



UDP Protocol Specification v1.2

René Görlich, Paul Kanevsky and Peter Simpson

March 2018

Gemini UDP Protocol Specification v1.2

Introduction

This document defines Gemini's UDP protocol specification and establishes the processes that clients must implement in order to use the protocol. The protocol has been developed by René Görlich, Paul Kanevsky and Peter Simpson for use with the Gemini 2 telescope control system; it is not and cannot be supported by the earlier Gemini 1 hardware and v4.1 firmware.

The UDP network protocol is described as are the reasons for its selection over TCP; some .NET example code is also provided in Appendix 2. For support please post on the Gemini II Yahoo group at: <http://tech.groups.yahoo.com/group/Gemini-II/messages>

What is UDP?¹

The **User Datagram Protocol (UDP)** is one of the core members of the [Internet Protocol Suite](#), the set of network protocols used for the [Internet](#). With UDP, computer applications can send messages, in this case referred to as [datagrams](#), to other hosts on an [Internet Protocol](#) (IP) network without requiring prior communications to set up special transmission channels or data paths. The protocol was designed by [David P. Reed](#) in 1980 and formally defined in [RFC 768](#).

UDP uses a simple transmission model without implicit [handshaking](#) dialogues for providing reliability, ordering, or data integrity. Thus, UDP provides an unreliable service and datagrams may arrive out of order, appear duplicated, or go missing without notice. UDP assumes that error checking and correction is either not necessary or performed in the application, avoiding the overhead of such processing at the network interface level. Time-sensitive applications often use UDP because dropping packets is preferable to waiting for delayed packets, which may not be an option in a real-time system.^[1] If error correction facilities are needed at the network interface level, an application may use the [Transmission Control Protocol](#) (TCP) or [Stream Control Transmission Protocol](#) (SCTP) which are designed for this purpose.

UDP's [stateless](#) nature is also useful for servers answering small queries from huge numbers of clients. Unlike [TCP](#), UDP supports [packet broadcast](#) (sending to all on local network) and [multicasting](#) (send to all subscribers).^[2]

Common network applications that use UDP include: the [Domain Name System](#) (DNS), [streaming media](#) applications such as [IPTV](#), [Voice over IP](#) (VoIP), [Trivial File Transfer Protocol](#) (TFTP), [IP tunneling](#) protocols and many [online games](#).

UDP applications use [datagram sockets](#) to establish host-to-host communications. An application binds a socket to its endpoint of data transmission, which is a combination of an [IP address](#) and a service port. A port is a software structure that is identified by the [port number](#), a 16 [bit](#) integer value, allowing for port numbers between 0 and 65535, that allows for multiplexing many

¹ Reproduced from [Wikipedia](#), please see document licensing section for further information.

applications on a host. Port 0 is reserved, but is a permissible source port value if the sending process does not expect messages in response.

Comparison of TCP and UDP

This comparison is based on four features of the two protocols

	TCP	UDP
Lightweight	Relative to UDP, a heavier weight protocol because it supports reliable and ordered communications in a very flexible and self adaptive manner.	Lightweight protocol because it is simple, connectionless and does not support reliable and ordered communication.
Responsive	Optimised for reliable bulk transfer operations where multiple packet sends can be acknowledged in one return packet. For "highly interactive, chatty" applications, can be less responsive than UDP.	Low intrinsic protocol overhead, suited to "chatty" applications where fast round trip response times are required.
Reliable	TCP manages message acknowledgment, retransmission and timeout. Multiple attempts to deliver the message are made. If it gets lost along the way, the server will re-request the lost part. In TCP, there's either no missing data, or, in case of multiple timeouts, the connection is dropped.	When a message is sent, it cannot be known if it will reach its destination; it could get lost along the way. There is no concept of acknowledgment, retransmission or timeout.
Ordered	The bytes send in a stream are handed over in the same order. When data segments arrive in the wrong order, TCP buffers the out-of-order data until all data can be properly re-ordered and delivered to the application.	If two messages are sent to the same recipient, the order in which they arrive cannot be predicted.

In the world of TCP/IP, with which we are most familiar, TCP provides reliable data communication streams and as application users we do not need to concern ourselves with issues such as packets arriving out of order or being lost in transit.

Why UDP, why not TCP?

