineStat() is sent to a **PIC-SERVO** module).

Example

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

```
NmcDefineStat(1, SERVO_SEND_POS|SERVO_SEND_POSEERROR);
```

Nmclnit
Initializes the NMC controller and software serial port

Function Prototype

```
void Nmclnit(void);
```

File Name

```
nmccom.c
```

Include

```
nmccom.h
```

Return Value

```
None
```

Arguments

```
None
```

Description

Nmclnit() initializes the NMC controller and establishes NMC communications with external modules. It sets the variable “nummod” to the number of modules found, fills the array “modtype[]” with the type of each module found, and initializes each module to return no status data (other than the default status byte).

Example

To initialize the NMC controller and start NMC communications:

```
Nmclnit();
```

NmcNoOp
No operation – returns the current status data

Function Prototype

```
byte NmcNoOp(byte addr);
```

File Name

```
nmccom.c
```

Include

```
nmccom.h
```

Return Value

```
0 = Successful return  
-1 = Receive status timeout error  
-2 = Status checksum error
```

Arguments

```
addr – Module address  
Module address (1 – (MAXMOD-1))
```

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 ) ;
```

NmcReadStat

Reads the specified status items once

Function Prototype

```
byte NmcReadStat(byte addr, byte statusitems);
```

File Name

nmccom.c

Include

nmccom.h

Return Value

0 = Successful return
-1 = Receive status timeout error
-2 = Status checksum error

Arguments

addr – Module address

Module address (1 – (MAXMOD-1))

statusitems – Status Items to be returned

---- Logical OR of the following status item bits ----

---- Optional status items common to all modules

SEND_ID

Send the device type and version number (2 bytes)

---- Optional **PIC-I/O** status items

IO_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 the input values for I/O bits 9-12 in the lower nibble)

IO_SEND_AD1

Send A/D 1 value (1 byte)

IO_SEND_AD2

Send A/D 2 value (1 byte)

IO_SEND_AD3

Send A/D 3 value (1 byte)

IO_SEND_TMR

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

IO_SEND_SYNCH_INPUTS

Send input bit values captured with the NmcSynchSave() command (2 bytes)

IO_SEND_SYNCH_TMR

Send counter/timer value captured with the NmcSynchSave() command (4 bytes)

---- Optional **PIC-SERVO** status items

SERVO_SEND_POS

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

SERVO_SEND_AD

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

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

SERVO_SEND_AUX
Send auxiliary status byte (1 byte)

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

SERVO_SEND_POSEERROR
Send servo position error (2 bytes)

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

---- Optional **PIC-STEP** status items

STEP_SEND_POS
Send position (4 bytes)

STEP_SEND_AD
Send A/D value (1 byte)

STEP_SEND_TC
Send current initial timer count (2 bytes)

STEP_SEND_INPUTS
Send inputs byte (1 byte)

STEP_SEND_HOMEPOS
Send home position (4 bytes)

Description

NmcReadStat() is used to read specific status data from a module just one time; that is, the status packet returned in response to NmcReadStat() 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 NmcDefineStat() call. For example, the NmcReadStat() 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 sent in the order listed above (**not** the order they are listed in the function call) and will follow the default status byte which is always sent with each packet. 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 3*.

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 (for example, the SERVO_ defines must be used when NmcDefineStat() is sent to a **PIC-SERVO** module).

Example

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

```
NmcReadStat(1, STEP_SEND_AD|STEP_SEND_HOMEPOS);
```

NmcReset
Reset a group of controllers

Function Prototype

```
void NmcReset(byte addr);
```

File Name

```
nmccom.c
```

Include

```
nmccom.h
```

Return Value

```
None
```

Arguments

```
addr – Module address
```

```
Module address (1 – (MAXMOD-1)) or group address (0x80 – 0xFF)
```

Description

NmcReset() resets a group of modules to their 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 universal reset address of 0xFF. (For special reset features of the **PIC-SERVO SC** modules, please see ServoHardReset().)

NOTE: The universal reset address 0xFF is recognized only by new versions of **PIC-SERVO** and **PIC-STEP** modules. It is not recognized in older **PIC-SERVO** and **PIC-STEP** versions, or any **PIC-I/O** versions. To reset modules which do not recognize the universal reset address, it is best to change the group address back to the default of 0xFF and then call NmcReset() using the address 0xFF.

Example

To reset all controllers on the network:

