# EDC SPEAKERS NETWORK API

This document summarizes the Ethernet protocol and commands relevant to control systems requiring access to monitoring, status and operational parameters.

The information contained is commercial-in-confidence.

Revision Control

| Date | Notes |
|---|---|
| 23rd June 2018 | Initial Release |
| 14th January 2019 | Added POBJ Packet Information<br>Added Section on Mix Block Control |
| | |
| 28th December 2019 | Large Changes – All Network Packets and GOBJ and POBJ blocks added. |
| | |

# CONTENTS

# NETWORKING

This section contains a guide to the UDP packets supported in the EDC speakers, as well as procedures to discover and configure the IP settings of the units.

## PROTOCOL INFORMATION

The communications protocol used with the hardware devices is UDP. The user data of the UDP packet consists of a header followed by the data.

The EDC speaker hardware receives data on UDP port **12128** and transmits data by default on UDP port **12129**.

The header for each packet is 10 bytes in length and consists of the following:

| Field | Size | Description |
|---|---|---|
| Protocol ID | 2 bytes | 0x5E65 EDC Column/Square, 4-Beam<br>0x5E66 EDC Subwoofer Series<br>0x5EFF Target all SE Devices on the Network |
| Sub Type | 2 bytes | Set to 0x0001 for packets from a master<br>Set to 0x0100 for packets from a slave |
| Sequence Number | 2 bytes | Any value from 0x0001 to 0xFFFF |
| Reserved | 2 bytes | Set to 0x0000 |
| Chunk Length | 2 bytes | Length in bytes of data after this header (does not include the length of the header) |

The `ProtocolId` field differentiates this packet from others with a similar format used by other Secure Electronics products. Its value is specific and unique to these devices. Devices will discard any packets that do not have the correct protocol ID. It should be noted that 0x5EFF is a broadcast protocol ID and all devices will attempt to process these packets.

The `SubType` field should be set to 0x0001 for all packets originating from a controller (or master). A master is typically a software GUI, iPad/iPhone application or Crestron/AMX controller. The hardware units, in their reply packets, will have this field set to 0x0100. Masters should discard any packets received that do not have this field set to 0x0100.

The `SequenceNumber` is a field that can be used by masters to place a unique ID on the packet. The hardware units will reply to a particular command with a matching SequenceNumber such that a master can match up a device reply to a previously sent packet. The sequence number can be any 16-bit number except 0. Typically a good way to use the sequence number is to have a static class that provides the number and increments it each time a number is provided. Using sequence numbers allows for several packets to be sent at the same time, and the responses back to the master then matched up with the correct sequence to ensure all packets have been answered.

The `Reserved` field should be set to 0x0000. Note that for **advanced** applications, this field can contain a 16-bit number that the speakers will use as a transmit port for the UDP packets (rather than 12129).

Lastly, the `ChunkLength` field contains the length of ALL SUBSEQUENT data in the packet (in bytes). Data chunks following the header are restricted to 272 bytes.

## CRC8 CALCULATIONS

To provide a simple data validation check, a CRC8 has been implemented. This is done with a look-up table. It is deliberately simple to take some processing load of the small microcontroller in the hardware units. There are two types that can be calculated – Type 1 and 2 – and both are computed the same way, with different start seeds and end seeds.

A static C# class implementation is provided. This should be easily portable to other coding environments. The functions GetCrc8Type1FromDataList and GetCrc8Type2FromDataList contain the method to create the CRC8 values from a byte array (first call the StartCrc8TypeX, then UpdateCrc8TypeX with data bytes, then EndCrc8TypeX).

```
public static class Crc8
{
    private const byte CRC8_SEED1 = 0x55;
    private const byte CRC8_UPDATER1 = 0x00;
    private const byte CRC8_MASK1 = 0x80;

    private const byte CRC8_SEED2 = 0xAA;
    private const byte CRC8_UPDATER2 = 0x18;
    private const byte CRC8_MASK2 = 0x01;

    private static byte[] _crc8Lookup = new byte[256]
    {
         0,  94, 188, 226,  97,  63, 221, 131, 194, 156, 126,  32, 163, 253,  31,  65,
       157, 195,  33, 127, 252, 162,  64,  30,  95,   1, 227, 189,  62,  96, 130, 220,
        35, 125, 159, 193,  66,  28, 254, 160, 225, 191,  93,   3, 128, 222,  60,  98,
       190, 224,   2,  92, 223, 129,  99,  61, 124,  34, 192, 158,  29,  67, 161, 255,
        70,  24, 250, 164,  39, 121, 155, 197, 132, 218,  56, 102, 229, 187,  89,   7,
       219, 133, 103,  57, 186, 228,   6,  88,  25,  71, 165, 251, 120,  38, 196, 154,
       101,  59, 217, 135,   4,  90, 184, 230, 167, 249,  27,  69, 198, 152, 122,  36,
       248, 166,  68,  26, 153, 199,  37, 123,  58, 100, 134, 216,  91,   5, 231, 185,
       140, 210,  48, 110, 237, 179,  81,  15,  78,  16, 242, 172,  47, 113, 147, 205,
        17,  79, 173, 243, 112,  46, 204, 146, 211, 141, 111,  49, 178, 236,  14,  80,
       175, 241,  19,  77, 206, 144, 114,  44, 109,  51, 209, 143,  12,  82, 176, 238,
        50, 108, 142, 208,  83,  13, 239, 177, 240, 174,  76,  18, 145, 207,  45, 115,
       202, 148, 118,  40, 171, 245,  23,  73,   8,  86, 180, 234, 105,  55, 213, 139,
        87,   9, 235, 181,  54, 104, 138, 212, 149, 203,  41, 119, 244, 170,  72,  22,
       233, 183,  85,  11, 136, 214,  52, 106,  43, 117, 151, 201,  74,  20, 246, 168,
       116,  42, 200, 150,  21,  75, 169, 247, 182, 232,  10,  84, 215, 137, 107,  53,
    };

    public static byte StartCrc8Type1()
    {
        return CRC8_SEED1;
    }

    public static byte StartCrc8Type2()
    {
        return CRC8_SEED2;
    }

    public static byte UpdateCrc8Type1(byte crc, byte data)
    {
        return (_crc8Lookup[crc ^ ((byte)(data + CRC8_UPDATER1))]);
    }

    public static byte UpdateCrc8Type1(byte crc, byte[] data)
    {
        byte newCrc = crc;

        foreach (byte b in data)
        {
            newCrc = UpdateCrc8Type1(newCrc, b);
        }

        return newCrc;
```

```
    }

    public static byte UpdateCrc8Type2(byte crc, byte data)
    {
        return (_crc8Lookup[crc ^ ((byte)(data + CRC8_UPDATER2))]);
    }

    public static byte UpdateCrc8Type2(byte crc, byte[] data)
    {
        byte newCrc = crc;

        foreach (byte b in data)
        {
            newCrc = UpdateCrc8Type2(newCrc, b);
        }

        return newCrc;
    }

    public static byte EndCrc8Type1(byte crc)
    {
        return ((byte)(crc ^ CRC8_MASK1));
    }

    public static byte EndCrc8Type2(byte crc)
    {
        return ((byte)(crc ^ CRC8_MASK2));
    }

    /// <summary>
    /// Returns a CRC-8 (type 1) on a list of bytes.
    /// </summary>
    /// <param name="dataList"></param>
    /// <returns></returns>
    public static byte GetCrc8Type1FromDataList(List<byte> dataList)
    {
        byte crc = Crc8.StartCrc8Type1();

        for (int i = 0; i < dataList.Count; i++)
        {
            crc = Crc8.UpdateCrc8Type1(crc, dataList[i]);
        }
        crc = Crc8.EndCrc8Type1(crc);

        return crc;
    }

    /// <summary>
    /// Returns a CRC-8 (type 1) on a list of bytes.
    /// </summary>
    /// <param name="dataList"></param>
    /// <returns></returns>
    public static byte GetCrc8Type2FromDataList(List<byte> dataList)
    {
        byte crc = Crc8.StartCrc8Type2();

        for (int i = 0; i < dataList.Count; i++)
        {
            crc = Crc8.UpdateCrc8Type2(crc, dataList[i]);
        }
        crc = Crc8.EndCrc8Type2(crc);

        return crc;
    }
}
```

## COMMAND PACKETS: ACKN, NACK AND BACK

Description:

These three packets are common replies to requests sent from a master. ACKN means the request was acknowledged and processed, whereas a NACK means it was not acknowledged and a NACK code is sent in reply. The BACK packet is a broadcast ACKN. As a data field it contains the IP address of the speaker that sends it, such that multiple masters on a network can determine if the source of the BACK is relevant to their operations.

