

---

## 1. Overview

---

The DxP Protocol is packet based protocol designed to be extensible. This protocol is transmitted over via TCP on a port selected by the user. The factory default port is 9100.

The protocol uses a Hello handshake to establish unique sequence numbers to allow for advanced security when AES encryption is used. With AES enabled, all messages must be encrypted with the AES Passphrase set in the device.

After the Hello, a Command and Response sequence follows. Any number of Command – Response sequences are permitted after Hello.

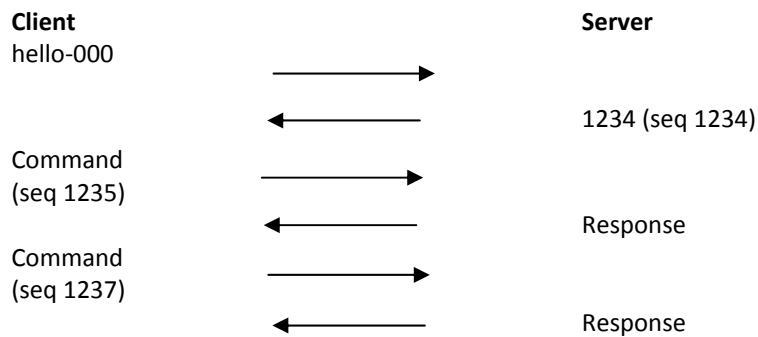
---

## 2. Hello Handshake

---

The client sends a Hello message in the form of the text string 'hello-000'. The DxP enabled device will respond with a packet containing the unsigned 16 bit sequence number. This sequence number is incremented by the client and server with each correct packet sent.

### 2.1. EXAMPLE



---

## 3. DxP Packet

---

The packet is broken up into 2 parts. The first part is the header and the second is the payload. Each of these elements are described below.

### 3.1. Header

The header is used to carry general information such as shown in the 'C' programming structure below;

```
typedef struct {
    eCmdnd command;
    char[21] uName;
    char[21] password;
    uChar desc;
    uChar param;
    uint16 seq;
}THeader
```

#### command

The command variable is an enumerated type that tells the DxP server what class of command is being sent. See **Commands** for a full list of command classes..

#### uName

This variable is reserved for future use. It will contain the user name of a user on the ipIO that is being accessed.

#### password

This variable is reserved for future use. It will contain the password of the user above.

#### desc

This variable is the command descriptor that describes the individual command within a command class. By extension it lets the server know what the payload is. There is a different set of descriptors for each command class. See **Descriptors** for a full list of descriptors by command class.

#### param

This is an optional parameter that can be passed to the server in addition to the descriptor. Reserved for future use.

#### seq

This is the packet's sequence number. It is used as part of the security scheme.

### 3.2. Payload

The payload is determined by a combination of the command class and the descriptor. The payloads are described with the descriptor. See **descriptors** for details.

---

## 4. Commands

---

Currently there are 7 command classes. All classes are defined in the 'C' programming enumerated type definition below.

```
typedef enum {  
    eCmnd_null,  
    eCmnd_set,  
    eCmnd_get,  
    eCmnd_io,  
    eCmnd_keepAlive,  
    eCmnd_rss,  
    eCmnd_rcu  
} eCmnd;
```

- |   |                 |   |
|---|-----------------|---|
| 0 | eCmnd_null      | This is a null command and should not be sent to the server.  |
| 1 | eCmnd_set       | This command is used to set programmable variables on the server  |
| 2 | eCmnd_get       | This command is used to get programmable variables from the server.   |
| 3 | eCmnd_io        | This command is used to monitor and control the I/O on the server.  |
| 4 | eCmnd_keepAlive | This command is sent to the server as a means of the client validating the communications path to the server. |
| 5 | eCmnd_rss       | This command class is used to control the RSS nest using the RCU.   |
| 6 | eCmnd_rcu       | This command class is used to update the display of the RCU.  |

*Note:* The RSS and RCU commands were added for a specific project and are not for general use.

---

## 5. Descriptors

---

Descriptors are used to describe the individual command within a command class and the payload that the packet contains. All of the descriptors and their payloads are outlined by command class below.

### 5.1. eCmnd\_set

The descriptors for this command class will be product specific.

### 5.2. eCmns\_get

The descriptors for this command class will be product specific.

### 5.3. eCmnd\_io

```
typedef enum {  
    eIO_null,  
    eIO_changeRelay,  
    eIO_changeRelays,  
    eIO_getRelay,  
    eIO_getRelays,  
    eIO_getInput,  
    eIO_getInputs,  
    eIO_pulseRelay,  
}eIO;
```

#### eIO\_changeRelay

This command is used to change the status of an individual relay. It carries the TChangeRelay payload. See **Payloads** for details. The server will respond to this command with a single byte of 0 or 1. 0 if the command was successful, and 1 if there was an error;

#### eIO\_changeRelays

This command is used to set ALL of the relays on a device. It carries the TChangeRelays payload. See **Payloads** for details. The server will respond to this command with a single byte of 0 or 1. 0 if the command was successful and 1 if there was an error.

