Distributed Display Protocol (DDP)

Last update 2022-09-05

Introduction

DDP was designed for sending real-time data to distributed lighting displays where synchronization may be important. It can also be used to control other types of devices needing a similar protocol.

DDP is being used today, but some of the advanced features have not yet been decided on. Items in this document will be in italics to indicated they have not been decided upon or standardized.

History

Traditionally, the DMX protocol over RS-485 was used to control small numbers of dimmable lights. Because of the relation of DMX to lighting, people have tried to apply DMX to controlling much larger arrays of lights, for example RGB LED displays. However, DMX has many limitations including small universe/data-packet sizes, low update rates, and no methods for display synchronization. There are two popular open protocols that transport DMX over ethernet (Art-Net and E1.31), but neither of these protocols solve all the problems. DMX fails in these applications but people continue to use it for lack of a better standard.

For example, if you tried to use DMX (over Art-Net) to control a large scale distributed RGB LED lighting system, you would run into the following issues:

- Small number of lights can be controlled per DMX universe. Assuming 24-bit RGB data, only 170 nodes per DMX universe. Even less if you wanted to use 36-bit or higher data widths.
- Small number of DMX universes (256), so a maximum of 256*170=43,520 nodes possible.
- Low maximum update rate of around 45hz, not suitable for special applications. Unspecified control of actual data rate.
- No synchronization across DMX universes. Data sent across multiple DMX universes at different times would not be displayed at the same time, which is not suitable for large displays. [note in 2022: Art-Net v4 was updated to address some of these issues]

Efficiency

Let's look at the efficiency of DDP vs Art-Net vs E1.31. We define efficiency as the amount of useable data (RGB pixel data) received divided by the total amount of data that needs to be transmitted. For example, if a protocol sends 100 bytes to get 50 bytes of useable data, then it is 50% efficient. Ethernet packets have an overhead of 66 bytes (8 byte preamble, 14 byte addresses, 4 byte crc, 12 byte gap). There is also overhead for IP (20 bytes) and UDP (8 bytes). So total overhead before adding in each specific protocol is 66 bytes. For example, for E1.31 you need to send 66 bytes (enet+ip+udp) + 126 bytes (e1.31) + 512 bytes (dmx data). So efficiency is 512/(66+126+512) = 72.7%

We will also compute the maximum possible number of RGB pixels that could be addressed at 45fps on 100Mbit ethernet. 100Mbit = 12.5M bytes per second. We divide this by each protocols total packet size, for instance, 584 bytes for E1.31. This gives the total number of packets that could be sent in 1 second. We multiply this by the useable data size divided by 3 for RGB to calculate how many pixels can be addressed per second, and then divide by 45 frames per second to calculate how many pixels can be address at 45fps.

protocol	header len	max data per packet	Efficiency	pixels at 45fps on 100M net
E1.31	126	512	72.7%	67,340
Art-Net	18	512	85.9%	79,542
DDP	10	1440	94.9%	87,950

You may also hear arguments that these protocols are more efficient because they support broadcasting or multicasting. In some special cases this may be true. Consider a case where you have 170 separate RGB devices (for example, a color settable wall-washer) and each one needs 3 data channels to operate. Each device has its own network interface and IP address. With Art-Net or E1.31 you could put all these channels in a single DMX universe and broadcast that packet once to all 170 devices. Without broadcasting, you would need to unicast that data to all 170 devices. So for very small networks of devices broadcasting could be more efficient, but this isn't true beyond that.

Consider that when driving RGB pixel displays, you will most likely have more than 170 pixels connected to a single device (pixel controller). So you already need to send a full packet to each device. Now, consider the data you are displaying is most likely different for every pixel on every device (in some cases you might just want all your pixels to be the same color, but not likely for long). So using broadcast or multicast would not reduce the number of packets that need to be sent to update all your pixels. In fact, doing so is likely to interrupt and slow down many receivers that get overloaded processing these broadcasts.

