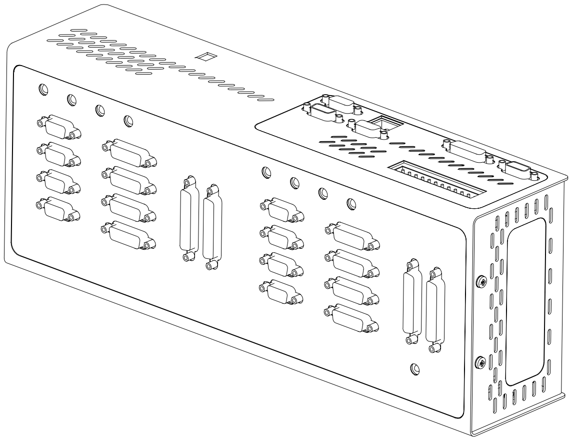
6K

USER GUIDE ADDENDUM

This is an addendum to the 6K product user documentation, and covers these topics:



Parker
Automation

Ship Kit

This material supersedes the information on page 3 of the *6K Series Hardware Installation Guide* (88-017547-01A).

A ship kit is automatically included with the standard product part number (i.e., 6K2, 6K4, 6K6, or 6K8). If you do not need a ship kit, you may order the "no kit" (-NK) option (i.e., 6K2-NK, 6K4-NK, 6K6-NK, or 6K8-NK).

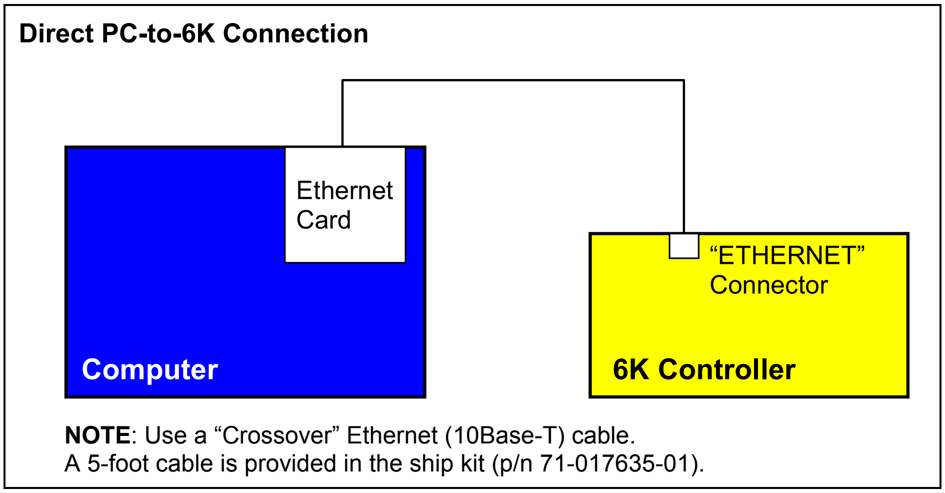
The corrected ship kit table is provided below:

Part Name	Part Number
One of the following 6K products:	
6K2 two-axis controller with ship kit (see 6K-KIT list below)	6K2
6K2 without ship kit	6K2-NK
6K4 four-axis controller with ship kit (see 6K-KIT list below)	6K4
6K4 without ship kit	6K4-NK
6K6 six-axis controller with ship kit (see 6K-KIT list below)	6K6
6K6 without ship kit	6K6-NK
6K8 eight-axis controller with ship kit (see 6K-KIT list below)	6K8
6K8 without ship kit	6K8-NK
Ship kit items (6K-KIT): *	
6K Series Hardware Installation Guide	88-017547-01
6K Series Command Reference	88-017136-01
6K Series Programmer's Guide	88-017137-01
Motion Planner CD-ROM	95-017633-01
Ethernet cable (5-foot, RJ-45, cross-over)	71-017635-01
Peel-and-stick labels for onboard I/O cables	87-017636-01

* The panel mounting kit (part number 74-018177-01), which includes two mounting brackets and four screws (6-32 x ¼), is included with all 6K shipments, independent of the 6K-KIT.

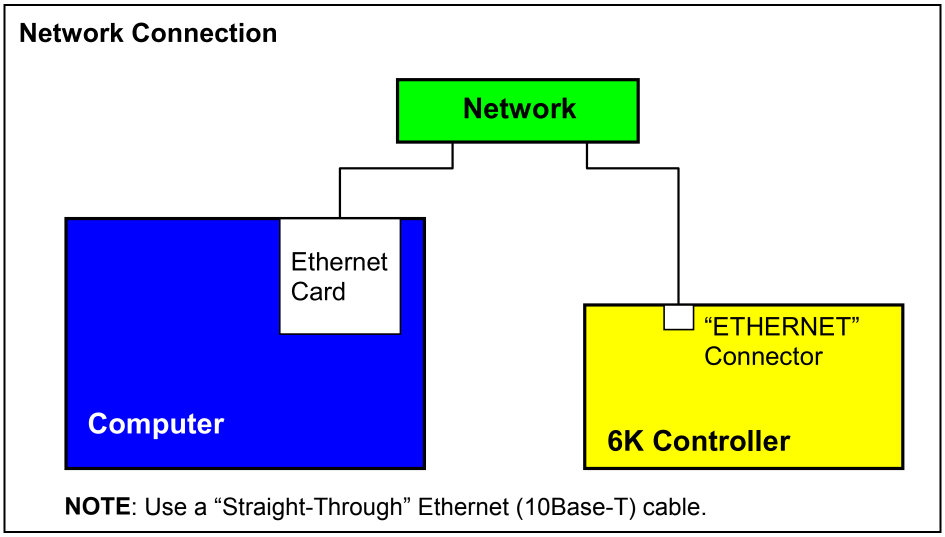
Ethernet Configuration

This material supplements the information on page 25 of the *6K Series Hardware Installation Guide* (88-017547-01A).



NOTE: Use a "Crossover" Ethernet (10Base-T) cable.
A 5-foot cable is provided in the ship kit (p/n 71-017635-01).

Use Procedure A
(page 4)



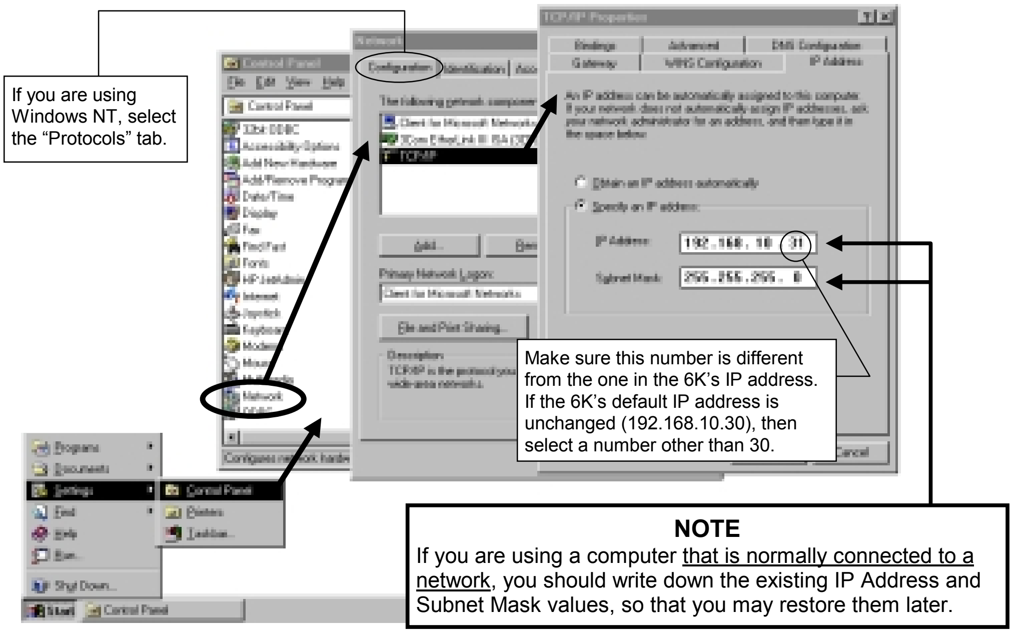
NOTE: Use a "Straight-Through" Ethernet (10Base-T) cable.

Use Procedure B
(page 5)

Procedure A *(Direct PC-to-6K Connection)*

1. Install your Ethernet card and configure it for TCP/IP protocol. Refer to your Ethernet card's user documentation for instructions.

2. (**see illustration below**) Configure your Ethernet card's TCP/IP properties so that your computer can communicate directly to the 6K controller.
   a. Access the Control Panels directory.
   b. Open the Network control panel.
   c. In the Network control dialog, select the Configuration tab (95/98) or the Protocols tab (NT) and double-click the TCP/IP network item to view the TCP/IP Properties dialog.
   d. In the TCP/IP Properties dialog, select the IP Address tab, **see note below**, select "Specify an IP Address", type in 192.168.10.31 in the "IP Address" field, and type in 255.255.255.0 in the "Subnet Mask" field.
   e. Click the OK buttons in both dialogs to finish setting up your computer's IP address.



IP Address: 192.168.10.31

Subnet Mask: 255.255.255.0

If you are using Windows NT, select the "Protocols" tab.

Make sure this number is different from the one in the 6K's IP address. If the 6K's default IP address is unchanged (192.168.10.30), then select a number other than 30.

NOTE
If you are using a computer that is normally connected to a network, you should write down the existing IP Address and Subnet Mask values, so that you may restore them later.

3. Establish an RS-232 communication link between the 6K and your computer (connect to the 6K's "RS-232" connector according to the instructions in the *6K Series Hardware Installation Guide*).

4. Install Motion Planner on your computer, and launch Motion Planner. Click on the Terminal tab to view the terminal emulator.

5. In the Terminal window, click on the button to view the Communications Settings dialog. Select the Port tab and select the COM port that is connected to the 6K's "RS-232" connector (see Step 3 above). Click OK.

6. Enable Ethernet communication: type the `NTFEN2` command and press ENTER.

7. Follow the ARP -S Static Mapping procedure on page 5.

8. Connect the 6K Controller to your computer using a cross-over 10Base-T cable (5-foot cable provided in ship kit).

9. In Motion Planner's Terminal window, click on the button to view the Communications Settings dialog. Select the Port tab, select "Network" and type the IP address (192.168.10.30) in the text field. Click OK.

10. You may now communicate to the 6K controller over the Ethernet interface. Reminder: You cannot communicate to the 6K with simultaneous transmissions over both the "ETHERNET" and "RS-232" (`PORT1`) connections.

Ethernet Connection Status LEDs (located on the RJ-45 "ETHERNET" connector):
- Green LED turns on to indicate the Ethernet physical connection is OK.
- Yellow LED flashes to indicate the 6K is transmitting over the Ethernet interface.

Procedure B *(Network Connection)*

1. Connect the 6K Controller to your network.
2. Have your network administrator assign a 6K IP address and subnet mask. The factory default 6K IP address is 192.168.10.30; the default mask is Class C.

   If the default address and mask are not compatible with your network, you may change them with the `NTADDR` and `NTMASK` commands, respectively (see *6K Series Command Reference* for details on the `NTADDR` and `NTMASK` commands). To ascertain the 6K's Mac address, use the `TNTMAC` command. The `NTADDR`, `NTMASK` and `TNTMAC` commands may be sent to the 6K controller over an RS-232 interface (see Steps 3-5). **NOTE**: If you change the 6K's IP address or mask, the changes will not take affect until you cycle power or issue a `RESET` command.
3. Establish an RS-232 communication link between the 6K and your computer (connect to the 6K's "RS-232" connector according to the instructions in the *6K Series Hardware Installation Guide*).
4. Install Motion Planner on your computer, and launch Motion Planner. Click on the Terminal tab to view the terminal emulator.
5. In the Terminal window, click on the button to view the Communications Settings dialog. Select the Port tab and select the COM port that is connected to the 6K's "RS-232" connector (see Step 3 above). Click OK.
6. Enable Ethernet communication: type the `NTFEN2` command and press ENTER.
7. Follow the ARP -S Static Mapping procedure on page 5.
8. In the Terminal window, click on the button to view the Communications Settings dialog. Select the Port tab, select "Network" and type the IP address in the text field (use 192.168.10.30 unless you changed it with the `NTADDR` command). Click OK.
9. You may now communicate to the 6K controller over the Ethernet interface. Reminder: You cannot communicate to the 6K with simultaneous transmissions over both the "ETHERNET" and "RS-232" (`PORT1`) connections.

Ethernet Connection Status LEDs (located on the RJ-45 "ETHERNET" connector):

- Green LED turns on to indicate the Ethernet physical connection is OK.
- Yellow LED flashes to indicate the 6K is transmitting over the Ethernet interface.

ARP –S Static Mapping Procedure

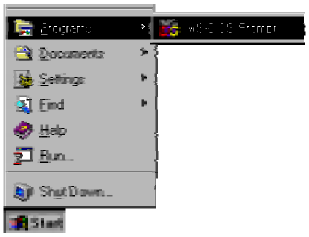
If you are using the `NTFEN2` setting, follow this procedure to use ARP -S to statically map the 6K's Ethernet MAC address to its IP address.

Static mapping eliminates the need for the PC to ARP the 6K controller. Resolving the MAC address to an IP address is known as *ARPing*. ARPing can be done automatically (dynamically) by the TCP/IP stack. Alternatively, the user can define the mapping by statically mapping the MAC address to the IP address. Static mapping has the benefit that it can reduce communication overhead.

NOTE: Static mapping is necessary only if you are using the `NTFEN2` setting.

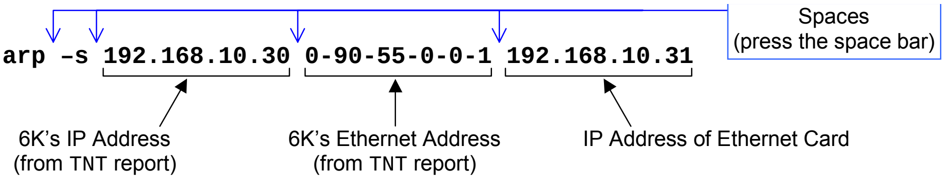
1. If you have not already done so, follow the Ethernet connection instructions in the *6K Hardware Installation Guide*. Then, follow the appropriate Ethernet configuration procedure — start on page 3.

2. In Motion Planner's Terminal window, type `TNT` and press ENTER. The response includes the 6K IP address, and the 6K Ethernet address value in hex (this is also known as the "MAC" address). Write down the IP address and the Ethernet address (hex value) for later use in the procedure below.



3. Start a DOS window. The typical method to start a DOS window is to select MS-DOS Prompt from the Start/Programs menu (see illustration at left).

4. At the DOS prompt, type the **`arp -s`** command (see example below) and press ENTER.



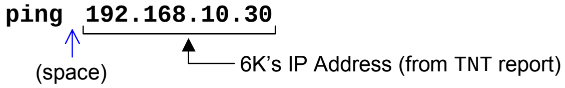
```
arp -s 192.168.10.30 0-90-55-0-0-1 192.168.10.31
```

5. To verify the mapped addresses, type the **`arp -a`** command and press ENTER.

If you receive the response "`No ARP Entries Found`":

a. Switch to the Motion Planner Terminal window, type `NTFEN1` and press ENTER, then type `RESET` and press ENTER.

b. Switch to the DOS window, type the **`ping`** command and press ENTER:



```
ping 192.168.10.30
```

If your PC responds with "`Request Timed Out`", check your Ethernet wiring and IP address setting.

c. Repeat the **`arp -s`** command as instructed above. Use **`arp -a`** to verify.

d. Switch to the Motion Planner Terminal window, type `NTFEN2` and press ENTER, then type `RESET` and press ENTER.

6. (OPTIONAL) Automate the **`arp -s`** static mapping command. *This allows your PC to automatically perform the static mapping when it is booted; otherwise, you will have to manually perform static mapping (Steps 2-5 above) every time you boot your PC.*

   - Windows 95/98: Add the **`arp -s`** command to the Autoexec.bat file.
   - Windows NT: Create a batch file that contains the **`arp -s`** command. Save the file (name the file "6KARP.BAT") to the root directory on the C drive. Using Windows Explorer, locate the 6KARP.BAT file, create a shortcut, then cut and paste the shortcut into the StartUp directory. Windows NT has several StartUp directories to accommodate various user configurations. We recommend using the Administrators or All Users locations. For example, you can paste the shortcut into the WinNt\Profiles\AllUsers\StartMenu\Programs\StartUp directory, allowing all users to statically map the IP and Mac addresses whenever the PC is booted.

7. Return to procedure A or B (page 4 or 5) to finish the configuration process.

Servo Tuning Procedure

This material supersedes the information on page 51 of the *6K Series Hardware Installation Guide* (88-017547-01A) and page 70 of the *6K Series Programmer's Guide* (88-017137-01A):

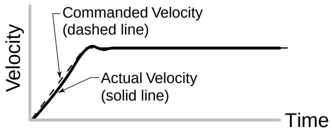
To assure optimum performance you should tune your servo system. The goal of the tuning process is to define the gain settings, servo performance, and feedback setup (see command list below) that you can incorporate into your application program. (Typically, these commands are placed into a setup program). Servo tuning should be performed as part of the application *setup process*, as described below

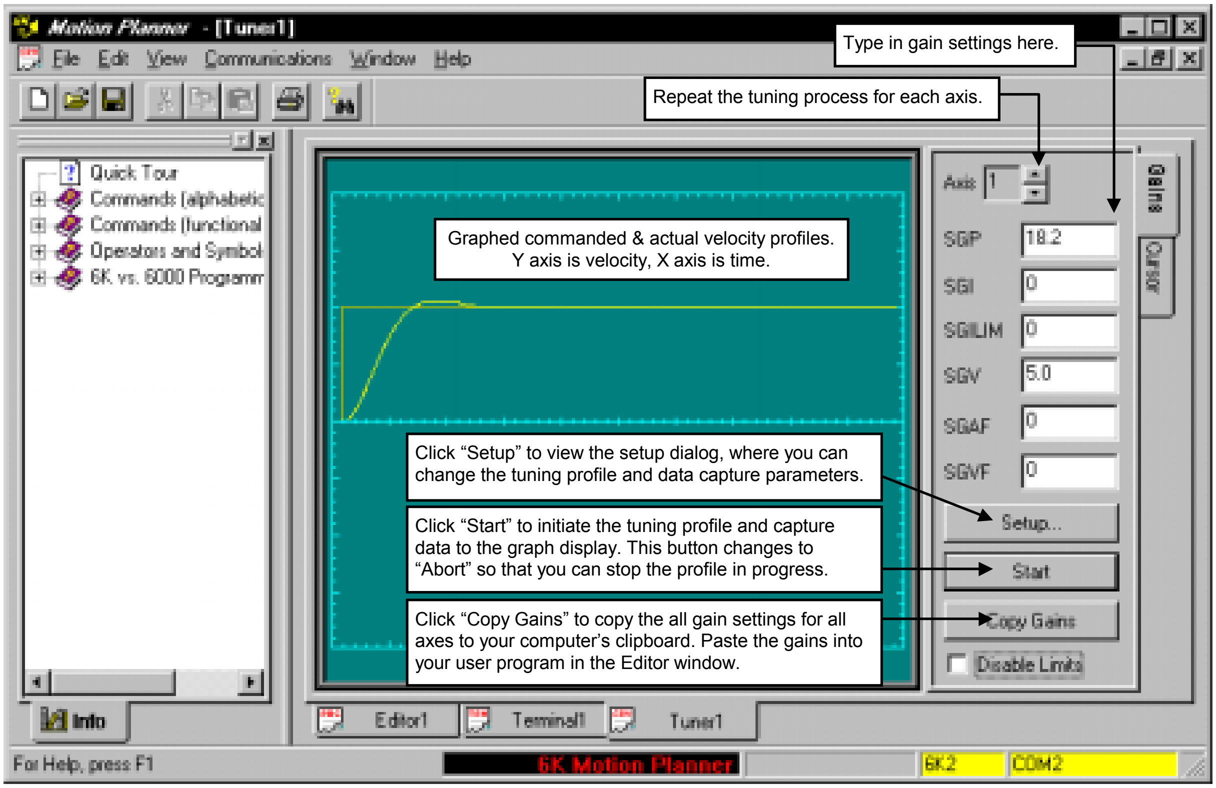
To tune your servo system: (4-step process)

1. After you launch Motion Planner, you will see the Editor window. Click on the "Tuner" window tab to bring the servo tuning utility to the front.
2. Click the "Start" button to send the pre-programmed step output to the drive. Notice that the graph display draws the commanded and actual velocity profiles so that you can graphically tune your servo system.

   Optimize the proportional (SGP) and velocity (SGV) values by iteratively changing gains and viewing the results on the graph display. The object is to achieve a 1st order response (minimal overshoot and close position tracking). The typical process is illustrated in the flow diagram on the next page.

1st Order Response:





SGP 18.2
SGI 0
SGILIM 0
SGV 5.0
SGAF 0
SGVF 0

Type in gain settings here.

Repeat the tuning process for each axis.

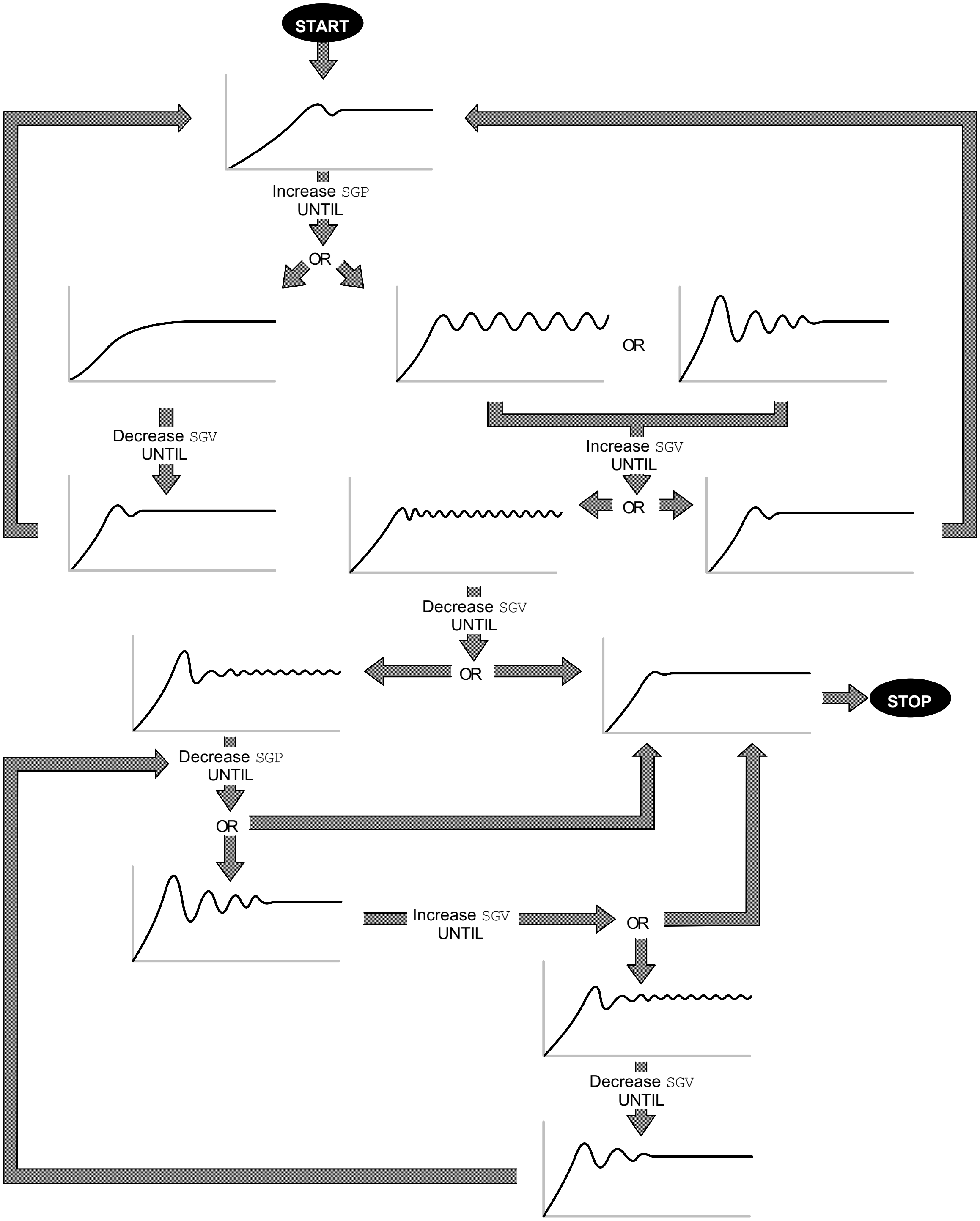
Graphed commanded & actual velocity profiles. Y axis is velocity, X axis is time.