Key requirements for the Gemini protocol are that it is:

- **Lightweight:** It places low demand on the network and Gemini hardware and the connecting operating system
- **Responsive:** Gemini receives commands and clients receive responses quickly
- **Reliable:** The protocol must detect and correct communications errors
- **Ordered:** Commands are processed by Gemini in the order in which they are sent

TCP based protocols are available to access Gemini, but have certain disadvantages that should be considered:

1. **Lightweight**: High impact on the Gemini CPU when using the built-in TCP stack. The necessary buffering limits the amount of available sockets.
2. **Responsive**: The TCP implementation decides how long data is buffered before sending. Slow responses when using Windows clients²

These were traced to:

1. A TCP stack needs resources to implement the features described above, impacting overall responsiveness and throughput.
2. The TCP stack decides when a data segment is sent. The default value of the Microsoft TCP stack `TcpAckFrequency` setting (2), resulted in a 200ms delay in the round trip time to Gemini because the stack was waiting for a second packet from Gemini before sending the ACK packet for the first packet. Gemini on the other hand was waiting for an ACK to the first packet before sending the second packet... deadlock! The deadlock was always broken after 200ms by the Microsoft stack sending an ACK, even if it had not received a second packet within that time. So the system worked reliably but with frequent delays of 200ms resulting in low throughput.

When `TcpAckFrequency` is set to a value of 1, performance is fast, however, it was not felt appropriate to require all Gemini uses to change the Microsoft default setting and therefore to carry the risk that some other aspect of Windows would not perform as expected.

Testing with UDP was much more successful, providing a higher throughput than TCP, even with `TcpAckFrequency` set to 1 together with lower processor impact due to the lower protocol overhead for Gemini and because it bypassed the higher levels of the Microsoft TCP stack at the client.

UDP thus meets the **Lightweight** and **Responsive** Gemini protocol requirements but does not meet the **Reliable** and **Ordered** requirements; consequently we need to provide these at the application level.

Implementation is straightforward since we don't need all the frills that TCP provides and just need a simple, low overhead process. The following sections describe Gemini's UDP protocol and both the client's and Gemini's roles in the error recovery process.

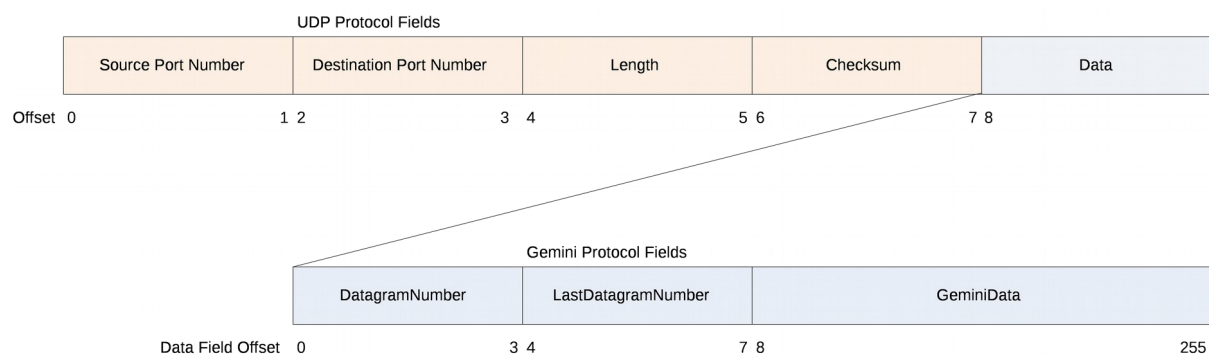
Access to Gemini's UDP Server

Gemini creates a UDP listener socket by default on port 11110; there is no security access control, login or similar concept. If you can reach port 11110 then you can immediately send commands to Gemini and receive responses. The UDP port number used by the Gemini server can be changed through the Network Settings page of the Gemini's built-in web application.

² In testing René used a Linux client and did not see the 200ms ACK delay, this issue was only observed in the Microsoft TCP/IP stack.

UDP Datagram Format

This diagram shows the UDP protocol fields and the Gemini protocol fields that are defined within the UDP Data field.



