Ultimate Networking Command Technical Reference

Scott Hutter

Draft 1.0

This document is meant to explain the usage of the networking capabilities of the U64 and Ultimate II/+ cartridges. It is, admittedly, absolutely terrible documentation. But the hope is to get an understanding out on how to use these functions, and not so much any unnecessary detail beyond that.

The Ultimate devices communicate via the following registers:

Control Register	0xDF1C
Command Data Register	0xDF1D
Status Data Register	0xDF1F

Targets

When sending commands to the Ultimate devices, the Control Register is used. The first byte to send is known as the Command Target. They are:

0x01 – DOS1

0x02 – DOS2

0x03 – Network

0x04 – Control

Targets 1 and 2 represent the built in UltimateDOS instances. Two targets allow for communicating to the file system in multiple ways. Functions include mounting disks, reading directory contents, etc. The standard documentation for UltimateDOS covers usage of these targets and is not covered here.

Target 3 is the target that concerns this document. Using this target, we can send various networking commands to the device, such as connect, read and send data through a socket connection.

Target 4 is the Control target, which represents the entire cartridge device control functions. With this target, it is possible to send command which will do things such as force the system into the freeze menu. Again, these functions are not covered in this document, and are considered undocumented commands (therefore could change).

Ultimate Networking Command Technical Reference

Scott Hutter

Draft 1.0

Sending commands to the network target

The process for sending a connection request would be as follows:

1) Wait for an idle state on the Status register (just a loop checking the status register will do):
    a. while (!(((*statusreg & 32) == 0) && ((*statusreg & 16) == 0))) { … just wait.. };
2) Send the command to the command register:
    a. Byte 0: 0x03 = Network Target
    b. Byte 1: 0x07 = Connect command
    c. Byte 2: 0x00 = Port Lo Byte (example 6400)
    d. Byte 3: 0x19 = Port Hi Byte
    e. Bytes 4 – X = Hostname, null terminated
3) After sending the command bytes through the command register, we read the Data Register as long as data is available. For this command, this register will contain the socket number.
    a. Is data available? If so, read the register and append its contents to a buffer
        i. If(((*statusreg & 128) == 128))
            1. YES: read a byte from the data register and append to our data buffer, loop
            2. NO: return
4) Next, we next read the status register until this is no more data available from it. The status register will contain various ASCII responses to the commands being sent.
    a. Is data available?
        i. If(((*statusreg & 128) == 128))
            1. YES: read a byte from the status register and append to our status buffer, loop
            2. NO: return
5) Finally, we “accept” the response, by sending an acknowledgement back through the control register via bit 1 :
    a. *controlreg |= 0x02
6) At this point, we are either connected to the host or a connection error has occurred. The status register will have the detail
7) If we are connected, the Data Register will contain a socket number that we can use for accessing this socket via read, write, and close commands

This pattern is duplicated for all commands (and even other command targets)

Ultimate Networking Command Technical Reference

Scott Hutter

Draft 1.0

Sending data to the network target

The process for sending data to a connected host would be:

1) Wait for an idle state on the Status register (just a loop checking the status register will do)
    a. while (!(((*statusreg & 32) == 0) && ((*statusreg & 16) == 0))) { … just wait.. };
2) Send the command to the command register:
    a. Byte 0: 0x03 = Network Target
    b. Byte 1: 0x11 = Socket write command
    c. Byte 2: 0x?? = Socket Number (returned from connect command)
    d. Byte 3– X = data to send, null terminated
3) After sending the command bytes through the command register, we read the Data Register as long as data is available. This command does not return data on the data register, but I call it anyway.
    a. Is data available? If so, read the register and append its contents to a buffer
        i. If(((*statusreg & 128) == 128))
            1. YES: read a byte from the data register and append to our data buffer, loop
            2. NO: return
4) Next, we next read the status register until this is no more data available from it. The status register will contain various ASCII responses to the commands being sent.
    a. Is data available?
        i. If(((*statusreg & 128) == 128))
            1. YES: read a byte from the status register and append to our status buffer, loop
            2. NO: return
5) Finally, we “accept” the response, by sending an acknowledgement back through the control register via bit 1 :
    a. *controlreg |= 0x02

Reading data to the network target

The process for sending data to a connected host would be:

1) Wait for an idle state on the Status register (just a loop checking the status register will do)
    a. while (!(((*statusreg & 32) == 0) && ((*statusreg & 16) == 0))) { … just wait.. };
2) Send the command to the command register:
    a. Byte 0: 0x03 = Network Target
    b. Byte 1: 0x10 = Socket read command
    c. Byte 2: 0x?? = Socket Number (returned from connect command)
    d. Byte 3: 0x?? = Lo Byte of length of data to read (ie number of bytes to read)
    e. Byte 4: 0x?? = Hi Byte of length of data to read
2) After sending the command bytes through the command register, we read the Data Register as long as data is available.
    a. Is data available? If so, read the register and append its contents to a buffer
        i. If(((*statusreg & 128) == 128))
            1. YES: read a byte from the data register and append to our data buffer, loop
            2. NO: return
3) Next, we next read the status register until this is no more data available from it. The status register will contain various ASCII responses to the commands being sent.
    b. Is data available?
        i. If(((*statusreg & 128) == 128))
            1. YES: read a byte from the status register and append to our status buffer, loop
            2. NO: return
4) Finally, we “accept” the response, by sending an acknowledgement back through the control register via bit 1 :
    c. *controlreg |= 0x02

Closing a socket on the network target

The process for closing a connection would be:

1) Wait for an idle state on the Status register (just a loop checking the status register will do)
    a. while (!(((*statusreg & 32) == 0) && ((*statusreg & 16) == 0))) { … just wait.. }
2) Send the command to the command register:
    d. Byte 0: 0x03 = Network Target
    e. Byte 1: 0x09 = Socket close command
    f. Byte 2: 0x?? = Socket Number (returned from connect command)
3) After sending the command bytes through the command register, we read the Data Register as long as data is available. Normally this command will not return data.
    a. Is data available? If so, read the register and append its contents to a buffer
        i. If(((*statusreg & 128) == 128))
            1. YES: read a byte from the data register and append to our data buffer, loop
            2. NO: return
3) Next, we next read the status register until this is no more data available from it. The status register will contain various ASCII responses to the commands being sent.
    b. Is data available?
        i. If(((*statusreg & 128) == 128))
            1. YES: read a byte from the status register and append to our status buffer, loop
            2. NO: return
4) Finally, we “accept” the response, by sending an acknowledgement back through the control register via bit 1 :
    c. *controlreg |= 0x02

Ultimate Networking Command Technical Reference

Scott Hutter

Draft 1.0

Command summary:

Command	Byte
0x02	Get Interface Count (only 1 currently)
0x05	Get IP Address
0x07	Connect
0x09	Close connection
0x10	Read data from socket
0x11	Write data to socket

A C wrapper for these functions as well as UltimateDOS for the cc65 compiler is available at:

https://github.com/xlar54/ultimateii-dos-lib