Click "Setup" to view the setup dialog, where you can change the tuning profile and data capture parameters.

Click "Start" to initiate the tuning profile and capture data to the graph display. This button changes to "Abort" so that you can stop the profile in progress.

Click "Copy Gains" to copy the all gain settings for all axes to your computer's clipboard. Paste the gains into your user program in the Editor window.

Basic Tuning Process



3. Repeat step 2 for each axis.

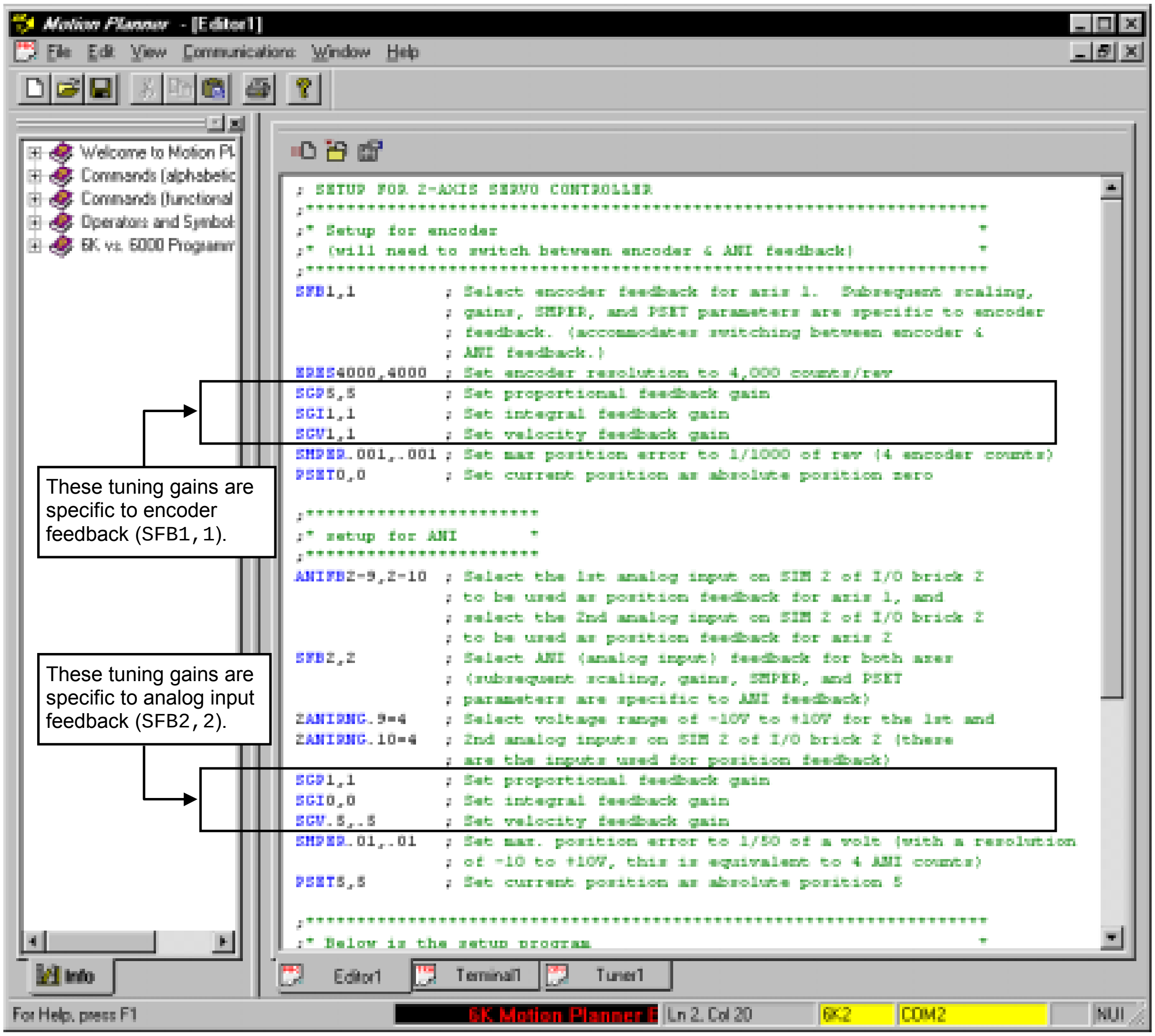
4. When you have determined which tuning gains are best for your application's performance, insert the gain commands into your setup program: **(refer also to the illustration below)**

   a. Click the "Copy Gains" to Clipboard button. This copies the gain commands to your computer's clipboard.
   b. Click the "Editor" tab to bring the program editor to the front.
   c. Place the cursor at the location in your program where you wish to insert the gain commands (see NOTE below).
   d. Paste the gain commands at the location of the cursor. Use the <ctrl>V keystroke shortcut or use the "Paste" command from the "Edit" pull-down menu.

NOTE

The tuning gains are specific to the feedback source selection in effect at the time the gain commands are executed. The factory default feedback source (selected with the SFB command) is encoder feedback. The illustration below demonstrates where to insert the gain commands relative to the SFB command.

If your application requires you to switch between feedback sources for the same axis, then for each feedback source you must select the source with the SFB command and then execute the tuning gain commands relevant to the feedback source (an example is provided in the illustration below).



```
; SETUP FOR 2-AXIS SERVO CONTROLLER
;*****************************************************************
;* Setup for encoder                                             *
;* (will need to switch between encoder & ANI feedback)          *
;*****************************************************************
SFB1,1          ; Select encoder feedback for axis 1.  Subsequent scaling,
                ; gains, SMPER, and PSET parameters are specific to encoder
                ; feedback. (accommodates switching between encoder &
                ; ANI feedback.)
ERES4000,4000   ; Set encoder resolution to 4,000 counts/rev
SGP5,5          ; Set proportional feedback gain
SGI1,1          ; Set integral feedback gain
SGV1,1          ; Set velocity feedback gain
SMPER.001,.001  ; Set max position error to 1/1000 of rev (4 encoder counts)
PSET0,0         ; Set current position as absolute position zero

;***********************
;* setup for ANI       *
;***********************
ANIFB2-9,2-10   ; Select the 1st analog input on SIM 2 of I/O brick 2
                ; to be used as position feedback for axis 1, and
                ; select the 2nd analog input on SIM 2 of I/O brick 2
                ; to be used as position feedback for axis 2
SFB2,2          ; Select ANI (analog input) feedback for both axes
                ; (subsequent scaling, gains, SMPER, and PSET
                ; parameters are specific to ANI feedback)
2ANIRNG.9=4     ; Select voltage range of -10V to +10V for the 1st and
2ANIRNG.10=4    ; 2nd analog inputs on SIM 2 of I/O brick 2 (these
                ; are the inputs used for position feedback)
SGP1,1          ; Set proportional feedback gain
SGI0,0          ; Set integral feedback gain
SGV.5,.5        ; Set velocity feedback gain
SMPER.01,.01    ; Set max. position error to 1/50 of a volt (with a resolution
                ; of -10 to +10V, this is equivalent to 4 ANI counts)
PSET5,5         ; Set current position as absolute position 5

;*****************************************************************
;* Below is the setup program                                    *
```

These tuning gains are specific to encoder feedback (SFB1,1).

These tuning gains are specific to analog input feedback (SFB2,2).

Tuning-Related Commands (see *6K Series Command Reference* for details)

Tuning Gains:

`SGP`...........Sets the proportional gain in the **P**IV&F servo algorithm.

`SGI`...........Sets the integral gain in the P**I**V&F servo algorithm.

`SGV`...........Sets the velocity gain in the PI**V**&F servo algorithm.

`SGAF`.........Sets the acceleration feedforward gain in the PIV&$\mathbf{F_a}$ algorithm.

`SGVF`.........Sets the velocity feedforward gain in the PIV&$\mathbf{F_v}$ algorithm.

`SGILIM` ....Sets a limit on the correctional control signal that results from the integral gain action trying to compensate for a position error that persists too long.

`SGENB`.......Enables a previously-saved set of PIV&F gains. A set of gains (specific to the current feedback source selected with the `SFB` command) is saved using the `SGSET` command.

`SGSET`.......Saves the presently-defined set of PIV&F gains as a *gain set* (specific to the current feedback source on each axis). Up to 5 gain sets can be saved and enabled at any point in a move profile, allowing different gains at different points in the profile.

Feedback Setup:

`SFB` .... Selects the servo feedback device (encoder or analog input). To use analog input feedback, you must first use the `ANIFB` command to configure the targeted analog input to be used for feedback.
IMPORTANT: Parameters for scaling, tuning gains, max. position error (`SMPER`), and position offset (`PSET`) are specific to the feedback device selected (with the `SFB` command) at the time the parameters are entered (see programming examples in the 6K *Programmer's Guide*).

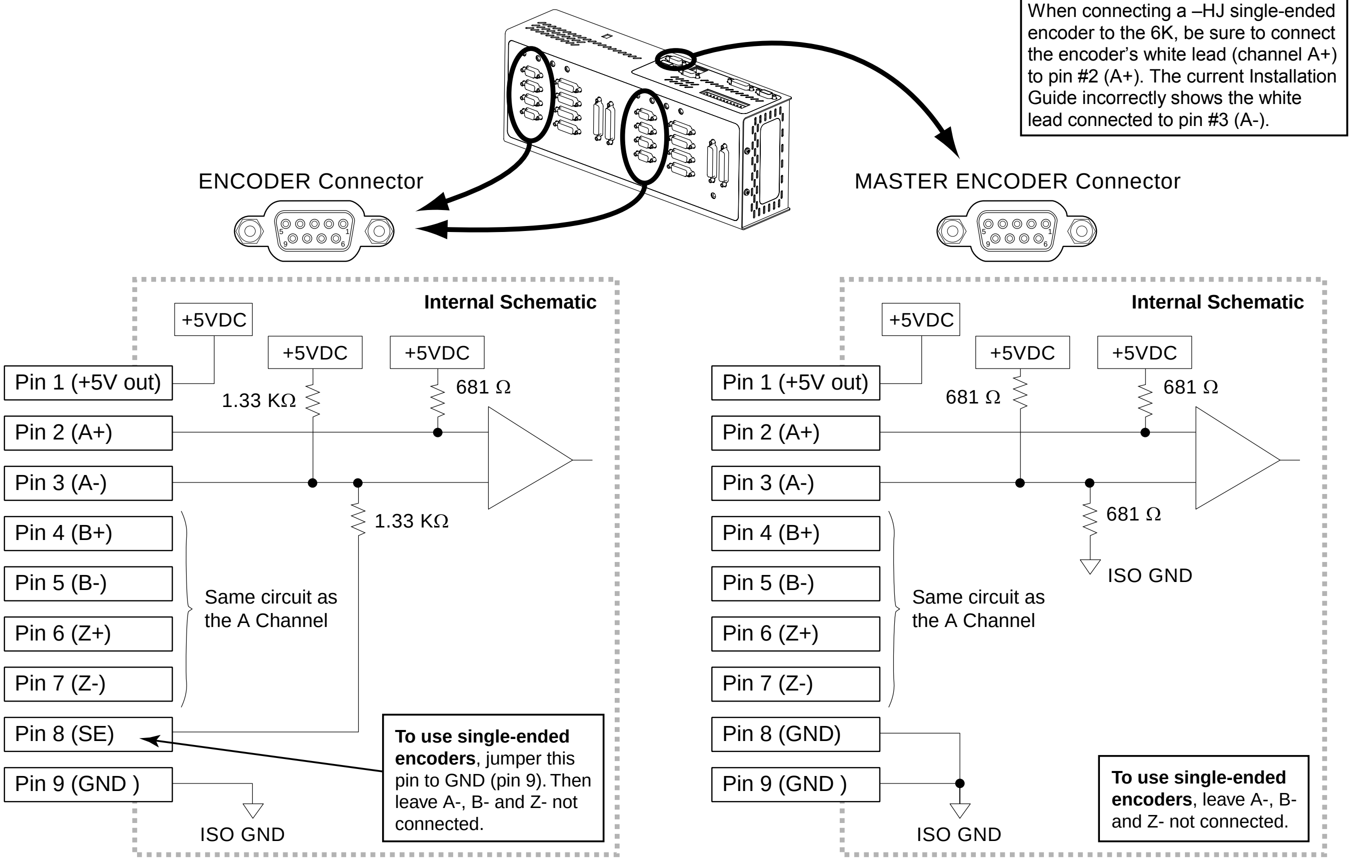
`ERES` .... Encoder resolution.

`SMPER` ...... Sets the maximum allowable error between the commanded position and the actual position as measured by the feedback device. If the error exceeds this limit, the controller activates the Shutdown output and sets the DAC output to zero (plus any `SOFFS` offset). If there is no offset, the motor will freewheel to a stop. You can enable the `ERROR` command to continually check for this error condition (`ERROR.12-1`), and when it occurs to branch to a programmed response defined in the `ERRORP` program.

Encoder Schematic Corrections

This material supersedes the schematic on page 19 of the *6K Series Hardware Installation Guide* (88-017547-01A).

Additional Correction:
When connecting a –HJ single-ended encoder to the 6K, be sure to connect the encoder's white lead (channel A+) to pin #2 (A+). The current Installation Guide incorrectly shows the white lead connected to pin #3 (A-).



To use single-ended encoders, jumper this pin to GND (pin 9). Then leave A-, B- and Z- not connected.

To use single-ended encoders, leave A-, B- and Z- not connected.

Communications Server (COM6SRVR)

This material supersedes the information on pages 35-40 of the *6K Series Programmer's Guide* (88-017137-01A):

Programming Samples

Examples may be installed with Motion Planner and are located in the Motion Planner directory (\Motion Planner\Samples):

- Visual Basic 5.0 sample
- Visual C++ sample
- Delphi 3.0 sample
- Panel (.pnl) examples in VBScript that you can edit in Motion Planner's PanelMaker utility.

NOTE: The samples are not installed as part of the "typical" installation; use the "custom" installation option.

The 6K Communications Server (COM6SRVR.EXE) is an 32-bit OLE automation server which facilitates communications between 6K controllers and PC software applications. It is compatible with any 32-bit software application or programming environment which can utilize an OLE automation component, including:

- Visual Basic
- Visual C++
- Delphi
- Software packages that support Microsoft's Component Object Model (COM):
  - Wonderware's Factory Suite 2000
  - National Instruments LabVIEW

The Motion Planner installation program installs COM6SRVR.EXE in the Windows\System (Windows 95/98) or WinNt\System32 (Windows NT) directory.

To begin communications, an application simply needs to request a connection to a 6K controller through the Communications Server. The Communications Server manages the actual connection to each 6K controller, and can feed information from a particular controller to all client applications which require the information.

Although the Communications Server only makes one connection to each 6K controller, it can feed the information from that one connection to multiple client applications. This means, for example, that a terminal application created in Visual Basic and a terminal in Motion Planner can be connected to the same 6K at the same time. They will both receive the same responses coming from the controller, instead of competing for the data. It is also possible for an application to request connections to multiple 6K units via the Communications Server. Each connection can be either Ethernet or RS-232.

For RS-232 connections, you need to specify the PC COM port on which to connect. For Ethernet connections, you need to specify the controller's IP address. Each controller is set with a default IP address (192.168.10.30). If there is an address conflict with other devices on the network, you can change the 6K's address with the NTADDR command (you must cycle power to invoke the new address — refer also to the configuration procedures on page 3). The Communications Server can handle up to two RS-232 connections and unlimited Ethernet connections (to different IP addresses).

The syntax for requesting a connection to the Communications Server varies depending on the programming environment being used. Below are examples in the Visual Basic, Visual C++, and Delphi programming formats (refer also to the samples in the Motion Planner directory). To disconnect, refer to "How to Disconnect" instructions on page 13.

COM6SRVR Application Programming Interface (API): Once the proper object variable has been created and a connection is established, there is a standard set of methods and properties which the client application(s) can access.

- For RS-232 methods, refer to page 14.
- For Ethernet methods and properties, refer to page 16.

Visual Basic Connection Example

```
'create an object variable, initialize it to an
'Ethernet interface and make a connection

Dim commserver As Object
Dim ConnectReturnValue As Integer
Set commserver = CreateObject("COM6SRVR.NET")
ConnectReturnValue = commserver.Connect("192.168.10.30")

'-----------------------------------------------

'create an object variable, initialize it to a
'RS-232 interface and make a connection to PC COM1

Dim MyMachine As Object
Dim ConnectReturnValue As Integer
Set MyMachine = CreateObject("COM6SRVR.RS232")
ConnectReturnValue = MyMachine.Connect(1)
```

Note: When using VBScript, the syntax is identical to the example above, except that the variable declaration should omit the "`As Object`" and "`As Integer`" keywords.

Visual C++ Connection Example

```
/* create an object variable, initialize it to an
Ethernet interface and make a connection */

INet commserver;
commserver.CreateDispatch ("COM6SRVR.NET");
int ConnectReturnValue = commserver.Connect("192.168.10.30");

/*==============================================*/

/* create an object variable, initialize it to a
RS-232 interface and make a connection to COM2 */

IRS232 MyMachine;
MyMachine.CreateDispatch ("COM6SRVR.RS232");
int ConnectReturnValue = MyMachine.Connect(2);
```

Delphi Connection Example

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ComObj;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    CommServer: Variant;         { Create the object variable }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  { Initialize CommServer object to an Ethernet interface }
  CommServer := CreateOleObject('COM6SRVR.NET');
  { For RS-232, use CommServer := CreateOleObject('COM6SRVR.RS232');  }
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  { Make a connection to the 6k Controller at IP 192.168.10.30 }
  CommServer.Connect('192.168.10.30');
  { To connect via RS-232 on COM2, use CommServer.Connect(2);  }
end;

end.
```

How to Disconnect

The 6K Communications Server is designed as an “EXE” (out-of-process) server rather than a “DLL” (in-process) server. This means that it runs independently of the client application’s process. This feature allows the same data from the 6K Communications Server to be shared among several clients. It also provides a more secure connection model by insulating the 6K Communications Server from failure on any singular client.

With the use of an *in-process* server, the server itself runs in the client’s process. If the client application fails or shuts down, the server will be shutdown along with the client. With the use of an *out-of-process* server, the server runs independently of the client and is therefore insulated from a failure in the client’s process. If a particular client application fails, the server will continue to run and provide data to any other client applications requiring its service.

As an out-of-process server, the 6K Communications Server does not shutdown until all client applications have disconnected from the server. In many cases, a proper disconnect does not take place if an unhandled error occurs in the client application and the program exits abnormally. This means that care must be exercised on the part of the client program to disconnect from the server on such occasions or when its services are no longer needed.

VB and VBScript

For VB/VBScript applications, an object variable is typically released when the variable loses scope. However, it is always a good practice to explicitly release the object by setting it to nothing.

```
'assuming the commserver is an object variable
'representing a 6K Communications Server connection

Set commserver = Nothing;   'free the object - disconnect from the server
```

C++

In C++, the same rule applies to the scope of an object variable, but again it is good programming practice to explicitly release the object.

```
//assuming the commserver is an object variable
//representing a 6K Communications Server connection

commserver.ReleaseDispatch();     // release the IDispatch connection
```

Delphi

Again, in Delphi, the same rule applies.

```
{ assuming the CommServer is an object variable       }
{ representing a 6K Communications Server connection   }

CommServer := UnAssigned;     { release the connection }
```

Be Aware of Background Commands

During some operations in the Communications Server, it is necessary for the server to send setup commands to the 6K. These commands generally affect communication port settings and are necessary for proper communications between the server and the 6K controller.

The use of these commands may affect settings previously established in the 6K controller by a user program, so it may be necessary to adjust the settings after certain methods in the Communications Server are exercised. The use of these commands will also affect the command count data available in FastStatus Ethernet property.

The Communications Server methods which invoke background commands are:

RS-232 Methods:	**Connect**, **GetFile**, **SendFile**, and **SendOS**
Ethernet Methods:	**Connect**, **GetFile**, and **SendFile**

For details on the background commands sent, refer to the description (below) for the respective method.

Using COM6SRVR with RS-232

RS-232 Methods

NOTE

This section covers RS-232 methods, there are no RS-232 properties for the Communications Server.

Connect (port)

Description: The Connect method opens a connection to a 6K controller.
Visual Basic: `object.Connect(port as Integer) As Integer`
Visual C++: `short object.Connect(short commport)`
Delphi: `Smallint_variable := Object_variable.Connect(port as Smallint)`
Parameter: *port* (or *commport*) .... Short integer.
Represents the PC's COM port number (1-4).
Return Type: Short Integer.
If the connection is successfully opened, the method returns a positive value representing the number of connected clients. If the connection is unsuccessful, then an error code is returned (see table on page 28).
Remarks: The Server can handle up to two RS-232 connections. The RS232 server assumes 9600 Baud operation.

Background Commands: After a successful connection is made, a "`PORT0:`" command is sent to the controller.

Flush

Description: The Flush method removes all characters from the client's receive buffer. This method allows the programmer to clear the receive buffer prior to making a read.
Visual Basic: `object.Flush`
Visual C++: `void object.Flush()`
Delphi: `Object_variable.FLUSH`
Parameter : NONE
Return Type: NONE
Remarks: **USE WITH CAUTION**. This method allows the programmer to clear the receive buffer, such that a subsequent Read call can yield a clean response. However, data arriving in the receive buffer is asynchronous to the application program and a thorough understanding of how the application program is structured is necessary to use this method correctly (for example, it would not be beneficial to Flush the buffer if only a partial response has been received).

GetFile (filename)

Description: The GetFile method is used to upload programs currently stored in the controller.
Visual Basic: `object.GetFile(filename as String) As Long`
Visual C++: `long object.GetFile(LPCTSTR lpFileName)`
Delphi: `Longint_variable := Object_variable.GetFile(filename as String)`
Parameter: *filename*... String.
Represents the name of the file to store the uploaded programs. If the filename is an empty string, then the user will be prompted for the filename.
Return Type: Long integer.
The method returns a positive value if the operation is successful; otherwise, it returns an error code (see table on page 28).
Remarks: Background Commands: At the beginning of a file upload operation, these commands are sent to the controller:

```
!PORT0
!ECHO0
!ERRLVL0
!EOT1,0,0
!EOL10,0,0
!TDIR
```

For each program selected for upload, a "`!TPROG`" command is also sent to the controller.

After the upload process is completed, these commands are sent to the controller:

```
     !PORT0
     !EOT13,0,0
     !EOL13,10,0
     !ERRLVL4
     !ECHO1
```

Read ( )

Description: The Read method retrieves command responses from the controller.
Visual Basic: `object.`**`Read`**`() As String`
Visual C++: `object.CString `**`Read`**`()`
Delphi: `String_variable := Object_variable.`**`Read`**
Parameter: NONE
Return Type: String.
The read method does not wait for incoming responses from the controller. It returns immediately with a string containing the controller's response at the time of the request. If no response is available, this method will return an empty string. The Read method response is limited to 256 characters. If the response is longer than 256 characters, the excess characters will remain in the COM6SRVR buffer. Multiple reads are necessary for long responses.
Remarks: You should disable Timer events in VB5 and VBScript when reading and writing to the COM6SRVR (see Microsoft Support Online Article ID176399).

SendFile (filename)

Description: The SendFile method is used to download program files to the controller.
Visual Basic: `object.`**`SendFile`**`(`*`filename`*` as String) As Long`
Visual C++: `long object.`**`SendFile`**`(LPCTSTR `*`lpFileName`*`)`
Delphi: `Longint_variable := Object_variable.`**`SendFile`**`(`*`filename`*` as String)`
Parameter: *`filename`*... String.
Represents the name of the program file (containing 6K programs/code) to be downloaded. If the filename is an empty string, then the user will be prompted for the filename.
Return Type: Long integer.
The method returns a positive value if the operation is successful; otherwise, it returns an error code (see table on page 28).
Remarks: To speed up downloads, the SendFile method strips comments from the downloaded 6K code. That is, all text between the comment delimiter (semi-colon) and the command delimiter (carriage return or line feed) is removed.

NOTE: The SendFile method should be called when motion is not in progress and programs are not running.

Background Commands: At the beginning of a file download operation, these commands are sent to the controller:

```
     !PORT0
     !ECHO0
     !ERRLVL0
     !EOT1,0,0
     !EOL10,0,0
     !TDIR
```

After the download process is completed, these commands are sent to the controller:

