# 1.0 ASCII MODULE sETUP

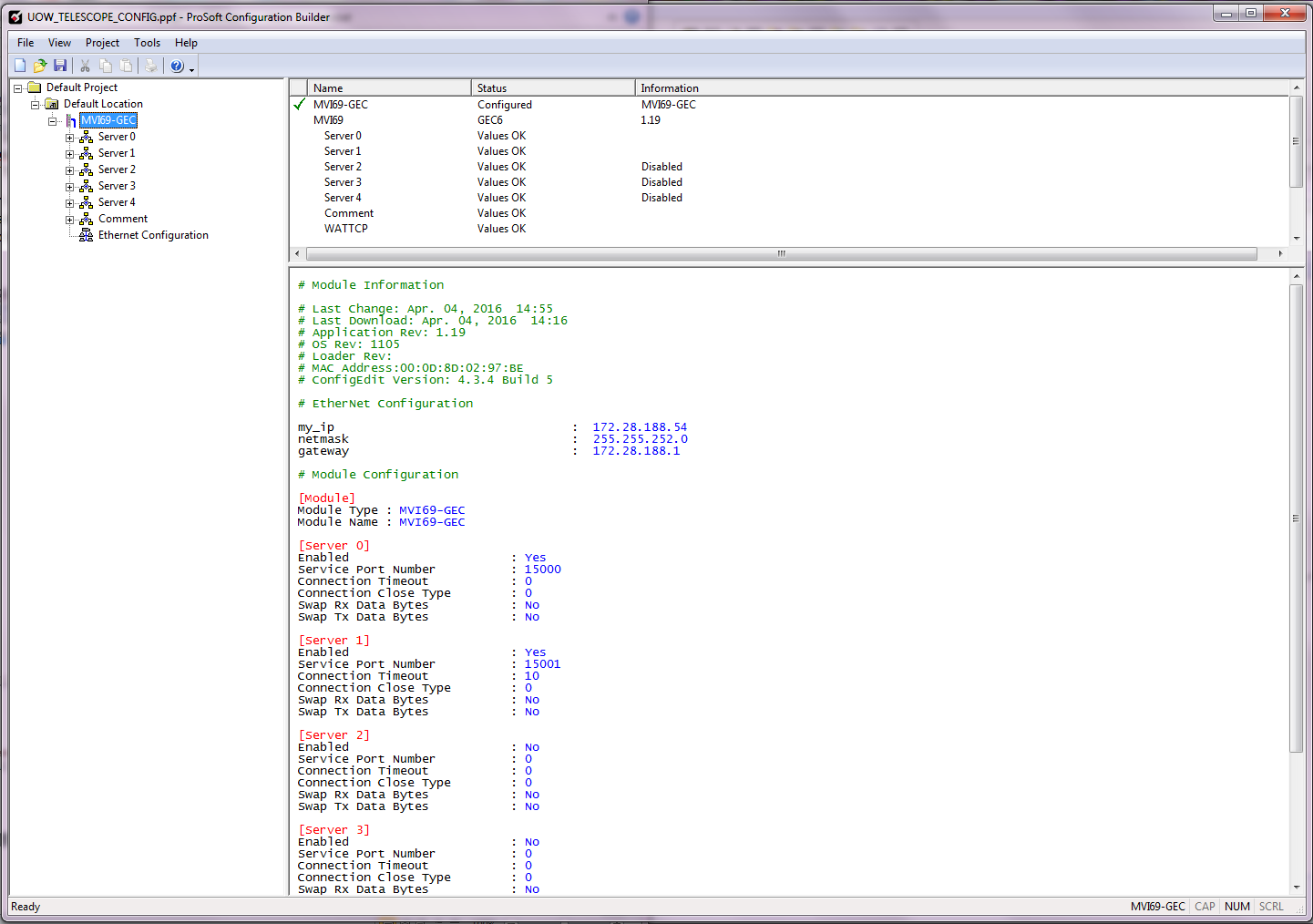
The Prosoft MVI69-GEC ASCII communications module resides in the PAC (Programmable Automation Controller) rack at slot 6. The module must be configured before it is able to communicate to a remote Client. Configuration consists of settings in the module using ProSoft’s “Configuration Builder” software as well as programming in the PAC.