Packet Structure (ACKN):

| Packet Field | Size | Description |
|---|---|---|
| SE Master Header | 10 bytes | Standard Secure Electronics UDP header (sent from slave) |
| Command | 4 bytes | ACKN (ASCII 0x0x41434B4E) |

Packet Structure (NACK):

| Packet Field | Size | Description |
|---|---|---|
| SE Master Header | 10 bytes | Standard Secure Electronics UDP header (sent from slave) |
| Command | 4 bytes | NACK (ASCII 0x4E41434B) |
| NACK Code | 4 bytes | 32-bit NACK code value (big-endian). Packet specific. |

NACK codes will be explained as required further in this document in relation to return responses from packets.

Packet Structure (BACK):

| Packet Field | Size | Description |
|---|---|---|
| SE Master Header | 10 bytes | Standard Secure Electronics UDP header (sent from slave) |
| Command | 4 bytes | BACK (ASCII 0x4241434B) |
| Sending IP Address | 4 bytes | The IP address of the unit sending the BACK packet. |

## COMMAND PACKET: PING (PING HARDWARE)

Description:

This command is used to discover information about a device (or devices) that are present on the network. It includes an option to force the hardware to return a broadcast (instead of unicast) UDP response packet for use in detecting hardware devices that may exist on the same physical network but on a different subnet.

Packet Structure (unicast reply requested):

| Packet Field | Size | Description |
|---|---|---|
| SE Master Header | 10 bytes | Standard Secure Electronics UDP header (sent from master) |
| Command | 4 bytes | PING (ASCII 0x50494E47) |

Packet Structure (broadcast reply requested):

| Packet Field | Size | Description |
|---|---|---|
| SE Master Header | 10 bytes | Standard Secure Electronics UDP header (sent from master) |
| Command | 4 bytes | PING (ASCII 0x50494E47) |
| Broadcast Flag | 1 byte | Set to 0x11 (all other values will result in a unicast reply) |

The reply to a PING command when a unicast reply is requested is a WHAT command packet.

The reply to a PING command when a broadcast reply is requested is an EXPL command packet.

## COMMAND PACKET: WHAT (HARDWARE ID REPLY)

Description:

This packet is received from a hardware unit in response to a PING packet (in a non-broadcast request). It contains device information including firmware version, MAC address and IP details as well as any build-state considerations.

Packet Structure:

| Packet Field | Size | Description |
|---|---|---|
| SE Slave Header | 10 bytes | Standard Secure Electronics UDP header (sent from slave) |
| Command | 4 bytes | WHAT (ASCII 0x57484154) |
| Firmware Info (Major) | 1 byte | Value ranges from 0-255 |
| Firmware Info (Minor) | 1 byte | Value ranges from 0-255 |
| Firmware Info (Rev) | 2 bytes | Value ranges from 0-65535 |
| MAC Address | 6 bytes | Network MAC address of the hardware unit |
| IP Address | 4 bytes | Network IP address (IPv4) of the hardware unit |
| Subnet Mask | 4 bytes | Subnet mask IP address (IPv4) of the hardware unit |
| Default Gateway | 4 bytes | Default gateway address (IPv4) of the hardware unit |
| DHCP Enabled Flag | 1 byte | 0 if the hardware unit has its DHCP client disabled<br>1 if the hardware unit has its DHCP client enabled |
| Custom Mode Info | 1 byte | Data byte used by the hardware unit for identification purposes |
| Lock Out Flag | 1 byte | 0 if the hardware unit is free to communicate with the master<br>1 if this master is locked out from the hardware unit |
| Device Name | 32 bytes | The manufacturer/product ID string (ASCII string) |
| User Hardware Name | 81 bytes | 81 byte length UTF-8 encoded string. Strings not requiring the full length are padded with bytes set to 0x00. |

Note that the Custom Mode Info field varies depending on the type of hardware unit.

For EDC speakers this usually reflects the subtype of a speaker – for instance, in the column/square variants, it would indicate a small, medium or large column, or a square:

1 = Small Column (SC-30)

2 = Medium Column (MC-60)

3 = Large Column (LC-90)

4 = Square (SQ-90)

For EDC subs, it would indicate an SS-3, SS-9, or SS-1:

1 = SS-3

2 = SS-9

3 = SS-1

## COMMAND PACKET: EXPL (HARDWARE ID REPLY)

Description:

This packet is received from a hardware unit in response to a PING packet (in a broadcast request). It contains device information including firmware version, MAC address and IP details as well as any build-state considerations. It differs from the WHAT packet by including the IP address of the original requesting master in the data chunk, such that a master can determine if the reply was sent to it (which would normally be impossible to tell from the standard UDP header due to the fact that this packet is broadcast).

Packet Structure:

| Packet Field | Size | Description |
|---|---|---|
| SE Slave Header | 10 bytes | Standard Secure Electronics UDP header (sent from slave) |
| Command | 4 bytes | EXPL (ASCII 0x4558504C) |
| Firmware Info (Major) | 1 byte | Value ranges from 0-255 |
| Firmware Info (Minor) | 1 byte | Value ranges from 0-255 |
| Firmware Info (Rev) | 2 bytes | Value ranges from 0-65535 |
| MAC Address | 6 bytes | Network MAC address of the hardware unit |
| IP Address | 4 bytes | Network IP address (IPv4) of the hardware unit |
| Subnet Mask | 4 bytes | Subnet mask IP address (IPv4) of the hardware unit |
| Default Gateway | 4 bytes | Default gateway address (IPv4) of the hardware unit |
| DHCP Enabled Flag | 1 byte | 0 if the hardware unit has its DHCP client disabled<br>1 if the hardware unit has its DHCP client enabled |
| Custom Mode Info | 1 byte | Data byte used by the hardware unit for identification purposes |
| Lock Out Flag | 1 byte | 0 if the hardware unit is free to communicate with the master<br>1 if this master is locked out from the hardware unit |
| Device Name | 32 bytes | The user defined name of the hardware unit (ASCII string) |
| Target IP Address | 4 bytes | Network IP address (IPv4) of the master that this packet has been specifically sent at |
| User Hardware Name | 81 bytes | 81 byte length UTF-8 encoded string. Strings not requiring the full length are padded with bytes set to 0x00. |

Note that the Custom Mode Info field varies depending on the type of hardware unit. See the WHAT packet definition for a list of what this field can contain.

## COMMAND PACKET: BOOT (HARDWARE RESTART)

Description:

This packet is used to command a hardware unit to perform a restart. Note that as this packet can be sent broadcast (for instance when committing IP changes to a device on another Ethernet subnet) it contains a target IP address. Hardware units will compare this IP address with their own IP address and only restart if it matches.

Packet Structure:

| Packet Field | Size | Description |
|---|---|---|
| SE Master Header | 10 bytes | Standard Secure Electronics UDP header (sent from master) |
| Command | 4 bytes | BOOT (ASCII 0x424F4F54) |
| Target IP Address | 4 bytes | The IP address of the unit to be restarted. |

## COMMAND PACKET: SMON (SIGNAL MONITORING)

Description:

This packet is sent by a master to instruct the hardware unit to start (or maintain) sending broadcast monitoring packets. These packets contain data that represents the current signal level at various points of the audio structure in the device.

Packet Structure (master sent):

| Packet Field | Size | Description |
|---|---|---|
| SE Master Header | 10 bytes | Standard Secure Electronics UDP header (sent from master) |
| Command | 4 bytes | SMON (ASCII 0x534D4F4E) |
| Sub Command | 1 byte | 0x11 (Enable broadcast monitoring) |

The hardware unit will then reply with an ACKN packet to confirm the command was received:

| Packet Field | Size | Description |
|---|---|---|
| SE Slave Header | 10 bytes | Standard Secure Electronics UDP header (sent from slave) |
| Command | 4 bytes | ACKN (ASCII 0x41434B4E) |

Once triggered the monitoring packets will be sent from the hardware automatically for 13 seconds. Upon receiving another SMON packet as outlined above, this counter will restart and packets will continue.

These broadcast packets containing the signal monitoring data have the following structure:

| Packet Field | Size | Description |
|---|---|---|
| SE Slave Header | 10 bytes | Standard Secure Electronics UDP header (sent from slave) |
| Command | 4 bytes | SMON (ASCII 0x534D4F4E) |
| Monitoring Data | <x bytes> | Monitoring data<br>Specific to hardware |
| Sending IP Address | 4 bytes | The IP address of the hardware sending this reply packet |

The length and contents of the monitoring data varies based on the speaker type. The details of the reply packet are provided in the section exclusive to the COLUMN/SQUARE or SUB speaker variants.