```
     !PORT0
     !EOT13,0,0
     !EOL13,10,0
     !ERRLVL4
     !ECHO1
```

NOTE: If the download process is canceled, an "`END`" command is sent to the controller.

SendOS (filename)

Description: The SendOS method downloads the soft operating system to a 6K controller.
Visual Basic: `object.SendOS(filename as String) As Boolean`
Visual C++: `BOOL object.SendOS(LPCTSTR lpFileName)`
Delphi: `Boolean_variable := Object_variable.SendOS(filename as String)`
Parameter: *filename*... String. Represents the name of soft operating system file. If filename is an empty string then the user will be prompted for the operating system file name.
Return Type: Boolean. (This method returns a Boolean value.)
The method returns a TRUE value if the operation is successful; otherwise, a FALSE value is returned.
Remarks: After downloading a new operating system, the appropriate `NTFEN` command must be sent to the controller (see page 5) — this applies only if you will be using Ethernet communication.

Background Commands: The operating system download process, COM6SRVR sends several setup commands to the 6K, followed by a `reset` command. NOTE – The download process uses a baud rate of 38400. This allows for fast download times. After the download process is completed, the 6K's previous baud rate is reinstated. The Communications server ALWAYS uses 9600 baud for normal communications.

Write (cmd)

Description: The Write method is used to send commands to the controller.
Visual Basic: `object.Write(cmd as String) As Long`
Visual C++: `long object.Write(LPCTSTR cmd)`
Delphi: `Longint_variable := Object_variable.Write(cmd as String)`
Parameter: *cmd*............. String. A string of commands to be sent. Multiple commands can be sent, but each command should be separated with a valid 6K command delimiter (colon, carriage return, or line feed). The command string should be limited to 256 characters or less. Larger command strings may cause an overflow in the 6K's command buffer.
Return Type: Long integer.
This method returns a positive value corresponding to the number of bytes sent, or a negative error code (see table on page 28).
Remarks: You should disable Timer events in VB5 and VBScript when reading and writing to the COM6SRVR (see Microsoft Support Online Article ID176399).

Using COM6SRVR with Ethernet

Ethernet Methods

Connect (netaddress)

Description: The Connect method opens a connection to a 6K controller
Visual Basic: `object.Connect(netaddress as String) As Integer`
Visual C++: `short object.Connect(LPCTSTR netaddress)`
Delphi: `Smallint_variable := Object_variable.Connect(netaddress as String)`
Parameter: *netaddress*.......String. Represents the target controller's IP address.
Return Type: Short Integer.
If the connection is successfully opened, the method returns a positive value representing the number of connected clients. If the connection is unsuccessful, then an error code is returned (see table on page 28).
Remarks: The Server can handle unlimited Ethernet connections (to different IP addresses). The 6K takes up to one minute for an Ethernet connection to truly expire and be available for a new connection.

Background Commands: After a successful connection is made, the following commands are sent to the controller:

```
!PORT0
!ERRLVL4
!EOT13,0,0
!EOL13,10,0
```

`ECHO` mode is initially disabled (`ECHO0`) by the 6K during Ethernet communications.

Flush

Description: The Flush method removes all characters from the client's receive buffer. This method allows the programmer to clear the receive buffer prior to making a read.

Visual Basic: `object.`**`Flush`**

Visual C++: `void object.`**`Flush`**`()`

Delphi: `Object_variable.`**`FLUSH`**

Parameter : NONE

Return Type: NONE

Remarks: Use with caution. This method allows the programmer to clear the receive buffer, such that a subsequent Read call can yield a clean response. However, data arriving in the receive buffer is asynchronous to the application program and a thorough understanding of how the application program is structured is necessary to use this method correctly (for example, it would not be beneficial to Flush the buffer if only a partial response has been received).

GetFile (filename)

Description: The GetFile method is used to upload programs currently stored in the controller.

Visual Basic: `object.`**`GetFile`**`(`*`filename`*` as String) As Long`

Visual C++: `long object.`**`GetFile`**`(LPCTSTR `*`filename`*`)`

Delphi: `Longint_variable := Object_variable.`**`GetFile`**`(`*`filename`*` as String)`

Parameter: *`filename`*... String.
Represents the name of the file to store the uploaded programs. If the filename is an empty string, then the user will be prompted for the filename.

Return Type: Long integer.
The method returns a positive value if the operation is successful; otherwise, it returns an error code (see table on page 28).

Remarks: Background Commands: At the beginning of a file upload operation, these commands are sent to the controller:

```
!PORT0
!ERRLVL0
!EOT1,0,0
!EOL10,0,0
!ECHO0
!TDIR
```

For each program selected for upload, a "`!TPROG`" command is also sent to the controller.

After the upload process is completed, these commands are sent to the controller:

```
!ERRLVL4
!EOT13,0,0
!EOL13,10,0
```

IsWatchdogTimedOut

Description: The IsWatchdogTimedOut method interrogates the current status of the Ethernet Watchdog. The Ethernet Watchdog is a handshake established between the COM6SRVR and the 6K to monitor that the Ethernet connection is still active and "connected".

Visual Basic: `object.`**`IsWatchdogTimedOut`**` As Boolean`

Visual C++: `BOOL `**`IsWatchdogTimedOut`**`()`

Delphi: `Boolean_variable := Object_variable.`**`IsWatchdogTimedOut`**

Parameter: None

Return Type: Boolean.
A True indicates that the Ethernet connection has been lost (possible causes: the 6K was reset, or the Ethernet connection was broken). The property is cleared

when a new Ethernet connection is established.

Remarks: For further information, refer to the SetWatchdog method on page 20.

Read ( )

Description: The Read method retrieves command responses from the controller.

Visual Basic: `object.Read() As String`

Visual C++: `CString object.Read()`

Delphi: `String_variable := Object_variable.Read`

Parameter: NONE

Return Type: String.

The read method does not wait for incoming responses from the controller. It returns immediately with a string containing the controller's response at the time of the request. If no response is available, this method returns an empty string. The Read method response is limited to 256 characters. If the response is longer than 256 characters, the excess characters will remain in the COM6SRVR buffer. Multiple reads are necessary for long responses.

Remarks: You should disable Timer events in VB5 and VBScript when reading and writing to the COM6SRVR (see Microsoft Support Online Article ID176399).

RequestFastStatusUpdate

Description: The RequestFastStatusUpdate method allows the COM6SRVR to request a fast status update as needed, without having to enable the fast status "Streaming Mode" (FSEnabled) or set an update interval (FSUpdateRate).

Visual Basic: `object.RequestFastStatusUpdate As Integer`

Visual C++: `short object.RequestFastStatusUpdate`

Parameter: NONE

Return Type: Short integer. If the RequestFastStatusUpdate call is successful the method returns the number of bytes sent. If the call is unsuccessful, the method returns a negative error code (see error code table on page 28).

Remarks: This method is one of two "On Demand" fast status update options. The other option is for the 6K to execute the `NTSFS` command (see page 41). Using an On Demand update technique is more efficient for interactive PC applications than the Streaming Mode, and reduces network traffic. For an overview of using the fast status, refer to page 31.

SendFile (filename)

Description: The SendFile method is used to download program files to the controller.

Visual Basic: `object.SendFile(`*`filename`*` as String) As Long`

Visual C++: `long object.SendFile(LPCTSTR `*`filename`*`)`

Delphi: `Longint_variable := Object_variable.SendFile(`*`filename`*` as String)`

Parameter: *`filename`*... String. Represents the name of the program file (containing 6K programs/code) to be downloaded. If the filename is an empty string, then the user will be prompted for the filename.

Return Type: Long integer. The method returns a positive value if the operation is successful; otherwise, it returns an error code (see table on page 28).

Remarks: To speed up downloads, the SendFile method strips comments from the downloaded 6K code. That is, all text between the comment delimiter (semicolon) and the command delimiter (carriage return or line feed) is removed.

Background Commands: At the beginning of a file download operation, these commands are sent to the controller:

```
!PORT0
!ERRLVL0
```

After the download process is completed, these commands are sent to the controller:

```
!PORT0
!ERRLVL4
!EOT13,0,0
!EOL13,10,0
```

NOTE: If the download process is canceled, an "`END`" command is sent to the controller.

SendVariable (nVariableMask, vaValue)

Description: The SendVariable method sends one variable from the variable packet to the 6K controller.

Visual Basic: `object.SendVariable(nVariableMask As Long, vaValue As Variant) As Integer`

Visual C++: `short object.SendVariable(long nVariableMask, const VARIANT FAR& vaValue)`

Parameter: *nVariableMask* ..... Long integer.

Specifies the one variable to be sent. Constants are defined for the mask bits (mask bits for Visual Basic and Visual C++ are provided below). Only one bit can be set in the nVariableMask.

vaValue .... Variant.

Specifies the value of the variable to be sent. The actual variable being sent is specified by the nVariableMask. Because the SendVariable Method can be used to send integer, real or binary variables, the data type can either be a long integer or a double floating point value. Using a Variant parameter allows the flexibility of sending any integer type, while allowing the COM6SRVR to cast the Variant into the appropriate data type.

Return Type: Short integer.

If the SendVariable call is successful, the method returns the number of bytes sent. If the call is unsuccessful, the method returns a negative error code (see error code table on page 28). Errors codes are returned if more than one bit is set in the nVariableMask or if the Variant data type is incompatible. Error codes are also returned if there are Ethernet communications errors.

Remarks: Refer to page 33 for an overview of using Send Variables packets.

The data range of real variables in the 6K and the number of significant figures available in a double data type in the PC programming language may cause some rounding errors. The 6K can store data with greater significance, but with a smaller range of values (refer to the VAR command in the *6K Series Command Reference* and to your PC programming language reference).

Variable Packet Mask Bits for Visual Basic

```
Public Const VARI1 As Long = 1
Public Const VARI2 As Long = 2
Public Const VARI3 As Long = 4
Public Const VARI4 As Long = 8
Public Const VARI5 As Long = 16
Public Const VARI6 As Long = 32
Public Const VARI7 As Long = 64
Public Const VARI8 As Long = 128
Public Const VARI9 As Long = 256
Public Const VARI10 As Long = 512
Public Const VARI11 As Long = 1024
Public Const VARI12 As Long = 2048

Public Const VAR1 As Long = 4096
Public Const VAR2 As Long = 8192
Public Const VAR3 As Long = 16384
Public Const VAR4 As Long = 32768
Public Const VAR5 As Long = 65536
Public Const VAR6 As Long = 131072
Public Const VAR7 As Long = 262144
Public Const VAR8 As Long = 524288
Public Const VAR9 As Long = 1048576
Public Const VAR10 As Long = 2097152
Public Const VAR11 As Long = 4194304
Public Const VAR12 As Long = 8388608

Public Const VARB1 As Long = 16777216
Public Const VARB2 As Long = 33554432
Public Const VARB3 As Long = 67108864
Public Const VARB4 As Long = 134217728
Public Const VARB5 As Long = 268435456
Public Const VARB6 As Long = 536870912
Public Const VARB7 As Long = 1073741824
Public Const VARB8 As Long = &H80000000
```

Variable Packet Mask Bits for Visual C++

```
#define VARI1  0x00000001
#define VARI2  0x00000002
#define VARI3  0x00000004
#define VARI4  0x00000008
#define VARI5  0x00000010
#define VARI6  0x00000020
#define VARI7  0x00000040
#define VARI8  0x00000080
#define VARI9  0x00000100
#define VARI10 0x00000200
#define VARI11 0x00000400
#define VARI12 0x00000800

#define VAR1   0x00001000
#define VAR2   0x00002000
#define VAR3   0x00004000
#define VAR4   0x00008000
#define VAR5   0x00010000
#define VAR6   0x00020000
#define VAR7   0x00040000
#define VAR8   0x00080000
#define VAR9   0x00100000
#define VAR10  0x00200000
#define VAR11  0x00400000
#define VAR12  0x00800000

#define VARB1  0x01000000
#define VARB2  0x02000000
#define VARB3  0x04000000
#define VARB4  0x08000000
#define VARB5  0x10000000
#define VARB6  0x20000000
#define VARB7  0x40000000
#define VARB8  0x80000000
```

SendVariablePacket
(vaPacket)

Description: The SendVariablePacket method sends a packet of variables to the 6K controller. A complete packet of variables (comprising 6K integer variables 1-12, real variables 1-12 and binary variables 1-8) are always sent. (Refer to the Variable Structures listed below for VB and VC++.) Also included in the packet is a mask, which allows specific variables to be write-protected or over-written.

Visual Basic: `object.`**`SendVariablePacket`**`(`*`vaPacket`*` As Variant) As Integer`

Visual C++: `short object.`**`SendVariablePacket`**` (const VARIANT FAR& `*`vaPacket`*`)`

Parameter: *`vaPacket`*........ Variant.

An array of bytes representing the SendVariable packet. The array of bytes comprise: mask bits, reserved elements and bytes of data for the variables.

To send a variable packet:

1. Create a structure (TypeDef) and populate the structure with the mask and the variable values.
2. Create an array of bytes from the structure (VB uses Windows API function CopyMemory, Visual C++ uses SAFEARRAYS).
3. Pass the array of bytes as a Variant to the SendVariablePacket method.

Refer also to the examples in the SimpleOnePlus sample VB application.

Return Type: Short integer.

If the SendVariablePacket call is successful the method returns the number of bytes sent. If the call is unsuccessful, the method returns a negative error code (see error code table on page 28). Errors codes are returned if the variant data type is incompatible or if there are Ethernet communication errors.

Remarks: Refer to page 33 for an overview of using Send Variables packets.

A list of mask bits for Visual Basic and Visual C++ is provided in the SendVariables method description above.

The data range of real variables in the 6K and the number of significant figures available in a double data type in the PC programming language may cause some rounding errors. The 6K can store data with greater significance, but with a smaller range of values (refer to the VAR command in the *6K Series Command Reference* and to your PC programming language reference).

Variable Structure for Visual Basic

```
Type SendVariableStructure
  Mask As Long
  Reserved1 As Long
  Reserved2 As Long
  Reserved3 As Long
  VarI(1 To 12) As Long
  VarR(1 To 12) As Double
  VarB(1 To 8) As Long
End Type
```

Variable Structure for Visual C++