## Module Configuration

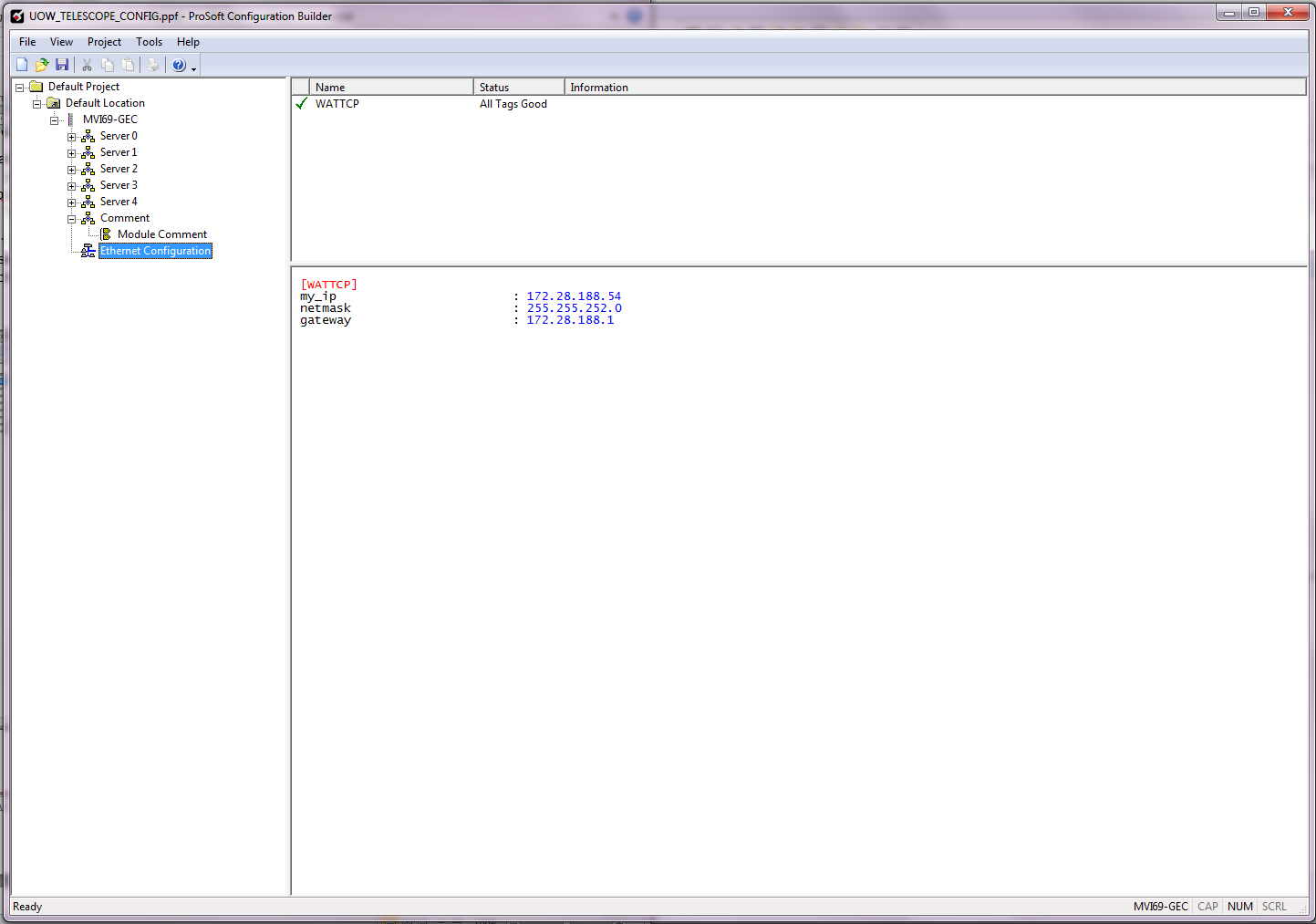
Configuration of up to 5 servers (0 - 4) is possible in the ASCII module. For this application two servers are used (servers 0 and 1). In order to add additional servers the module must be configured using Prosoft’s “Configuration Builder” software and the two serial cables provided with the module. The Configuration Builder software is included with the CD provided with the module.