```
NmcReset (0xFF) ;
```

NmcSendCmd (Internal Library Function)

Low level routine for sending a command packet
--

Function Prototype

byte NmcSendCmd(byte addr, byte cmd, byte n, byte *cdata, byte saddr);

File Name

nmccom.c

Include

nmccom.h

Return Value

0 = Successful return
-1 = Receive status timeout error
-2 = Status checksum error

Arguments

addr – Module address

Module address (1 – (MAXMOD-1)) or group address (0x80 – 0xFF)

cmd – Command type

-- commands for all modules --

SETADDR - set module address

DEFINESTAT - define status to be returned

READSTAT - read specified status once

SYNCHSTART - start multiple motions

SETBAUD - set baud rate

SYNCHSAVE - synchronously save data

NOOP - no operation but return defined status

HARDRESET – reset module

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

IOSETDIR – set I/O bit directions

IOSETPWM – set PWM output values

IOSETOUTPUTS – set output bit values

IOSETSYNCHOUT – store output bit and PWM values for later output

IOTMRMODE – set timer/counter mode

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

SERVORESETPOS – reset position counter to 0

SERVOLOADTRAJ – set motion trajectory command data

SERVOSETGAIN – set PID servo filter operating parameters

SERVOSTOPMOTOR – stop motor and enable/disable amplifier

SERVOIOCTRL – set amplifier output mode

SERVOSETHOMING – set homing mode parameters

SERVOCLRBITS – clear the latched status bits

SERVOADDPPOINTS – add path points

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

STEPRESETPOS – reset position counter to 0

STEPLOADTRAJ – set motion parameters

STEPSETPARAM – set control parameters
 STEPSTOPMOTOR – stop motor and enable/disable amplifier
 STEPSETOUTPUTS – clear or set output pins
 STEPSETHOMING - - set homing parameters
 n – Number additional bytes
 Number of additional bytes required for the additional command data
 cdata – Additional data
 String containing additional command data
 saddr – Status data address
 If command is sent to a group address, this is the individual address of the group leader (use 0xFF if no group leader). Otherwise, use the same value for addr and saddr.

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 0xFF, NmcSendCmd() will wait for and store returned module status data.

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 be sent with the command. Additional data (cdata) 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 (saddr) 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, saddr should be set to 0xFF.

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 *SSA-485 Motion Control 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 status data “datastr”:

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

NmcSetBaud

Change the communications baud rate

Function Prototype

```
void NmcSetBaud(byte brate);
```

File Name

nmccom.c

Include

nmccom.h

Return Value

None

Arguments

brate – Baud rate
PICBAUD19200 or PICBAUD57600

Description

NmcSetBaud() sets the communications baud rate. The default baud rate is 19200. Because all control modules on the network must have their baud rates changed at the same time, this command sent to the default group address 0xFF. No status packet is returned. NmcSetBaud() is typically called immediately after NmcInit().

NOTE: If an application program uses NmcSetBaud() to change the baud rate to 57600, then it is recommended that the baud rate be changed back to 19200 before the application exits. This will prevent a baud rate mismatch communication failure when the application is restarted: for example, applications starting at the default rate of 19200 will not be able to communicate with any modules still running at 57600. Alternately, modules may be power cycled to change their baud rate back to the default 19200.

Example

To change the network baud rate to 57600:
`NmcSetBaud(PICBAUD57600);`

NmcSetGroupAddress

Set the group address for a module

Function Prototype

```
byte NmcSetGroupAddress(byte addr, byte groupaddr, byte groupleader);
```

File Name

nmccom.c

Include

nmccom.h

Return Value

0 = Successful return
-1 = Receive status timeout error
-2 = Status checksum error

Arguments

addr – Module address
Module address (1 – (MAXMOD-1))
groupaddr – Group address
Module's group address (0x80 – 0xFF)
groupleader – Group leader
Module's group leader status (1 = group leader, 0 = group member)

Description

NmcSetGroupAddress() is used to set the group address of a module 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. It is also a good idea to 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:

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

NmcSyncSave

Synchronously saves data for multiple modules

Function Prototype