Monitoring data at various points in the signal processing is returned as a 32-bit integer. To convert this integer value to dB, apply the following equation:

dB = 20*log10(x/0x01000000)

where x is the 32-bit value read from the packet.

Note that a value of 0x01000000 equates to 0dBFS.

## COMMAND PACKET: STAT (STATUS REQUEST)

Description:

This packet is sent by a master to obtain the hardware status of the device. A basic summary or more detailed packet can be requested.

Packet Structure for a basic status request (master sent):

| Packet Field | Size | Description |
|---|---|---|
| SE Master Header | 10 bytes | Standard Secure Electronics UDP header (sent from master) |
| Command | 4 bytes | STAT (ASCII 0x53544154) |

Packet Structure for a detailed status request (master sent):

| Packet Field | Size | Description |
|---|---|---|
| SE Master Header | 10 bytes | Standard Secure Electronics UDP header (sent from master) |
| Command | 4 bytes | STAT (ASCII 0x53544154) |
| Sub Command | 1 byte | 0x22 |

The reply from the hardware will be:

| Packet Field | Size | Description |
|---|---|---|
| SE Slave Header | 10 bytes | Standard Secure Electronics UDP header (sent from slave) |
| Command | 4 bytes | STAT (ASCII 0x53544154) |
| Status Data | <x bytes> | Status data<br>Specific to hardware |

The length and contents of the monitoring data varies based on the speaker type. The details of the reply packet are provided in the section exclusive to the COLUMN/SQUARE or SUB speaker variants.

## COMMAND PACKET: FCMD (NV READING/WRITING)

Description:

This packet is sent by a master to obtain the contents of NV memory in the hardware. It can be used to both write and read data, and poll if the EEPROM is busy.

Packet Structure for an EEPROM write/read process:

| Packet Field | Size | Description |
|---|---|---|
| SE Master Header | 10 bytes | Standard Secure Electronics UDP header (sent from master) |
| Command | 4 bytes | FCMD (ASCII 0x46434D44) |
| FCMD Sub Command | 1 byte | FCMD Packet Type<br>0x12 = EEPROM Read<br>0x35 = EEPROM Write |
| Start Address | 4 bytes | Start Address to read/write (32-bit number, big-endian) |
| Length | 2 bytes | Length to read (1 to 256 bytes, big-endian) |

Note that as EEPROMs can be busy if writing a lengthy data stream, a poll command can be used to determine if it is busy. Alternatively, for simpler programming, setting the highest bit of the address (i.e. ORing 0x80000000 with the address) will force a write/read to wait for the EEPROM to become available.

Packet Structure for an EEPROM poll process:

| Packet Field | Size | Description |
|---|---|---|
| SE Master Header | 10 bytes | Standard Secure Electronics UDP header (sent from master) |
| Command | 4 bytes | FCMD (ASCII 0x46434D44) |
| Sub Command | 1 byte | FCMD Packet Type<br>0x02 = EEPROM Poll |

Note that if a NACK code of 0x00020001 is received, the actual sub-command of FCMD packet type was invalid. Use only the sub-command values as indicated in the packet structures.

For a WRITE process, an ACKN will be sent if successful. If not successful, the following NACK codes may be received to indicate the issue:

0x00020006 – Length of write not between 1 and 256.

0x00020007 – Write does not start on a page boundary. As the page size is 256, addresses should start on multiples of 256.

0x00020008 – A fault occurred in the write unexpectedly. Potentially a hardware issue.

0x00020002 – The EEPROM was not ready to be written to.

For a POLL process, an ACKN will be sent if successful. If not successful, a NACK code of 0x00020002 will be sent back.

For a READ process, an FCMD packet reply will be received if successful. If not successful, the following NACK codes may be received to indicate the issue:

0x00020006 – Length of read not between 1 and 256.

0x00020009 – A fault occurred in the write unexpectedly. Potentially a hardware issue.

0x00020002 – The EEPROM was not ready to be read from.

An FCMD reply packet to a successful read will be structured as:

| SE Master Header | 10 bytes | Standard Secure Electronics UDP header (sent from master) |
|---|---|---|
| Command | 4 bytes | FCMD (ASCII 0x46434D44) |
| FCMD Sub Command | 1 byte | FCMD Packet Type<br>0x12 = EEPROM Read |
| Start Address | 4 bytes | Start Address of the read (32-bit number, big-endian) |
| Length | 2 bytes | Length to read (1 to 256 bytes, big-endian) |
| Read Data | <x> bytes | Data read from the EEPROM of user defined length |

For both EDC column/square and sub speaker variants, the FCMD packets can be used to backup/restore operating parameters.

Reading from address 0x00000800 to address 0x00020000 will provide a backup of operating memory and writing back to this address range will restore the memory.

It is suggested that after a write process, that the speaker be restarted to engage the restored memory (alternatively, a PSET packet with an ID of 0xFE for force reload can be sent).

## COMMAND PACKET: PSET (PRESET RECALL)

Description:

This packet is sent by a master to recall a preset (if relevant) in a hardware unit. It can also be used to force a reload of the live preset (default parameter space). In the case of the EDC speaker variants, this "forced reload" will be the main use of this packet. For example, when a long FIR filter is written directly to the EEPROM memory, the forced reload will then transfer the EEPROM parameter data into the running data of the DSP.

Packet Structure:

| Packet Field | Size | Description |
|---|---|---|
| SE Master Header | 10 bytes | Standard Secure Electronics UDP header (sent from master) |
| Command | 4 bytes | PSET (ASCII 0x50534554) |
| Preset ID #1 | 1 byte | 0x01 – Max Num Stored Presets (for preset recall)<br>0xFE (for forced reload) |
| Preset ID #2 | 1 byte | Must match byte Preset ID #1 |
| Reserved | 1 byte | Set to 0x00 |

If a preset recall is attempted with the hardware unit and fails, the following NACK codes may be returned:

| | |
|---|---|
| NACK_CODE_PSET_INCORRECT_STATE | 0x00090001 |
| NACK_CODE_PSET_BAD_PRESET_ID | 0x00090002 |
| NACK_CODE_PSET_BAD_PRESET_CMD | 0x00090003 |
| NACK_CODE_PSET_BAD_NVOP_SUBCMD | 0x00090004 |
| NACK_CODE_PSET_BAD_NVOP_LENGTH | 0x00090005 |
| NACK_CODE_PSET_BAD_NVOP_ADDR | 0x00090006 |
| NACK_CODE_PSET_EEPROM_OP_FAILED | 0x00090007 |
| NACK_CODE_PSET_PRESETS_NOT_SUPPORTED | 0x00090008 |

Note that the EDC speakers will only support the forced reload command – do not send a preset recall that is not of ID 0xFE.

## COMMAND PACKET: POBJ (PRESET OBJECT) - WRITE

Description:

This packet is sent by a master to write a preset object within the hardware.

Packet Structure, Write (master sent):

| Packet Field | Size | Description |
|---|---|---|
| SE Master Header | 10 bytes | Standard Secure Electronics UDP header (sent from master) |
| Command | 4 bytes | POBJ (ASCII 0x504F424A) |
| Read Request | 1 byte | 0x00 (Write) |
| Preset ID | 1 byte | 0x00 = Live Preset<br>0x01 and upwards = Stored Preset |
| Preset Object Number | 2 bytes | 16-bit number of the Object ID to write |
| NV Commit Flag | 1 byte | 0x00 = Do not store into non-volatile memory<br>0x01 = Store into non-volatile memory |
| Object Data | <x> bytes | Data for the object (specific to the type of object) |

The hardware unit will then reply with an ACKN packet to confirm the command was received. If a NACK is received, an issue occurred. Error codes for a NACK may be:

| | |
|---|---|
| NACK_CODE_POBJ_CORRUPT_PACKET | 0x00040001 |
| NACK_CODE_POBJ_BAD_OBJECT_ID | 0x00040002 |
| NACK_CODE_POBJ_NV_OP_FAILED | 0x00040003 |
| NACK_CODE_POBJ_RAM_OP_FAILED | 0x00040004 |
| NACK_CODE_POBJ_INCORRECT_STATE | 0x00040005 |
| NACK_CODE_POBJ_BAD_PRESET_ID | 0x00040006 |
| NACK_CODE_POBJ_DUMP_ONLY_OBJECT | 0x00040007 |

The object numbers and contents of the data varies based on the speaker type.