Figure 1.1-1 shows the configuration of Servers 0 and 1. The IP address and Service Port Number must be set before using the module (Figure 1.1-2). Refer to the MVI69-GEC user manual on the CD (chapters 1 and 2) for instructions on how to configure the module.

**Figure 1.1-1 Prosoft Configuration Builder**



**Figure 1.1-2 Prosoft Configuration Builder**



# 2.0 Client / pac communication structure

All ASCII communications between a Client device and the PAC controller are on two server/client ports (Server 0, port 15000 and Server 1, port 15001). Server 0 is the primary means for the Client to communicate with the PAC. Server 1 is provided as a means to reset the ASCII module should it become necessary. The Client is responsible for establishing the connection to the ASCII card. The ASCII card monitors for a connected Client and incoming messages on the Servers.

All valid commands are echoed back to the Client device from the PAC. When the command execution is finished,”OK” is returned by the PAC to the Client to let the user know that the command is finished. There are no unsolicited responses to the Client.

Each of the Servers have different commands available for the Client to use. The following commands are available to be sent to the PAC from the Client:

**Table 2.0-CLIENT COMMANDS**

|  |  |
| --- | --- |
| **SERVER 0** | **SERVER 1** |
| MOVE | WBOOT |
| STOP | CBOOT |
| SPEED | STATUS |
| UPDATE |  |
| STATUS |  |

## MOVE Command

The MOVE command is issued from the remote Client device and is available only on server 0. A MOVE command issued on Server 1 will cause an ERROR INVALID COMMAND response. The structure of the command is the word MOVE, followed by a space, then the destination position in millimeters. For example:

**MOVE xxx.yyy**

Where xxx.yyy is any positive REAL number. For example, MOVE 10.5 will move the thread ring to 10.5 millimeters from the home or zero position.

When the command is entered at the Client the PAC will echo the command back to the Client to indicate the command was received. Also, the PAC will respond with the current speed setpoint after the command echo, followed by OK.

The PAC will reply with ERROR $$$$$ ($$$$$ is a string description of the error as described below) if the command is invalid or there is a fault condition. Entering a negative destination is an error condition.

If the command is valid, the thread ring will begin to move to the commanded position. Every second the thread ring is moving, (or whatever time interval is entered at the HMI touch screen or via the SPEED command) the PAC will write the current position to the Client until the commanded position is reached. The current position will display in the form: CURRENT\_POSITION xxx.yyy (xxx.yyy is the positon in millimeters – i.e. 1.2). When the move is complete, OK will be returned to the Client.

The following error conditions may be returned by the PAC in response to a MOVE command:

**Table 3.1-1 MOVE Instruction ERRORS**

|  |  |
| --- | --- |
| **ERROR INVALID COMMAND** | A command other than MOVE, STOP, or STATUS has been received by the PAC. |
| **ERROR OUT OF RANGE** | A destination was entered that is outside of the acceptable range of movement. |
| **ERROR BUSY MOVING** | A command was entered while a previous MOVE command is still in progress. |
| **ERROR OVERTRAVEL** | Thread Ring has moved to the point of contacting the over travel limit switch. |

OK is written to the Client after the MOVE command is complete or after an error condition ends the MOVE process.

1. A typical successful move will look something like this (Italics is sent from the Client, Bold is returned by the PAC):

*move 10.5*

**MOVE 10.5**

**SPEED 0.1**

**CURRENT\_POSITION 2.1**

**CURRENT\_POSITION 4.2**

**CURRENT\_POSITION 6.3**

**CURRENT\_POSITION 8.4**

**CURRENT\_POSITION 10.5**

**OK**

1. A typical invalid command will look something like this (Italics is sent from the Client, Bold is returned by the PAC): (for Server 0, any command sent from the client that is not MOVE, STOP, SPEED, UPDATE, or STATUS is invalid. STAT or STATU are ok instead of STATUS since only the first 4 characters are compared. Also UPDA or UPDAT will be the equivalent of UPDATE and SPEE will be valid as well as SPEED). For Server 1, any command sent from the client that is not WBOOT, CBOOT, or STATUS is invalid). WBOO works as well as WBOOT, CBOO is the same as CBOOT.