```
byte NmcSynchSave(byte addr, byte leaderaddr);
```

File Name

nmccom.c

Include

nmccom.h

Return Value

0 = Successful return
-1 = Receive status timeout error
-2 = Status checksum error

Arguments

addr – Module address

Module address (1 – (MAXMOD-1)) or group address (0x80 – 0xFF)

leaderaddr – Group leader address

If addr is individual use: leaderaddr = addr

If addr is group use: leaderaddr = group leader's individual address

(If no group leader use: leaderaddr = 0xFF)

Description

NmcSyncSave() 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, NmcSyncSave() is used to synchronously save the current position of the motor in the home position register. When issued to a group of modules, NmcSyncSave() will cause them to store their current positions in their corresponding home position registers. The home position registers 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, NmcSyncSave() causes the current input bit values and the counter/timer value to be synchronously stored in the **PIC-IO**'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 (addr) 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 0xFF 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:

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

NmcSyncStart

Synchronously starts preloaded motions

Function Prototype

```
byte NmcSynchStart(byte addr, byte leaderaddr);
```

File Name

nmccom.c

Include

nmccom.h

Return Value

0 = Successful return
-1 = Receive status timeout error
-2 = Status checksum error

Arguments

addr – Module address

Module address (1 – (MAXMOD-1)) or group address (0x80 – 0xFF)

leaderaddr – Group leader address

If addr is individual use: leaderaddr = addr

If addr is group use: leaderaddr = group leader address

(If no group leader use: leaderaddr = 0xFF)

Description

The NmcSyncSave() 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 NmcSyncStart() 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 NmcSyncStart() command is received. If another Load Trajectory command is received before the NmcSyncStart() command, it will overwrite all of the previous trajectory data.

For **PIC-I/O** modules, the NmcSyncStart() function synchronously sets the output bit values and PWM values previously stored with the IoSetSynchOutput() 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 0xFF. 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:

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

ServoAddPathPoints

Add path points to path point buffer, or start path execution

Function Prototype

```
byte ServoAddPathpoints(byte addr, byte npoints, int *pointlist);
```

File Name

picservo.c

Include

picservo.h

Return Value

0 = Successful return
-1 = Receive status timeout error
-2 = Status checksum error

Arguments

addr – Module address
Module address (1 – (MAXMOD-1)) or group address (0x80 – 0xFF)
npoints – Number of points
Number of points in the list (0 – 7).
If npoints = 0, then path execution will begin.
pointlist – Path points list
Path points list formatted as differential position data.
saddr – Module address of group leader (if sent to group)
Module address (1 – (MAXMOD-1)) if sent to a single module
Group leader's address (1 – (MAXMOD-1)) if sent to a group

Description

ServoAddPathPoints() is used to add points to the path point buffer for path control mode. Up to 7 path points can be added with a single command.

Path point data is specified differentially (i.e., you specify the distance between the previous path point position and the current path point position) using the following format:

30 Hz Path:

P ₁₃ P ₁₂ P ₁₁ P ₁₀ P ₉ P ₈ P ₇ P ₆	P ₅ P ₄ P ₃ P ₂ P ₁ P ₀ F D
<i>most significant byte</i>	<i>least significant byte</i>

60 Hz Path:

P ₁₂ P ₁₁ P ₁₀ P ₉ P ₈ P ₇ P ₆ P ₅	P ₄ P ₃ P ₂ P ₁ P ₀ 0 F D
<i>most significant byte</i>	<i>least significant byte</i>

where P₀ - P₁₃ are the 14 bits of differential position data, F is the path frequency bit (0 = 60 Hz, 1 = 30 Hz) and D is the direction bit (0 = forward, 1 = reverse). Note that for a 60 Hz path, the position data bits are shifted to the left, and bit 2 is always zero.

If “fast path” mode has been selected using the I/O Control command, bit F will be set to 0 for 120 Hz or 1 for 60 Hz, and the path point data will have the following format:

60 Hz Path (fast path mode):	
P ₁₂ P ₁₁ P ₁₀ P ₉ P ₈ P ₇ P ₆ P ₅	P ₄ P ₃ P ₂ P ₁ P ₀ 0 F D
<i>most significant byte</i>	<i>least significant byte</i>
120 Hz Path (fast path mode):	
P ₁₁ P ₁₀ P ₉ P ₈ P ₇ P ₆ P ₅ P ₄	P ₃ P ₂ P ₁ P ₀ 0 0 F D
<i>most significant byte</i>	<i>least significant byte</i>

This compact format minimizes the amount of data sent.

Starting the Path Motion - Sending a ServoAddPathPoints() command with no additional data bytes will cause the path motion to start executing, and 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 NmcReadStat() 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 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 status byte will be cleared.

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 NmcReadStat() command and then adding them together. You will then specify your path points starting from there.

Example

For module address 1 which is currently at position 0, add four path points -100, -201, -303, -406 (the differential data for these path points would be 100, 101, 102, 103 with a direction bit of 1) at 60 Hz:

```
byte plist[8] = { 0x0321, 0x0329, 0x0331, 0x0339 };
                  point1   point 2   point 3   point 4
ServoAddPathPoints(1, 4, plist, 1);
```

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

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

*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
Clear the latched status bits

Function Prototype

```
byte ServoClearBits(byte addr);
```

File Name

```
picservo.c
```

Include

```
picservo.h
```

Return Value

```
0 = Successful return  
-1 = Receive status timeout error  
-2 = Status checksum error
```

Arguments

```
addr – Module address  
Module address (1 – (MAXMOD-1))
```

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);
```

ServoHardReset (Only valid for <i>PIC-SERVO SC</i> – v.10 and greater)

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

Function Prototype

```
byte ServoHardReset(byte addr, byte rmode);
```

File Name

```
picservo.c
```

Include

```
picservo.h
```

Return Value

```
0 = Successful return  
-1 = Receive status timeout error  
-2 = Status checksum error
```

Arguments

addr – Module address

Module address (1 – (MAXMOD-1))

rmode – Reset mode control byte

---- Logical OR of the following reset control bits ----

SAVE_DATA - save config. data in EPROM

RESTORE_ADDR - restore addresses 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 NmcReset() 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.

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 and no others.

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) ;
```

ServoloCtrl

Set amplifier output mode and other I/O options

Function Prototype

```
byte ServoloCtrl(byte addr, byte iomode);
```

File Name

picservo.c

Include

picservo.h

Return Value

0 = Successful return
-1 = Receive status timeout error
-2 = Status checksum error

Arguments

addr – Module address

Module address (1 – (MAXMOD-1))

iomode – 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 for 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**, ServIoCtrl() is used to set a number of operating options. The ServIoCtrl() 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

<i>PIC-SERVO V.4 and earlier</i>	<i>PIC-SERVO V.5</i>	<i>PIC-SERVO SC</i>
SET_OUT1	SET_OUT1	LIMSTOP_OFF
SET_OUT2	SET_OUT2	LIMSTOP_ABRUPT
IO1_IN	IO1_IN	THREE_PHASE
IO2_IN	IO2_IN	ANTIPHASE
.	FAST_PASS	FAST_PATH
.	.	STEP_MODE

Bits IO1_IN, IO2_IN, SET_OUT1 and SET_OUT2 are used in ***PIC-SERVO*** version 5 and earlier 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);
```

ServoLoadTraj

Send motion trajectory command data for PWM, velocity, and trapezoidal profile modes

Function Prototype

```
byte ServoLoadTraj(byte addr, byte tmode, long pos, long vel,  
                  long accel, byte pwm);
```

File Name

picservo.c

Include

picservo.h

Return Value

0 = Successful return
-1 = Receive status timeout error
-2 = Status checksum error

Arguments

addr – Module address

Module address (1 – (MAXMOD-1))

tmode – 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 – use reverse direction

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 NmcSyncStart() 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 NmcSyncStart() 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 NmcSyncStart() 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 NmcSyncStart() command is issued. If you send a new Load Trajectory before sending a NmcSyncStart() 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 3.1 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);
```

ServoMoveData (Internal Library Function)

Moves **PIC-SERVO** status data from the temporary status buffer to the specified module status buffer

Function Prototype

```
byte ServoMoveData(byte addr);
```

File Name

picservo.c

Include

picservo.h

Return Value

None

Arguments

addr – Module address
Mode address (1 – (MAXMOD-1))

Description

When a module returns status data in response to a command, the *SSA-485 Motion Control Library* stores the status data in the temporary status buffer `moddata[0]`. After being stored in the temporary status buffer, the status data must be moved to the appropriate module status buffer. `ServoMoveData()` is used to move **PIC-SERVO** status item data from the temporary status buffer `moddata[0]` to the specified module status buffer `moddata[addr]`.

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

Example

To move the **PIC-SERVO** status data stored in temporary status buffer to the module 1 status buffer:

```
ServoMoveData(1);
```

ServoNumStat (Internal Library Function)

Returns the number of additional status packet bytes for a <i>PIC-SERVO</i> module
--

Function Prototype

```
byte ServoNumStat(byte addr);
```

File Name

```
picservo.c
```

Include

```
picservo.h
```

Return Value

Returns the number of additional status packet bytes for a *PIC-SERVO* module.

Arguments

addr – Module address
Module address (1 – (MAXMOD-1))

Description

Modules may be programmed to return additional status information using the `NmcDefineStat()` and `NmcReadStat()` commands. `ServoNumStat()` is used to calculate and return the number bytes required to send the additional status packet information for the *PIC-SERVO* module with address “addr”.

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

Example

To return the number of additional status packet bytes for *PIC-SERVO* module 1:

```
ServoNumStat (1) ;
```

ServoResetPos
Resets position counter to zero

Function Prototype

```
byte ServoResetPos(byte addr);
```

File Name

```
picservo.c
```

Include

```
picservo.h
```

Return Value

```
0 = Successful return  
-1 = Receive status timeout error  
-2 = Status checksum error
```

Arguments

```
addr – Module address  
Module address (1 – (MAXMOD-1))
```

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 the PIC-SERVO module 1:  
ServoResetPos(1);
```

ServoSetGain

Set PID servo filter operating parameters

Function Prototype

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

File Name

picservo.c

Include

picservo.h

Return Value

0 = Successful return
-1 = Receive status timeout error
-2 = Status checksum error

Arguments

addr – Module address
Module address (1 – (MAXMOD-1))

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.

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

Set homing mode parameters for capturing the home position

Function Prototype

```
byte ServoSetHoming(byte addr, byte hmode);
```

File Name

picservo.c

Include

picservo.h

Return Value

0 = Successful return
-1 = Receive status timeout error
-2 = Status checksum error

Arguments

addr – Module address

Module address (1 – (MAXMOD-1))

hmode – 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_CUR_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 NmcReadStat() 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);
```

ServoStopMotor

Stop the motor and enable or disable the amplifier

Function Prototype

```
byte ServoStopMotor(byte addr, byte smode);
```

File Name

picservo.c

Include

picservo.h

Return Value

0 = Successful return
-1 = Receive status timeout error
-2 = Status checksum error

Arguments

addr – Module address

Module address (1 – (MAXMOD-1))

smode – 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

STOP_HERE - set to stop at position (not currently supported)

ADV_FEATURE - enable features in ver.5 CMC

Description

The Stop Motor command is used to both stop the motor in one of four 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 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* version 5 CMC (this bit applies only to the **PIC-SERVO** version 5). Once the advanced features are enabled, they remain enabled until the **PIC-SERVO** v.5 chip is reset..

Only one of the stopping mode bits 1 - 4 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, to turn off the motor and disable the amplifier, use the following command string:

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

For module address 1, smoothly decelerate to a stop:

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

StepLoadTraj

Send motion trajectory command data for velocity, and trapezoidal profile modes

Function Prototype

```
byte StepLoadTraj(byte addr, byte tmode, long pos, byte speed, byte acc);
```

File Name

picservo.c

Include

picservo.h

Return Value

0 = Successful return
-1 = Receive status timeout error
-2 = Status checksum error

Arguments

addr – Module address
Module address (1 – (MAXMOD-1))

tmode - 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_TC – load initial time count (not supported – clear to 0)
STEP_REVERSE – 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_VEL 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)

Description

All motion parameters are set with this command. This command is also used to select either trapezoidal profile or profiled velocity mode. (Unprofiled velocity modes are not supported.)

Setting one of the LOAD_POS, LOAD_VEL, or LOAD_ACC bits in the tmode will cause the corresponding data to be sent to the **PIC-STEP**. Setting the LOAD_POS bit will also force the **PIC-STEP** into trapezoidal profile mode. If LOAD_POS is not set, profiled velocity mode will be used. Table 2 specifies the allowable transitions from one operating mode to another.

The position data is used as the goal position in trapezoidal profile 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 position value can be either positive or negative and represents 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 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 NmcSyncStart() command is issued.

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_VEL|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_VEL|REVERSE|START_NOW,
              0, 100, 0);
```