```
typedef struct VARIABLEPACKETStruct {
 int nVariableMask;
 int nReserved1;
 int nReserved2;
 int nReserved3;
 int VARI[12];
 double VAR[12];
 int VARB[8];
} VARIABLEPACKET, *LPVARIABLEPACKET;
```

SetWatchdog
(wTimeout, wTicker)

Description: The SetWatchdog method enables Ethernet watchdog hand-shaking between the COM6SRVR and the 6K Controller.

Visual Basic: `object.`**`SetWatchdog`**`(`*`wTimeout`*` as Integer, `*`wTicker`*` as Integer) as Integer`

Visual C++: `short `**`SetWatchdog`**`(short `*`wTimeout`*`, short `*`wTicker`*`)`

Delphi:
```
Smallint_variable := Object_variable.SetWatchdog(wTimeout as Smallint,
         wTicker as Smallint)
```

Parameters: *`wTimeout`*............ Timeout period in seconds (see guidelines below)

`wTicker`.............. Number of “heartbeat” packets to send during the timeout period

Return Type: Short integer.

Returns zero if successful, or a negative error value (usually –11, which indicates that an invalid configuration was specified).

Remarks: The Ethernet watchdog allows the COM6SRVR and 6K Controller to gracefully recover when communication between the 6K and COM6SRVR is lost. Such situations might arise from the loss of power to the 6K or to the PC while an Ethernet connection was active. By enabling the Watchdog, a *heartbeat* packet is sent periodically by the COM6SRVR. The 6K detects the heartbeat and echoes it back to the COM6SRVR. If the COM6SRVR does not detect the echoed heartbeat (within the constraints set by the `wTimeout` and `wTicker` parameters), the watchdog is considered timed out. If the 6K does not receive the heartbeat (within the same `wTimeout` and `wTicker` constraints), the 6K considers the watchdog timed out.

Loss or delay of a single echoed heartbeat could happen quite frequently on a busy network connection. Therefore, we provide a method whereby a number of re-tries are attempted over a specific *timeout period*. If all re-tries fail within the timeout period, then the watchdog is considered to have *timed out*. This functionality is provided by the `wTimeout` and `wTicker` parameters. The constraints for these parameters are as follows:

- To enable the watchdog, set `wTimeout` > 0 > `wTicker`.
- To disable the watchdog, set `wTimeout` = 0 and set `wTicker` = 0.
- The `wTimeout/wTicker` ratio must be ≤ 65.

RECOMMENDATION: Set `wTimeout` = 100 and `wTicker` = 5, which provides a heartbeat once every twenty seconds (100 seconds / 5 tries = 20 seconds/attempt). If none of the 5 heartbeats are acknowledged in 100 seconds, the watchdog times out.

WHEN A WATCHDOG TIMEOUT OCCURS:

- In the 6K: When the 6K detects a watchdog timeout, it attempts to send an *alarm packet* to the COM6SRVR (AlarmStatus bit #22 – see page 22). It then closes the Ethernet connection and reports "`disconnected`" in the `TNT` report. If the user has enabled error-checking bit #22 (`ERROR.22-1`), the 6K will execute a GOSUB branch to the `ERRORP` program. Within the `ERRORP` program, the watchdog timeout can be cleared by disabling `ERROR` bit #22 (`ERROR.22-0`).
- In the COM6SRVR: When the COM6SRVR detects a watchdog timeout, the IsWatchdogTimedOut method (see page 17) returns TRUE. (If the COM6SRVR receives the alarm packet from the 6K, it will also display an alert dialog to the user.) A client application can poll the IsWatchdogTimedOut. When a timeout is detected by the COM6SRVR, the Client application should "disconnect" the COM6SRVR (if using VB, set COM6SRVR object to Nothing. If using VC++, use ReleaseDispatch). After the COM6SRVR has been disconnected, creating a new Com6srvr object and "connecting" Ethernet will clear the watchdog timeouts. All client applications for that particular 6K Ethernet connection should be disconnected.

Write (cmd)

Description: The Write method is used to send commands to the controller.

Visual Basic: `object.Write(cmd as String) As Integer`

Visual C++: `short object.Write(LPCTSTR cmd)`

Delphi: `Smallint_variable := Object_variable.Write(cmd as String)`

Parameter: *cmd*............. String. A string of commands to be sent. Multiple commands can be sent, but each command should be separated with a valid 6K command delimiter (colon, carriage return, or line feed). The command string should be limited to 256 characters or less. Larger command strings may cause an overflow in the 6K's command buffer.

Return Type: Short integer.
This method returns a positive value corresponding to the number of bytes sent, or a negative error code (see table on page 28).

Remarks: You should disable Timer events in VB5 and VBScript when reading and writing to the COM6SRVR (see Microsoft Support Online Article ID176399).

Ethernet Properties

Bit Status Convention

When retrieving bit-oriented properties (e.g., AxisStatus, ErrorStatus, Limits, SystemStatus, etc.) note that the convention in the 6K programming language differs from the convention used for C and Assembly programming languages. Compumotor's 6K convention is to refer to the bits within a 32 bit long integer as bits 1 through 32 (left to right). The C and Assembler Programmer's convention refers to these as bits 0 through 31 (right to left). When masking these bits, you should be aware of this subtle difference when referring to 6K documentation.

AlarmStatus (bit)

Description: The AlarmStatus property returns the state of the controller's alarm status.

Visual Basic: `object.AlarmStatus(bit As Integer) as Long`

Visual C++: `long object.GetAlarmStatus(short bit)`

Delphi: `Longint_variable := Object_variable.AlarmStatus(bit as Smallint)`

Parameter: `bit`............. Short Integer.

Specifies the status bit of the alarm status to return. Range is 0-32, where values in the 1-32 range represent the alarm bits as described in the table below (refer also to the `INTHW` command). Specifying a bit value of 0 returns the entire 32-bit alarm status as a long value; otherwise, a value of 1 or 0 is returned to indicate the state of any single bit. When any single bit status is retrieved using the AlarmStatus property, that bit status is automatically cleared by the Communications Server. If a bit value of 0 is used then all alarm status bits are cleared.

Return Type: Long Integer.

Remarks: When the 6K sends an alarm packet to the COM6SRVR, the FastStatus structure (see page 24) is automatically updated, regardless of state of FSEnabled.

Bit #	Function **	Bit #	Function
1	Software (forced) Alarm #1	17	Reserved
2	Software (forced) Alarm #2	18	Reserved
3	Software (forced) Alarm #3	19	Limit Hit - hard or soft limit, on any axis
4	Software (forced) Alarm #4	20	Stall Detected (stepper) or Position Error (servo) on any axis
5	Software (forced) Alarm #5		
6	Software (forced) Alarm #6	21	Timer (`TIMINT`)
7	Software (forced) Alarm #7	22	Ethernet fail (`RESET` or `ER.22` occurred) (also invokes an error dialog)
8	Software (forced) Alarm #8		
9	Software (forced) Alarm #9	23	Input - any of the inputs defined by `INFNCi-I` or `LIMFNCi-I`
10	Software (forced) Alarm #10		
11	Software (forced) Alarm #11	24	Command Error
12	Software (forced) Alarm #12	25	Motion Complete on Axis 1
13	Command Buffer Full	26	Motion Complete on Axis 2
14	ENABLE input Activated	27	Motion Complete on Axis 3
15	Program Complete	28	Motion Complete on Axis 4
16	Drive Fault on any Axis	29	Motion Complete on Axis 5
		30	Motion Complete on Axis 6
		31	Motion Complete on Axis 7
		32	Motion Complete on Axis 8

** Bits 1-12: software alarms are forced with the `INTSW` command.

AnalogInput (channel)

Description: The AnalogInput property returns the value (counts) of the specified analog input.

Visual Basic: `object.AnalogInput(channel As Integer) As Long`

Visual C++: `short object.GetAnalogInput(short channel)`

Delphi: `Smallint_variable := Object_variable.AnalogInput(channel as Smallint)`

Parameter: `channel`..... Short Integer.

Specifies the analog input channel (channel 1 or 2) value to return. This property uses only the first two analog inputs detected on an I/O brick connected to the 6K, regardless of the `ANIEN` (analog input enable) setting.

Return Type: Short integer.

The method returns the specified analog input value in counts.

Remarks: Requires fast status to be enabled (see FSEnabled property – page 25).

AxisStatus (axis)

Description: The AxisStatus property retrieves the current axis status for the specified axis.

Visual Basic: `object.AxisStatus(axis As Integer) As Long`

Visual C++: `long object.GetAxisStatus(short axis)`

Delphi: `Longint_variable := Object_variable.AxisStatus(axis as Smallint)`

Parameter: `axis` .... Short Integer.
Specifies the axis about which the status pertains. The range for this value is 1-8.

Return Type: Long Integer.
The long integer value represents the current axis status for the specified axis. Refer to the TAS command description for a list of the status elements.

Remarks: Requires fast status to be enabled (see FSEnabled property – page 25).

CommandCount

Description: Use the CommandCount property to ascertain how many 6K commands have been executed (outside of defined programs) since the 6K controller was powered up.

Visual Basic: `object.CommandCount As Long`

Visual C++: `long object.GetCommandCount()`

Delphi: `Longint_variable := Object_variable.CommandCount`

Parameter: NONE

Return Type: Long Integer.
The value represents the number of 6K commands which have been executed **outside of defined programs**, since the 6K controller was powered up.

Remarks: Read Only.
This property allows users to track when commands being sent to the controller (via the communications ports) have been executed. The value is reset to zero each time power is cycled on the 6K. The return value is affected by any background commands sent in conjunction with the Connect, GetFile, and SendFile methods.
Requires fast status to be enabled (see FSEnabled property – page 25).

Counter

Description: The Counter property returns the current Time Frame Counter value.

Visual Basic: `object.Counter As Integer`

Visual C++: `short object.GetCounter()`

Delphi: `Smallint_variable := Object_variable.Counter`

Parameter: NONE

Return Type: Short integer.
The values represents the current Time Frame Counter value.

Remarks: Read Only.
The Time Frame Counter is a free-running timer in the controller. The Counter is updated at the *System Update Rate* (2 milliseconds).
Requires fast status to be enabled (see FSEnabled property – page 25).

EncoderPos (axis)

Description: The EncoderPos property returns the current encoder position (TPE) in counts for the specified axis.

Visual Basic: `object.EncoderPos(axis As Integer) As Long`

Visual C++: `long object.GetEncoderPos(short axis)`

Delphi: `Longint_variable := Object_variable.EncoderPos(axis)`

Parameter: `axis` .... Short Integer.
Specifies the axis number of the encoder. The range for this value is 1-8.

Return Type: Long integer.
The value represents the current encoder position (TPE) in counts for the specified axis.

Remarks: Read Only.
Requires fast status to be enabled (see FSEnabled property – page 25).

ErrorStatus

Description: The ErrorStatus property returns the current error status (TER) of task 0 only.
Visual Basic: `object.`**`ErrorStatus`**` As Long`
Visual C++: `long object.`**`GetErrorStatus`**`()`
Delphi: `Longint_variable := Object_variable.`**`ErrorStatus`**
Parameter: NONE
Return Type: Long Integer.
The values represents the current error status (TER) of task 0.
Remarks: Requires fast status to be enabled (see FSEnabled property – page 25).

FastStatus

Description: The FastStatus property returns the entire FastStatus data structure.
Visual Basic: `object.`**`FastStatus`**` As Variant`
Visual C++: `VARIANT object.`**`GetFastStatus`**`()`
Delphi: `Variant_variable := Object_variable.`**`FastStatus`**
Parameter: NONE
Return Type: Variant.
The variant represents the value of the entire FastStatus data structure.
Remarks: This property allows for faster, more efficient retrieval of the FastStatus information if multiple FastStatus items need to be checked at once. The variant is essentially a byte array which can be mapped into a FastStatus structure (see table below for FastStatus structure). The Fast Status structure includes ten integer (VARI) variables and ten binary (VARB) variables that you can use to customize the Fast Status content (see *Customizing Fast Status* on page 32).

Refer to the VB5 sample application SimpleOne in the subroutine `cmdGetData_Click()` for details of how to convert the byte array data into a Fast Status structure (User Defined Type). Refer to the VC5 sample application VC_Ethernet in the subroutine `MakeFastStatus` for details on how to convert from a byte array into a Fast Status TypeDef. VBScript does not allow use of structures – use the properties Inputs() and MotorPos().

Requires fast status to be enabled (see FSEnabled property – page 25).

NOTE: When the object is first created, the FastStatus data will read zeroes. There after, it will report the most recently updated values. When FSEnabled is set to FALSE, the FastStatus structure will retain the values from the last update. When the 6K sends an alarm packet to the COM6SRVR, the FastStatus structure is automatically updated, regardless of state of FSEnabled.

Fast Status — Packet Data Definition (280 bytes total)

Type	Description	Bytes
WORD wUpdateID	Unique update ID for synch channel	2
WORD wCounter	Time Frame Counter	2
DWORD dwMotorPos[8]	Commanded Position (TPC)	32
DWORD dwEncPos[8]	Encoder Position (TPE)	32
DWORD dwMotorVel[8]	Commanded Velocity (TVEL)	32
DWORD dwAxisStatus[8]	Axis Status (TAS)	32
DWORD dwSystemStatus	System Status (TSS)	4
DWORD dwErrorStatus	Error Status (TER)	4
DWORD dwUserStatus	User Status (TUS)	4
DWORD dwTimer	Timer (TTIM)	4
DWORD dwLimits	Limit Status (TLIM)	4
DWORD dwInputs[4]	Input Status (TIN)	16
DWORD dwOutputs[4]	Output Status (TOUT)	16
DWORD dwTriggers	Trigger Status (TTRIG)	4
WORD wAnalogIn[2]	Analog Input Value (TANI - in counts)	4
DWORD dwVarb[10]	Binary Variable Values (VARB1 through VARB10)	40
DWORD dwVari[10]	Integer Variable Values (VARI1 through VARI10)	40
DWORD dwIPAddress	IP Address (NTADDR)	4
DWORD dwCmdCount	Command Count	4

FSEnabled

Description: The FSEnabled property sets or returns the state of FastStatus polling.

Visual Basic: object.**FSEnabled** As Boolean

Visual C++: Read: BOOL object.**GetFSEnabled**()
Write: void object.**SetFSEnabled**(BOOL *state*)

Delphi: Read: Boolean_variable := Object_variable.**FSEnabled**
Write: Object_variable.**FSEnabled :=** (*state* as Boolean)

Parameter: Boolean (read/write property).

Return Type: Boolean (read/write property).

Remarks: The table above lists the items in the FastStatus structure. If the FSEnabled property is set to TRUE, then FastStatus information is automatically retrieved from the controller on a continual basis. **BE AWARE** that enabling automatic FastStatus polling provides fresh data from the controller on a continual basis, but this will impair the controller's processing time and create a high volume of traffic over the Ethernet network interface. If you intend to enable automatic FastStatus polling, be sure to first set the FSUpdateRate property accordingly. If the FSEnabled property is set to FALSE, automatic FastStatus polling is turned off (but the FastStatus structure will retain the values from the last update).

FSUpdateRate

Description: The FSUpdateRate property is used to set the millisecond interval on which the controller automatically updates its FastStatus information.

Visual Basic: object.**FSUpdateRate** As Integer

Visual C++: Read: short object.**GetFSUpdateRate**()
Write: void object.**SetFSUpdateRate**(short *rate*)

Delphi: Read: Smallint_variable := Object_variable.**FSUpdateRate**
Write: Object_variable.**FSUpdateRate :=** (*rate* as Smallint)

Parameter: Short integer (read/write property).

Return Type: Short integer (read/write property).

Remarks: This property should be set before the FSEnabled property is set to TRUE. Setting a larger value for this property means that information will be updated less frequently, thereby consuming less of the controller's processing resources. A small value will provide for more frequent updates, but consume more processing time. Valid values for this property are from 10 to 65536 milliseconds. This is a read/write property.

Visual Basic Users: COM6SRVR interprets the FSUpdateRate as an unsigned 16-bit integer value. Visual Basic does not support use of unsigned data types. Therefore, you have to pass a signed 16-bit integer and allow the COM6SRVR to interpret it as unsigned. Thus, to allow slower update intervals than 32767 ms, a VB programmer would pass a negative value (see examples below):

Value passed is -1................result is 65535 ms/update
Value passed is -32768 ....result is +32768 ms/update
Value passed is -30000 ....result is +35536 ms/update
Value passed is -25536 ....result is +40000 ms/update
Value passed is +32767 ....result is +32767 ms/update
Value passed is +10.............result is +10 ms/update

Inputs (brick)

Description: Use the Inputs property to check the current state of the inputs (TIN) on a specific brick.
Visual Basic: object.**Inputs**(*brick* As Integer) As Long
Visual C++: long object.**GetInputs**(short *brick*)
Delphi: Longint_variable := Object_variable.**Inputs**(*brick* as Smallint)
Parameter: *brick* .... Short Integer.
Represents the number of the brick where the inputs reside. Range is 0-3. Brick 0 represents the onboard inputs. Bricks 1-3 represent expansion I/O bricks 1-3.
Return Type: Long Integer.
The value represents the current state of the inputs (TIN) for the specified brick.
Remarks: Read Only.
Requires fast status to be enabled (see FSEnabled property – page 25).

IPAddress

Description: The IPAddress property returns the controller's IP Address (NTADDR).
Visual Basic: object.**IPAddress** As Long
Visual C++: long object.**GetIPAddress**()
Delphi: Longint_variable := Object_variable.**IPAddress**
Parameter: NONE
Return Type: Long Integer.
The value represents the controller's IP Address (NTADDR).
Remarks: Read Only.
Requires fast status to be enabled (see FSEnabled property – page 25).

Limits

Description: The Limits property returns the current limit status (TLIM).
Visual Basic: object.**Limits** As Long
Visual C++: long object.**GetLimits**()
Delphi: Longint_variable := Object_variable.**Limits**
Parameter: NONE
Return Type: Long Integer.
The value represents the current limit status (TLIM).
Remarks: Read Only.
Requires fast status to be enabled (see FSEnabled property – page 25).

MotorPos (axis)

Description: The MotorPos property returns the current commanded position (TPC) for the specified axis.
Visual Basic: object.**MotorPos**(*axis* As Integer) As Long
Visual C++: long object.**GetMotorPos**(short *axis*)
Delphi: Longint_variable := Object_variable.**MotorPos**(*axis* as Smallint)
Parameter: *axis* .... Short Integer. Specifies the axis number (range is 1-8).
Return Type: Long Integer.
The value represents the current commanded position (TPC) in counts for the specified axis.
Remarks: Read Only.
Requires fast status to be enabled (see FSEnabled property – page 25).

MotorVel (axis)

Description: The MotorVel property returns the current commanded motor velocity (`TVEL`) for the specified axis.

Visual Basic: `object.`**`MotorVel`**`(`*`axis`*` As Integer) As Long`

Visual C++: `long object.`**`GetMotorVel`**`(short `*`axis`*`)`

Delphi: `Longint_variable := Object_variable.`**`MotorVel`**`(`*`axis`*` as Smallint)`

Parameter: *`axis`* .... Short Integer.
Specifies the axis number. The range for this value is 1-8.

Return Type: Long Integer.
The value represents the current commanded velocity (`TVEL`) in counts for the specified axis.

Remarks: Read Only.
Requires fast status to be enabled (see FSEnabled property – page 25).

Outputs (brick)

Description: The Outputs property returns the state of the outputs (`TOUT`) on the specified brick.

Visual Basic: `object.`**`Outputs`**`(`*`brick`*` As Integer) As Long`

Visual C++: `long object.`**`GetOutputs`**`(short `*`brick`*`)`

Delphi: `Longint_variable := Object_variable.`**`Outputs`**`(`*`brick`*` as Smallint)`

Parameter: *`brick`* .... Short Integer. Represents the number of the brick where the outputs reside. Range is 0-3. Brick 0 represents the onboard outputs. Bricks 1-3 represent expansion I/O bricks 1-3.

Return Type: Long Integer.
The value represents the state of the outputs (`TOUT`) on the specified brick.

Remarks: Read Only.
Requires fast status to be enabled (see FSEnabled property – page 25).

SystemStatus

Description: The SystemStatus property returns the system status (`TSS`) for task 0 only.

Visual Basic: `object.`**`SystemStatus`**` As Long`

Visual C++: `long object.`**`GetSystemStatus`**`()`

Delphi: `Longint_variable := Object_variable.`**`SystemStatus`**

Parameter: NONE

Return Type: Read Only.
Long Integer.
The value represents the system status (`TSS`) for task 0 only.

Timer

Description: The Timer property returns the current Timer value (`TTIM`) for task 0 only.

Visual Basic: `object.`**`Timer`**` As Long`

Visual C++: `long object.`**`GetTimer`**`()`

Delphi: `Longint_variable := Object_variable.`**`Timer`**

Parameter: NONE

Return Type: Long integer.
Represents the current Timer value (`TTIM`) for task 0 only.

Remarks: Read Only.
Requires fast status to be enabled (see FSEnabled property – page 25).

Triggers

Description: The Triggers property returns the Trigger Interrupt Status (`TTRIG`).

Visual Basic: `object.`**`Triggers`**` As Long`

Visual C++: `long object.`**`GetTriggers`**`()`

Delphi: `Longint_variable := Object_variable.`**`Triggers`**

Parameter: NONE

Return Type: Long integer.
The value represents the current state of the Trigger Interrupts (`TTRIG`).

Remarks: Read Only.
Requires fast status to be enabled (see FSEnabled property – page 25).

UserStatus

Description: The UserStatus property returns the current state of the user status register (`TUS`).
Visual Basic: `object.`**`UserStatus`**` As Long`
Visual C++: `long object.`**`GetUserStatus`**`()`
Delphi: `Longint_variable := Object_variable.`**`UserStatus`**
Parameter: NONE
Return Type: Long integer.
The value represents the current state of the user status register (`TUS`).
Remarks: Read Only.
Requires fast status to be enabled (see FSEnabled property – page 25).

VarB (varnum)

Description: The VarB property returns the value of the specified binary variable (`VARB`).
Visual Basic: `object.`**`VarB`**`(`*`varnum`*` As Integer) As Long`
Visual C++: `long object.`**`GetVarB`**`(short `*`varnum`*`)`
Delphi: `Longint_variable := Object_variable.`**`VarB`**`(`*`varnum`*` as Smallint)`
Parameter: *`varnum`* .... Short Integer. Represents number of the binary variable (`VARB`*`varnum`*). Range is 1-10.
Return Type: Long integer.
The value represents the value of the specified binary variable (`VARB`).
Remarks: Read Only.
Requires fast status to be enabled (see FSEnabled property – page 25). Refer to page 32 for information on using `VARB` variables to customize the Fast Status structure.

VarI (varnum)

Description: The VarI property returns the value of the specified integer variable (`VARI`).
Visual Basic: `object.`**`VarI`**`(`*`varnum`*` As Integer) As Long`
Visual C++: `long object.`**`GetVarI`**`(short `*`varnum`*`)`
Delphi: `Longint_variable := Object_variable.`**`VarI`**`(`*`varnum`*` as Smallint)`
Parameter: *`varnum`* .... Short Integer. Represents number of the integer variable (`VARI`*`varnum`*). Range is 1-10.
Return Type: Long Integer.
The value represents the value of the specified integer variable (`VARI`).
Remarks: Read Only.
Requires fast status to be enabled (see FSEnabled property – page 25). See page 32 for information on using `VARI` variables to customize the Fast Status structure.

COM6SRVR Error Codes

Error Code	Description
-1	Bad Ethernet connection due to socket error
-2	Ethernet connection was shut down
-3	Connection attempt failed
-4	Maximum number of Ethernet connections exceeded
-5	Ethernet or RS232 connection not yet established
-6	No filename specified
-7	Unable to locate specified file
-8	Unable to open specified file
-9	Unable to ping Ethernet connection
-10	Unable to create Ethernet socket
-11	Invalid parameter passed to function
-12	Unable to create or connect Ethernet watchdog socket
-13	Unable to create or connect Ethernet fast status socket
-14	Unable to create or connect Ethernet alarm socket
-15	Unable to create or connect Ethernet command socket
-16	Unable to create client ring buffer for Ethernet command socket
-17	SetWatchdog returns this error when Windows runs out of timers.

COM6SRVR Programming Notes

Calls to COM6SRVR

All calls to the COM6SRVR are *blocking* calls. Programming control does not return to the client program until the requested operation has been completed in the COM6SRVR. During the call, the Windows operating system continues to process other system calls and timer messages.

Be careful to avoid multiple, simultaneous calls to the server from within the same process. This situation typically arises if there are multiple timer messages being processed by the client application. Because of the nature of COM design, an error would be generated if a timer message initiates a request to the server while another server request from the same client is already in progress.

How to Include the COM6SRVR in a Visual Basic application

All of the Visual Basic samples use a technique known as *late binding* to interface with the COM6SRVR. The COM6SRVR is not linked at compile time; rather, the link is created dynamically at run time. Unlike an OCX control that needs to be added to a VB form, the COM6SRVR does not require to be added to a form. Creating and using the COM6SRVR is all performed in software. Creating the COM6SRVR is performed with the `CreateObject` VB function.

How to upgrade a Visual Basic application to use the latest COM6SRVR

You should create your Visual Basic application using the *late binding* technique to interface with the COM6SRVR (see above). The benefit, here, is that whenever a new version of the COM6SRVR is available or installed on the PC, all applications written in Visual Basic with the *late binding* technique should continue to run satisfactorily.

> The latest COM6SRVR may be downloaded from the Support portion of the Compumotor web site at http://www.compumotor.com.

How to include the COM6SRVR in a Visual C++ application

The process below demonstrates how to create a minimal dialog-based application that includes the Com6srvr. Refer also to the sample applications installed in the Motion Planner\Samples\Vc5 directory. (NOTE: The samples are installed only if you use the "custom" installation option.)

1. Using Visual C++ AppWizard …
   a. Create an MFC Exe application.
   b. Select the "Dialog based" option.
   c. In the Wizard steps it is not necessary to select the "ActiveX" or "Automation" check boxes.
2. When the application is created, include the afxole.h and afxdisp.h header files. These header files add libraries required for Ole Automation and the COleDispatchDriver class.
3. Initialize the Ole libraries with a call to AfxOleInit. (Refer to sample applications.)
4. Using Visual C++ Class Wizard …
   a. Click the "Add Class…" button and select "From a Type Library" from the drop-down menu.
   b. In the "Import From Type Library" dialog, locate the Com6srvr.tlb file and click the Open button. (The Com6srvr.tlb file is included on the Motion Planner CD-ROM. Recommendation: Copy this file to your project directory.)
   c. Select the relevant interface class ("INet" for 6K Ethernet, "IRS232" for 6K RS232, or "IGemini" for the Gemini drives). Once the class is imported, the wizard creates two new files and adds them to the project: Com6srvr.cpp and Com6srvr.h.
5. Add a class instance for the interface class you selected in Step 4.c. above (INet, IRS232 or IGemini). Refer to the m_NetServer variable in the VC_Ethernet sample.
6. To understand how to use the libraries, study the sample applications. Pay particular attention to the code related to the Connect, Write and Read methods.

How do I rebuild my Visual C++ project with the new COM6SRVR?

If the COM6SRVR “Interface” changes (because Compumotor has added new properties or methods), it is necessary to re-build your VC++ application and link in the new COM6SRVR to take advantage of any of the new features. Use this step-by-step procedure:

1. Make a backup of your project.
2. Make a backup of your current Com6srvr.exe file (typically located in the Windows\System\ directory or in the Program Files\Compumotor\Motion Planner\ directory).
3. Replace the Com6srvr.exe by overwriting the existing version with the new one from the CD-ROM. Register the new Com6srvr by executing the Com6srvr.exe once. The mouse pointer may change to an hourglass for a couple of seconds while it’s being registered.
4. Start VC++ and open the Project Workspace (**File > Open Workspace**).
5. Use menu item **View > Workspace** and select the File View tab.
6. Expand the Source Files node and highlight the Com6srvr.cpp file. Remove the file from the project by pressing the DEL key.
7. Expand the Header Files node in the Workspace window. Highlight the Com6srvr.h file and remove it from the project by pressing the DEL key.
8. Save the Workspace (**File > Save Workspace**) and exit VC++.
9. Start Windows Explorer and locate the project directory.
10. Delete the Com6srvr.cpp, Com6srvr.h and Com6srvr.tlb files.
11. Copy the new Com6srvr.tlb from the CD-ROM into the project directory.
12. Locate the project’s Class Wizard database file (.clw file) and delete it.
13. Start VC++ and Open the project Workspace.
14. Note that the COM6SRVR Interfaces (IGemini, IRS232 or INet) no longer appear in the Workspace Window’s Class View.
15. Run the class wizard (**View > Class Wizard**).
16. A dialog appears stating that the class wizard database file does not exist and prompts you to re-build it from your source files. Click the YES button.
17. A “Select Source Files” dialog appears. Click OK.
18. The MFC Class Wizard dialog now appears. Click the Add Class button and select “From a type library” on the drop-down menu.
19. Locate the Com6srvr.tlb file and click the Open button.
20. A “Confirm Classes” dialog appears. All classes (IRS232, INet and Igemini) are highlighted. Select only the classes that are of particular interest. Accept the recommended Header and Implementation File names of Com6srvr.h and Com6srvr.cpp. Click the OK button.
21. Back at the MFC Class Wizard, click OK button.
22. Rebuild the complete project (**Build > Rebuild All**).
23. Save the Workspace (**File > Save Workspace**).

Overview of the VB5 Samples on the CD-ROM

The VB5 samples are installed if you select Custom installation when installing Motion Planner. The samples are installed on the hard disk in the …\Motion Planner\Samples\VB5 directory. Each sample has its own sub-directory.

The VB5 samples are located on the CD-ROM in the directory 6K\Samples\VB5. Again, each sample has its own sub-directory. To copy the files from the CD-ROM use Windows Explorer to copy the sub-directory and its contents to a new location on your hard disk. NOTE, however, that the file attributes will be set to read-only, because the CD-ROM is read-only media. To change the attributes: Using Windows Explorer, locate the sub-directory. Open the sub-directory such that all files are now visible. Use the **Edit > Select All** menu to select all files. Then, use the menu **File > Properties**, the dialog shows a Read-Only box with a check mark against it. Uncheck the box and click OK. Now it is possible to open the files and save them in Visual Basic. When using Motion Planner's Custom install to install the samples, the read-only attribute is automatically reset.

There are three VB5 Samples:

- SimpleOne
- SimpleOnePlus
- Terminal
- 6K_Capture_Sample

To become familiar with the COM6SRVR and how it is used in Visual Basic, review the SimpleOne application. This sample demonstrates the absolute basics. It deals with creating an instance of the COM6SRVR using Visual Basic's `CreateObject` function. Then it demonstrates how to make a connection to the 6K and how to close or disconnect a connection. The main form also has two buttons, "Send Command" and "Read Response," to demonstrate the Write and Read methods of the COM6SRVR. The "Easy Get Data" and "Get Data" buttons demonstrate two techniques to get data from the Ethernet Fast Status.

The Terminal application builds upon the basics introduced by the SimpleOne application. It creates a Terminal to communicate with the 6K via Ethernet or RS-232. Buttons to demonstrate the `SendFile`, `GetFile` and `SendOS` methods are included. The Fast Status and Alarms are demonstrated in the form frmFastStatus.

6K_Capture_Sample is a utility as much as a sample. It can be used to capture fast status packets from the 6K and store the data for later use.

Overview: Using Fast Status

Fast Status is a tool you can use to keep the COM6SRVR informed of conditions within the 6K (conditions such as the values of motor position, encoder position, axis status, and the current value of some of the integer and binary variables). A client application can interrogate the COM6SRVR's fast status data to check various 6K conditions (input states, variable values, system conditions, etc.) in two ways:

- Interrogate individual fast status elements via their respective properties. This is the easiest to implement.
- Interrogate the entire fast status data structure. This is more complex, is advantageous when you need to check many data elements at one time.

Interrogating Individual Fast Status Elements

The fast status data structure comprises many elements. Each element may be interrogated through the use of its respective property. The table below lists each fast status data element and its respective property.

Fast Status — Packet Data Definition (280 bytes total)

Type	Description	Bytes	Property (see pages 22-28)
WORD wUpdateID	Unique update ID for synch channel	2	*No property available*
WORD wCounter	Time Frame Counter	2	**Counter**
DWORD dwMotorPos[8]	Commanded Position (TPC)	32	**MotorPos**
DWORD dwEncPos[8]	Encoder Position (TPE)	32	**EncoderPos**
DWORD dwMotorVel[8]	Commanded Velocity (TVEL)	32	**MotorVel**
DWORD dwAxisStatus[8]	Axis Status (TAS)	32	**AxisStatus**
DWORD dwSystemStatus	System Status (TSS)	4	**SystemStatus**
DWORD dwErrorStatus	Error Status (TER)	4	**ErrorStatus**
DWORD dwUserStatus	User Status (TUS)	4	**UserStatus**
DWORD dwTimer	Timer (TTIM)	4	**Timer**
DWORD dwLimits	Limit Status (TLIM)	4	**Limits**
DWORD dwInputs[4]	Input Status (TIN)	16	**Inputs**
DWORD dwOutputs[4]	Output Status (TOUT)	16	**Outputs**
DWORD dwTriggers	Trigger Status (TTRIG)	4	**Triggers**
WORD wAnalogIn[2]	Analog Input Value (TANI - in counts)	4	**AnalogInput**
DWORD dwVarb[10]	Binary Variable Values (VARB1 – VARB10)	40	**VarB**
DWORD dwVari[10]	Integer Variable Values (VARI1 – VARI10)	40	**VarI**
DWORD dwIPAddress	IP Address (NTADDR)	4	**IPAddress**
DWORD dwCmdCount	Command Count	4	**CommandCount**

NOTE: Each call to the COM6SRVR incurs overhead; therefore, you should be aware that interrogating many different fast status elements at one time is slower than interrogating the entire data structure (see below).

Interrogating the Entire Fast Status Structure

Accessing data in the fast status structure is more complex than interrogating individual elements, but it has the advantage that only one call to the COM6SRVR is required to access the entire fast status structure — this reduces overhead when multiple data elements are required.

The COM6SRVR FastStatus property (see page 24) returns a VARIANT data type, which is actually an array of bytes. The array of bytes is copied into a structure. Ordering the array of bytes and structure elements is very important to ensure that the correct data bytes are copied into the correct structure elements. (Do not change the fast status structure or TypeDef.) In Visual Basic, use the Windows API CopyMemory function to copy bytes into the structure; in Visual C++, the copying is performed through use of SAFEARRAYS (refer to the MakeFastStatus function in the VC_Ethernet sample provided).

There are two techniques for updating the Fast Status: *Streaming* and *On Demand*.

- The *Streaming* technique is particularly useful for HMI type applications where a constant stream of data at a set interval is required. To use streaming, set a streaming interval with the FSUpdateRate property (see page 25) and enable streaming by setting the FSEnabled property to TRUE (see page 25).
- The *On Demand* technique can reduce network traffic by sending fast status packets only when required, rather than at a pre-defined interval. The fast status packet can be updated by a call to the COM6SRVR RequestFastStatusUpdate method (see page 18), or under 6K program control with the 6K command `NTSFS` (see page 41). Similarly, when a 6K generates an alarm, such as `INTSW1`, a fast status packet (280 bytes) is automatically sent to the COM6SRVR. Note that the On Demand technique operates independent of the FSEnabled property and the FSUpdateRate property. The sample VB program called SimpleOnePlus demonstrates the use of RequestFastStatusUpdate.

Customizing Fast Status

The Fast Status structure (see page 24) provides most of the frequently required system parameters (e.g., motor position and axis status). However, there are times when a specific status condition or parameter is required, but not provided by default. One such example might be Following Status (`TFS`).

The Fast Status packet always includes ten integer variables (`VARI1` – `VARI10`) and ten binary variables (`VARB1` – `VARB10`). This allows you to customize part of the Fast Status packet by copying status conditions of interest into the `VARI` or `VARB` variables. Below are two scenarios that demonstrate two methods of using variables to customize the Fast Status.

Scenario 1: An HMI application must inform the operator of the total number of products made, the number of products that meet specification (passes), and number of products that fail to meet specification. In this type of scenario, the Total Number is assigned to `VARI1`, the number of Passes to `VARI2`, and the number of failures to `VARI3`. These variables are then updated as needed. For example, when a product is made, `VARI1` is incremented. This insures the data is automatically updated in the Fast Status packet; then the HMI application can use the VARI(1) property to interrogate the `VARI1` value.

Scenario 2: Another method to maintain the information is to use a `PLCP` program, launched in the Scan Mode with the `SCANP` command. The PLCP program updates the variable (`VARI` or `VARB`). In this manner, the parameter of interest can be mapped to a specific variable. Example: Use `VARB` to allow monitoring of Following Status (`TFS`) and allow `VARI1` to auto-increment (see code at left).

Then use the “ActiveXFastPanel” in Motion Planner’s Panel Maker to monitor the “Variables” grid. You’ll notice that `VARI1` is continually incrementing and that as you enable and disable Following on axis 1 (`FOLEN1` / `FOLEN0`), bit 6 of `VARB1` is set and cleared.