Your UDP protocol stack will take care of the first 8 bytes (0 to 7) of the UDP protocol fields, which record the source and destination IP ports, the length of the datagram and a checksum over the entire datagram. The Data field (shown in blue), starting at offset 8 is where the Gemini protocol fields are located.

The Gemini protocol defines three fields within the UDP datagram Data field:

Field	Offset	Purpose
DatagramNumber	0	<p>On sending a command, this must be a unique number for this datagram from this client. It sequentially increments by one for each datagram starting at 0.</p> <p>On receipt of a datagram from Gemini, this will contain the DatagramNumber of the command for which this is the response.</p>
LastDatagramNumber	4	<p>This should be set to zero for commands to Gemini and will normally be zero in responses from Gemini.</p> <p>It is set to the DataGramNumber of the last received command from the client, in response to a NAK command from the client. (See error handling protocol section)</p>
GeminiData	8	<p>Command to Gemini, using the Gemini Serial Command syntax or response from Gemini, in both cases terminated by the NULL character (0x00).</p> <p>The maximum size of the GeminiData field is 255 bytes; the total size of Gemini command strings, including NULL characters, should not exceed this size.</p>

Client Protocol

The overall Gemini client - server protocol is set out diagrammatically in Appendix 1. The happy path is straightforward comprising:

- Open a socket connection
- Loop as needed:
 - Construct a datagram containing a Gemini serial protocol command
 - Send the datagram to Gemini
 - Listen for the response datagram
 - Extract Gemini's response from the return datagram and pass to the application
- Close the socket connection

Sending Individual Serial Commands to Gemini

In these cases the data to be sent is an individual serial command as defined in the serial protocol, which must be terminated with a NULL (0x00) character. In this example you want to send the :GR# command. So, you would determine the next unused DatagramNumber by incrementing that last used DatagramNumber and then construct a GeminiData field comprising:

- 4 bytes of DatagramNumber field
- 4 bytes of 0x00 for the LastDatagramNumber field
- 4 characters of serial command :GR#
- 1 byte of NULL character(0x00) as terminator

and send it to Gemini as a UDP datagram.

In response, you will receive a datagram containing something like:

- 4 bytes of DatagramNumber field (same as the one you sent)
- 4 bytes of 0x00 for the LastDatagramNumber field
- 9 characters of response 13:45:23#
- 1 byte of NULL character(0x00) as terminator

Sending Multiple Gemini Serial Commands to Gemini

The UDP protocol supports sending multiple Gemini Serial Commands in one datagram. For example, if you send the commands :

- 4 bytes of DatagramNumber field
- 4 bytes of 0x00 for the LastDatagramNumber field
- 17 characters of serial command :GR#:GD#:GS#:GVP#
- 1 byte of NULL character(0x00) as terminator

in one datagram you will receive a response datagram containing something like:

- 4 bytes of DatagramNumber field (same as the one you sent)
- 4 bytes of 0x00 for the LastDatagramNumber field
- 43 characters of response 13:45:23#75:34:09#09:56:09#Losmandy Gemini#
- 1 byte of NULL character(0x00) as terminator

Serial Commands with no Response

Some serial protocol commands have no response e.g. :Q#, :RS# If one of these is sent on its own, there should be no response for the client and the client will be unable to determine that the command has actually been received and acted upon.

To cater for this situation, Gemini will respond with a datagram containing an ACK (0x06) character terminated with a NULL (0x00) character whenever there is no serial command response to return. Thus the client should wait for a response datagram from Gemini under all circumstances. It will either contain the response or the ACK character if there is no serial command response.

If multiple commands are sent, one of which does not have a response, the ACK character will not be returned as the response to the other commands indicates that the response-less command was received and processed OK.

Datagram Reliability - Arrival Order

Since UDP does not guarantee packet arrival order, critical sequences must be managed at the application level. An example of a critical sequence is the "Equatorial Slew" sequence where the Serial Protocol specifies that the :Sr command must be sent before the :Sd command.

This can be achieved in two ways:

1. Ensure that your client always waits for the response to a command before sending the next command, i.e. always have only one datagram "in flight" at a time.
2. Place all critical sequence commands into one datagram using the "Multiple Gemini Serial Commands approach described above. This will ensure that they all either arrive together and are processed in the order that you placed them in the datagram, or they do not arrive at all and are resent together through the "Errors and Losses" process described below.