## COMMAND PACKET: POBJ (PRESET OBJECT) - READ

Description:

This packet is sent by a master to read a preset object from the hardware.

Packet Structure, Read (master sent):

| Packet Field | Size | Description |
|---|---|---|
| SE Master Header | 10 bytes | Standard Secure Electronics UDP header (sent from master) |
| Command | 4 bytes | POBJ (ASCII 0x504F424A) |
| Read Request | 1 byte | 0x01 (Read) |
| Preset ID | 1 byte | 0x00 = Live Preset<br>0x01 and upwards = Stored Preset |
| Preset Object Number | 2 bytes | 16-bit number of the Object ID to read |

The hardware unit will then reply with a POBJ packet to confirm the command was received (if the preset ID and object number are valid – if not, a NACK will be returned).

Packet Structure, Reply (hardware sent):

| Packet Field | Size | Description |
|---|---|---|
| SE Master Header | 10 bytes | Standard Secure Electronics UDP header (sent from slave) |
| Command | 4 bytes | POBJ (ASCII 0x504F424A) |
| Read Request | 1 byte | 0x01 (Read) |
| Preset ID | 1 byte | 0x00 = Live Preset<br>0x01 and upwards = Stored Preset |
| Preset Object Number | 2 bytes | 16-bit number of the Object ID being returned |
| Reply Data | <x> bytes | Data for the object (specific to the type of object) |

# COMMAND PACKET: GOBJ (GLOBAL OBJECT) - WRITE

Description:

This packet is sent by a master to write a global object within the hardware.

Packet Structure, Write (master sent):

| Packet Field | Size | Description |
|---|---|---|
| SE Master Header | 10 bytes | Standard Secure Electronics UDP header (sent from master) |
| Command | 4 bytes | GOBJ (ASCII 0x474F424A) |
| Read Request | 1 byte | 0x00 (Write) |
| Global Object Number | 2 bytes | 16-bit number of the Object ID to write |
| NV Commit Flag | 1 byte | 0x00 = Do not store into non-volatile memory<br>0x01 = Store into non-volatile memory |
| Object Data | <x> bytes | Data for the object (specific to the type of object) |

The hardware unit will then reply with an ACKN packet to confirm the command was received. If a NACK is received, an issue occurred. Error codes for a NACK may be:

| | |
|---|---|
| NACK_CODE_GOBJ_CORRUPT_PACKET | 0x00030001 |
| NACK_CODE_GOBJ_BAD_OBJECT_ID | 0x00030002 |
| NACK_CODE_GOBJ_NV_OP_FAILED | 0x00030003 |
| NACK_CODE_GOBJ_RAM_OP_FAILED | 0x00030004 |
| NACK_CODE_GOBJ_INCORRECT_STATE | 0x00030005 |

The object numbers and contents of the data varies based on the speaker type.

## COMMAND PACKET: GOBJ (GLOBAL OBJECT) - READ

Description:

This packet is sent by a master to read a global object from the hardware.

Packet Structure, Read (master sent):

| Packet Field | Size | Description |
|---|---|---|
| SE Master Header | 10 bytes | Standard Secure Electronics UDP header (sent from master) |
| Command | 4 bytes | GOBJ (ASCII 0x474F424A) |
| Read Request | 1 byte | 0x01 (Read) |
| Global Object Number | 2 bytes | 16-bit number of the Object ID to read |

The hardware unit will then reply with a GOBJ packet to confirm the command was received (if the object number is valid – if not, a NACK will be returned).

Packet Structure, Reply (hardware sent):

| Packet Field | Size | Description |
|---|---|---|
| SE Master Header | 10 bytes | Standard Secure Electronics UDP header (sent from slave) |
| Command | 4 bytes | GOBJ (ASCII 0x474F424A) |
| Read Request | 1 byte | 0x01 (Read) |
| Global Object Number | 2 bytes | 16-bit number of the Object ID being returned |
| Reply Data | <x> bytes | Data for the object (specific to the type of object) |

## COMMAND PACKET: NVCF (NON-VOLATILE CONFIGURATION) - WRITE

Description:

This packet is used by the master to write various configuration settings to the hardware device. Typically, these include factory settings (not listed in this document) as well as IP settings and hardware names.

Packet Structure, Write (master sent):

| Packet Field | Size | Description |
|---|---|---|
| SE Master Header | 10 bytes | Standard Secure Electronics UDP header (sent from master) |
| Command | 4 bytes | NVCF (ASCII 0x4E564346) |
| Read Request | 1 byte | 0x00 (Write) |
| Item ID | 1 byte | 0x03 – IP Address<br>0x04 – Subnet Mask<br>0x05 – Default Gateway<br>0x07 – DHCP Enabled Flag<br>0x0C – User Hardware Name |
| Configuration Data | <x> bytes | Data for the item (specific to the item ID) |

If the NVCF packet is successful, an ACKN is returned. If unsuccessful, a NACK will be sent:

| | |
|---|---|
| NACK_CODE_NV_XCLCFG_CORRUPT_PACKET | 0x00010001 |
| NACK_CODE_NV_XCLCFG_UNKNOWN_ITEM | 0x00010002 |
| NACK_CODE_NV_XCLCFG_BAD_ITEM_LENGTH | 0x00010003 |
| NACK_CODE_NV_XCLCFG_ITEM_WRITE_FAILED | 0x00010004 |

A further description of the items is provided below:

0x03 – IP Address – length 4 bytes. The IP address for the hardware unit to use (big-endian, MSB first).

0x04 – Subnet Mask – length 4 bytes. The subnet mask for the hardware unit to use (big-endian, MSB first).

0x05 – Default Gateway – length 4 bytes. The gateway address for the hardware unit to use (big-endian, MSB first).

0x07 – DHCP Enabled – length 1 byte. If set to 0x00, static IP addressing is used as defined by the IP, Subnet and Gateway configuration parameters. If set to 0x01, the hardware will attempt to source IP details from a DHCP server on the network. If no DHCP server is found, an auto-IP address (169.254.xxx.xxx) will be assigned.

0x0C – User Hardware Name – length 81 bytes. Note! To provide localization, this is stored as UTF-8! The maximum length of this parameter cannot exceed 81 bytes, so when a character string is converted to UTF-8, ensure that the conversion to a byte array is within these bounds.

## COMMAND PACKET: NVCF (NON-VOLATILE CONFIGURATION) - READ

Description:

This packet is sent by a master to read configuration settings from the hardware device.

Packet Structure, Read (master sent):

| Packet Field | Size | Description |
|---|---|---|
| SE Master Header | 10 bytes | Standard Secure Electronics UDP header (sent from master) |
| Command | 4 bytes | NVCF (ASCII 0x4E564346) |
| Read Request | 1 byte | 0x01 (Read) |
| Item ID | 1 byte | 0x03 – IP Address<br>0x04 – Subnet Mask<br>0x05 – Default Gateway<br>0x07 – DHCP Enabled Flag<br>0x0C – User Hardware Name |

The hardware unit will then reply with a NVCF packet to confirm the command was received (if the item ID is valid – if not, a NACK will be returned).

Packet Structure, Reply (hardware sent):

| Packet Field | Size | Description |
|---|---|---|
| SE Master Header | 10 bytes | Standard Secure Electronics UDP header (sent from slave) |
| Command | 4 bytes | NVCF (ASCII 0x4E564346) |
| Read Request | 1 byte | 0x01 (Write) |
| Item ID | 1 byte | 0x03 – IP Address<br>0x04 – Subnet Mask<br>0x05 – Default Gateway<br>0x07 – DHCP Enabled Flag<br>0x0C – User Hardware Name |
| Configuration Data | <x> bytes | Data for the item (specific to the item ID) |

## COMMAND PACKET: FIPC (FULL IP CONFIG)

Description:

This packet is used by the master to write all relevant IP settings and the user hardware name to a hardware device. It can be broadcast, as it contains a target IP in the payload of the packet that devices will check against to ensure the data is for itself. It is useful in conjunction with the BOOT packet to reconfigure a device on the network that is on a different subnet.