*mov 10.5*

**MOV 10.5**

**ERROR INVALID COMMAND**

**OK**

1. A MOVE command that is issued with the destination outside of the valid range will look something like this (assume range is 0 – 50 or some number less than 100). (Italics is sent from the Client, Bold is returned by the PAC):

*move 100.5*

**MOVE 100.5**

**SPEED 0.2**

**ERROR OUT OF RANGE**

**OK**

1. Any command issued while a move is in progress will generate a BUSY response and will look something like this (Italics is sent from the Client, Bold is returned by the PAC):

*move 10.5*

**MOVE 10.5**

**SPEED 0.1**

**CURRENT\_POSITION 2.1**

**CURRENT\_POSITION 4.2**

*Status*

**STATUS**

**ERROR BUSY MOVING**

**CURRENT\_POSITION 6.3**

**CURRENT\_POSITION 8.4**

**CURRENT\_POSITION 10.5**

**OK**

1. A MOVE command that is issued that results in the overtravel limit switch activating will look something like this. (Italics is sent from the Client, Bold is returned by the PAC):

*move 10.5*

**MOVE 10.5**

**SPEED 0.2**

**CURRENT\_POSITION 2.1**

**CURRENT\_POSITION 4.2**

**CURRENT\_POSITION 6.3**

**ERROR OVERTRAVEL**

**OK**

## STOP Command

The STOP command is issued from the remote Client to the PAC. This is only a valid command on Server 0. If a STOP command is issued on Server 1 then an ERROR INVALID COMMAND response is sent. There are no parameters to pass with the STOP command. The command is merely STOP. Once the PAC receives the STOP command the MOVE will immediately end and the Thread Ring axis will stop.

When the command is entered at the Client the PAC will echo the command back to the Client to indicate the command was received.

1. A typical successful STOP of a previously entered MOVE will look something like this (Italics is sent from the Client, Bold is returned by the PAC):

*move 10.5*

**MOVE 10.5**

**SPEED 2.5**

**CURRENT\_POSITION 2.1**

**CURRENT\_POSITION 4.2**

*stop*

**STOP**

**OK**

## SPEED Command

The SPEED command is issued from the remote Client to the PAC. SPEED is only a valid command on Server 0. If a SPEED command is issued on Server 1 then an ERROR INVALID COMMAND response is sent. This command will set the speed setpoint for all MOVE commands issued to the thread ring axis. This speed setpoint will be used until a new speed is set with the SPEED command or is entered at the HMI.

When the SPEED command is sent to the PAC, the PAC will echo the command back to the client followed by OK.

1. A typical successful SPEED command will look something like this (Italics is sent from the Client, Bold is returned by the PAC):

*speed 1.5*

**SPEED 1.5**

**OK**

1. Any MOVE command after that will show the new speed as 1.5 mm/sec. For example:

Move 20.0

**MOVE 20.0**

**SPEED 1.5**

**.**

**.**

**.**

**Etc.**

## UPDATE Command

The UPDATE command is issued from the remote Client device to the PAC. UPDATE is only a valid command on Server 0. If an UPDATE command is issued on Server 1 then an ERROR INVALID COMMAND response is sent. This command sets the MOVE positon update time from the PAC to the Client. The structure of the command is the word UPDATE, followed by a space, then the update time in milliseconds. For example:

**UPDATE xxxx**

xxxx is any positive whole number. For example, UPDATE 3000 will set the thread ring axis positon update time to 3000 milliseconds (3 seconds).

When the command is entered at the Client the PAC will echo the command back to the Client to indicate the command was received followed by OK.

The PAC will reply with ERROR $$$$$ ($$$$$ is a string description of the error as described below) if the command is invalid. Entering a negative time is an error condition.

When the command is entered at the Client the PAC will echo the command back to the Client to indicate the command was received. OK will be returned to the Client by the PAC to indicate the command has been completed.

1. A typical successful UPDATE command will look something like this (Italics is sent from TCC, Bold is returned by PAC):

*Update 3500*

**UPDATE 3500**

**OK**

## STATUS Command

The STATUS command is issued from the remote Client to the PAC. This command is available with both Server 0 and Server 1.There are no parameters to pass with the command. The command is merely STATUS. STAT or STATU will also work as a valid command since the PAC only looks at the first four characters to determine a valid command.

When the command is entered at the Client the PAC will echo the command back to the Client to indicate the command was received. Status of the system is reported after the command is received. At the end of the status list, OK will be returned to the Client by the PAC to indicate the command has been completed.