Broadcasting or multicasting is also less reliable on Wifi networks. To quote a White Paper from Cisco (Optimizing Enterprise Video Over Wireless LAN White Paper):

> Packet Loss
>
> Given the combination of collisions, fades, and data rate selection, it is not at all uncommon for Wi-Fi to operate with an underlying packet error rate (PER) that can approach 5 percent. To compensate, Wi-Fi uses a retransmission mechanism whereby packets that are not successfully received and acknowledged are resent. This mechanism generally serves to reduce the final packet loss rate (PLR) to less than 0.1 percent.
>
> Multicast Unreliability
>
> The underlying packet error rate plays an even more prominent role for Wi-Fi multicast traffic. For multicast transmissions (with multiple receivers), Wi-Fi does not provide a retransmission mechanism. As a result, the PLR for multicast traffic is equal to the PER. In other words, it would not be uncommon for Wi-Fi multicast traffic to experience a packet loss rate of 5 percent. This is a serious problem for video, where loss of even a single packet can result in an error that propagates for many video frames.

DDP Design Philosophy

To create a small, simple, extensible protocol for distributing and synchronizing data across multiple display devices.

Implementations can pick various levels of protocol support depending on the application and processing resources available.

The protocol should be scalable to support applications as simple as a light switch controlling a light bulb, a controllable holiday light tree, all the way to a large system controlling an entire venue consisting of RGB video displays, light strings, strobes, moving lights and foggers.

The protocol supports data sourced from the packet, or from storage on the display device.

Data is typed and a full range is supported, from on/off, greyscale, RGB, etc. Data does not have to contain light or color info but could be commands, sequence numbers, directional information, or a mix.

The protocol should allow for status, configuration and management of devices.

Definitions used in this document

Controller:
: generally, a system that sends data to and controls a Display

Display:
: generally, a system that receives and displays data from Controllers

A device can operate as both a Controller and a Display.

Protocol Operation

The protocol is packet based and thus works nicely over UDP, which all implementations should support. The protocol can also be used over TCP for applications needing higher reliability or for other reasons.

The protocol uses a single packet header definition for all packets, for ease of implementation.

Many display applications can tolerate a low level of packet loss. Use TCP if needed.

Data is generally sent using direct IP destination addresses, and broadcasts of data should not be used unless the same data is meant for all destinations.

Data is transmitted to displays in blocks specifying their length and offset within an output frame buffer. They can be sent in any order. Multiple blocks can be sent until the buffer is filled. The buffer is not cleared between display commands, so it is possible to send just the data that has changed between frames. *this could be a configuration option, what action to take after displaying an output frame buffer. leave alone, blank, etc*

It is also possible to specify that data comes from Storage instead of from the packet. This could be used to display data that is stored on local disks or flash cards, for example.

A display not using synchronization can display data as it is received.

Synchronization of display output across multiple devices is supported in two ways, both optional: using the Push flag, or using a Timecode.

PUSH FLAG
: Sending the Push flag tells a device to display its data buffer immediately. When sending to a single display device, the Push flag can be sent along with a data packet, often the last packet of multiple blocks used to fill a frame buffer.

  When sending to multiple display devices, the Push flag can be broadcast with no data. All devices will then display their frame buffers simultaneously.

  The Push method can be used when all display devices are on a single network that supports broadcasts and latency is minimal.

TIMECODE
: A timecode is sent (along with the Push Flag) that tells the display device at what future time to do the final Push and display the data. An NTP based time format is used. To use this feature all your display devices will need to be synchronized using NTP.

Source and Destination IDs

A Display uses source and destination IDs to specify where data is read from or written to. For example, a Display may have multiple physical output devices, and each one can use a different ID. IDs are also used for reading and writing configuration and status info. They can be used for numerous purposes, such as specifying flash memory for remote codes updates, or even sub-regions of an output display.