Table 2 - PIC-STEP Operating Mode Transition Table

Goal Mode	Starting Mode		
	Stopped	Trap. Profile	Vel. Profile
Stopped	---	Use Stop Command	Use Stop Command
Trap. Profile	<ul style="list-style-type: none"> ✓ Position ○ Velocity ○ Acceleration ✗ Tmr. Count 	Not Allowed	Not Allowed
Vel. Profile	<ul style="list-style-type: none"> ✗ Position ○ Velocity ○ Acceleration ✗ Tmr. Count 	<ul style="list-style-type: none"> ✗ Position ○ Velocity ○ Acceleration ✗ Tmr. Count 	<ul style="list-style-type: none"> ✗ Position ○ Velocity ○ Acceleration ✗ 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.

StepMoveData (Internal Library Function)

Moves **PIC-STEP** status data from the temporary status buffer to the specified module status buffer

Function Prototype

```
byte StepMoveData(byte addr);
```

File Name

```
picstep.c
```

Include

```
picstep.h
```

Return Value

None

Arguments

addr – Module address
Module address (1 – (MAXMOD-1))

Description

When a module returns status data in response to a command, the *SSA-485 Motion Control Library* stores the status data in the temporary status buffer `moddata[0]`. After being stored in the temporary status buffer, the status data must be moved to the appropriate module status buffer. `ServoMoveData()` is used to move **PIC-STEP** status item data from the temporary status buffer `moddata[0]` to the specified module status buffer `moddata[addr]`.

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

Example

To move the **PIC-STEP** status data stored in temporary status buffer to the module 1 status buffer:

```
StepMoveData (1) ;
```

StepNumStat (Internal Library Function)
--

Returns the number of additional status packet bytes for a PIC-STEP module

Function Prototype

```
byte StepNumStat(byte addr);
```

File Name

```
picstep.c
```

Include

```
picstep.h
```

Return Value

Returns the number of additional status packet bytes for a **PIC_STEP** module.

Arguments

addr – Module address
Module address (1 – (MAXMOD-1))

Description

Modules may be programmed to return additional status information using the NmcDefineStat() and NmcReadStat() commands. StepNumStat() is used to calculate and return the number bytes required to send the additional status packet information for the **PIC-STEP** module with address “addr”.

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

Example

To return the number of additional status packet bytes for **PIC-STEP** module 1:

```
StepNumStat (1) ;
```

StepSetOutputs

Sets or clears the general purpose output pins
--

Function Prototype

```
byte StepSetOutputs(byte addr, byte outbyte);
```

File Name

```
picstep.c
```

Include

```
picstep.h
```

Return Value

```
0 = Successful return  
-1 = Receive status timeout error  
-2 = Status checksum error
```

Arguments

addr – Module address

Module address (1 – (MAXMOD-1))

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);
```

StepResetPos
Resets position counter to zero

Function Prototype

```
byte StepResetPos(byte addr);
```

File Name

```
picstep.c
```

Include

```
picstep.h
```

Return Value

```
0 = Successful return  
-1 = Receive status timeout error  
-2 = Status checksum error
```

Arguments

```
addr – Module address  
Module address (1 – (MAXMOD-1))
```

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

```
byte StepSetHoming(byte addr, byte hmode);
```

File Name

picstep.c

Include

picstep.h

Return Value

0 = Successful return
-1 = Receive status timeout error
-2 = Status checksum error

Arguments

addr – Module address

Module address (1 – (MAXMOD-1))

hmode – 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);
```

StepSetParam

Set the **PIC-STEP** operating parameters

Function Prototype

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

File Name

picstep.c

Include

picstep.h

Return Value

0 = Successful return
-1 = Receive status timeout error
-2 = Status checksum error

Arguments

addr – Module address
Module address (1 – (MAXMOD-1))
mode – Operating mode
---- Logical OR of the following load trajectory 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, 0x0a, 0x64, 0x32, 0);
```

StepStopMotor
Stops the motor

Function Prototype

```
byte StepStopMotor(byte addr, byte smode);
```

File Name

```
picstep.c
```

Include

```
picstep.h
```

Return Value

```
0 = Successful return
-1 = Receive status timeout error
-2 = Status checksum error
```

Arguments