1. A typical successful STATUS command will look something like this (Italics is sent from TCC, Bold is returned by PAC):

*status*

**STATUS**

**THREAD\_RING\_AXIS:**

**\_\_ACTUAL\_POSITION 10.5**

**\_\_TARGET\_POSITION 10.5**

**\_\_DRIVE\_STATUS: OFF**

**\_\_MOTOR\_CURRENT 0.0**

**\_\_DRIVE\_SPEED 0.0**

**\_\_DRIVE\_ACCEL 3**

**\_\_DRIVE\_DECEL 3**

**\_\_MOVE\_RANGE 0.0 – 100.0**

**\_\_HARDWARE\_FAULT 0**

**\_\_INSTRUCTION\_FAULT 0**

**LOCK\_RING\_AXIS:**

**\_\_ACTUAL\_POSITION 0.0**

**\_\_TARGET\_POSITION 0.0**

**\_\_OPEN\_SETPOINT: 150.0**

**\_\_LOCKED\_SETPOINT: 18.0**

**\_\_DRIVE\_STATUS: OFF**

**\_\_MOTOR\_CURRENT 0.0**

**\_\_DRIVE\_SPEED 0.0**

**\_\_DRIVE\_ACCEL 3**

**\_\_DRIVE\_DECEL 3**

**\_\_MOVE\_RANGE 0.0 – 10.0**

**\_\_HARDWARE\_FAULT 0**

**\_\_INSTRUCTION\_FAULT 0**

**\_\_THREADRING\_OVERTRAVEL\_OFF**

**WINCH\_AXIS:**

**\_\_ACTUAL\_POSITION 100.5**

**\_\_TARGET\_POSITION 1000.0**

**\_\_UP\_SETPOINT: 0.0**

**\_\_TO\_CART\_SETPOINT: 1560.0**

**\_\_ON\_CART\_SETPOINT: 1652.0**

**\_\_RELEASE\_SETPOINT: 1695.0**

**\_\_DRIVE\_STATUS: RUNNING**

**\_\_MOTOR\_CURRENT 2.4**

**\_\_DRIVE\_SPEED 20**

**\_\_DRIVE\_ACCEL 3**

**\_\_DRIVE\_DECEL 3**

**\_\_MOVE\_RANGE 0.0 – 2000.0**

**\_\_HARDWARE\_FAULT 0**

**\_\_INSTRUCTION\_FAULT 0**

**SCALE\_1: 45.1**

**SCALE\_2: 46.0**

**SCALE\_3: 45.3**

**CARTRIDGE\_ID: 1**

**\_\_ID\_SW: 0 1 2 3 4 5 6 7 8**

**1 0 0 0 0 0 0 0 0**

**\_\_POS\_SW: 1 2 3**

**0 0 0**

**WINCH\_HOOK\_SENSOR: OFF**

**WINCH\_ENCODER\_1\_POS: 880.1**

**WINCH\_ENCODER\_2\_POS: 881.9**

**WINCH\_ENCODER\_3\_POS: 880.9**

**OK**

Table 2.5-1 lists the STATUS conditions.