For example, an RGB video display might use ID 1. You could defined ID 2 to be a small corner of that display region to allow updates to a portion of the display. The data for ID 2 might come from a different source Controller than for ID 1.

Status, Configuration and Management

The protocol for data transmission from Controller to Display is kept short and simple as this is the most important function. Optional higher level functions are implemented with JSON formatted data.

Port Numbers

Displays always receive packets on UDP/TCP Port 4048. Controllers can send from any source port. A dual Controller/Display device might use port 4048 for both source and destination packets.

Packet Format

```
All packets sent and received have a 10 or 14 byte header followed by optional data.

byte  0:    flags: V V x T S R Q P

            V V:    2-bits for protocol version number, this document specifies version 1 (01).

            x:      reserved

            T:      timecode field added to end of header
                    if T & P are set, Push at specified time

            S:      Storage.  If set, data comes from Storage, not data-field.

            R:      Reply flag, marks reply to Query packet.
                    always set when any packet is sent by a Display.
                    if Reply, Q flag is ignored.

            Q:      Query flag, requests len data from ID at offset (no data sent)
                    if clear, is a Write buffer packet

            P:      Push flag, for display synchronization, or marks last packet of Reply


byte  1:    x x x x n n n n
            x: reserved for future use (set to zero)
            nnnn: sequence number from 1-15, or zero if not used
              the sequence number should be incremented with each new packet sent.
              a sender can send duplicate packets with the same sequence number and DDP header for redundancy.
              a receiver can ignore duplicates received back-to-back.
              the sequence number is ignored if zero.

byte  2:    data type
            set to zero if not used or undefined, otherwise:
            bits: C R TTT SSS
             C is 0 for standard types or 1 for Customer defined
             R is reserved and should be 0.
             TTT is data type
              000 = undefined
              001 = RGB
              010 = HSL
              011 = RGBW
              100 = grayscale
             SSS is size in bits per pixel element (like just R or G or B data)
              0=undefined, 1=1, 2=4, 3=8, 4=16, 5=24, 6=32

byte  3:    Source or Destination ID
            0 = reserved
            1 = default output device
            2=249 custom IDs, (possibly defined via JSON config)
            246 = JSON control (read/write)
            250 = JSON config  (read/write)
            251 = JSON status  (read only)
            254 = DMX transit
            255 = all devices

byte  4-7:  data offset in bytes
            32-bit number, MSB first

byte  8-9:  data length in bytes (size of data field when writing)
            16-bit number, MSB first
```

```
            for Queries, this specifies size of data to read, no data field follows header.

if T flag, header extended 4 bytes for timecode field (not counted in data length)
byte 10-13: timecode

byte 10 or 14: start of data
```

To Write To an ID

Send data packets with type, ID, offset, length and data fields, as many as wanted to fill remote frame buffers.

Send last data packet with Push flag set (or broadcast to multiple devices).

To Read From an ID

An ID can be read from, if supported. This can be used to read from a frame buffer, JSON config or status, etc.

Send packet with Query flag, ID to read from, offset=starting offset, data length=number of bytes to read. *[length should fit into UDP packet, what about fragmentation, MTU, etc?]*

Device will reply with data packet with Reply bit set. Data offset specifies where in device buffer data came from and should match Query. Data length might be shorter than requested or zero if no data. If Push flag is set it marks end of device data.

If reading a device is not supported, Reply should be sent with offset=0, length=0, Push flag.

Discovery

Send directed Read packet for ID=[JSON Status] Reply tells you if server exists.

A Display that has powered up can broadcast a short [JSON Status] update Reply:

```
{"status":{"update":"change","state":"up"}}
```

A Controller can scan all IP address on a local network to discover devices, or use a list configured by the user.

Discovery by Broadcasting

A [JSON Status] query can also be broadcast, and replies collected to discover all Displays. It is suggested that if a Display receives such a query, that it delay its reply by the number of milliseconds of the last byte of its MAC address. As such, Controllers should wait a half-second for any replies.