```
DEL PLCP1
DEF PLCP1
  VARI1 = VARI1 + 1
  VARB1 = 1FS
END

PCOMP PLCP1

VARI1=0

SCANP PLCP1
```

Overview: Using Variable Packets

Using variable packets, you can quickly and efficiently transfer large amounts of data from the COM6SRVR to the 6K. The variables packet comprises: integer variables 1-12 (VARI1 – VARI12), real variables 1-12 (VAR1 – VAR12), and binary variables 1-8 (VARB1 – VARB8).

Variable packets can be sent by one of two COM6SRVR methods:

- The SendVariable method (see page 19) allows transmission of one variable. To use SendVariable, you define: (a) a mask to specify which variable to update, and (b) the value of the variable. Mask bits for Visual Basic and Visual C++ are provided below.
- The SendVariablePacket method (see page 20) allows one or all variables to be sent in one packet. The SendVariablePacket method requires a SendVariable structure to be populated, copied into a variant and then pass the variant to the SendVariablePacket Method. The SendVariable structure comprises several elements: a Mask, several reserved elements and an array of twelve elements for integer variables, an array of twelve elements for real variables and an array of eight elements for binary variables (see structure lists below). Several mask constants can be Or'ed together to allow 6K variables to be updated (see mast lists below).

TIP: Using the a variable packet method (SendVariable or SendVariablePacket) and an *On Demand* Fast Status interrogation technique (RequestFastStatusUpdate method or the 6K command `NTSFS`) provides a clean and efficient communication tool between the client application and the 6K program. Although variables can be sent as a command using the *Write* method, the time taken to parse the command and check data validity is longer than the time to send an entire variable packet.

A sample VB program called SimpleOnePlus is provided; it demonstrates the use of the SendVariable and SendVariablePacket COM6SRVR methods.

Variable Structure for Visual Basic

```
Type SendVariableStructure
  Mask As Long
  Reserved1 As Long
  Reserved2 As Long
  Reserved3 As Long
  VarI(1 To 12) As Long
  VarR(1 To 12) As Double
  VarB(1 To 8) As Long
End Type
```

Variable Structure for Visual C++

```
typedef struct VARIABLEPACKETStruct {
 int nVariableMask;
 int nReserved1;
 int nReserved2;
 int nReserved3;
 int VARI[12];
 double VAR[12];
 int VARB[8];
} VARIABLEPACKET, *LPVARIABLEPACKET;
```

Variable Packet Mask Bits for Visual Basic

```
Public Const VARI1 As Long = 1
Public Const VARI2 As Long = 2
Public Const VARI3 As Long = 4
Public Const VARI4 As Long = 8
Public Const VARI5 As Long = 16
Public Const VARI6 As Long = 32
Public Const VARI7 As Long = 64
Public Const VARI8 As Long = 128
Public Const VARI9 As Long = 256
Public Const VARI10 As Long = 512
Public Const VARI11 As Long = 1024
Public Const VARI12 As Long = 2048

Public Const VAR1 As Long = 4096
Public Const VAR2 As Long = 8192
Public Const VAR3 As Long = 16384
Public Const VAR4 As Long = 32768
Public Const VAR5 As Long = 65536
Public Const VAR6 As Long = 131072
Public Const VAR7 As Long = 262144
Public Const VAR8 As Long = 524288
Public Const VAR9 As Long = 1048576
Public Const VAR10 As Long = 2097152
Public Const VAR11 As Long = 4194304
Public Const VAR12 As Long = 8388608

Public Const VARB1 As Long = 16777216
Public Const VARB2 As Long = 33554432
Public Const VARB3 As Long = 67108864
Public Const VARB4 As Long = 134217728
Public Const VARB5 As Long = 268435456
Public Const VARB6 As Long = 536870912
Public Const VARB7 As Long = 1073741824
Public Const VARB8 As Long = &H80000000
```

Variable Packet Mask Bits for Visual C++

```
#define VARI1  0x00000001
#define VARI2  0x00000002
#define VARI3  0x00000004
#define VARI4  0x00000008
#define VARI5  0x00000010
#define VARI6  0x00000020
#define VARI7  0x00000040
#define VARI8  0x00000080
#define VARI9  0x00000100
#define VARI10 0x00000200
#define VARI11 0x00000400
#define VARI12 0x00000800

#define VAR1   0x00001000
#define VAR2   0x00002000
#define VAR3   0x00004000
#define VAR4   0x00008000
#define VAR5   0x00010000
#define VAR6   0x00020000
#define VAR7   0x00040000
#define VAR8   0x00080000
#define VAR9   0x00100000
#define VAR10  0x00200000
#define VAR11  0x00400000
#define VAR12  0x00800000

#define VARB1  0x01000000
#define VARB2  0x02000000
#define VARB3  0x04000000
#define VARB4  0x08000000
#define VARB5  0x10000000
#define VARB6  0x20000000
#define VARB7  0x40000000
#define VARB8  0x80000000
```

OS 5.1.0 Firmware Enhancements

OS 5.1.0 provides these firmware enhancements:

New Commands

`ANO`	Set Analog Output Value	page 58
`[ANO]`	Analog Output Value (assignment/comparison operand)	page 58
`DSTALL`	Drive Input Stall Detection (enable/disable)	page 42
`IN`	Virtual Input Override	page 35
`INVARI`	Map Inputs to Integer Variable	page 45
`KIOEN`	Kill on EVM32 I/O Disconnect (enable/disable)	page 59
`NTSFS`	Ethernet Send Fast Status Packet	page 41
`PGOWHN`	Path Conditional Go	page 37
`PVF`	Path Final Velocity	page 36
`[SC]`	Controller Status (assignment/comparison operand)	page 44
`[SCAN]`	PLC Scan Runtime (assignment/comparison operand)	page 39
`TANO`	Display Status of Analog Outputs	page 59
`TNT`	Display Ethernet Status	page 40
`TSC`	Display Controller Status	page 43
`TSCF`	Display Controller Status (full-text report)	page 44

Virtual Inputs

The virtual input override feature, implemented with the `IN` command allows you to use any 32-bit data as an I/O brick containing up to 32 inputs.

IN Virtual Input Override

		Product	Rev
Type	`Inputs`	6K	5.1
Syntax	`<!><B>IN<=xx>`		

Units	`B = I/O brick number`
	`xx = 32-bit data operand (see list below)`		

Range	`B = 1-8`		
Default	`n/a`		
Response	`n/a`		
See Also	`[ IN ], INDEB, INEN, INFNC, INPLC, INSTW, TIN, TIO, VARSHI, VARSHO`		

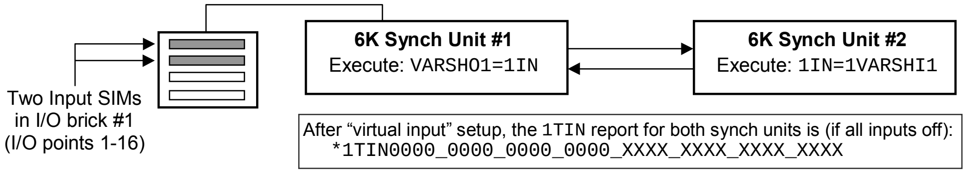
The Virtual Input Override (`IN`) command allows you to substitute almost any 32-bit data parameter as a virtual input brick of 32 inputs. The virtual inputs behave similar to real inputs in that they are affected by `INEN` and `INLVL`, and they affect `INFNC`, `INPLC`, `INSTW`, `INDUST`, `ONIN`, and `GOWHEN(<B>IN=b<bbbb>)` commands. Unlike real inputs, virtual inputs are not affected by the `INDEB` debounce setting. The data operands allowed for virtual input assignments are: `A`, `AD`, `ANI`, `ANO`, `AS`, `ASX`, `D`, `DAC`, `DKEY`, `ER`, `FB`, `FS`, `IN`, `INO`, `LIM`, `MOV`, `NMCY`, `OUT`, `PANI`, `PC`, `PCCn`, `PCEn`, `PCME`, `PE`, `PER`, `PMAS`, `PME`, `PSHF`, `PSLV`, `SC`, `SCAN`, `SEG`, `SS`, `SWAP`, `SYNCH`, `TASK`, `TIM`, `TRIG`, `US`, `V`, `VARI`, `VARB`, `VEL`, `VELA`, `VMAS`, and `VARSHI`.

NOTE: A virtual input can only be defined for I/O bricks that are not connected on the serial I/O network (remember that up to 8 I/O bricks are allowed). For example, if your 6K unit has two I/O bricks, you can designate I/O bricks 3-8 as virtual I/O bricks.

There are two main uses of the virtual input override feature:

- Virtual inputs are helpful for systems using the Synchronous Bus option (details in the *6K Synchronization Bus Guide*, p/n 88-018179-01). Instead of requiring each 6K unit to have its own I/O brick with inputs, one "global" input brick can be shared across the synchronous bus using the `VARSHO<x>=<B>IN` command, and the other synch units can use this data as a virtual input brick using the `<B>IN=<i>VARSHI<x>` command.

  For example, suppose two 6K controllers are connected on a synch bus and synch unit #1 has one I/O brick (input brick #1) with 2 input SIMs in slots 1 and 2. Synch unit #2 has no I/O bricks connected, but needs to base its motion control of the state of the inputs on synch unit #1's I/O brick. Synch unit #1 executes `VARSHO1=1IN` to assign the state of all 32 bits of input data from brick #1 (`1IN`) to shared output variable #1 (`VARSHO1`). Synch unit #2 executes `1IN=1VARSHI1` to assign synch unit #1's `VARSHO1` data (which is `1IN`) to be its "virtual" input brick #1 (identical to synch unit #1).



After "virtual input" setup, the `1TIN` report for both synch units is (if all inputs off):
`*1TIN0000_0000_0000_0000_XXXX_XXXX_XXXX_XXXX`

- Virtual inputs also provide programming input functionality for data or external events (see operand list above) that are not ordinarily represented by inputs.
  For example, suppose a PLC is sending binary data via the `VARB1` command to the 6K. If the binary state of `VARB1` is assigned to input brick 2 (`2IN=VARB1`), the 6K can respond based on programmable input functions set up with the `INFNC` command.

```
2TIN               ; Brick 2 is not connected; therefore, the 6K will
                   ; respond with an error message: "*INCORRECT I/O BRICK"
2IN=VARB1          ; Map the binary state of VARB1 to be the
                   ; input state of "virtual" input brick 2 (2IN)
VARB1=b10100000    ; Change "virtual" input brick 2IN to a new VARB1 value
2TIN               ; Check the input status. The response will be:
                   ; "*2IN1010_0000_0000_0000_0000_0000_0000_0000"
```

Contouring Enhancements

Two commands, PVF and PGOWHN, were added improve the flexibility and ease-of-use when implementing contouring motion programs.

PVF Path Final Velocity

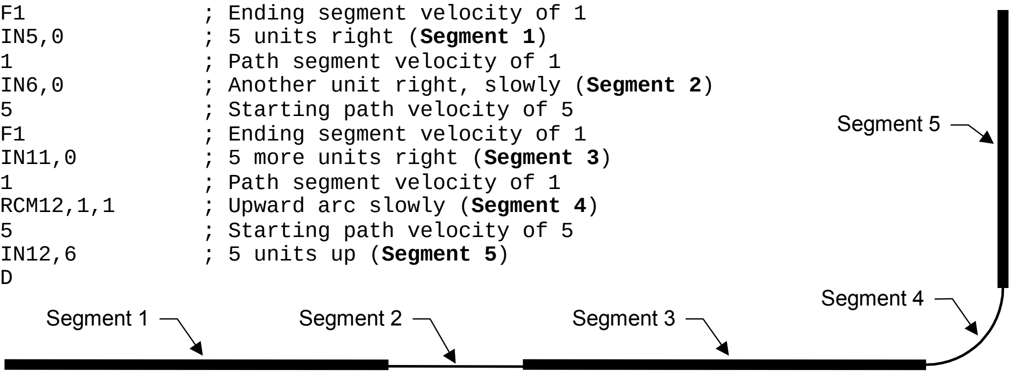
Type	Path Contouring	**Product**	**Rev**
Syntax	<!>PVF<r>	6K	5.1
Units	r = units/sec (scalable with SCLD)		
Range	Stepper axes: 0.0000-2048.0000 (max. depends on SCLD & PULSE) Servo axes: 0.0000-6500.0000 (max. depends on SCLD)		
Default	n/a		
Response	n/a		
See Also	PA, PAD, PV, SCALE, SCLD		

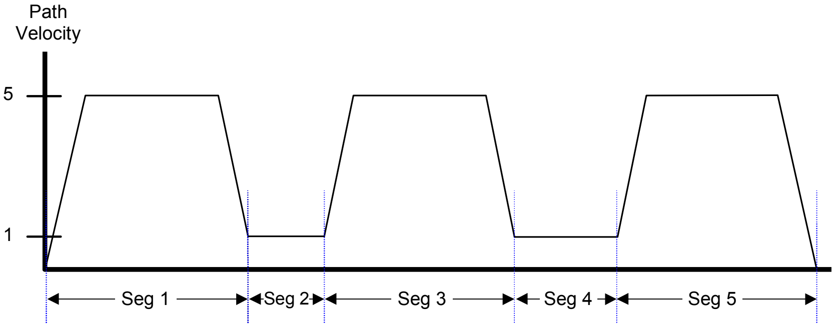
The PVF command allows a line or arc segment to terminate with a final segment velocity (PVF) which may be different from the velocity traveled for the majority of that segment (specified with PV). PVF must be smaller than or equal to PV, and the path velocity change will take place at the PAD deceleration rate. Like the other path motion parameters (e.g., PV, PA, PAD), the PVF velocity is applied to the next line or arc compiled; however, unlike these other commands, the PVF command applies only to the next line or arc compiled. All subsequent lines and arcs terminate at the PV value currently in effect. For each line or arc that needs to terminate at a velocity different than PV, a new PVF command must be issued, even if the PVF value has not changed.

The most common use for the PVF command will be to cause a preceding line segment to decelerate to the path velocity at which the next line or arc needs to travel. In this case, the PVF value for the preceding line would be the same as the PV value of the reduced speed segment. This feature eliminates the need to create a special deceleration segment in the path in order to have the entire subsequent line or arc travel at the reduced speed.

Example: In the example below, segments 2 and 4 must travel at a path velocity of 1, while the remainder to the path travels at a path velocity of 5.

```
DEF CORNER
PAB1                ; Use absolute positions
PA10                ; Path acceleration of 10
PAD10               ; Path deceleration of 10
PV5                 ; Starting path velocity of 5
PVF1                ; Ending segment velocity of 1
PLIN5,0             ; 5 units right (Segment 1)
PV1                 ; Path segment velocity of 1
PLIN6,0             ; Another unit right, slowly (Segment 2)
PV5                 ; Starting path velocity of 5
PVF1                ; Ending segment velocity of 1
PLIN11,0            ; 5 more units right (Segment 3)
PV1                 ; Path segment velocity of 1
PARCM12,1,1         ; Upward arc slowly (Segment 4)
PV5                 ; Starting path velocity of 5
PLIN12,6            ; 5 units up (Segment 5)
END
```

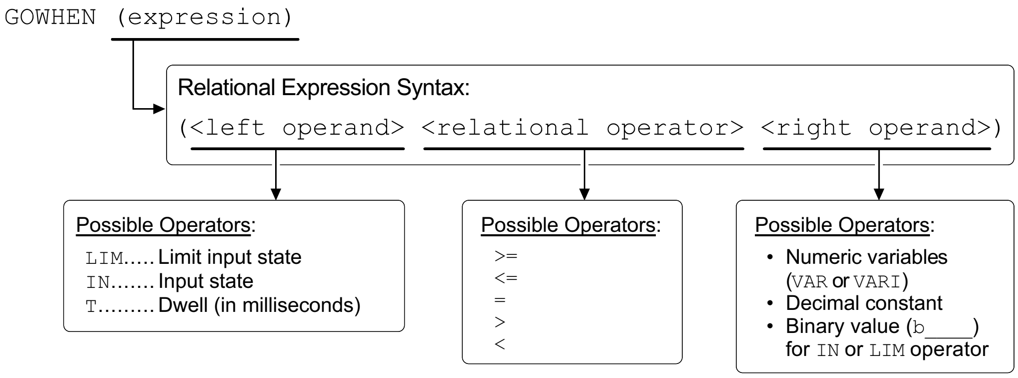




PGOWHN Path Conditional Go

		Product	Rev
Type	`Path Contouring`	6K	5.1
Syntax	`<!>PGOWHN(expression)`		
Units	`n/a`		
Range	`Up to 80 characters (including parentheses)`		
Default	`n/a`		
Response	`n/a`		
See Also	`PA, PAD, PV, PVF, POUT`		

The `PGOWHN` command allows a fixed or conditional delay in path travel to occur as part of the contouring motion profile, without requiring program execution to monitor the `PGOWHN` conditions. Combined with the `PVF` command, `PGOWHN` provides a convenient method of imbedding a dwell within a contour.


GOWHEN (expression)

Relational Expression Syntax:

(<left operand> <relational operator> <right operand>)

Possible Operators:

LIM..... Limit input state
IN....... Input state
T......... Dwell (in milliseconds)

Possible Operators:

>=
<=
=
>
<

Possible Operators:

- Numeric variables (VAR or VARI)
- Decimal constant
- Binary value (b____) for IN or LIM operator

The expression in the `PGOWHN` command may be specified as a time (dwell) in milliseconds (e.g., `PGOWHN(T=2000)`), or as inputs or limits matching the specified binary pattern (e.g., `PGOWHN(LIM.3=B1)` or `PGOWHN(2IN.6=B0)`). All participating axes will be at rest during a dwell, even if the previous segment had not ended in zero path velocity. In the latter case (limit or input condition), there will be an abrupt change to zero path velocity, until the `PGOWHN` condition is satisfied.

When the `PGOWHN` condition is satisfied, path velocity will ramp to the next segment's `PV` value. Although no motion occurs during a `PGOWHN` segment, it occupies one segment of compiled memory. Like the line and arc segments, an output pattern may be asserted for the duration of the `PGOWHN` dwell by preceding the `PGOWHN` statement with a `POUT` statement (see example below).

Example:

In the example below, the total path travel is six inches. At five inches, motion must stop until input 3 goes active. During this time, output 2 must be asserted. When input 3 goes active, output 2 must go off and motion then resumes for the final inch.

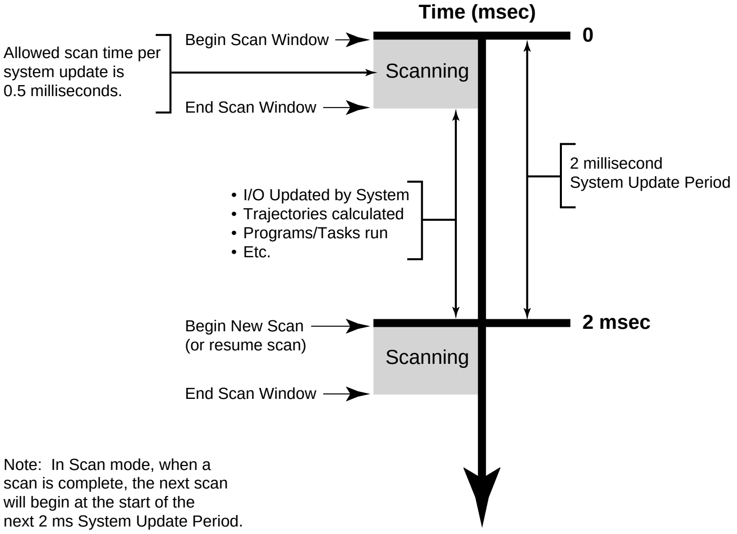
```
DEF BISEG
PAB1             ; Use absolute positions
PA10             ; Path acceleration of 10
PAD10            ; Path deceleration of 10
PV5              ; Starting path velocity of 5
PVF0             ; Ending segment velocity of 0
POUT.2-0         ; Start with onboard output 2 off
PLIN5,0          ; 5 units right
POUT.2-1         ; Output 2 on during dwell
PGOWHN(IN.3=B1)  ; Dwell until onboard input 3 is active
POUT.2-0         ; Output 2 off during motion
PLIN6,0          ; Another unit right to finish
END
```

PLC Scan Mode Enhancements

The internal scan method was changed to improve throughput and ease of use.

Before 5.1.0 (Old Method)

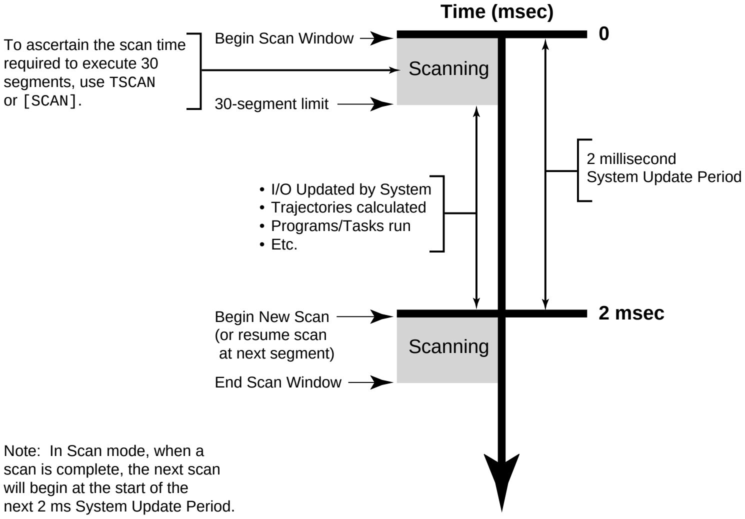
The PLCP program scan was allowed a 0.5 ms window at the beginning of each 2-ms system update period. If the contents of the PLCP program required more than 0.5 ms to scan, it was paused for the remaining 1.5 ms of the 2-ms update period and then resumed at the next 0.5 ms scan window in the subsequent 2-ms update period.


Time (msec)
0
Allowed scan time per system update is 0.5 milliseconds.
Begin Scan Window
Scanning
End Scan Window
2 millisecond System Update Period
• I/O Updated by System
• Trajectories calculated
• Programs/Tasks run
• Etc.
Begin New Scan (or resume scan)
Scanning
End Scan Window
2 msec

Note: In Scan mode, when a scan is complete, the next scan will begin at the start of the next 2 ms System Update Period.

5.1.0 (New Method)

Instead of waiting for a certain amount of time before stopping a scan, and then continuing at the next 2-ms update, the scan will stop after 30 segments have been executed and resume at the next 2-ms update. PLCP programs, when compiled, are comprised of a linked list of PLC segments. During a scan, each segment is counted until the total number of segments executed exceeds 30.


Time (msec)
0
To ascertain the scan time required to execute 30 segments, use TSCAN or [SCAN].
Begin Scan Window
Scanning
30-segment limit
2 millisecond System Update Period
• I/O Updated by System
• Trajectories calculated
• Programs/Tasks run
• Etc.
Begin New Scan (or resume scan at next segment)
Scanning
End Scan Window
2 msec

Note: In Scan mode, when a scan is complete, the next scan will begin at the start of the next 2 ms System Update Period.

Reminder: Most statements (commands) allowed in a PLCP program consume one segment of compiled memory after the program is compiled with `PCOMP`; the exceptions are `VARI` and `VARB` (each consume 2 segments) and `IF` statements. Each `IF` conditional statement consumes 2 segments, plus 1 segment for each additional evaluation in compound statements. For example, `IF(IN.1=b1)` consumes 2 segments, and `IF(IN.1=b1 AND 1AS.1=b0)` consumes three segments.

If the 30-segment limit occurs while executing a multi-segment statement, then that statement will finish no matter how many segments it executes. For example, if 29 segments have executed, then the next segment will cause the scan to pause until the next 2-ms update. If that next statement is VARI1=1PE, which executes 2 segments, then VARI1=1PE will complete its operations before pausing the scan.

What does this all mean? It means that the scan functionality of the 6K is now deterministic in its run time. Each pass, if taking the same path through the conditional branches (IF statements), will always report the same TSCAN value.

Additional features have been added to OS 5.1.0 in support of PLCP programs:

- Before OS 5.1.0, TSCAN was the only way to ascertain the scan runtime, but a new parameter, [SCAN] – see description below, has been added such that the scan runtime can be assigned to a variable (e.g., VARI1=SCAN).
- As of OS revision 5.1.0, when a PLCP program is being executed in Scan Mode, bit #3 is set in the Controller Status register (reported with the <u>new</u> TSC, TSCF and SC commands).
- As of OS revision 5.1.0, PLCP programs may now include HALT, BREAK, TIMST, and TIMSTP and commands.

  HALT: If the PLCP program is executed with PRUN, HALT will stop the PLC program (and the program that executed the PRUN statement) running in that task. If the PLCP program is executed with SCANP, HALT will kill the scan.

  BREAK: If the PLCP program is executed with PRUN, BREAK will stop a PLC program which is in process. If the PLCP program is executed with SCANP, HALT will end that scan and the scan will restart at the next 2-millisecond update.

[SCAN] Scan Time

Type	PLC Scan Program; Assignment or Comparison	**Product**	**Rev**
Syntax	See below	6K	5.1
Units	n/a		
Range	n/a		
Default	n/a		
Response	n/a		
See Also	PLCP, SCANP, TSCAN		

Use the SCAN operand to assign the PLCP runtime (the duration it took the last PLCP program scan to complete) to a variable, or to make a comparison against another value. A compiled PLCP program is launched into Scan mode using the SCANP command. During each 2-ms update, the PLCP program is scanned until 30 segments have been executed. It the PLCP program takes more than 30 segments to complete, the program will be paused and then resumed at the next 2-ms system update. The SCAN value is in multiples of the 2-ms system update period.

Example:

```
SCANP PLCP1   ; Start execution of compiled PLCP program 1 in
              ; Scan mode
VARI1=SCAN    ; Assign the duration of the last scan to integer
              ; variable VARI1
```

Ethernet Enhancements

Four enhancements were added.

- A handshake between the communications server (COM6SRVR) and the 6K detects for both sides whether or not the Ethernet connection goes down. In the 6K, if the handshake is missed from the COM6SRVR, error bit 22 (`ER.22` and `TER.22`) is set. Bit #22 is also added to the error-checking list; thus, if this bit is enabled with `ERROR`, the 6K will branch (GOSUB) to the `ERRORP` program if the Ethernet connection fails. To recover, disable the relevant error checking bit (`ERROR.22-0`), re-establish the Ethernet connection, then re-enable the error checking bit (`ERROR.22-1`).
- The new `TNT` reports the current Ethernet conditions (see command description below).
- The `NTSFS` command as added to allow you to send a fast status packet to the COM6SRVR as needed, without having to use the COM6SRVR to request the update.
- An additional option was added to `NTFEN` (see description on page 41). `NTFEN2` is recommended as the standard Ethernet communication mode, even if you are using a closed network and no file sharing. `NTFEN2` is especially helpful if you are using Windows NT, or Windows 95/98 with File Sharing and/or an open network. **NOTE**: When using `NTFEN2`, you must also follow the ARP -S Static Mapping procedure (see page 5). Refer also to the Ethernet configuration procedures on page 5.

TNT Transfer Ethernet Status

		Product	Rev
Type	`Transfer`	6K	5.1
Syntax	`<!>TNT`		
Units	`n/a`		
Range	`n/a`		
Default	`n/a`		
Response	`TNT: (see sample response below)`		
See Also	`NTADDR, NTFEN`		

The `TNT` command reports the current Ethernet conditions (see sample response below).

```
*6K ETHERNET STATUS
*Ethernet enabled: NTFEN2
*6K IP address: 172.20.34.156
*6K Ethernet address: 0-144-85-0-0-1 (decimal)
*6K Ethernet address: 0-90-55-0-0-1 (hex)
*6K Ethernet connected
```

How to interpret the status report

- **Ethernet enabled (`NTFEN`):** `NTFEN2` is recommended as the standard Ethernet communication mode, even if you are using a closed network and no file sharing. `NTFEN2` is especially helpful if you are using Windows NT, or Windows 95/98 with File Sharing and/or an open network. **NOTE**: When using `NTFEN2`, you must also follow the ARP -S Static Mapping procedure (see page 5).
  - **6K IP address:**
  This is the IP address that you specified (`NTADDR` command) in the configuration process (see configuration procedure on page 5).
- **6K Ethernet address:**
  This is the Ethernet “MAC” address that is given to the 6K unit at the factory.
- **6K Ethernet connected/not connected:**
  Ethernet is “Connected” if the Ethernet hardware connection is good and your PC is communicating to the 6K over a valid Communication Server/Motion Planner connection.

NTSFS Ethernet Send Fast Status Packet

		Product	Rev
Type	`Communication`	6K	5.1
Syntax	`<!>NTSFS`		
Units	`n/a`		
Range	`n/a`		
Default	`n/a`		
Response	`n/a`		
See Also	`NTFEN`		

The `NTSFS` command allows a 6K program to send a Fast Status packet, as needed, from the 6K controller to a PC application (COM6SRVR). In a typical application, a PC may be controlling a machine and using the COM6SRVR's Variable Packets to send variables and "job" instructions to the 6K. When the job is complete, the 6K could execute the `NTSFS` command to send a Fast Status packet back to the PC, thus indicating that the job was completed. Using the VarI and VarB elements of the Fast Status packet allows the 6K to communicate that the job was completed satisfactorily or that a specific error caused the job to fail.

NTFEN Ethernet Communication Enable

		Product	Rev
Type	`Communication Interface`	6K	5.1
Syntax	`NTFEN<b>`		
Units	`b = Ethernet enable bit`		

Range	`0 (disabled),`
	`1 (enabled for Windows 95/98, no File Sharing, closed network),`
	`2 (enabled – recommended for most Ethernet configurations)`		

Default	`0 (Ethernet communication is disabled)`		
Response	`NTFEN:   *0`		
See Also	`NTADDR, NTMASK, TNT, TNTMAC`		

Use the `NTFEN` command to enable or disable Ethernet communication. The factory default configuration is that Ethernet communication is disabled (`NTFEN0`). To enable Ethernet communication, you must connect to the 6K's "RS-232" serial port and send the `NTFEN` command which is appropriate for your Ethernet configuration (see options below). Then you can communicate over the Ethernet connection.

Options:

`NTFEN0`: Use `NTFEN0` to disable Ethernet communication.

`NTFEN1`: `NTFEN1` is primarily intended for use with Windows 95/98 with no File Sharing and a closed network.

`NTFEN2`: <u>`NTFEN2` is recommended as the standard Ethernet communication mode, even if you are using a closed network and no file sharing.</u> `NTFEN2` is especially useful if you are using Windows NT, or Windows 95/98 with File Sharing and/or an open network.
NOTE: When using `NTFEN2`, you must also follow the ARP -S Static Mapping procedure (see page 5).

NOTE: You can communicate to <u>either</u> the "ETHERNET" port <u>or</u> the "RS-232" port at one time.

The `NTFEN` setting is automatically saved in battery backed RAM. Thus, if you cycle power, Ethernet communication will still be enabled.

If the Ethernet connection fails, the 6K will set error status bit #22 (see `ER`, `TER`, and `TERF`) if error-checking bit #22 is enabled with the `ERROR` command. Also, if this error occurs, the 6K will branch to the `ERRORP` program.

Gemini Drive Stall Detection

Compumotor's Gemini GT Series of digital drives provide an encoder-less stall detect feature. When the GT detects a stall, it activates the Stall Output, which is connected to the 6K via pin #4 ("Stall Input") on the **DRIVE** connector.

To provide additional control in the event of a GT-detected stall, OS 5.1.0 introduces the DSTALL command to enable Drive Input Stall Detection (see description below), which functions much like the 6K's encoder stall detection feature (ESTALL). Note that even if DSTALL is not enabled, the state of the Stall Input can still be monitored at all times with Extended Axis Status bit #7 (reported with TASX, TASXF, and ASX).

DSTALL Drive Input Stall Detection

		Product	**Rev**
Type	Controller Configuration		
Syntax	<!><@><a>DSTALL<b><b><b><b><b><b><b><b>	6K	5.1
Units	b = enable bit		
Range	b = 0 (disable), 1 (enable), or X (don't change)		
Default	0 (disabled)		
Response	DSTALL: *DSTALL0000_0000 1DSTALL: *DSTALL0		
See Also	[AS], [ASX], [ER], ERROR, ERRORP, ESK, ESTALL, TAS, TASX, TER		

The DSTALL command determines if the Stall Input on pin #4 of the **DRIVE** connector will be checked as Drive Stall indicator.

The state of the Stall Input can be monitored at all times with Extended Axis Status bit #7 (reported with TASX, TASF, and the ASX assignment/comparison operand); if left unconnected, the input is low, and status bit #7 will be set (reports a "1"). If this input is enabled as a drive stall indicator with the DSTALL1 command, a low input will be interpreted as a Drive Stall.

When a Drive Stall is detected, the 6K responds as follows: (this response is the same as that for Encoder Stall Detection, which is enabled with the ESTALL command)

- The stall is reported with Axis Status bit #12 (reported with TAS, TASF, and AS).
- If ERROR error-checking bit #1 is enabled (ERROR.1-1):
  - The stall is reported with Error Status bit #1 (reported with TER, TERF, and ER).
  - The 6K branches to the assigned ERRORP program.
- If the Kill-on-Stall feature is enabled (ESK1), the 6K immediately stops pulses from being sent to the affected axis.

Example:

```
DEF SETUP1
WAIT(2ASX.7=B0 AND 3ASX.7=B0)   ; Be sure stall inputs are not active
DSTALL0110    ; Enable checking the Stall Input on axes 2 and 3
ENCCNT1001    ; Enable encoder counting on axes 1 and 4
ESTALL1001    ; Enable checking for 6K encoder stall on axes 1 and 4
ESK1111       ; Enable Kill On Stall function for axes 1-4
END
```

TAS, TASX, TFS – Added Status

Additional Axis Status bits (AS, TAS, TASF):

- Bit #29 is set if the input state or position relationship specified in the GOWHEN command is already true when the GO, GOL, FGADV, FSHFC, or FSHFD command is issued.
- Bit #31 is set while a compiled GOBUF profile is executing.

Additional Extended Axis Status bits (ASX, TASX, TASXF):

- Bit #7 reports the state of the Stall Input (pin #4 on the **DRIVE** connector), regardless of the DSTALL setting.

Additional Following Axis Status bits (FS, TFS, TFSF):

- Bit #26 now indicates that either FMAXA or FMAXV is limiting the current Following motion.

Controller Status Commands

OS 5.1.0 introduces a new status register for information that is not specific to an axis or a task. TCS provides a binary status report, TSCF provides a full-text report, and SC is an operand you can use to assign or compare the status of one or more binary status bits.

TSC Transfer Controller Status

Type	`Transfer`	**Product**	**Rev**
Syntax	`<!>TSC<.i>`	6K	5.1
Units	`i = controller status bit number`		
Range	`1-32`		
Default	`n/a`		

Response	`TSC:   *TSC0000_0000_0000_0000_0000_0000_0000_0000`
	`TSC.1: *0`		

See Also	`[ SC ], SCANP, SYNCH, TSCF`		

The TSC command provides a binary report of the current "Controller Status" register. If you would like a more descriptive text-based report, use the TSCF command.

```
*TSCbbbb_bbbb_bbbb_bbbb_bbbb_bbbb_bbbb_bbbb
```

Bit 1 Bit 32

Bit #	**Function** (1 = Yes; Ø = No)
1	RESERVED
2	RESERVED
3	A PLC program is being executed in Scan Mode (SCANP).
4	RESERVED
5	Synch mode is enabled (SYNCH1). ((applicable only to the Synch Option of the 6K, not the standard 6K))
6	Fieldbus is operational. ((applicable only to the Fieldbus Option of the 6K, not the standard 6K))
7	RESERVED
8-32	RESERVED

TSCF Transfer Controller Status (full-text report)

Type	`Transfer`	**Product**	**Rev**
Syntax	`<!>TSCF`	6K	5.1
Units	`n/a`		
Range	`n/a`		
Default	`n/a`		
Response	`TSCF: (see example below)`		
See Also	`[ SC ], SCANP, SYNCH, TSC`		

The `TSCF` command provides a full-text report of the current “Controller Status” register. This is an alternative to the binary report (`TSC`).

Example `TSCF` response:

```
*TSCF
*Reserved            NO
*Reserved            NO
*SCANP Running       NO
*Reserved            NO
*
*Synch Mode Enabled  NO
*Field Bus Running   NO
```

[SC] Controller Status

Type	`Assignment or Comparison`	**Product**	**Rev**
Syntax	`See below`	6K	5.1
Units	`n/a`		
Range	`n/a`		
Default	`n/a`		
Response	`n/a`		
See Also	`SCANP, SYNCH, TSC, TSCF, VARB`		

Use the `SC` operator to assign the controller status bits to a binary variable (`VARB`) or to make a comparison against a binary or hexadecimal value. To make a comparison against a binary value, the letter b (b or B) must be placed in front of the value. The binary value itself must only contain ones, zeros, or Xs (1, 0, X, x). To make a comparison against a hexadecimal value, the letter h (h or H) must be placed in front of the value. The hexadecimal value itself must only contain the letters A through F, or the numbers 0 through 9.

Syntax: `VARBn=SC` where “n” is the binary variable number, or `SC` can be used in an expression such as `IF(SC=b0100)`, or `IF(SC=h7F)`. If it is desired to assign only one bit of the controller status value to a binary variable, instead of all 32, the bit select (`.`) operator can be used. For example, `VARB1=SC.3` assigns controller status bit #12 to binary variable 1: `*VARB1=XX0X_XXXX_XXXX_XXXX_XXXX_XXXX_XXX_XXXX`.

The function of each controller status bit is shown below.

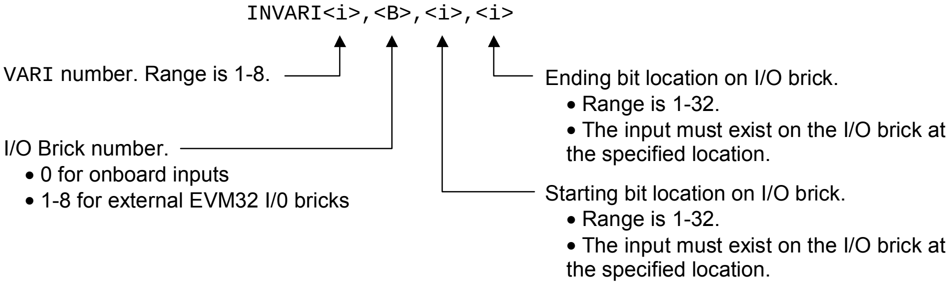
Bit #	**Function** (1 = Yes; Ø = No)
1	RESERVED
2	RESERVED
3	A PLC program is being executed in Scan Mode (`SCANP`).
4	RESERVED
5	Synch mode is enabled (`SYNCH1`). ((applicable only to the Synch Option of the 6K, not the standard 6K))
6	Fieldbus is operational. ((applicable only to the Fieldbus Option of the 6K, not the standard 6K))
7	RESERVED
8-32	RESERVED

Mapping Inputs to Integer Variables

INVARI Map Inputs to Integer Variable

		Product	Rev
Type	`Variable`	6K	5.1
Syntax	`<!>INVARI<i>,<B>,<i>,<i>`		
Units	`(see below)`		
Range	`(see below)`		
Default	`n/a`		
Response	`INVARI: *INVARI1,1,1,12`		
See Also	`INDEB, INEN, INLVL, VARI`		

The `INVARI` command allows a selected group of contiguous inputs to be interpreted as a binary number and continuously assigned to the selected integer variable (`VARI`). The inputs are specified by I/O brick number, starting bit number, and ending bit number. All inputs to be mapped to a `VARI` must be contiguous and on the same I/O brick. These inputs are read and masked internally, then shifted such that the starting bit is the low-order bit of the resulting binary value. A change in the starting bit input will always result in a change of ±1 in the resulting `VARI`, even if the starting bit number is not 1.



```
INVARI<i>,<B>,<i>,<i>
```

`VARI` number. Range is 1-8.

I/O Brick number.
- 0 for onboard inputs
- 1-8 for external EVM32 I/0 bricks

Ending bit location on I/O brick.
- Range is 1-32.
- The input must exist on the I/O brick at the specified location.

Starting bit location on I/O brick.
- Range is 1-32.
- The input must exist on the I/O brick at the specified location.

The 6K inputs are read every 2 milliseconds, modified by `INLVL` and `INEN`, and debounced with the time specified in `INDEB`. The inputs specified by `INVARI` are monitored after they are modified by `INLVL` and `INEN`, but before the debounce. Thus, the `VARI` variable specified by the `INVARI` command will be updated every 2 milliseconds.

The `VARI` variables are not updated with inputs unless the `INVARI` is issued with valid values for starting and ending bits. If the specified input bit does not exist onboard or on the I/O brick, an error message ("`*INVALID DATA`") will result.

Specifying bit 0 for both starting and ending bits disables the `INVARI` mapping (`INVARI<i>,<b>,0,0`).

Example:

A following application requires 2 axes to follow a master source that has position information presented as a 12-bit binary number on digital outputs. Each axis must have its own master source. An external I/O brick with three 8-bit digital input SIMs is used to read the 24 bits of the two sources. The first source is wired to bits 1-12, and the second source to bit 13-24.

```
DEF SETUP
INVARI4,1,1,12       ; VARI4 reflects bits 1-12 of brick 1
INVARI5,1,13,24      ; VARI5 reflects bits 13-24 of brick 1
FOLMAS48,58          ; Axis 1 follows VARI4, axis 2 follows VARI5
FOLEN11              ; Both axes in Following mode.
END
```

Test: (Assert bits 2, 3, 15, and 16)