Packet Structure:

| Packet Field | Size | Description |
|---|---|---|
| SE Master Header | 10 bytes | Standard Secure Electronics UDP header (sent from master) |
| Command | 4 bytes | FIPC (ASCII 0x46495043) |
| Target IP Address | 4 bytes | The IP address of the unit to be restarted. |
| IP Config Block | 138 bytes | IP Address – 4 bytes<br>IP Address CRC8_1 and CRC8_2 – 2 bytes<br>Subnet Mask – 4 bytes<br>Subnet Mask CRC8_1 and CRC8_2 – 2 bytes<br>Default Gateway – 4 bytes<br>Default Gateway CRC8_1 and CRC8_2 – 2 bytes<br>Device Name – 32 bytes (ASCII)<br>Device Name CRC8_1 and CRC8_2 – 2 bytes<br>DHCP Enabled Flag – 1 byte<br>DHCP Enabled CRC8_1 and CRC8_2 – 2 bytes<br>User Hardware Name – 81 bytes (UTF8)<br>User Hardware Name CRC8_1 and CRC8_2 – 2 bytes |

If the FIPC packet is successful, a BACK packet is returned. If unsuccessful, a NACK will be sent:

| | |
|---|---|
| NACK_CODE_NV_XCLCFG_CORRUPT_PACKET | 0x00010001 |
| NACK_CODE_NV_XCLCFG_UNKNOWN_ITEM | 0x00010002 |
| NACK_CODE_NV_XCLCFG_BAD_ITEM_LENGTH | 0x00010003 |
| NACK_CODE_NV_XCLCFG_ITEM_WRITE_FAILED | 0x00010004 |

Note that as this function is effectively copying a block of memory into the EEPROM of the hardware device, the master must calculate the CRC8 values (a type 1 and type 2 sequentially) for each item. Consult the CRC8 section of this document for information on those calculations. For example, the data bytes of the IP Address are run through the Type 1 and Type 2 CRC8 methods, with the values being appended after the IP Address. Then the Subnet Mask is appended in the same way, and so forth.

Typically, the device name should be written with the device name value read from a device in a returned EXPL packet. Typically, it would be a short string to describe the speaker, e.g. “EDC SC-30” or “EDC SQ-90”.

The user hardware name can be defined to anything by the user (with an 81-byte UTF8 length restriction).

## PROCEDURE: NETWORK DISCOVERY

The easiest way to discover units on a network is to send a broadcast PING command on to the network. All hardware units that receive this packet will reply with an EXPL command that provides information on the device's IP details, MAC address, ID name and firmware version.

As UDP delivery is not guaranteed, it is generally worth sending three or so PING commands within a second to maximize the chance of not missing any return packets.

The master can then select the hardware unit to communicate with (or allow the user to make this choice). Using a broadcast address of 255.255.255.255 will return all devices on the immediate network, but some of these devices will not be able to communicate if they are located on a different subnet.

If the user has a fixed IP address to communicate with, then the master can send the PING in non-broadcast mode to that specific address and wait and see if a WHAT command is received. Non-broadcast PING packets reply with WHAT commands instead of EXPL commands.

Note that a master can screen the hardware devices found based on firmware version, such that if a master has been programmed to suit a particular firmware version it will not communicate with a device of a different firmware version. In this situation the user should be instructed that this is the case.

In summary:

1) Master issues a broadcast PING packet.
2) Master waits for replies (EXPL packets) from remote hardware devices.
3) Repeat steps 1-2 as needed (say 2-3 times).
4) Master collates devices that replied.

# PROCEDURE: UPDATING IP ADDRESS DETAILS

To update IP Address details on a device, the best way is to use the information derived from an EXPL packet from a discovery process.

Changes can be made to these values and compiled into a FIPC packet. FIPC packets can be sent as broadcast (with a target IP address contained) meaning that devices on different subnets that usually cannot be reached can be reconfigured to be on the same subnet as the master device. Once the FIPC packet has been sent and a BACK packet received, the device can be restarted with a BOOT packet.

In summary:

1) Take an EXPL packet and gather IP data as required to assemble a FIPC packet (IP Address, Subnet Mask, Default Gateway, Name, DHCP Enabled and User Hardware Name).
2) Alter the required data fields.
3) Assemble the FIPC packet with the data fields, calculating the CRC8 Type 1 and 2 values as defined in the FIPC packet definition.
4) Send the FIPC packet as broadcast.
5) Wait for a BACK packet in response.
6) Send a BOOT packet as broadcast.
7) Wait for the BACK packet in response.
8) After the hardware device reboots (5-10 seconds) it will now be communicating with the new IP settings.

# DSP FIXED POINT MATHS

The DSP used in the EDC speakers is a fixed-point processor. This means that conversion is needed between stored parameters in master software and what is stored in the hardware device's memory. The DSP stores numbers in an 8.24 format.

The code provided below is a C# class used to convert between the fixed-point of the DSP and numbers more commonly used in software.

Porting this code to a different environment should be well supported due to the generic use of the standard Math library in C#.

```
public static class FixedPointConversion
{
    /// <summary>
    /// Converts a double value (in dBFS) into a gain value scaled
    /// into an 8.24 digital number.
    /// </summary>
    /// <param name="dbfs"></param>
    /// <returns></returns>
    public static UInt32 DecibelsToFixed24(double dbfs)
    {
        return FracToFixed24(Math.Pow(10, (dbfs / 20.0D)));
    }

    /// <summary>
    /// Converts an 8.24 digital number into a decibel value.
    /// </summary>
    /// <param name="vol"></param>
    /// <returns></returns>
    public static double Fixed24ToDecibels(UInt32 vol)
    {
        return 20.0D * Math.Log10(Fixed24ToFrac(vol));
    }

    /// <summary>
    /// Converts a fractional number (within the range of -128.0 to +128.0)
    /// into an 8.24 fixed point number.
    /// </summary>
    /// <param name="frac"></param>
    /// <returns></returns>
    public static UInt32 FracToFixed24(double frac)
    {
        return Convert.ToUInt32(frac * Math.Pow(2, 24));
    }

    /// <summary>
    /// Converts a fractional number (within the range of -128.0 to +128.0)
    /// into an 8.24 fixed point number (signed).
    /// </summary>
    /// <param name="frac"></param>
    /// <returns></returns>
    public static Int32 FracToSignedFixed24(double frac)
    {
        return Convert.ToInt32(frac * Math.Pow(2, 24));
    }

    /// <summary>
    /// Converts an 8.24 signed fixed point number into a fractional number.
    /// </summary>
    /// <param name="fp"></param>
```

```
    /// <returns></returns>
    public static double SignedFixed24ToFrac(Int32 fp)
    {
        return (Convert.ToDouble(fp) / Math.Pow(2, 24));
    }

    /// <summary>
    /// Converts an 8.24 fixed point number into a fractional number.
    /// </summary>
    /// <param name="fp"></param>
    /// <returns></returns>
    public static double Fixed24ToFrac(UInt32 fp)
    {
        return (Convert.ToDouble(fp) / Math.Pow(2, 24));
    }

    /// <summary>
    /// Converts a filter coefficient into an 8.24 digital number.
    /// </summary>
    /// <param name="coeff"></param>
    /// <returns></returns>
    public static UInt32 FilterCoeffToFixed24(double coeff)
    {
        return (UInt32)Convert.ToInt32(coeff * Math.Pow(2, 24));
    }

    /// <summary>
    /// Converts a frequency into an 8.24 digital number (assumes 48kHz sampling rate).
    /// </summary>
    /// <param name="freq"></param>
    /// <returns></returns>
    public static UInt32 FrequencyToFixed24_48k(double frequency, bool halfSampleRate)
    {
        double rate;

        if (halfSampleRate)
            rate = AudioSysConstants.SampleRate48k / 2.0D;
        else
            rate = AudioSysConstants.SampleRate48k;

        return Convert.ToUInt32((frequency / rate) * Math.Pow(2, 24));
    }

    /// <summary>
    /// Converts a frequency into an 8.24 digital number (assumes 48kHz sampling rate).
    /// </summary>
    /// <param name="freq"></param>
    /// <returns></returns>
    public static double Fixed24ToFrequency_48k(UInt32 frequency, bool halfSampleRate)
    {
        double temp = Convert.ToDouble(frequency / Math.Pow(2, 24));

        if (halfSampleRate)
        {
            double rate = AudioSysConstants.SampleRate48k / 2.0D;
            return (temp * rate);
        }
        else
        {
            double rate = AudioSysConstants.SampleRate48k;
            return (temp * rate);
        }
    }
}
```

# COMMON GLOBAL OBJECTS

This section outlines the types of Global Objects that are used in the hardware devices that are common across the range. These objects usually get compiled into a large consecutive block used in conjunction with preset objects to configure the hardware device.