Displaying data from local storage

If S flag is set then data is read from a storage unit instead of from packet data field. Data field defines storage unit, by name, number, URL or whatever mechanism wanted. Data Offset defines where to start reading from within storage unit. Display reads enough data to fill output buffer for the particular ID.

Timecode Usage

Use and support of timecodes is optional.

The Timecode is the 32 middle bits of 64-bit NTP time: 16-bits for seconds and 16-bits for fraction of seconds.

Full 64-bit NTP time is not used because DDP is a real-time protocol, and 16 bits for seconds (1092 minutes) is plenty to tell if the packet is early or late. In addition, 16 bits for fractions of a second (15 microsecond resolution) is more than enough needed for display use.

To use timecode when writing data, set Timecode flag. Append 4 bytes (MSB first) of timecode at end of packet header.

Timecode specifies the time the Display should output data at (only when used with Push flag!). Data received after specified time has past should still be placed into device frame buffer, but receiver can decide whether or not to display it, and/or how long late means. *The parameters could be defined by JSON config.*

DMX Legacy Mode

It is the intention of this protocol to replace other DMX over ethernet protocols, and eventually DMX itself. However, for certain legacy applications you may want to use DDP to transport DMX. We define a standard way to be followed:

Send DMX packets to the DMX transit ID (254). Use the data offset field for the DMX universe number (converted to 32-bit unsigned integer), and place the DMX data including the START code in the data field (up to 513 bytes).

A Controller may transmit DMX packets to a Display. A Display may send unsolicited Replies with DMX packets back to a Controller. This allows transit support of DMX RDM protocol.

Configuration, Status and Control

Higher level management functions are sent and received using JavaScript Object Notation (JSON). Use of JSON is optional.

However, for discovery you should support the minimum specified in Discovery section above.

This section describes the JSON schema used for status and configuration, and is currently implemented in Minleon products.

JSON can be used for configuration and status.

STATUS:

```
   info is read-only,
   can be returned by device when queried
   device can send updates when status changes if configured to do so by JSON config commands
   examples include temperature monitoring, light failure, power-up,

suggested standard schema (and used by Minleon NDB):
 {"status":
   {
   "man"    : "device-manufacturer-string",
   "mod"    : "device-model-string",
   "ver"    : "device-version-string",
   "mac"    : "xx:xx:xx:xx:xx:xx:xx:xx",
   "push"   : true,      (if PUSH supported)
   "ntp"    : true       (if NTP supported)
   }
 }
```

A status request may also contain a JSON string of hints needed by the Display device in order to send a reply. So the data field of the STATUS request can be set to {"mac":"xx:xx:xx:xx:xx:xx"} along with the appropriate data field length.

Minleon products may require the mac address to be sent so the device can reply. The reason for this is that a new unconfigured device may not yet have its own IP address assigned or know the correct IP network number to use. Once the device is discovered, an IP address could be assigned with the CONFIG message.

The STATUS request can be broadcast to discover all devices on a network.

CONFIG:

```
   read/write
   reading reads entire config, writing can replace all or single elements
   example configuration items:
     number of lights, strings, etc connected
     allowable data types and type# mapping
     ID list and setting of regions
     support Timecode field or not


schema as used in Minleon NDB:
{"config" :
   {
   "ip":       "a.b.c.d"    (IP address)
   "nm":       "a.b.c.d"    (netmask)
   "gw":       "a.b.c.d"    (gateway)
   "ports" : [    (array of output port info)
```

```
    {
      "port": N,  (port #)
      "ts"  : N,  (number of T's)
      "l"   : N,  (number of lights)
      "ss"  : N   (starting slot)
    }, ...
             ]
   }
}

This will reboot the Minleon NDB and re-initialize the light strings
{"config" : {"reboot":1 }}
```