Either method will work on its own but using both together is recommended.

A list of critical sequence commands is given in Appendix 3.

Datagram Reliability - Errors or Losses

The Gemini UDP protocol specifies that every command datagram will receive a response datagram, even if the Gemini serial command does not have an application response. When things go wrong we need to consider two scenarios:

1. Command never reaches Gemini, so no response is sent
2. Command is received by Gemini but response is lost in transit and never reaches the client

From the client's perspective, both scenarios look the same: I sent a UDP datagram and I never received a response datagram; so it has no idea whether or not Gemini received and acted on the command. To detect that something has gone wrong, the client should implement a timeout in its "wait for response UDP datagram" routine.

When this is activated, the client must query Gemini to find out why no response was received, fortunately Gemini can supply information that allows the client to determine whether scenario, 1. or 2. above applies.

So, on timing out, the client should send a 9 byte NACK datagram^{3,4} to Gemini:

- 4 bytes of DatagramNumber field
- 4 bytes of 0x00 for the LastDatagramNumber field
- 1 byte of NACK character(0x15)

Gemini will respond with a datagram like this:

- 4 bytes of DatagramNumber field (same as supplied in NACK datagram)
- 4 bytes of LastDatagramNumber (DatagramNumber of the last command received)
- Response to the command in the LastDatagramNumber datagram
- 1 byte of NULL character(0x00) as terminator

The client should examine the LastDatagramNumber field and compare it with the DatagramNumber of the command that timed out.

- If it is the same, Gemini did receive the command and the response was lost en-route from Gemini to the client. The required response is now in this datagram so it can be returned to the client application method that initiated the command. Job done!
- If it is different, then Gemini never received the command in the first place, so the client should resend the command that timed out, using a new DatagramNumber, and return the response to that datagram to the calling method. Job done!

Under network loss conditions, Gemini may time out on several consecutive attempts to retrieve the command response. In this circumstance, the communications routine should throw an exception back to the application indicating loss of communication to Gemini, so it can trigger appropriate messages to higher level applications, the user interface or the application log.

Macro Commands

For convenience, if you send a datagram that just contains the ENQ command shown below, you will receive the composite response shown.

Command	Response
0x05	Coordinates PRA, PDEC (as integer values) , RA, DEC, HA, AZ, EL (as double values), movement rate (N/T/G/C/S see :Gv#) and side of pier (W/E see :Gm#) each separated by a semicolon. Sidereal time and tracking time left until Safety Limit is reached. State bits (see <99), servo lags in both axes, PWM duty in both axes. e.g.: 1152000;1152000;0.907784;+90.000000;+6.000001;180.000000;+33.818611;N;N;N ;E;6.907785;0;32;26327.898667;1;01060100;0;0;0;0;

Future versions of the UDP protocol are expected to extend this command set.

³ There is no NULL terminator character in this datagram

⁴ The DatagramNumber field should be the next unused value in sequence, it must not be any previously used value