With the current firmware versions of the EDC speakers, there are four common global objects:

- Boolean Flag
- Name String
- Standby Record
- DSP Volume

There can be global objects specific to a certain device and if applicable these would be defined with specific API implementation for that device rather than in this section.

For the global object definitions, the first table is how the objects need to be stored in memory for the firmware. This byte stream would form the “Object Data” section of a GOBJ packet (when writing, the master software builds this stream and sends it in the packet, when reading, the hardware unit fills this section of the packet).

The second table is typical of how master software would store the data. Generally, it is anticipated that the master software converts between how it stores the data and the firmware byte stream. Where necessary conversion routines are indicated in the following definitions.

For brevity this document does not include information on how to compile the firmware parameters. Code can be supplied from EDC Acoustics in C# format that can be ported to different operating systems. Each of the code files inherits a base class that supports a method called `GetGlobalObjectWriteRawData()`. Implementing this function will yield a list of bytes (including the tailing Crc8 Type 1) suitable for use with firmware.

# GLOBAL OBJECT: BOOLEAN FLAG

Description:

This global object represents a parameter that has a true/false (or on/off) condition.

Firmware Byte Stream:

| Item | Size | Description |
|---|---|---|
| Flag | UInt8 | Boolean Flag Value<br>0x00 = Off or False<br>0x01 = On or True |
| CRC8 Type 1 | UInt8 | The CRC8 (Type 1) computation of all preceding bytes. |

Master Software Typical Storage:

| Item | Type | Description |
|---|---|---|
| Flag | Boolean | Value Range:<br>False or True |

## GLOBAL OBJECT: NAME STRING

Description:

This global object represents a UTF8 string typically used as a name for an input, output or descriptor within the hardware device.

Firmware Byte Stream:

| Item | Size | Description |
|---|---|---|
| Name String | 31x UInt8 | Stored as UTF8 – must not exceed the 31 byte limit. |
| CRC8 Type 1 | UInt8 | The CRC8 (Type 1) computation of all preceding bytes. |

Master Software Typical Storage:

| Item | Type | Description |
|---|---|---|
| Name String | String | Value Range:<br>Unicode characters in an a string that does not exceed 31 bytes when converted to a UTF8 byte stream. |

## GLOBAL OBJECT: STANDBY RECORD

Description:

This global object represents a record of parameters grouped together which control the aspects of how a hardware device will go into automatic standby (low power) mode.

Firmware Byte Stream:

| Item | Size | Description |
|---|---|---|
| Auto Standby Enable | UInt8 | 0x00 = Auto Standby Disabled<br>0x01 = Auto Standby Enabled |
| Trigger Flags | UInt8 | Bitfield of inputs used to wake up the device from standby<br>Bit 0 (0x01): Analog Input<br>Bit 1 (0x02): AES Left Input<br>Bit 2 (0x04): AES Right Input<br>Note: For multiple wake-up input sources, the bits can be ORed together. |
| Timeout | UInt2 | UInt32 value of system ticks (10msec) indicating the amount of time required to go into auto standby if the trigger signals have not exceeded the threshold in this time.<br>e.g. For 100 seconds, the value is 10000 (decimal). |
| Threshold | UInt32 | Fixed 8.24 number indicating the value in dB at which if an input surpasses will automatically wake up the hardware device.<br>Using fixed point conversion code outlined in this document, call DecibelsToFixed24. |
| CRC8 Type 1 | UInt8 | The CRC8 (Type 1) computation of all preceding bytes. |

Master Software Typical Storage:

| Item | Type | Description |
|---|---|---|
| Auto Standby Enable | Boolean | Value Range:<br>False (disabled) or True (enabled) |
| Trigger Flags | Byte | Value Range:<br>0x00 (no triggers) to 0x07 (all triggers) inclusive of values |
| Timeout | Int | Value Range:<br>10 second to 43200 seconds |
| Threshold | Double | Value Range:<br>-10.0dBFS (high) to -80dBFS (very low) |

## GLOBAL OBJECT: DSP VOLUME

Description:

This global object represents a volume level for a control within the DSP of the hardware device.

Firmware Byte Stream:

| Item | Size | Description |
|---|---|---|
| Volume | UInt32 | Fixed 8.24 number indicating the value in dB of the volume control in DSP.<br>Using fixed point conversion code outlined in this document, call DecibelsToFixed24. |
| CRC8 Type 1 | UInt8 | The CRC8 (Type 1) computation of all preceding bytes. |

Master Software Typical Storage:

| Item | Type | Description |
|---|---|---|
| Volume | Double | Value Range:<br>Depends on what the global object controls, but typically will be in the range of -100.0dBFS to +12.0dBFS. |

# COMMON PRESET OBJECTS

This section outlines the types of Preset Objects that are used in the hardware devices that are common across the range. These objects usually get compiled into a large consecutive block used in conjunction with global objects to configure the hardware device.

The key difference between a preset object and a global object is that a preset can be stored several times in memory space of the hardware device and recalled, whereas a global object has only one definition. However, in the EDC speakers, only one preset is used making this difference less relevant.

With the current firmware versions of the EDC speakers, there are fifteen common preset objects:

- Boolean Flag
- Crossover
- Advanced Crossover
- Delay
- Dynamics
- Filter (Biquad)
- FIR Filter – 2111 coeffs
- FIR Filter – 511 coeffs
- Fractional Delay
- Generic Value (Double)
- Generic Value (Byte)
- Signal Invert
- Volume
- Look-up Table Volume (or Volume LUT)

There can be preset objects specific to a certain device and, if applicable, these would be defined with a specific API implementation for that device rather than in this section.

For the preset object definitions, the first table is how the objects need to be stored in memory for the firmware. This byte stream would form the “Object Data” section of a POBJ packet (when writing, the master software builds this stream and sends it in the packet, when reading, the hardware unit fills this section of the packet).

The second table is typical of how master software would store the data. Generally, it is anticipated that the master software converts between how it stores the data and the firmware byte stream. Where necessary conversion routines are indicated in the following definitions.

For brevity this document does not include information on how to compile the firmware parameters. Code can be supplied from EDC Acoustics in C# format that can be ported to different operating systems (the code uses standard Maths libraries). Each of the code files inherits a base class that supports a method called `GetPresetObjectWriteRawData()`. Implementing this function will yield a list of bytes (including the tailing Crc8 Type 1) suitable for use with firmware.

## PRESET OBJECT: BOOLEAN FLAG

Description:

This preset object represents a parameter that has a true/false (or on/off) condition.

Firmware Byte Stream:

| Item | Size | Description |
|---|---|---|
| Flag | UInt8 | Boolean Flag Value:<br>0x00 = Off or False<br>0x01 = On or True |
| CRC8 Type 1 | UInt8 | The CRC8 (Type 1) computation of all preceding bytes |

Master Software Typical Storage:

| Item | Type | Description |
|---|---|---|
| Flag | Boolean | Value Range:<br>False or True |

## PRESET OBJECT: CROSSOVER

Description:

This preset object represents an audio crossover filter with support for up to 4th order filter responses.

Firmware Byte Stream:

| Item | Size | Description |
|---|---|---|
| Filter Info Pack | UInt32 | *Highest Byte:*<br>Value is the Crossover Filter Type<br>(as per Enum in Software Table)<br>If pass type is high-pass, OR with 0x40<br>If bypassed, OR with 0x80<br>*Lower Three Bytes:*<br>Frequency of Filter x10 (e.g. 1000Hz = 10000) |
| Filter Stage 1 Coeffs | 5x UInt32 | Biquad Filter Coefficients, Stage 1:<br>B0, B1, B2,<br>-A1, -A2 (note that these coeffs are negated)<br>In order, normalized to A0 |
| Filter Stage 2 Coeffs | 5x UInt32 | Biquad Filter Coefficients, Stage 2:<br>B0, B1, B2,<br>-A1, -A2 (note that these coeffs are negated)<br>In order, normalized to A0 |
| CRC8 Type 1 | UInt8 | The CRC8 (Type 1) computation of all preceding bytes |

Master Software Typical Storage:

| Item | Type | Description |
|---|---|---|
| Crossover Filter Type | Enum | Values:<br>0 = Butterworth 1st<br>1 = Butterworth 2nd<br>2 = Butterworth 3rd<br>3 = Butterworth 4th<br>4 = Bessel 2nd<br>5 = Bessel 3rd<br>6 = Bessel 4th<br>7 = Linkwitz-Riley 2nd<br>8 = Linkwitz-Riley 4th |
| Crossover Pass Type | Enum | Values:<br>0 = Low Pass<br>1 = High Pass |
| Frequency | Double | Value Range: 20Hz – 20000Hz |
| Bypass Flag | Boolean | Value Range: True (bypassed) or False (engaged) |

## PRESET OBJECT: ADVANCED CROSSOVER

Description:

This preset object represents an audio advanced crossover filter with support for up to 8th order filter responses.