CONTROL:

```
The Minleon WIFI controller accepts the following CONTROL packets on ID=246:
{"control":
   {
     "fx":  "effect-name"             (runs the named effect)
     "int": intensity                 (set intensity level from 0-100)
     "spd": speed                     (set speed from 1-100)
     "dir": direction                 (set effect direction to normal=0 or reverse=1)
     "colors": [{"r":nn,"g":nn,"b":nn},...]   (sets custom r,g,b color values for effects.
                                              up to 3 sets of rgb values can be set)
     "save": 1                        (saves the settings above so they resume after a power cycle)
     "power": n                       (turns power off=0, or on=1)
   }
}

note: "power" should be sent by itself with no other keywords.
      sending "fx" will turn power back on if it was off.


For example, you could send
  {"control":{"fx":"Multi Chaser","int":100,
              "colors":[{"r":255,"g":0,"b":0},{"r":0,"g":255,"b":0}]}}
  to pick the Multi Chaser effect at 100% intensity with custom colors RED and GREEN.


The Minleon WIFI controller also lets you set all or one item on the favorites list:

{"control":
  { "favorites" :[
    {"i":   index                     (index into favorites list from 1-10)
     "fx":  "effect-name"             (runs the named effect)
     "t":   time                      (run effect for this many seconds from 0-32500, 0 to disable)
     "int": intensity                 (set intensity level from 0-100)
     "spd": speed                     (set speed from 1-100)
     "dir": direction                 (set effect direction to normal=0 or reverse=1)
     "colors": [{"r":nn,"g":nn,"b":nn},...]   (sets custom r,g,b color values for effects.
                                              up to 3 sets of rgb values can be set)
    },... ]
  }
}

Fr example,
  {"control":{"favorites": [
                {"i":1,"fx":"Multi Chaser","t":60,"int":100,
                 "colors":[{"r":255,"g":0,"b":0},{"r":0,"g":255,"b":0}]},
                {"i":2,"fx":"Shift","t":60,"int":100,
                 "colors":[{"r":0,"g":255,"b":255},{"r":20,"g":20,"b":20}]} ]}}



You can also read the entire Favorites list by send a Query packet to the CONTROL ID.
```

Security

None provided. Many lighting systems will run on private networks not connected to the Internet. These networks are also likely to be dedicated for lighting control in order to ensure sufficient bandwidth for proper operation.

Open Protocol

DDP is an open protocol and may be freely used and implemented by anyone.

References

- JavaScript Object Notation (JSON)
- Network Time Protocol (NTP)
- DMX specification, DMX512-A. See Wikipedia.
- ANSI E.131 ACN spec. A committee attempt at proclaiming a new protocol to put a legacy protocol (DMX) over ethernet. See Wikipedia.
- Art-Net specification. Seach for it on the author's web site. Version 4 is the latest.
- DDP for Cinder and OF on Github.

Comments

Please address any comments or questions to mark at three(the digit)waylabs.com. You'll figure it out if you are human.

Code

Definitions

```
// DDP protocol header definitions

#define DDP_PORT 4048

#define DDP_HEADER_LEN (sizeof(struct ddp_hdr_struct))
#define DDP_MAX_DATALEN (480*3)   // fits nicely in an ethernet packet

#define DDP_FLAGS1_VER     0xc0   // version mask
#define DDP_FLAGS1_VER1    0x40   // version=1
#define DDP_FLAGS1_PUSH    0x01
#define DDP_FLAGS1_QUERY   0x02
#define DDP_FLAGS1_REPLY   0x04
#define DDP_FLAGS1_STORAGE 0x08
#define DDP_FLAGS1_TIME    0x10

#define DDP_ID_DISPLAY       1
#define DDP_ID_CONFIG      250
#define DDP_ID_STATUS      251


// DDP header format
// header is 10 bytes (14 if TIME flag used)
struct ddp_hdr_struct {
  byte flags1;
  byte flags2;
  byte type;
  byte id;
  byte offset1;  // MSB
  byte offset2;
  byte offset3;
  byte offset4;
  byte len1;     // MSB
  byte len2;
};

// for example code below:
struct ddp_hdr_struct dh;    // header storage
unsigned char *databuf;      // pointer to data buffer
```