Document Licensing

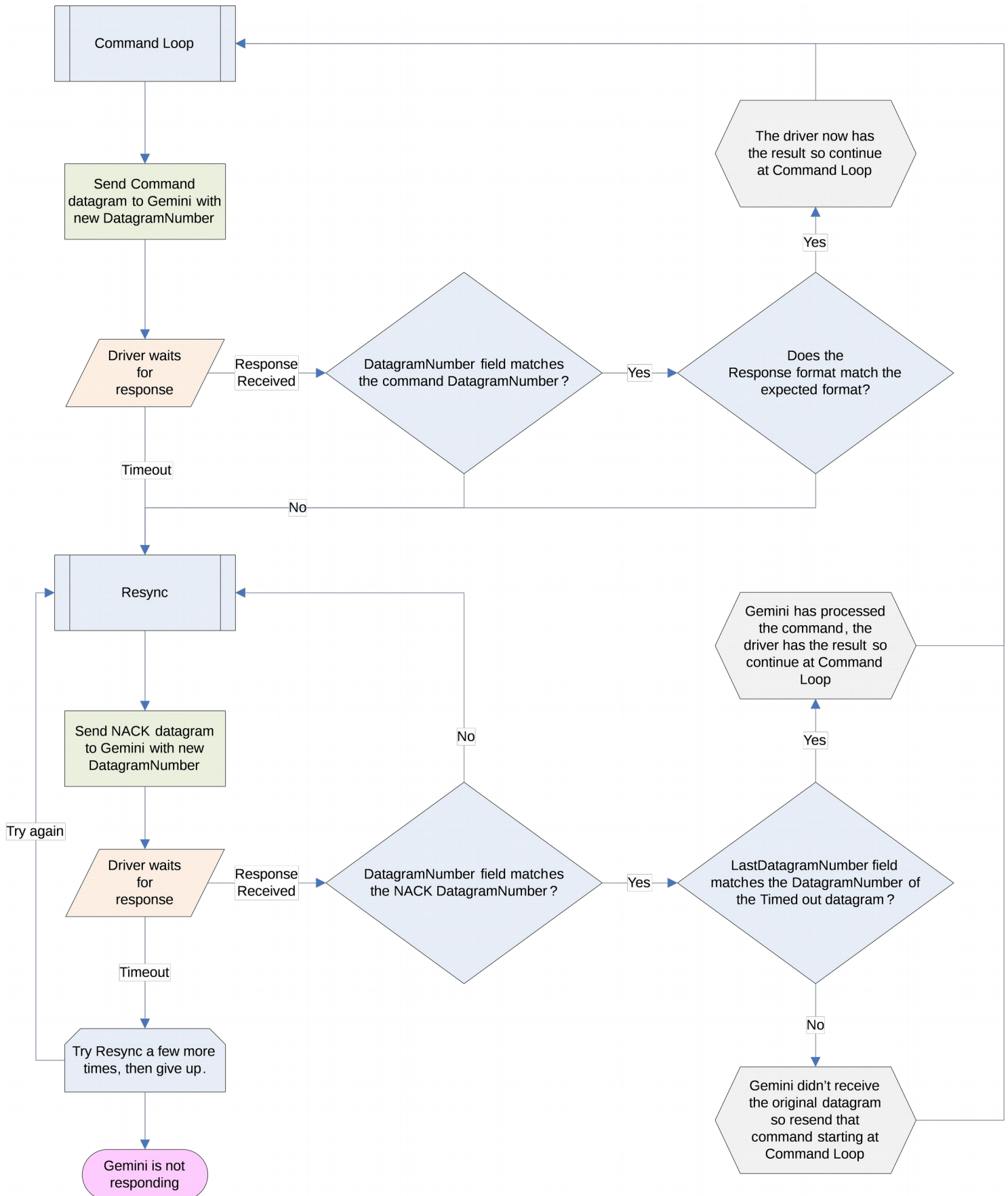
This document contains text from [Wikipedia](http://en.wikipedia.org/wiki/User_Datagram_Protocol) http://en.wikipedia.org/wiki/User_Datagram_Protocol

To comply with its licensing requirements, this document is distributed under the same Creative Commons Attribution ShareAlike 3.0 Unported (CC-BY-SA 3.0) license <http://creativecommons.org/licenses/by-sa/3.0/>, which makes it freely copyable and usable for commercial purposes, so long as appropriate attribution is made and derived works are published under the CC-BY-SA license.

Document History

Version	Release Date	Changes
0.1 Draft	15 th October 2011	Initial draft for review
0.2 Draft	23 rd October 2011	Tidied typos, created new comparison section for TCP and UDP, reworked the "Why UDP?" section. Refined code example and wording in protocol flowchart. Added title page. Improved UDP datagram format picture
1.0 Final	29 th October 2011	UDP Datagram format: Confirmed GeminiData field length as 255. Macros: Tidied to allow for future protocol extensions. Appendix 2 - .NET example code: Tidied variable names and added more comment. Appendix 3 - Critical sequences: Added native commands to critical sequence list and provided example.
1.1 Draft	3rd November 2011	Corrected spelling of René's name -oops, sorry! Corrected Macro command character to ENQ rather than ACK.
1.2 Draft	7 th March 2018	Extended ENQ Macro description.