Firmware Byte Stream:

| Item | Size | Description |
|---|---|---|
| Filter Info Pack | UInt32 | *Highest Byte:*<br>Value is the AdvCrossover Filter Type<br>(as per Enum in Software Table)<br>If pass type is high-pass, OR with 0x40<br>If bypassed, OR with 0x80<br>*Lower Three Bytes:*<br>Frequency of Filter x10 (e.g. 1000Hz = 10000) |
| Filter Stage 1 Coeffs | 5x UInt32 | Biquad Filter Coefficients, Stage 1:<br>B0, B1, B2,<br>-A1, -A2 (note that these coeffs are negated)<br>In order, normalized to A0 |
| Filter Stage 2 Coeffs | 5x UInt32 | Biquad Filter Coefficients, Stage 2:<br>B0, B1, B2,<br>-A1, -A2 (note that these coeffs are negated)<br>In order, normalized to A0 |
| Filter Stage 3 Coeffs | 5x UInt32 | Biquad Filter Coefficients, Stage 3:<br>B0, B1, B2,<br>-A1, -A2 (note that these coeffs are negated)<br>In order, normalized to A0 |
| Filter Stage 4 Coeffs | 5x UInt32 | Biquad Filter Coefficients, Stage 4:<br>B0, B1, B2,<br>-A1, -A2 (note that these coeffs are negated)<br>In order, normalized to A0 |
| CRC8 Type 1 | UInt8 | The CRC8 (Type 1) computation of all preceding bytes |

Master Software Typical Storage:

| Item | Type | Description |
|---|---|---|
| AdvCrossover<br>Filter Type | Enum | Values:<br>0 = Butterworth 1st<br>1 = Butterworth 2nd<br>2 = Butterworth 3rd<br>3 = Butterworth 4th<br>4 = Butterworth 5th<br>5 = Butterworth 6th<br>6 = Butterworth 7th<br>7 = Butterworth 8th<br>8 = Bessel 2nd<br>9 = Bessel 3rd<br>10 = Bessel 4th<br>11 = Bessel 6th<br>12 = Bessel 8th<br>13 = Linkwitz-Riley 2nd<br>14 = Linkwitz-Riley 4th |

| | | |
|---|---|---|
| | | 15 = Linkwitz-Riley 6th<br>16 = Linkwitz-Riley 8th |
| Crossover Pass Type | Enum | Values:<br>0 = Low Pass<br>1 = High Pass |
| Frequency | Double | Value Range: 20Hz – 20000Hz |
| Bypass Flag | Boolean | Value Range: True (bypassed) or False (engaged) |

# PRESET OBJECT: DELAY

Description:

This preset object represents an audio delay block. The block is generic and maximum delay for a hardware device must be enforced by the software. Note that this block has a maximum delay of 65535 samples which translates to 1.365 seconds. In audio applications, delay is typically represented to the user of a system as time (milliseconds or seconds) or distance (metres or feet). As the speed of sound varies with air temperature, this needs to be accounted for, although typically most calculations assume an ambient temperature of 25 degrees.

Firmware Byte Stream:

| Item | Size | Description |
|---|---|---|
| Bypass | UInt8 | Value Range:<br>0 = Delay enabled<br>1 = Delay bypassed |
| Delay | UInt16 | Value Range:<br>1 – 65535 (UInt16 maximum)<br>The minimum delay through the block is always 1. |
| CRC8 Type 1 | UInt8 | The CRC8 (Type 1) computation of all preceding bytes |

Master Software Typical Storage:

| Item | Type | Description |
|---|---|---|
| Delay | Double | Value Range: 1 – 65536 samples, block dependent |
| Bypass Flag | Boolean | Value Range: True (bypassed) or False (engaged) |

## PRESET OBJECT: DYNAMICS

Description:

This preset object represents an audio dynamics block. These blocks can implement a compressor, limiter or even expander to control the dynamic range of a signal.

Firmware Byte Stream:

| Item | Size | Description |
|---|---|---|
| Attack | UInt16 | Attack (in msec) multiplied x 10.<br>Value Range is between 10 and 1000 (1msec to 100msec). |
| Release | UInt16 | Release (in msec) multiplied x 10.<br>Value Range is between 2000 and 20000 (200msec to 2sec). |
| Threshold | UInt16 | Threshold (in dB) negated and multiplied by 100.<br>Value range is between 0 and 4800 (0dB to -48dB). |
| Ratio | UInt16 | Ratio (in dB) multiplied by 100.<br>Value range is between 100 and 10000 (1 to 100). |
| Post Gain | UInt32 | DecibelsToFixed24 conversion of post gain (in dB). |
| Enabled | UInt8 | Dynamics Enabled flag.<br>0 = Not enabled (bypassed).<br>1 = Enabled and active. |
| Transfer Curve | 24x UInt32 | The calculated transfer curve of the dynamics block.<br>Consult the EDC Acoustics provided C# code. |
| RMS Time Constant | UInt32 | The calculated RMS time constant of the dynamics block.<br>Consult the EDC Acoustics provided C# code. |
| Hold Time | UInt32 | Not used. Set to 0. |
| RMS Decay Time | UInt32 | The calculated RMS decay constant of the dynamics block.<br>Consult the EDC Acoustics provided C# code. |
| CRC8 Type 1 | 1 byte | The CRC8 (Type 1) computation of all preceding bytes |

Master Software Typical Storage:

| Item | Type | Description |
|---|---|---|
| Attack | Double | Value Range: 1msec – 100msec |
| Release | Double | Value Range: 200msec – 2sec |
| Threshold | Double | Value Range: -48dB to 0dB |
| Ratio | Double | Value Range: 1.0 to 100.0 |
| Post Gain | Double | Value Range: -20dB to 20dB |
| Enabled | Boolean | Value Range: 0 (not enabled) or 1 (enabled) |

# PRESET OBJECT: FILTER (BIQUAD)

Description:

This preset object represents a single Biquad IIR filter stage, generally used for parametric equalization.

Firmware Byte Stream:

| Item | Size | Description |
|---|---|---|
| Filter Info Pack 1 | UInt32 | *Highest Byte:*<br>Value is the Filter Type<br>(as per Enum in Software Table)<br>If bypassed, OR with 0x80<br>*Lower Three Bytes:*<br>Frequency of Filter x10 (e.g. 1000Hz = 10000) |
| Filter Info Pack 2 | UInt24 | *Upper 12 bits:*<br>(Gain of the Filter + 15.0dB) x100<br>*Lower 12 bits:*<br>Q of the Filter x100 |
| Filter Stage 1 Coeffs | 5x UInt32 | Biquad Filter Coefficients, Stage 1:<br>B0, B1, B2,<br>-A1, -A2 (note that these coeffs are negated)<br>In order, normalized to A0 |
| CRC8 Type 1 | UInt8 | The CRC8 (Type 1) computation of all preceding bytes |

Master Software Typical Storage:

| Item | Type | Description |
|---|---|---|
| Filter Type | Enum | Values:<br>0 = Bell<br>1 = Shelving Low Pass 1st<br>2 = Shelving Low Pass 2nd<br>3 = Shelving High Pass 1st<br>4 = Shelving High Pass 2nd<br>5 = Notch<br>6 = Butterworth Low Pass 1st<br>7 = Butterworth High Pass 1st<br>8 = Butterworth Low Pass 2nd<br>9 = Butterworth High Pass 2nd<br>10 = All Pass 2nd |
| Frequency | Double | Value Range: 20Hz – 20000Hz |
| Gain | Double | Value Range: -15.0 to +15.0 |
| Q | Double | Value Range: 0.1 to 10.0 |
| Bypass Flag | Boolean | Value Range: True (bypassed) or False (engaged) |

# PRESET OBJECT: FIR FILTER 2111

Description:

This preset object represents a FIR Filter with up to 2111 coefficients.