#### eIO\_getRelay

This command has not yet been implemented

#### eIO\_getRelays

This command is used to get the status of all the relays on the server. The server will respond with an array of bytes containing the status of each relay. The size of the array is dependent on the number of relays on the server.

#### eIO\_getInput

This command is not yet implemented.

#### eIO\_getInputs

This command is used to get the status of all input on the server. The server will return an array of bytes containing the status of each input on the server. The size of the array returned is dependent on the number of inputs on the server.

#### eIO\_pulseRelay

This command is used to pulse a relay. It carries the TPulseRelay payload. The server responds with a single byte of either 0 or 1, 0 upon success and 1 upon failure.

## 5.4. eCmnd\_keepAlive

```
typedef enum {  
    eKeepAlive_null;  
} eKeepAlive;
```

### 0 – eKeepAlive\_null

This is the only valid descriptor that the keep alive command supports. It is defined as null as it carries no payload. The server responds with a single byte of 0 or 1, 0 upon success and 1 upon failure.

## 5.5. eCmnd\_rss

These commands are sent from the RCU to the RSS.

```
typedef enum {  
    eRss_null,  
    eRss_switch,  
    eRss_lock,  
    eRss_unlock,  
    eRss_query  
} eRss;
```

### 1 – eRss\_switch

This command is sent to the RSS when an operator has pressed and released an RCU switch to either the A or B position. It carries the tSwitch payload. The server responds with an array of bytes containing the status of all 16 switches.

### 2- eRss\_lock

This command is sent to the RSS when the operator has held an RCU switch in either the A or B position for more than 3 seconds. Upon receipt of this command the RSS will lock the selected card. This command carries tSwitch as its payload. The server responds with an array of bytes containing the status of all 16 switches.

### 3 – eRss\_unlock

This command is sent to the RSS when the operator has pressed and released the switch of a locked card to either the A or B position. Upon receipt of this command the RSS will unlock the selected card. This command carries tSwitch as its payload. The server responds with an array of bytes containing the status of all 16 switches.

### 4 – eRss\_query

This command is sent to the RSS every X seconds to keep the RCU in sync with the RSS

## 5.6. eCmnd\_rcu

```
typedef enum {  
    eRcu_null,  
    eRcu_statusUpdate,  
    eRcu_frontPanel,  
    eRcu_rcu,  
    eRcu_autoSwitch,  
    eRcu_manualSwitch  
} eRcu;
```

### 1 – eRcu\_statusUpdate

This command is sent from the RSS controller to the RCU whenever one of the following changes

1. Control is changed to NMO
2. Control is changed to Front Panel
3. Control id changed to RCU
4. A switch changes position regardless of the cause (manual/automatic).
5. A switch is locked.

The command carries the TSwitchStatus structure as its payload.

---

## 6. Payloads

---

### 6.1. TChangeRelay

```
typedef struct{
    unsigned char relay;
    unsigned char state;
}TChangeRelay;
```

Where relay is the number of the relay to be affected-1 (i.e. 0 for relay 1 and 1 for relay 2)and state sets the state of the relay, 1=Energize 2=Relax.

### 6.2. TChangeRelays

```
typedef struct{
    unsigned char relayStates[32];
}TChangeRelays;
```

Where relayStates is an array of relay states as defined below:

```
#define NO_CHANGE 0
#define ENERGIZE 1
#define RELAX 2
```

This payload is supported by devices that support the DxP protocol with 2-32 controllable relays.

### 6.3. TPulseRelay

```
typedef struct {
    unsigned char relay;        //the relay to be pulsed
    unsigned char state;       //the state to pulse
    uint16 pulseWidth;        //the pulse width in seconds
}TPulseRelay;
```

Where relay is the number of the relay to be affected, state is the state to pulse, and pulseWidth is the time to pulse for in seconds.

### 6.4. TSwitch

```
typedef struct {
    unsigned char card;        //the card to be switched
    unsigned char pos;        //position to put the card in
} TSwitch;
```

Where card is the number of the card to be switched (1-16) and post is the position to place the card in (0=A 1=B).

## 6.1. TSwitchStatus

```
typedef struct {  
    unsigned char status[16];    //The status of all 16 cards  
}  
#define SWITCH_STATUS_A        0x00  
#define SWITCH_STATUS_B        0x01  
#define SWITCH_STATUS_LOCK     0x02  
#define SWITCH_STATUS_UNLOCK   0x00  
#define SWITCH_STATUS_NMO      0x04  
#define SWITCH_STATUS_RCU      0x08  
#define SWITCH_STATUS_FRONTANEL 0x10  
#define SWITCH_STATUS_NOCARD   0xFF
```

Where status is an array of bit fields used to reflect the status of each switch in the RSS nest. Slots without a card will be SWITCH\_STATUS\_NOCARD



### Technical Support

tech@dataprobe.com  
dataprobe.com/support

201-934-5111  
201-934-9944