Appendix 1 - Client Protocol Flowchart



Appendix 2 - .NET Example Code

```
Imports System.Net
```

```
Imports System.Net.Sockets
```

```
Imports System.Text
```

```
'This code shows how to open a UDP connection, send a command and receive a response from Gemini
```

```
Dim hostinfo As IPEndPoint, UDPsocket As Socket
```

```
Dim TransmitBytes(255), ReceiveBytes(255) As Byte, NumberOfBytes As Integer, ReceiveString As String
```

```
Dim GeminiCommand As String
```

```
'Constants defining field positions within the datagram and the length of both integer fields
```

```
Const DatagramNumberField As Integer = 0
```

```
Const LastDatagramNumberField As Integer = 4
```

```
Const GeminiDataField As Integer = 8
```

```
Const IntegerFieldLength As Integer = 4
```

```
GeminiCommand = "GVP#" 'Command to be sent to Gemini
```

Try

'Open a UDP socket to the Gemini server

UDPSocket = New Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp) ' Create a UDP socket

hostinfo = Dns.GetHostEntry("gemini") ' Find the IP address of Gemini

UDPSocket.Connect(hostinfo.AddressList, 11110) 'Connect the socket to the Gemini server

UDPSocket.ReceiveTimeout = 2000 'Set the receive timeout to 2 seconds

'Construct the datagram to be sent to Gemini in the TransmitBytes byte buffer

'Set the Gemini DatagramNumber to 1 as this is the first datagram and the LastDatagramNumber to 0

Array.Copy(BitConverter.GetBytes(1%), 0, TransmitBytes, DatagramNumberField, IntegerFieldLength)

Array.Copy(BitConverter.GetBytes(0%), 0, TransmitBytes, LastDatagramNumberField, IntegerFieldLength)

'Add the command to be sent and the trailing NULL character starting at offset 8

Array.Copy(Encoding.ASCII.GetBytes(GeminiCommand), 0, TransmitBytes, GeminiDataField, GeminiCommand.Length)

Array.Copy(Encoding.ASCII.GetBytes(Chr(0)), 0, TransmitBytes, GeminiDataField + GeminiCommand.Length, 1)

'Send the datagram to Gemini

UDPSocket.Send(TransmitBytes, DatagramNumberField, GeminiDataField + GeminiCommand.Length + 1, SocketFlags.None)

'Receive the returned datagram data field from Gemini into the ReceiveBytes byte buffer

```
NumberOfBytes = UDP Socket.Receive(ReceiveBytes)
```

'Extract the command response as a string and display it

'Ignore the first 8 bytes containing DatagramNumber and LastDatagramNumber

```
ReceiveString = Encoding.ASCII.GetString(ReceiveBytes, GeminiDataField, NumberOfBytes - GeminiDataField)
```

```
ReceiveString = ReceiveString.TrimEnd(Chr(0)) 'Remove the NULL character terminator added by Gemini
```

```
MsgBox("Gemini response to " & GeminiCommand & ": " & ReceiveString)
```

'Close the UDP socket connection

```
UDP Socket.Close()
```

Catch ex As Exception

```
MsgBox(ex.ToString)
```

End Try

Appendix 3 - Critical Sequence Commands

LX200-Like Commands

- :Os with :Od
- :OS with :OR
- :RC with :MA
- :RC/RG/RM/Rm/RS with :Me/Mw/Mn/Ms/Ma/Mi/Mg
- :SG with :SL/SC
- :Sr with :ON with :Sd (This order is recommended in the serial commands description)
- :Sz with :ON with :Sa

Gemini Native Command Sequences

For maximum UDP communication robustness, native commands in the following sequences should be grouped together. You do not have to send all of the commands in each sequence, but if you are setting several of these parameters as part of a single application level operation, it is advisable to send them all in one datagram.

- >1: to >26:
- >100: and >110:
- >120: to >172:
- >201: to >211:
- >220: to >223:
- >411: to >415:
- >501: to >504:
- >801: to >815:

E.g. if you wish to set the:

- mount type to MI250
- DEC and RA worm ratios to -360 and 25
- Move and slew speeds to 500
- PE counter to zero

You should, at a maximum, send these as three datagrams:

- Mount type and worm ratios because they are in the >1: to >26: range
- Both movement rates because they are in the range >120: to :172:
- Reset PE counter because it is the range >501: to >504:

Of course, there is nothing to stop you merging these into two or just one datagram so long as all the commands in each bullet above stay in the same datagram.