Discovery

```
To discover all DDP devices on a network, broadcast a DDP STATUS request:

dh.flags1 = DDP_FLAGS1_VER1 | DDP_FLAGS1_QUERY;
dh.id     = DDP_ID_STATUS;
dh.offset = 0;
dh.len    = 0;
UDP_SEND(255.255.255.255,DDP_PORT,dh,databuf);


Display devices will reply with something like:

dh.flags1 = DDP_FLAGS1_VER1 | DDP_FLAGS1_REPLY | DDP_FLAGS1_PUSH;
dh.id     = DDP_ID_STATUS;
dh.offset = 0;
dh.len    = length of returned JSON data:

databuf: "{"status":{"man":"Minleon","mod":"NDB","ver":"1.0"}}"
```

Configuration

Once devices have been discovered, their specific configuration can be read (or written) using DDP_ID_CONFIG in a similar mannner as above. Packets would be sent to individual devices and not broadcast.

Displaying Data

In this example, a large buffer of RGB data (RGBDATA) needs to be sent to a number of display devices.

```
int NDEVICES;          // number of display devices
int LIGHTS_PER_DEVICE; // how many RGB pairs are being sent to each display device
byte RGBDATA[BUFLEN];  // the data to send

rgbdata_index = 0;
for (devnum = 0; devnum < NDEVICES; devnum++)  // for each output device
{
   output_byte_count = 0;
   frame_offset = 0;
   for (i = 0; i < LIGHTS_PER_DEVICE; i++)  // copy RGB values to output buffer
   {
      databuf[output_byte_count++] = RGBDATA[rgbdata_index++];  // copy R
      databuf[output_byte_count++] = RGBDATA[rgbdata_index++];  // copy G
      databuf[output_byte_count++] = RGBDATA[rgbdata_index++];  // copy B
      if (output_byte_count > (DDP_MAX_DATALEN-3))  // if DDP packet full...
      {
         // send next DDP data packet to device
         dh.flags1 = DDP_FLAGS1_VER1;
         dh.id     = DDP_ID_DISPLAY;
         dh.type   = 1;
         dh.offset = frame_offset;
         dh.len    = output_byte_count;
         if ((NDEVICES == 1) && (i == (LIGHTS_PER_DEVICE-1)))
           dh.flags1 |= DDP_FLAGS1_PUSH;  // push if only 1 device and last packet
         UDP_SEND(ip_addr(devnum),DDP_PORT,dh,databuf);
         frame_offset += output_byte_count;
         output_byte_count = 0;
      }
   }
   if (output_byte_count > 0) // partial packet left to send?
   {
         // send last DDP data packet to device
         dh.flags1 = DDP_FLAGS1_VER1;
         dh.id     = DDP_ID_DISPLAY;
         dh.type   = 1;
         dh.offset = frame_offset;
         dh.len    = output_byte_count;
         if (NDEVICES == 1)
           dh.flags1 |= DDP_FLAGS1_PUSH;  // push if only 1 device and last packet
         UDP_SEND(ip_addr(devnum),DDP_PORT,dh,databuf);
   }
}

// sent data to all devices, now broadcast PUSH flag so they all display sync'd
if (NDEVICES > 1)  // if 1 device, already sent PUSH above
{
   dh.flags1 = DDP_FLAGS1_VER1 | DDP_FLAGS1_PUSH;
```

```
    dh.id     = DDP_ID_DISPLAY;
    dh.offset = 0;
    dh.len    = 0;
    UDP_SEND(local-IP-broadcast-address,DDP_PORT,dh,databuf);
}
```