```
addr – Module address
      Module address (1 – (MAXMOD-1))
smode – Stop mode
      ---- Logical OR of the following load trajectory mode 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);
```

3. Status Packet Description

The *SSA-485 Motion Control Library* includes a set of macros for accessing individual status fields returned in a status packet. The status fields and their corresponding macros are described below.

3.1 PIC-SERVO Status Packet

Status Packet Macros

Macro	Type	Status Field Description
ServoStatByte(addr)	byte	Status byte. See status byte bit field definitions.
ServoPos(addr)	long	Current position. Signed 32 bit integer.
ServoAD(addr)	byte	A/D value of voltage on CUR_SENSE pin. Range: 0 - 255
ServoVel(addr)	int	Velocity in encoder counts per servo cycle. Signed 16 bit integer.
ServoAux(addr)	byte	Auxiliary status byte. See auxiliary status byte bit field definitions.
ServoHomePos(addr)	long	Home position. Signed 32 bit integer.
ServoModType(addr)	byte	Module Type. 0=PIC-SERVO, 2=PIC-I/O, 3= PIC-STEP.
ServoModVer(addr)	byte	Module Version.
ServoPosErr(addr)	int	Position Error. Signed 16 bit integer.
ServoNPoints(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_ERROR	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	Set when the motor is accelerating, clear when decelerating. This bit has no meaning when stopped or at a constant velocity.
4	SLEW	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	

3.2 PIC-STEP Status Packet

Status Packet Macros

Macro	Type	Status Field Description
StepStatByte(addr)	byte	Status byte. See status byte bit field definitions.
StepPos(addr)	long	Current position. Signed 32 bit integer.
StepAD(addr)	byte	A/D value of voltage on CUR_SENSE pin. Range: 0 - 255
StepTC(addr)	int	Current initial timer count.
StepInputs(addr)	byte	Inputs byte. See inputs byte bit field definitions.
StepHomePos(addr)	long	Home position. Signed 32 bit integer.
StepModType(addr)	byte	Module Type. 0=PIC-SERVO, 2=PIC-I/O, 3= PIC-STEP.
StepModVer(addr)	byte	Module Version.

Status Byte Bit Fields

Bit	Name	Definition
0	MOTOR_MOVING	Motor is moving
1	CKSUM_ERROR	Set if there was a checksum error in the most recently received command packet.
2	AMP_ENABLED	Amplifier enable output signal is HIGH
3	PWR_SENSE_HIGH	The power sense input signal is HIGH
4	AT_SPEED	At commanded speed
5	VELOCITY_MODE	Velocity profile mode
6	TRAPEZOID_MODE	Trapezoidal profile mode
7	HOME_IN_PROG	Homing in progress

Inputs Byte Bit Fields

Bit	Name	Definition
0	ESTOP	Value of the E-Stop input.
1	IN1	Value of IN1 general purpose input.
2	IN2	Value of IN2 general purpose input.
3	LIMIT1	Value of limit switch 1 input.
4	LIMIT2	Value of limit switch 2 input.
5	HOME_SW	Value of home switch input.
6	not used	
7	not used	

3.3 PIC-I/O Status Packet

Status Packet Macros

Macro	Type	Status Field Description
IoStatByte(addr)	byte	Status byte. See status byte bit field definitions.
IoInputs(addr)	int	Input bit values. First byte contains input values for I/O bits 1-8, and the second byte contains input values for I/O bits 9-12 in the lower nibble.
IoAD1(addr)	byte	A/D value on analog input 1.
IoAD2(addr)	byte	A/D value on analog input 2.
IoAD3(addr)	byte	A/D value on analog input 3.
IoTmr(addr)	long	Counter/timer value.
IoModType(addr)	byte	Module Type. 0=PIC-SERVO, 2=PIC-I/O, 3= PIC-STEP.
IoModVer(addr)	byte	Module Version.
IoSynchInputs	int	Input values captured with NmcSychSave() command.
IoSynchTmr	long	Counter/timer value captured with NmcSyncSave() 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	

4.0 Contact Information

Additional information may be found from these sources:

JEFFREY KERR, LLC Web Site

www.jrkerr.com

Application notes, new products, and useful links can be found at this site. Technical support is provided via e-mail with contact information on the web page: www.jrkerr.com/contact.html.

Microchip

www.microchip.com

Visit Microchip's web site for information on their ICD2 development tools and their PIC18Fxxxx series of microcontrollers.