Appendix D:
Communications Server
(COM6SRVR.EXE)

The Communications Server (COM6SRVR.EXE) is a 32-bit OLE automation server which facilitates communications between Gemini drives (as well as 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 the Visual Basic, Visual C++, and Delphi. The Motion Planner installation program installs COM6SRVR.EXE in the Motion Planner directory.

To begin serial (RS-232 or RS-485) communications, an application simply needs to request a connection to a Gemini drive through the Communications Server. (You need to specify the PC COM port on which to connect.) The Communications Server manages the actual connection to each Gemini drive, and can feed information from a particular drive to all client applications which require the information.

Although the Communications Server only makes one connection to each Gemini drive, 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 Gemini at the same time. They will both receive the same responses coming from the drive, instead of competing for the data. It is also possible for an application to request multiple connections to multiple Gemini units via the Communications Server.

The syntax for requesting a connection to the Communications Server varies depending on the programming environment being used. Page 196 provides 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 197.

COM6SRVR Application Programming Interface (API): Once the proper object variable has been created and a connection is established, there is a standard set of Gemini methods which the client application(s) can access (see page 198).

Visual Basic Connection Example

```
'create an object variable, initialize it to a serial interface
'with the Gemini and make a connection to PC COM1

Dim MyMachine As Object
Dim ConnectReturnValue As Integer
Set MyMachine = CreateObject("COM6SRVR.GEMINI")
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

```
1. Add the class com6srvr.tlb to your project. This file is located in
C:\Window\System under the default Motion Planner installation.

2. In the project header file, add the line #include "com6srvr.tlb"

3. In the main project source file, there should be a function BOOL
C project App::InitInstance(). Add the line AfxO1eInit().

4. To initialize the communication server object and establish a
connection with RS-232 in your program, include the following lines:
     IGemini MyMachine;
     MyMachine.CreateDispatch ("COM6SRVR.GEMINI");
     int ConnectReturnValue = MyMachine.Connect(2);  /*connect to COM2*/
```

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 a Gemini interface }
  CommServer := CreateOleObject('COM6SRVR.GEMINI);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  { Make a serial connection to the Gemini drive on COM2 }
  CommServer.Connect(2);
end;

end.
```

How to Disconnect

The 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 Communications Server to be shared among several clients. It also provides a more secure connection model by insulating the 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 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 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 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 Communications Server connection   }

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

Gemini Methods

Read ( ) The Read method retrieves command responses from the drive. There are no arguments for this method. The read method does not wait for incoming responses from the drive. It returns immediately with a string containing the drive's response at the time of the request. If no response is available, this method will return an empty string.

If you were to call the Read method twice in rapid succession, you could get the response in one call to Read, and the `ERROK` characters or `ERRBAD` characters in the next call to Read. If this poses a problem, there are three primary alternatives:

- Call the Read method until you receive the `ERROK` or `ERRBAD` characters.
- Use the `ERRLVL0` setting or the `ERRLVL2` setting, both of which eliminate the transmission of `ERROK` and `ERRBAD` characters.
- Wait a longer period of time before the subsequent Read, but be aware that this may degrade performance to an unacceptable level.

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

USE WITH CAUTION. This method allows you 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).

SendFile (*filename*) The SendFile method is used to download files to the drive. The **filename** argument represents the name of the program file (containing Gemini commands) to be downloaded. If the filename is an empty string, then you will be prompted for the filename. The method returns a positive value (long integer) if the operation is successful; otherwise, it returns an error code (see error code table below).

SendOS (*filename*) The SendOS method downloads the Gemini drive's operating system. The **filename** argument represents the name of operating system file. If filename is an empty string, then you will be prompted for the operating system file name. The method returns a TRUE value if the operation is successful; otherwise, a FALSE value is returned. (This method returns a Boolean value.)

Write (cmd) The Write method is used to send commands to the drive. **cmd** is a string of commands to be sent. Multiple commands can be sent, but each command should be separated with a valid command delimiter (colon, carriage return, line feed). The command string should be limited to 1024 characters or less. Excessively large command strings, may cause an overflow in the drive's command buffer. This method returns a positive value (long integer) corresponding to the number of bytes sent, or a negative error code (see error code table below).

Error Codes

Error Code	Description
-1	Bad Connection
-2	Connection was shutdown
-3	Connection attempt failed
-4	Maximum number of connections exceeded
-5	Connection not yet established
-6	Reserved
-7	Unable to locate specified file
-8	Unable to open specified file