```
>VARI4
*VARI4=+6
>VARI5
*VARI5=+12
>TPMAS
*TPMAS+6,+12
```

String Variable (VARS) Changes

OS 5.1.0 provides the option of copying one `VARS` to another `VARS`. `VARSn=VARSm` may be used, as well as variable substitution for "n" or "m". Also, the length of string variables has been increased from 20 characters to 50 characters.

More Following Master Options

OS 5.1.0 adds two more options that can be used as a Following Master in the `FOLMAS` syntax:

- Option 7 (syntax is `FOLMASn7`). This option allows an axis to follow shared output variable #1 (`VARSHO1`) of the Synch Bus unit specified with "n". The range for "n" is 1-8 (the maximum number of Synch Bus units). For example, `FOLMAS,,,37` assigns axis #4 to follow the `VARSHO1` variable of Synch Bus unit #3.
  NOTE: This option is applicable only to the Synchronization Bus option of the 6K controller (for details, refer to the *6K Synchronization Bus Guide*).
- Option 8 (syntax is `FOLMASn8`). This option allows an axis to follow the integer variable (`VARI`) specified with "n"; that is, `VARIn`. The range for "n" is 1-8 (`VARI1` – `VARI8`). For example, `FOLMAS,48` assigns axis #2 to follow `VARI4`.

  This option is particularly useful in conjunction with the `INVARI` (Map Inputs to a `VARI` Variable) feature – see page 45. `INVARI` continuously updates a specified `VARI` variable with the value of a specified group of digital inputs, allowing an axis to follow a binary input pattern. Another useful way to update the value of the `VARI` variable is to calculate its value in a `PLCP` program (launched with the `SCANP` command).

  The `INVARI` and `SCANP` options for updating `VARI` are good choices, because both are performed every system update, thus facilitating smooth Following motion. It is also possible to use an extra task (multi-tasking) to calculate `VARI` values, but the resulting updates will not be as fast (not perfectly periodic); consequently, Follow motion will be less smooth.

Increased Memory Capacity

The 6K's memory has been increased from 150,000 bytes to 300,000 bytes. The default allocation is now `MEMORY150000,150000` (150,000 bytes allocated to programs and 150,000 bytes allocated to compiled programs and profiles).

EVM32-Related Enhancements

OS 5.1.0 supports the new EVM32 SIMs with the following new commands:

`ANO`	Set Analog Output Value	page 58
`[ANO]`	Analog Output Value (assignment/comparison operand)	page 58
`TANO`	Display Status of Analog Outputs	page 59
`KIOEN`	Kill on EVM32 I/O Disconnect (enable/disable)	page 59

The `TIO` report was updated to include the analog output values.

NOTE: The SIM8-AN-OUT and SIM8-OUT-RLY10 SIMs cannot be used with operating system versions earlier than OS 5.1.0.

Installation and programming information for the EVM32 enhancements begins on page 47.

EVM32 Enhancements

What's new?

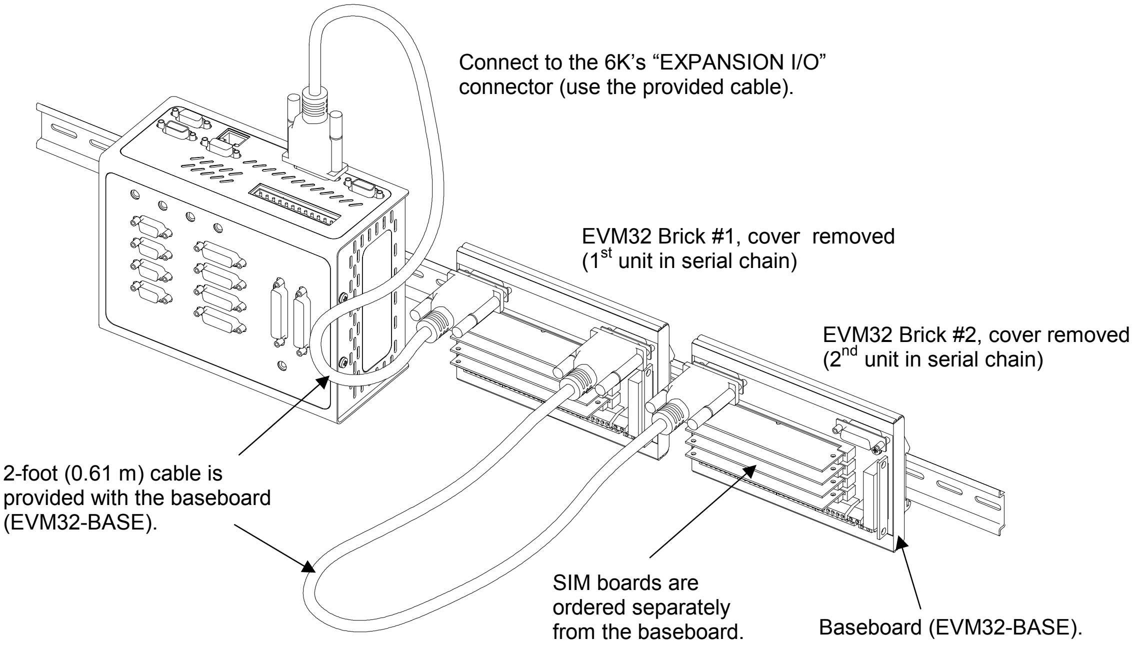
- Vented sheetmetal cover
- Adhesive labels to affix to the new sheetmetal cover (to identify the SIM configuration)
- SIM8-IN-EVM32: A new SIM that provides 8 digital inputs, with LEDs. (replaces SIM8-IN)
- SIM8-OUT-NPN: A new SIM that provides 8 digital sinking outputs, short-circuit protected, with LEDs
- SIM8-OUT-PNP: A new SIM that provides 8 digital sourcing outputs, short-circuit protected, with LEDs
- SIM8-OUT-RLY10: A new SIM that provides 8 reed relay outputs (10 watt max.), with LEDs
- SIM8-AN-OUT: A new SIM that provides 8 analog (±10V) outputs
- **NOTE**: The SIM8-AN-OUT and SIM8-OUT-RLY10 SIMs cannot be used with operating system versions earlier than OS 5.1.0.
- New commands (see descriptions starting on page 58):

`ANO`	Set Analog Output Value
`[ANO]`	Analog Output Value (assignment/comparison operand)
`TANO`	Display the Analog Output Value
`KIOEN`	Kill on EVM32 I/O Disconnect (enable/disable)

- The `TIO` report has been updated to include analog output values and reed relay output values.

Hardware Installation

This material supersedes the information on pages 43-49 of the *6K Series Hardware Installation Guide* (88-017547-01A).


Connect to the 6K's "EXPANSION I/O" connector (use the provided cable).
EVM32 Brick #1, cover removed (1st unit in serial chain)
EVM32 Brick #2, cover removed (2nd unit in serial chain)
2-foot (0.61 m) cable is provided with the baseboard (EVM32-BASE).
SIM boards are ordered separately from the baseboard.
Baseboard (EVM32-BASE).

EVM32 Description

The EVM32 is a family of I/O modules (or “bricks”) that is sold as accessories to the 6K Controllers. The purpose of the EVM32 is to provide more I/O than the 6K offers onboard. Up to eight DIN-rail mountable EVM32 bricks can be connected in a serial chain to the 6K. Each EVM32 brick can hold from one to four I/O SIM boards in any combination (each SIM board provides eight I/O points, for a total of 32 I/O points per I/O brick).

Order an EVM32 brick and up to four I/O SIM boards per brick (see table below).

Product (p/n)	Description
EVM32-BASE	EVM32 baseboard, extrusion with built-in DIN rail mount (includes 2-foot cable).
SIM8-IN-EVM32	SIM board with 8 digital inputs, with LEDs. Color code: RED.
SIM8-OUT-NPN	SIM board with 8 digital sinking outputs, with LEDs. Color code: BLUE.
SIM8-OUT-PNP	SIM board with 8 digital sourcing outputs, with LEDs. Color code: BLUE.
SIM8-OUT-RLY10	SIM board with 8 reed relay outputs, with LEDs. Color code: BLUE.
SIM8-AN-IN	SIM board with 8 analog inputs (12-bit, ±10V inputs). Color code: GREEN.
SIM8-AN-OUT	SIM board with 8 analog outputs (10-bit, ±10V inputs). Color code: BLACK.
71-016949-02	2-foot cable for connection to 6K or between I/O bricks (included with EVM32-BASE).
71-016949-100	100-foot cable for connection to 6K or between I/O bricks.

EVM32 Specifications

Parameter	Specification
Power (DC input)	
V+	User-supplied voltage that drives output circuitry.
V+ range	12-24VDC. (If using SIM8-AN-OUT, you must use a 24VDC supply.)
V+ current	1.8A @ 12VDC or 0.9A @ 24VDC; plus the sum of the load current on the PNP outputs.
Environmental	
Operating temperature	32 to 122°F (0 to 50°C)
Storage temperature	-22 to 185°F (-30 to 85°C)
Humidity	0 to 95% non-condensing
Dimensions	(see dimension drawing on page 50).
Digital Inputs (SIM8-IN-EVM32)	
Switching levels	Low ≤ 1/3 V+ voltage; High ≥ 2/3 V+ voltage.
Voltage range	Voltage range = 0-24VDC. Voltage of input signals should not exceed voltage level of V+. (Input circuitry of EVM32 has diodes to protect against voltages that exceed V+, but performance may degrade.)
Sinking/Sourcing	Sinking: Connect jumper for selected SIM board to position 1. Sourcing: Connect jumper for selected SIM board to position 3 (factory default).
Impedance	6 KΩ, minimum. Requires input current (sinking or sourcing) of 0.111mA per volt of user-supplied voltage to V+ (e.g., 2.67mA if V+ = 24V).
Active level	Set by the 6K controller (`INLVL` command setting) — default is active low, but can be set to active high.
Input frequency	50 kHz (the maximum frequency is limited practically to 500 Hz by the 2 ms update rate of the 6K controller).
Status	Check with the `TIO` command. LED illuminates when at least [2/3 * V+] volts is present on the input: • If sinking (jumper in position 1), the default LED state is off. The LED illuminates when the voltage at the input is at least [2/3 * V+] volts. • If sourcing (jumper in position 3), the default LED state is on. The LED goes off when the voltage at the input is below [1/3 * V+] volts.
Reed Relay Outputs (SIM8-OUT-RLY10)	
Current rating	Maximum of 10 Watts. • Switching voltage to 200VDC or 200VAC peak resistive • Switching current to 0.5A
Operate time, including bounce – typical	0.4 milliseconds
Release time – typical	0.1 milliseconds
Capacitance – typical	0.7 pF
Status	Check with the `TIO` command. LED is on when the relay contact is closed.

Digital Outputs (SIM8-OUT-NPN and SIM8-OUT-PNP)

Sinking/Sourcing .... SIM8-OUT-NPN provides 8 sinking outputs.
SIM8-OUT-PNP provides 8 sourcing outputs.

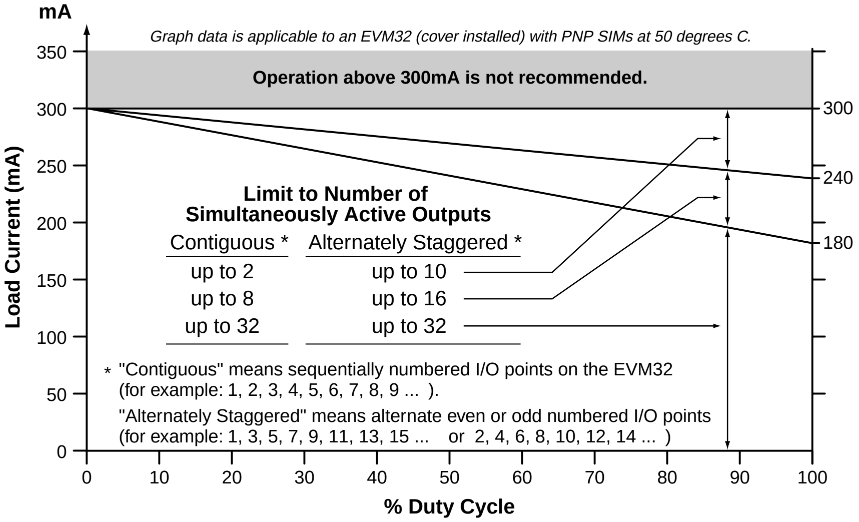
Voltage (sinking — SIM8-OUT-NPN) .... Output voltage level is less than or equal to 0.4VDC when sinking up to 50mA. (≤ 0.4 VDC for 50 mA).

Output voltage level is less than or equal to 2.5VDC when sinking up to 300mA. (≤ 2.5 VDC for 300 mA).

Voltage (sourcing — SIM8-OUT-PNP) ...... Output voltage level may be up to 2 volts less than the user-supplied voltage V+ when sourcing up to 50mA.

Output voltage level may be up to 2.5 volts less than the user-supplied voltage V+ when sourcing up to 300mA.

Current .... 300mA maximum per output; continuous duty at 50°C ambient temperature.
NOTE: For PNP outputs, the actual current is subject to derating, based on load current, duty cycle, and number of simultaneously active outputs (see graph below). Improved performance may be achieved by lowering the ambient temperature and/or staggering the physical order of the outputs that are simultaneously active.



Limit to Number of Simultaneously Active Outputs

Contiguous *	Alternately Staggered *
up to 2	up to 10
up to 8	up to 16
up to 32	up to 32

* "Contiguous" means sequentially numbered I/O points on the EVM32 (for example: 1, 2, 3, 4, 5, 6, 7, 8, 9 ...).
"Alternately Staggered" means alternate even or odd numbered I/O points (for example: 1, 3, 5, 7, 9, 11, 13, 15 ... or 2, 4, 6, 8, 10, 12, 14 ...)

Active level .... Set with the `OUTLVL` command. On power-up or reconnect, SIM8-OUT-NPN is set to active low (`OUTLVL0`), and SIM8-OUT-PNP is set to active high (`OUTLVL1`).

Thermal shutdown .... Thermal shutdown protects the drive devices from excessive heat. The NPN SIM has 2 drive devices (4 output channels per device); the PNP SIM has 4 drive devices (2 output channels per device). When a drive device reaches 165°C, it will shut down (PNP: shut down two output channels; NPN: shut down four channels). The device driver will again become active when its temperature cools to 150°C.

Short-circuit protection .... Digital outputs are short-circuit protected. Short-circuit protection only shuts down the affected output channel. To recover, remove the fault and cycle power to the EVM32.

Status........ Check with the `TIO` command.
With default `OUTLVL`, LED is on when output is active (set to 1 with `OUT` command).

Analog Inputs (SIM8-AN-IN)

Input voltage range .... 12-bit A/D converter, ±10VDC; unipolar/bipolar range selectable `ANIRNG` command.
Unipolar: 0V to 10VDC, or 0V to 5V;
Bipolar: -10 to +10V (factory default), or -5V to +5V.

Input current (worst case load) .... Unipolar: 720µA @ 0V to 10VDC range; 360µA @ 0V to 5V range.
Bipolar: -1200µA @ -10V and 720µA @ +10V; -600µA @ -5V and 360µA @ +5V.

Input dynamic resistance .... Unipolar: 21KΩ; Bipolar: 16KΩ

Fault tolerance .... ±16.5V

Sample rate .... Each input requires 2ms (e.g., 4 ms for 2 inputs, 16ms for 8 inputs); therefore, to maximize performance, you should disable unused inputs with `ANIEN` command.

Status........ Check with the `TIO` command.

Analog Outputs (SIM8-AN-OUT)

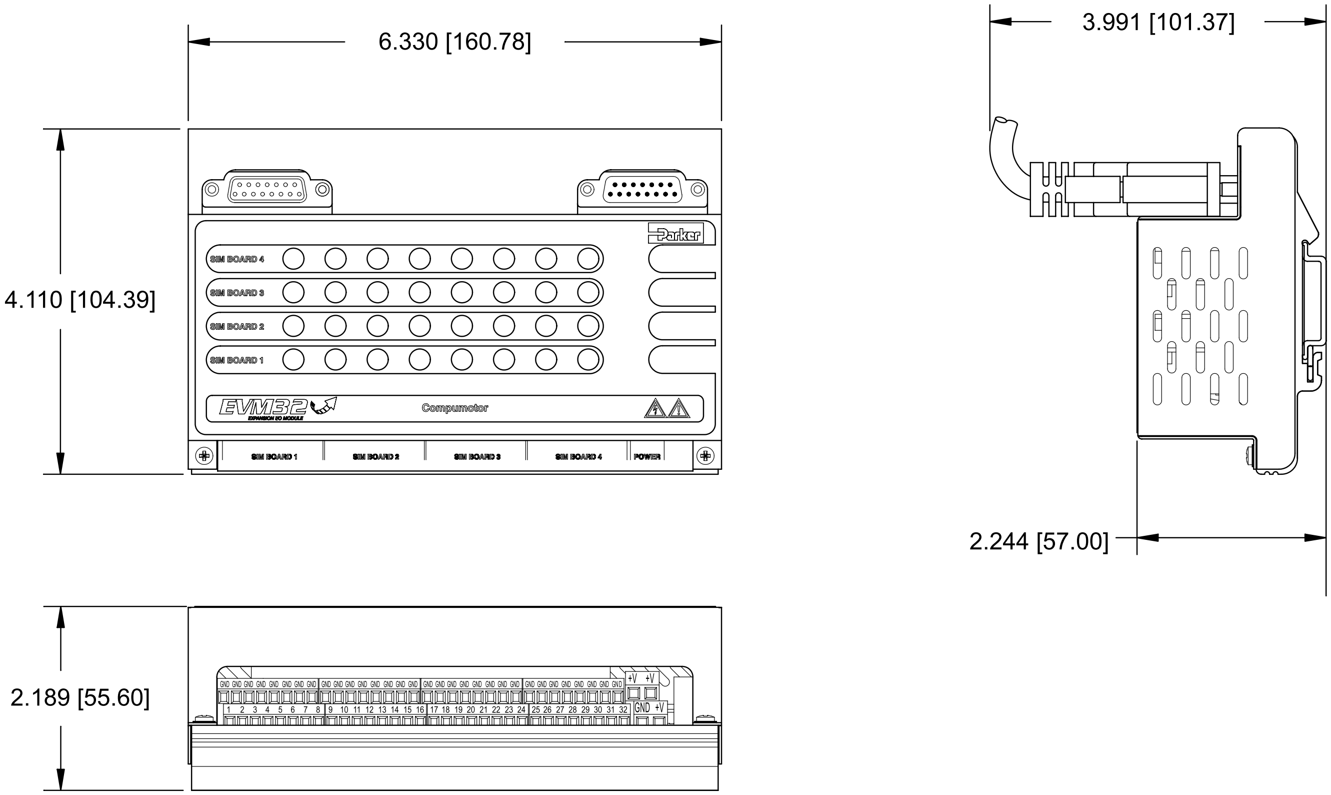
Output voltage range .... 10-bit DAC, ±10VDC, 8 channels total.

Load .... 2KΩ resistive at 5mA maximum.

Linearity error .... 0.66% (typical), 1.3% maximum, over the ±10VDC range.

Status........ Check with the `TIO` command or the `TANO` command.

EVM32 Dimensions



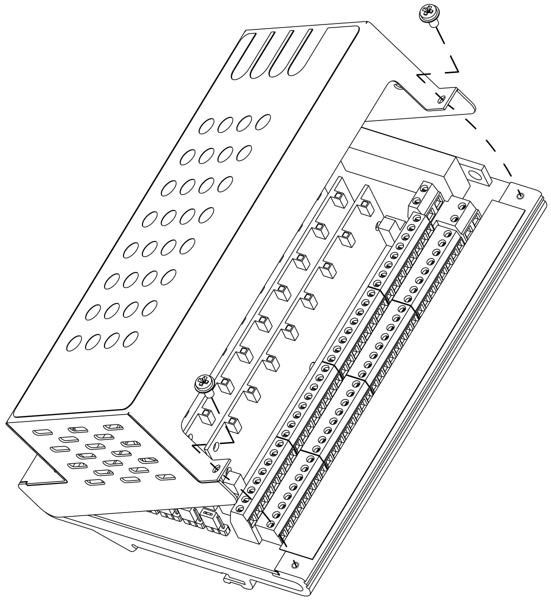
Installing the SIM Boards

Step 1: Remove the Cover

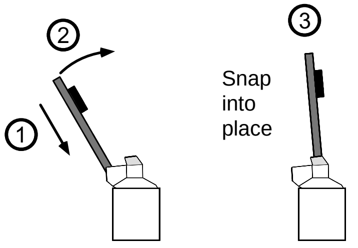
CAUTION:

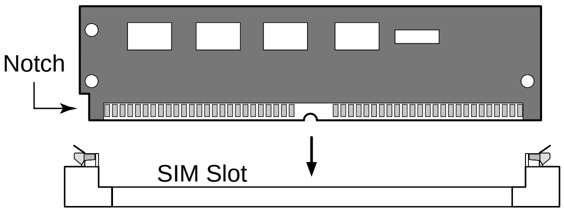
EVM32 SIM boards are static sensitive. Observe proper ESD handling precautions.

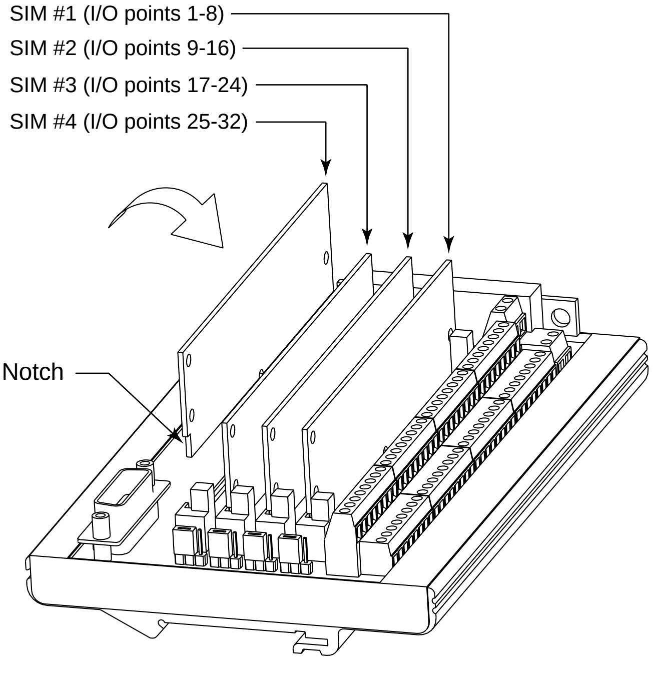
REMOVE POWER to the EVM32 baseboard before installing or removing SIM boards.



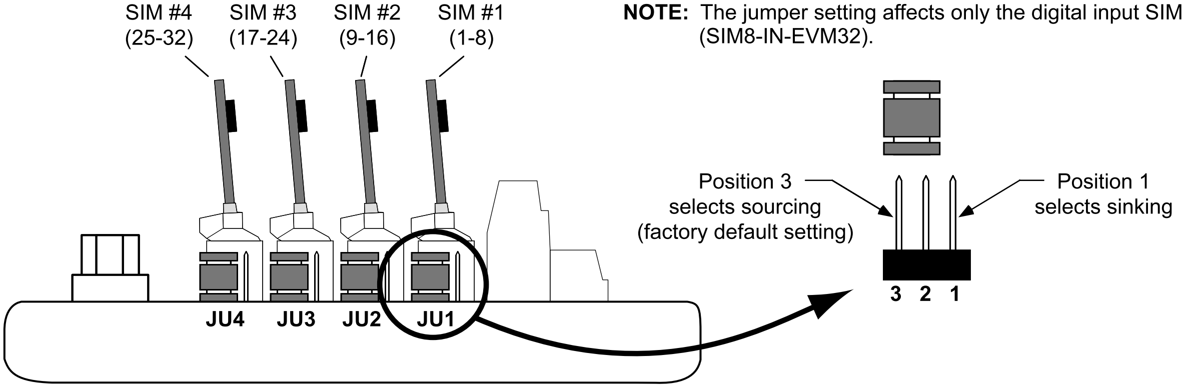
Step 2: Install the SIM boards







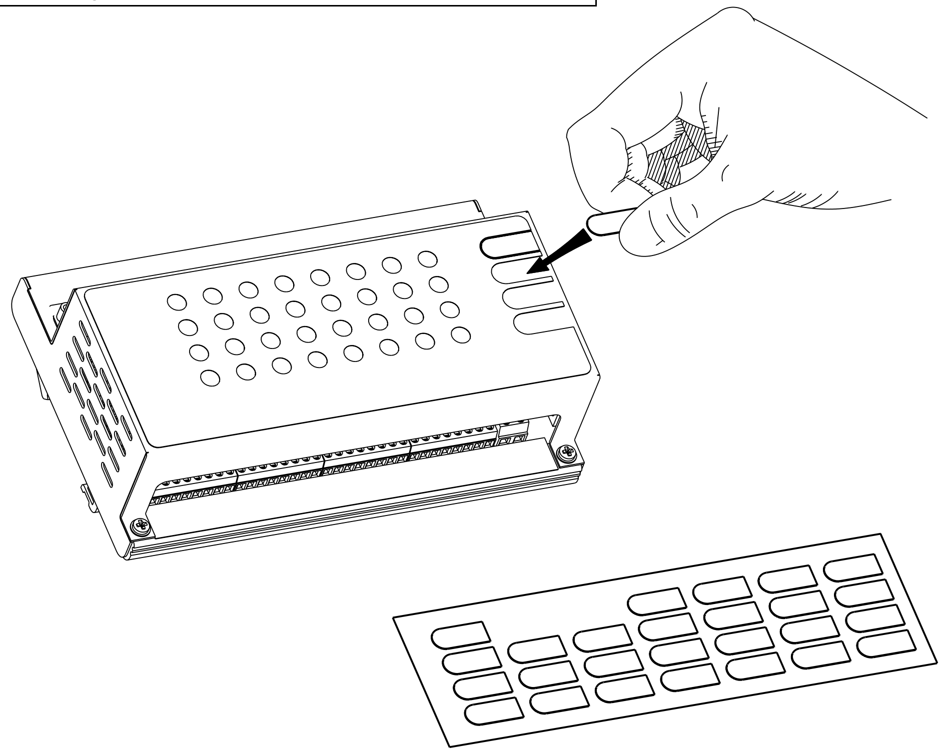
Step 3: (SIM8-IN-EVM32 only) Set the jumpers to select sinking or sourcing



NOTE: The jumper setting affects only the digital input SIM (SIM8-IN-EVM32).

Step 4: Replace the cover and label the SIM locations

SIM Board	Color	Label
SIM8-IN-EVM32 (digital inputs)	Red	8 IN
SIM8-OUT-NPN (digital outputs, sinking)	Blue	8 OUT (NPN)
SIM8-OUT-PNP (digital outputs, sourcing)	Blue	8 OUT (PNP)
SIM8-OUT-RLY10 (reed relay outputs)	Blue	RELAY
SIM8-AN-IN (analog inputs)	Green	ANALOG IN
SIM8-AN-OUT (analog outputs)	Black	ANALOG OUT



Electrical Connections

⚠ **CAUTION** ⚠

Remove power to the 6K controller and the EMV32 baseboard before:

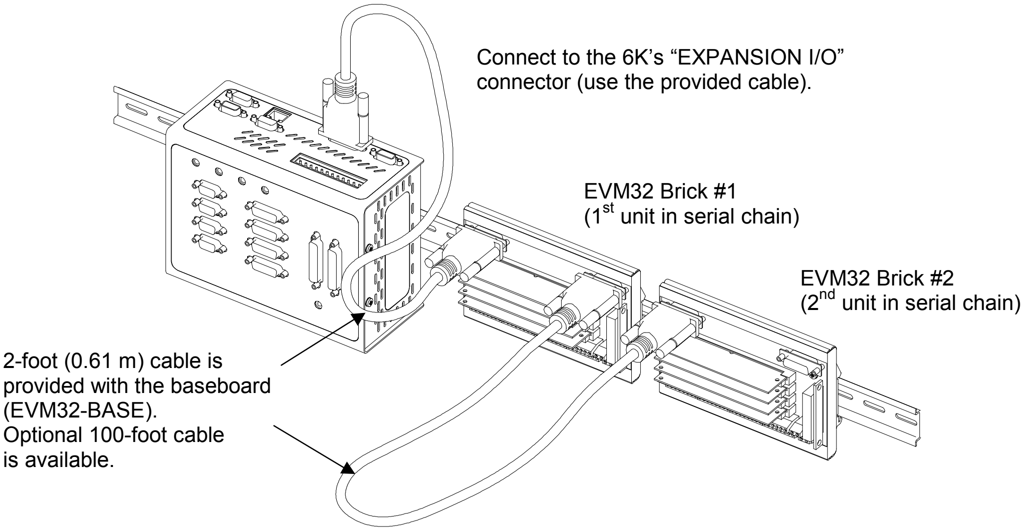
- Installing or removing SIM boards on the EVM32 baseboard
- Connecting or disconnecting the EVM32 baseboard to the 6K controller or to other EVM32 units
- Connecting inputs and outputs to the EVM32

Connection to the 6K controller and between EVM32 I/O bricks

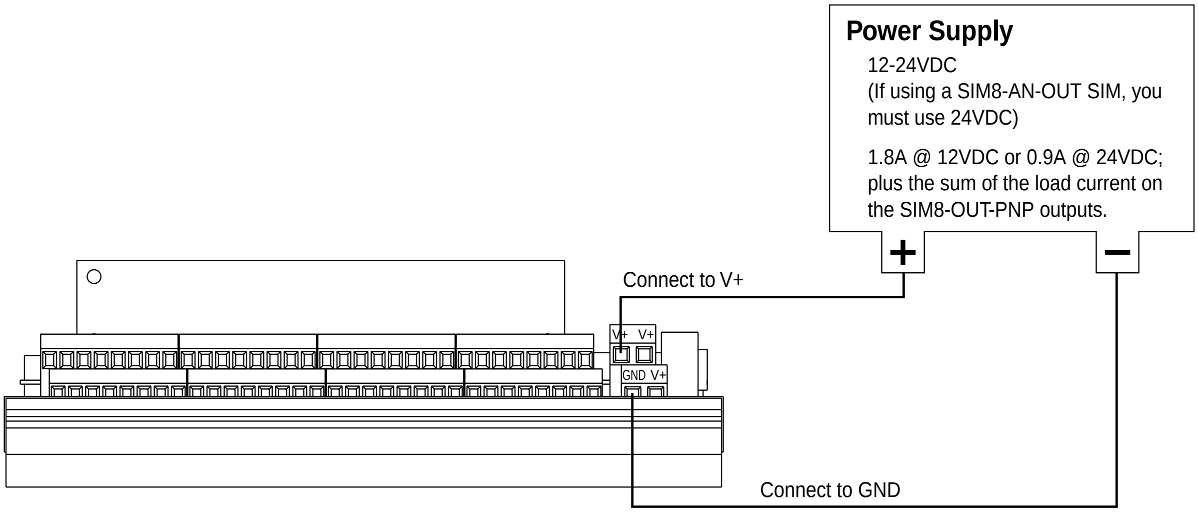
If the EVM32 I/O brick is disconnected (or if it loses power), the 6K will perform a kill (all motion and program execution on all tasks) and set error bit #18 (reported with the `TER`, `TERF` and `ER` commands). The 6K will remember the brick configuration (volatile memory) in effect at the time the disconnection occurred. When you reconnect the I/O brick, the controller checks to see if anything changed (SIM by SIM) from the state when it was disconnected. If an existing SIM slot is changed (different SIM, vacant SIM slot, or jumper setting), the controller will set the digital input SIMs and digital output SIMs to factory default `INEN` and `OUTLVL` settings, respectively. If a new SIM is installed where there was none before, the new SIM is auto-configured to factory defaults.

When the 6K powers up and detects a digital output SIM on a EVM32, it will set the active level (`OUTLVL` command) according to the type of SIM. `OUTLVL0`, active low, is selected for NPN SIMs; `OUTLVL1`, active high, is selected for PNP SIMs.

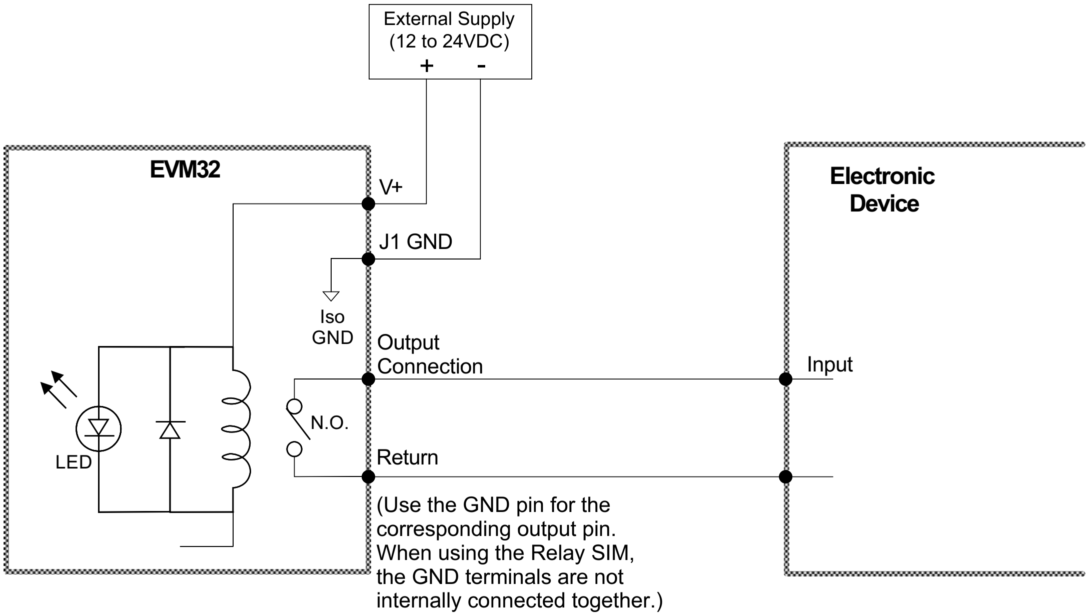
To check the status of one or more EVM32 I/O bricks, use the `TIO` command.


Connect to the 6K's "EXPANSION I/O" connector (use the provided cable).
EVM32 Brick #1 (1st unit in serial chain)
EVM32 Brick #2 (2nd unit in serial chain)
2-foot (0.61 m) cable is provided with the baseboard (EVM32-BASE). Optional 100-foot cable is available.

24VDC power input


Power Supply
12-24VDC
(If using a SIM8-AN-OUT SIM, you must use 24VDC)
1.8A @ 12VDC or 0.9A @ 24VDC; plus the sum of the load current on the SIM8-OUT-PNP outputs.
+
−
Connect to V+
V+ V+
GND V+
Connect to GND

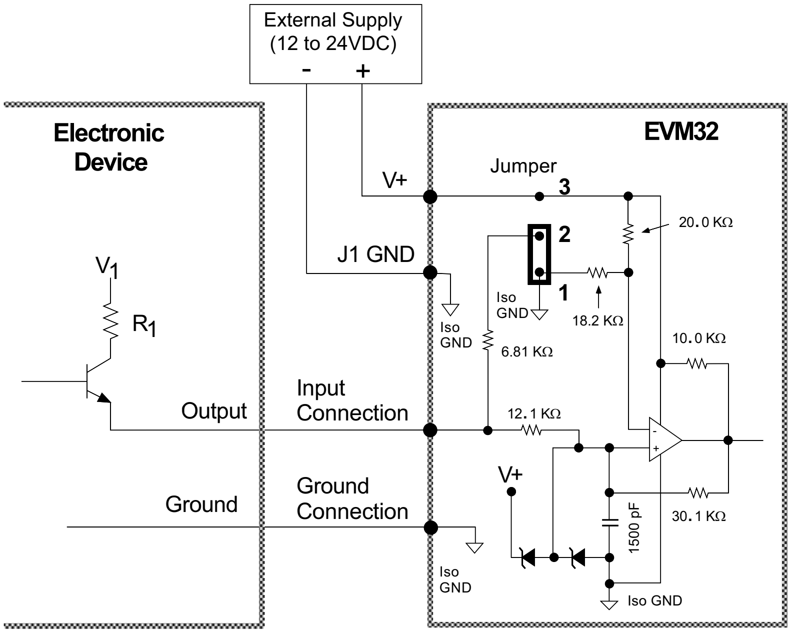
Reed Relay Outputs (SIM8-OUT-RLY10)


External Supply
(12 to 24VDC)
+
-
EVM32
V+
J1 GND
Iso
GND
Output
Connection
N.O.
LED
Return
(Use the GND pin for the corresponding output pin. When using the Relay SIM, the GND terminals are not internally connected together.)
Electronic
Device
Input

Digital Inputs (SIM8-IN-EVM32)

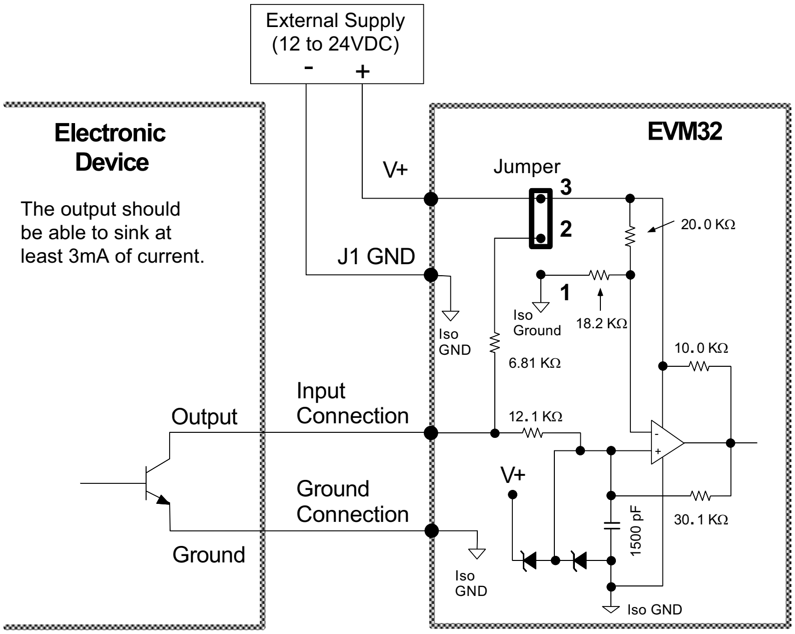
Sinking

(connecting to a sourcing output)



Sourcing

(connecting to a sinking output)



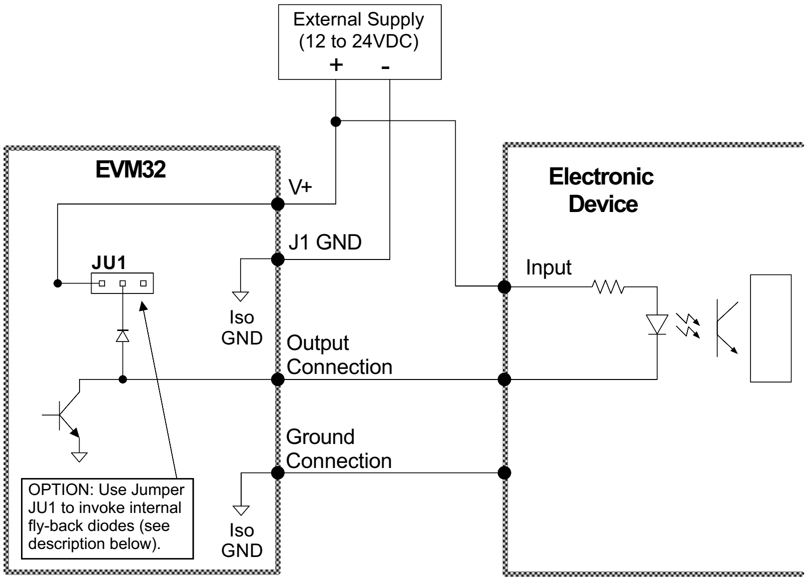
Interrelationships

Active Level *	Jumper Selection *	Switch	Voltage at Input	LED	IN/TIN/TIO Report
INLVL0 (active low)	Position 3 (sourcing)	Open	≥ 2/3 of V+	On	0
INLVL0 (active low)	Position 3 (sourcing)	Closed	< 1/3 of V+	Off	1
INLVL1 (active high)	Position 3 (sourcing)	Open	≥ 2/3 of V+	On	1
INLVL1 (active high)	Position 3 (sourcing)	Closed	< 1/3 of V+	Off	0
INLVL0 (active low)	Position 1 (sinking)	Open	< 1/3 of V+	Off	1
INLVL0 (active low)	Position 1 (sinking)	Closed	≥ 2/3 of V+	On	0
INLVL1 (active high)	Position 1 (sinking)	Open	< 1/3 of V+	Off	0
INLVL1 (active high)	Position 1 (sinking)	Closed	≥ 2/3 of V+	On	1

* Factory default: INLVL0 (active low) and jumper in position 3 (sourcing). Jumper location is illustrated on page 52.

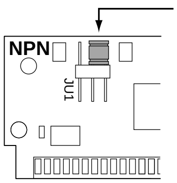
Digital Outputs (SIM8-OUT-NPN and SIM8-OUT-PNP)

SIM-OUT-NPN
(sinking outputs)

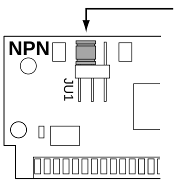


OPTION: Use Jumper JU1 to invoke internal fly-back diodes (see description below).

Fly-Back Diodes: The SIM card is sent from the factory with jumper JU1 in the “no connect” position (fly-back diodes not used). If you move the jumper to the left-hand position, eight fly-back diodes are invoked, one for each of the 8 output channels. **CAUTION**: If the power supply voltage for the remote device (to which the outputs are connected) is greater than the power supply voltage for the EVM32, <u>do not</u> use the fly-back diodes.



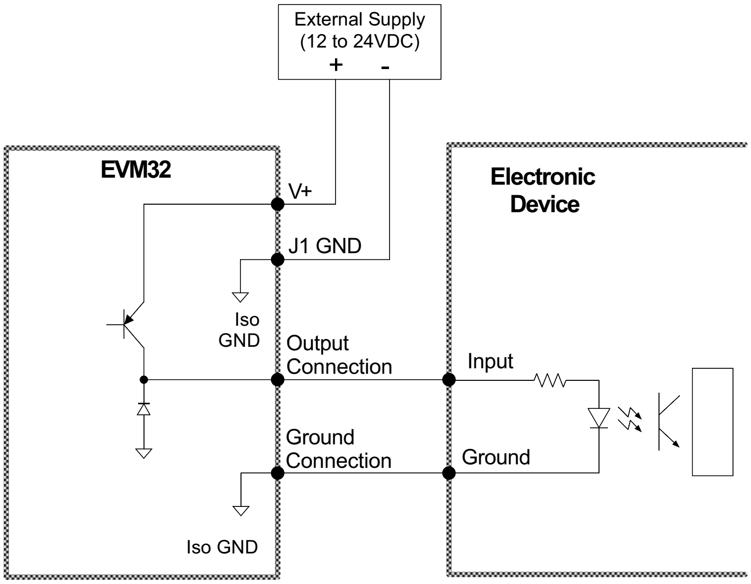
Factory default jumper position is "no connect" (no internal diodes used).



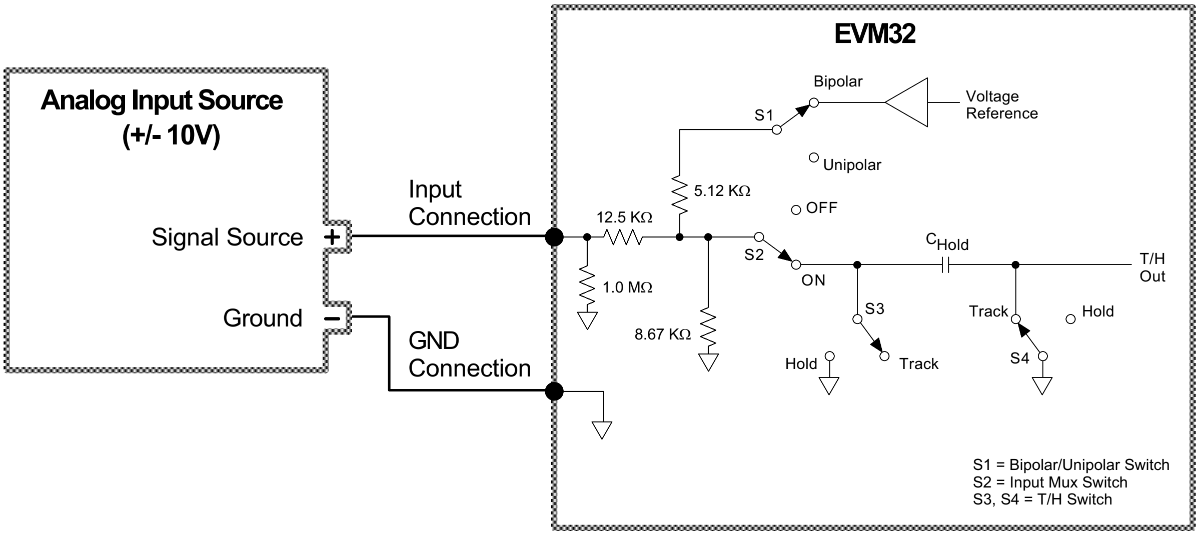
Move the jumper to the left-hand position to use internal fly-back diodes.

External Diodes: Use an external diode when driving inductive loads (you can do this only if you have not invoked the fly-back diodes with jumper JU1). Connect the diode in parallel to the inductive load.

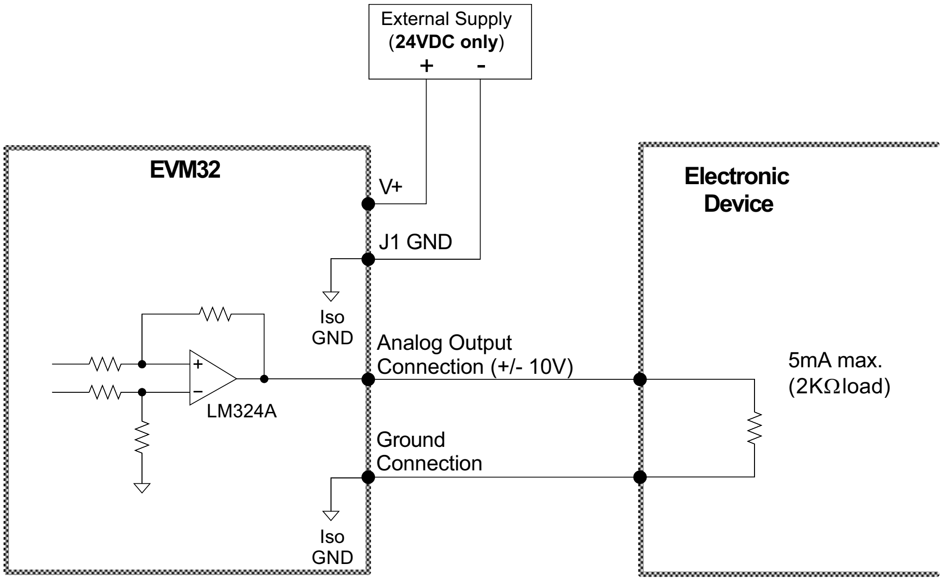
SIM8-OUT-PNP
(sourcing outputs)



Analog Inputs (SIM8-AN-IN)



Analog Outputs (SIM8-AN-OUT)



ANO Set Analog Output Value

Type	Output	**Product**	**Rev**
Syntax	<!><B>ANO<.i>=<r>	6K	5.1

Units	B = I/O brick number
	i = input output location on brick "B"
	r = volts		

Range	B = 1-8 (depending on I/O brick configuration)
	i = 1-32 (depending on I/O brick configuration)
	r = -10.00 to +10.00 (or 0 to 4095 in PLC mode)		

Default	0.0		
Response	n/a		
See Also	[ANO], TANO, TIO		

Use the Analog Output (ANO) command to assign a voltage of an analog output associated with an analog output SIM.

Performance: Each analog output channel is updated once every 16ms (2ms per analog input channel). Each DAC has 10-bit granularity, giving approximately 20 mV/bit over the 20 volt range.

PLC Mode: When commanding an analog output using PLC mode, the raw DAC value must be used. To calculate what value to program, use the following formula or table:

ANO Value = (Vout + 10) x 4095 / 20

Vout	DAC Value
-10V	0
-5V	1024
0V	2048
+5V	3071
+10V	4095

Example:

```
1ANO.9=8.5 ; Command 8.5 volts on analog output 9, I/O brick 1 (analog
           ; output SIM must be present in slot 2 of I/O brick 1)
```

[ANO] Analog Output Value

Type	Outputs; Assignment or Comparison	**Product**	**Rev**
Syntax	See below	6K	5.1
Units	Voltage		
Range	-10.00VDC to +10.00VDC		
Default	n/a		
Response	n/a		
See Also	ANO, TANO, TIO		

Use the ANO operand to assign the voltage level present at one of the analog outputs to a variable, or to make a comparison against another value.

The ANO value is derived from the voltage applied to the corresponding analog output and ground. The analog value is determined from a 10-bit digital-to-analog converter and the value set with the ANO command. The range of the ANO operand is -10.00VDC to +10.00VDC.

Syntax: VARn=<B>ANO.i where "n" is the variable number, "<B>" is the number of the I/O brick, and "i" is I/O brick address where the analog output resides; or ANO can be used in an expression such as IF(1ANO.2=2.3). If no brick identifier (<B>) is provided, it defaults to 1.

PLC Mode: When assigning or comparing an analog output value in PLC mode, the raw DAC value must be used. To calculate what this value will be, use the following formula or table:

ANO Value = (Vout + 10) x 4095 / 20

Vout	DAC Value
-10V	0
-5V	1024
0V	2048
+5V	3071
+10V	4095

Example:

```
IF(1ANO.9=8.5)   ; If the commanded analog output is greater than 8.5
  WRITE "HIGH"   ; write warning to screen
  NIF
IF(1ANO.9<-8.5)  ; If the commanded analog output is less than -8.5
  WRITE "LOW"    ; write warning to screen
  NIF
```

TANO Transfer Analog Output Value

Type	`Transfer`	**Product**	**Rev**
Syntax	`<!><B>TANO.i`	6K	5.1

Units	`B = I/O brick number`
	`i = input output location on brick “B”`		

Range	`B = 1-8 (depending on I/O brick configuration)`
	`i = 1-32 (depending on I/O brick configuration)`		

Default	`n/a`		
Response	`1TANO.9: *1TANO.9=0.00`		
See Also	`[ANO], ANO, TIO`		

Use the Transfer Analog Output (`TANO`) command to report the value of an analog output channel.

Example:

```
>1TANO.9   ; Report the commanded analog output voltage at bit 9 of
           ; I/O brick 1.
*1TANO.9=0.00
```

KIOEN Kill on EVM32 I/O Brick Disconnect – Enable

Type	`Controller Configuration; Inputs; Outputs`	**Product**	**Rev**
Syntax	`<!>KIOEN<b>`	6K	5.1
Units	`n/a`		
Range	`b = 0 (disable) or 1 (enable)`		
Default	`1`		
Response	`KIOEN:   *KIOEN1`		
See Also	`ERROR, TER, TIO`		

The `KIOEN` command allows you to control the functionality of the 6K when an *EVM32 I/O failure* has been detected (cause could be cable disconnection, or loss of power on the EVM32 I/O brick). The options are as follows:

- `KIOEN1` (factory default): The 6K will perform a kill (all motion and program execution on all tasks) if an EVM32 I/O failure is detected.
- `KIOEN0`: The 6K will not perform a kill. **(SEE WARNING NOTE BELOW)**

Regardless of the state of the `KIOEN` command, you can enable error bit 18 (`ERROR.18-1`) and use an error program to respond to an EVM32 I/O failure. Note that when operating in the

KIOEN0 mode, the branch to the error program is a GOSUB (instead of a GOTO branch under the factory default KIOEN1 mode).

WARNING

If you use the KIOEN0 mode (no kill if EVM32 I/O failure), bear in mind that when an EVM32 I/O failure occurs, the state of any external conditions becomes unknown to the 6K controller. For example, if an input on brick 2 is defined as a hardware end-of-travel limit, and if brick 2 loses power or disconnected from the 6K, the 6K will never see a state change on the limit.

The KIOEN0 mode is designed for use during system setup (for example, if you will be connecting and disconnecting I/O bricks in the process of wiring and programming your system).

Therefore, to help prevent damage to equipment and serious injury to personnel, we recommend leaving KIOEN at its default state (KIOEN1) during normal operation of your motion control system.

Changes to TIO:

In addition to the new commands (ANO, [ANO], and TANO), the TIO report will include analog output information similar to the way it displays analog input data. An example follows where brick 1 has an analog output SIM in slot 3 (voltage commanded on all outputs is +0.01V):

```
*BRICK 1:  SIM Type         Status       Function
*     1-8: NO SIM PRESENT
*    9-16: NO SIM PRESENT
*   17-24: ANALOG OUTPUTS   +0.01, +0.01, +0.01, +0.01, +0.01, +0.01, +0.01, +0.01
*   25-32: NO SIM PRESENT
```

Synchronous Serial Bus Interface (SSBI)

As of OS revision 5.2.0, the “Synch” hardware option is available for all 6K controller models (6K2-SYNC, 6K4-SYNC, and 6K8-SYNC). The Synch option allows you to connect up to eight 6K controllers together on a synchronized serial bus interface. New commands associated with the Synch option are:

SYNCH	Serial Synchronization Bus Enable
[SYNCH]	Serial Synchronization Address (assignment operand)
VARSHI	Shared Input Variable
VARSHO	Shared Output Variable

For details on the Synch option, refer to the *6K Synchronization Bus Guide* (part number 88-018179-01). To order this option, contact your local automation technology center (ATC) or distributor.