**Table 2.5-1 STATUS Conditions**

|  |  |
| --- | --- |
| **\_\_ACTUAL\_POSITION x.y** | Current position of the servo (Thread Ring, Lock Ring, Winch) in millimeters from home. |
| **\_\_TARGET\_POSITION x.y** | Most recent target destination for the axis (Thread Ring, Lock Ring, or Winch). |
| **\_\_OPEN\_SETPOINT: x.y** | Position in millimeters that will determine that the Lock Ring is OPEN. |
| **\_\_LOCKED\_SETPOINT: x.y** | Position in millimeters that will determine that the Lock Ring is LOCKED. |
| **\_\_UP\_SETPOINT: x.y** | Position in millimeters that will determine that the Winch is in the UP position. |
| **\_\_TO\_CART\_SETPOINT: x.y** | Position in millimeters that will determine that the Winch is in the To-Cart position. |
| **\_\_ON\_CART\_SETPOINT: x.y** | Position in millimeters that will determine that the Winch is in the On-Cart position. |
| **\_\_RELEASE\_SETPOINT: x.y** | Position in millimeters that will determine that the Winch is in the RELEASE position. |
| **\_\_DRIVE\_STATUS ON or RUNNING** | Current operating status of the axis (Thread Ring, Lock Ring, Winch): OFF or RUNNING. |
| **\_\_MOTOR\_CURRENT x.y** | Motor current of the axis (Thread Ring, Lock Ring, Winch) at time STATUS command is issued. |
| **\_\_DRIVE\_SPEED x.y** | Current configured speed of the servo drive (Thread Ring, Lock Ring, Winch) in mm/sec. |
| **\_\_DRIVE\_ACCEL x.y** | Current configured acceleration rate of the servo drive (Thread Ring, Lock Ring, Winch) in mm/sec2. |
| **\_\_DRIVE\_DECEL x.y** | Current configured deceleration rate of the servo drive (Thread Ring, Lock Ring, Winch) in mm/sec2. |
| **\_\_MOVE\_RANGE x.y – a.b** | Valid range of axis (Thread Ring, Lock Ring, Winch) movement in millimeters. |
| **\_\_HARDWARE\_FAULT xxx** | Decimal code of hardware fault (if any) for the axis (Thread Ring, Lock Ring, Winch). Zero is no fault. Refer to manufacturer’s documentation |
| **\_\_INSTRUCTION\_FAULT yyy** | Decimal code of the software instruction fault (if any) for the axis (Thread Ring, Lock Ring, Winch). Zero is no fault. Refer to manufacturer’s documentation |
| **SCALE\_1: x.y** | Current weight on Scale #1. |
| **SCALE\_2: x.y** | Current weight on Scale #2. |
| **SCALE\_3: x.y** | Current weight on Scale #3. |
| **CARTRIDGE\_ID: xxx** | Numerical value corresponding to the nine proximity switches that are set by targets on the cartridge. |
| **\_\_ID\_SW: 0 1 2 3 4 5 6 7 8**  **1 0 0 0 0 0 0 0 0** | Cartridge ID switch status: a 1 or 0 beneath the switch number indicates on and off respectively. |
| **\_\_POS\_SW: 1 2 3**  **1 0 0** | Cartridge Position switch status: a 1 or 0 beneath the switch number indicates on and off respectively. |
| **WINCH\_HOOK\_SENSOR: ON or OFF** | Indicates whether the proximity switch senses the winch hook or not. ON or OFF indicating switch status. |
| **WINCH\_ENCODER\_1\_POS: x.y** | Value representing the position of Linear Encoder #1 for winch position. |
| **WINCH\_ENCODER\_2\_POS: x.y** | Value representing the position of Linear Encoder #2 for winch position. |
| **WINCH\_ENCODER\_3\_POS: x.y** | Value representing the position of Linear Encoder #3 for winch position. |

## WBOOT Command

The WBOOT command (Warm Boot) is issued from the remote Client to the PAC only on Server 1. There are no parameters to pass with the command. The command is merely WBOOT. WBOO will also work as a valid command since the PAC only looks at the first four characters to determine a valid command. Issuing the WBOOT command on Server 0 will cause an ERROR INVALID COMMAND response.

When the command is entered at the Client the PAC will echo the command back to the Client to indicate the command was received. A warm boot of the PAC ASCII module will be initiated. This will have the effect of closing all connections to the module and resetting counters internal to the module. OK will be returned to the Client by the PAC to indicate the command has been completed.

1. A typical successful WBOOT command will look something like this (Italics is sent from TCC, Bold is returned by PAC):

*wboot*

**WBOOT**

**OK**

The client is disconnected immediately after the PAC executes the WBOOT command.

## CBOOT Command

The CBOOT command (Cold Boot) is issued from the remote Client to the PAC only on Server 1. There are no parameters to pass with the command. The command is merely CBOOT. CBOO will also work as a valid command since the PAC only looks at the first four characters to determine a valid command. Issuing the CBOOT command on Server 0 will cause an ERROR INVALID COMMAND response.

When the command is entered at the Client the PAC will echo the command back to the Client to indicate the command was received. A cold boot of the PAC ASCII module will be initiated. This will have the effect of closing all connections to the module and resetting counters internal to the module. OK will be returned to the Client by the PAC to indicate the command has been completed.

1. A typical successful CBOOT command will look something like this (Italics is sent from TCC, Bold is returned by PAC):

*cboot*

**CBOOT**

**OK**

The client is disconnected immediately after the PAC executes the CBOOT command.