Firmware Byte Stream:

| Item | Size | Description |
|---|---|---|
| Coefficients | 2111x UInt32 | The FIR coefficients. Note that a filter of less length than 2111 coefficients can be used – but all unused coefficients must be set to 0.<br>Note also that the DSP stores the coefficients in reverse order.<br>Use FracToSignedFixed24 to convert the coefficient from a double to an Int32, and then cast this Int32 to a UInt32 to be packed into the array. |
| Reserved | UInt16 | Set to 0 (note – two bytes). |
| Bypass | UInt8 | Filter bypass flag.<br>0 = Not bypassed (enabled).<br>1 = Bypassed (disabled). |
| CRC8 Type 1 | UInt8 | The CRC8 (Type 1) computation of all preceding bytes |

Master Software Typical Storage:

| Item | Type | Description |
|---|---|---|
| Coefficients | Array of Double | Coefficients can theoretically range from -128.0 to 127.0. Up to 2111 can be stored. |
| Bypass | Boolean | Filter bypass flag.<br>False = Not bypassed (enabled).<br>True = Bypassed (disabled). |

**NOTE!** This block is too big to send using the standard POBJ packet to write and read the coefficient. Instead, it must be sent via a series of FCMD packets that write or read directly from the EEPROM memory location. If being written, a PSET packet with a force recall option once the entire filter is written can be called to engage the filter. More on this procedure will be described under the hardware specific sections of this document.

# PRESET OBJECT: FIR FILTER 511

Description:

This preset object represents a FIR Filter with up to 511 coefficients.

Firmware Byte Stream:

| Item | Size | Description |
|---|---|---|
| Coefficients | 511x UInt32 | The FIR coefficients. Note that a filter of less length than 511 coefficients can be used – but all unused coefficients must be set to 0.<br>Note also that the DSP stores the coefficients in reverse order.<br>Use FracToSignedFixed24 to convert the coefficient from a double to an Int32, and then cast this Int32 to a UInt32 to be packed into the array. |
| Reserved | UInt16 | Set to 0 (note – two bytes). |
| Bypass | UInt8 | Filter bypass flag.<br>0 = Not bypassed (enabled).<br>1 = Bypassed (disabled). |
| CRC8 Type 1 | UInt8 | The CRC8 (Type 1) computation of all preceding bytes |

Master Software Typical Storage:

| Item | Type | Description |
|---|---|---|
| Coefficients | Array of Double | Coefficients can theoretically range from -128.0 to 127.0. Up to 511 can be stored. |
| Bypass | Boolean | Filter bypass flag.<br>False = Not bypassed (enabled).<br>True = Bypassed (disabled). |

**NOTE!** This block is too big to send using the standard POBJ packet to write and read the coefficient. Instead, it must be sent via a series of FCMD packets that write or read directly from the EEPROM memory location. If being written, a PSET packet with a force recall option once the entire filter is written can be called to engage the filter. More on this procedure will be described under the hardware specific sections of this document.

## PRESET OBJECT: FRACTIONAL DELAY

Description:

This preset object represents a delay block of which the delay is calculated as a percentage. 100% uses the full length of the delay buffer allocated, while 0% effectively bypasses the delay.

Firmware Byte Stream:

| Item | Size | Description |
|---|---|---|
| Percentage Delay | UInt32 | Value Range: 0.0 to 1.0, expressed as Fixed 8.24 number.<br>Use FracToFixed24 to convert the value. |
| Bypass | UInt8 | Delay bypass flag.<br>0 = Not bypassed (enabled).<br>1 = Bypassed (disabled). |
| CRC8 Type 1 | UInt8 | The CRC8 (Type 1) computation of all preceding bytes |

Master Software Typical Storage:

| Item | Type | Description |
|---|---|---|
| Percentage Delay | Double | Value between 0 and 100%. |
| Bypass | Boolean | Filter bypass flag.<br>False = Not bypassed (enabled).<br>True = Bypassed (disabled). |

## PRESET OBJECT: GENERIC VALUE

Description:

This preset object represents a generic value which can be sent to a DSP block.

Firmware Byte Stream:

| Item | Size | Description |
|---|---|---|
| Generic Value | UInt32 | Value Range: -128.0 to 127.0, expressed as Fixed 8.24 number. Use FracToFixed24 to convert the value. |
| CRC8 Type 1 | UInt8 | The CRC8 (Type 1) computation of all preceding bytes |

Master Software Typical Storage:

| Item | Type | Description |
|---|---|---|
| Generic Value | Double | Value between -128.0 and 127.0. |

## PRESET OBJECT: GENERIC VALUE (BYTE)

Description:

This preset object represents a generic value which can be sent to a DSP block, but limited to a single byte value.

Firmware Byte Stream:

| Item | Size | Description |
|---|---|---|
| Generic Value | UInt8 | Value Range: 0 to 255. |
| CRC8 Type 1 | UInt8 | The CRC8 (Type 1) computation of all preceding bytes |

Master Software Typical Storage:

| Item | Type | Description |
|---|---|---|
| Generic Value | UInt8 | Value between 0 and 255. |

# PRESET OBJECT: INVERT

Description:

This preset object is used to enable or disable a signal inversion in the DSP.

Firmware Byte Stream:

| Item | Size | Description |
|---|---|---|
| Invert | UInt8 | Value Range:<br>0 = No Signal Inversion<br>1 = Signal Inversion |
| CRC8 Type 1 | UInt8 | The CRC8 (Type 1) computation of all preceding bytes |

Master Software Typical Storage:

| Item | Type | Description |
|---|---|---|
| Invert | Boolean | Value Range:<br>False = No Signal Inversion<br>True = Signal Inversion |

## PRESET OBJECT: VOLUME

Description:

This preset object represents a volume or gain trim block.

Firmware Byte Stream:

| Item | Size | Description |
|---|---|---|
| Volume | UInt32 | Value Range: -100.0 to 24.0, expressed as Fixed 8.24 number.<br>Use DecibelsToFixed24 to convert the value. |
| Flags | UInt8 | Muting and Bypass Flags:<br>If muted, OR this with 0x01.<br>If bypassed, OR this with 0x02. |
| CRC8 Type 1 | UInt8 | The CRC8 (Type 1) computation of all preceding bytes |

Master Software Typical Storage:

| Item | Type | Description |
|---|---|---|
| Volume | Double | Value between -100.0 and +24.0dB. |
| Muted | Boolean | Block muted flag.<br>False = Not muted.<br>True = Muted. |
| Bypassed | Boolean | Block bypassed flag.<br>False = Not bypassed.<br>True = Bypassed (the volume will act as 0.0dB). |

## PRESET OBJECT: VOLUME LUT (LOOK-UP TABLE)

Description:

This preset object represents a volume or gain trim block that is a single byte value that equates to a volume ratio derived from a look-up table in the firmware.

Firmware Byte Stream:

| Item | Size | Description |
|---|---|---|
| Volume Index | UInt8 | Value Range: 0 (Off) to 249 (+24.0dB). |
| Flags | UInt8 | Muting and Bypass Flags:<br>If muted, OR this with 0x01.<br>If bypassed, OR this with 0x02. |
| CRC8 Type 1 | UInt8 | The CRC8 (Type 1) computation of all preceding bytes |

Master Software Typical Storage:

| Item | Type | Description |
|---|---|---|
| Volume Index | Double | Value between 0 (Off) to 249 (+24.0dB). |
| Muted | Boolean | Block muted flag.<br>False = Not muted.<br>True = Muted. |
| Bypassed | Boolean | Block bypassed flag.<br>False = Not bypassed.<br>True = Bypassed (the volume will act as 0.0dB). |

The Volume Index byte is a value ranging from 0 to 249, where 0 indicates mute, and values 1-249 represent -100.0dB to +24.0dB in 0.5dB steps. The following table shows the arrangement:

| Volume LUT Index | Audio dB Value |
|---|---|
| 0 | Mute |
| 1 | -100.0dB |
| 2 | -99.5dB |
| 3 | -99.0dB |
| … | … |
| 177 | -12.0dB |
| … | … |
| 189 | -6.0dB |
| … | … |
| 201 | 0.0dB |
| … | … |
| 248 | 23.5dB |
| 249 | 24.0dB |

# SPEAKER SPECIFIC INFORMATION

## EDC COLUMN/SQUARE VARIANT (FW VERSION 1.2.5 ONWARDS)

This section contains information specifically related to the Column/Square EDC speaker variants, including:

- Any specific global objects
- Any specific preset objects
- Full global object memory layout
- Full preset object memory layout
- Signal monitoring packet structure
- Status packet structure
- FIR filter deployment

# COL/SQ EEPROM LAYOUT

The Column/Square speaker variant stores global objects and preset objects within EEPROM (non-volatile memory). This table provides a complete layout of the global objects needed to configure a speaker.

NOTE: When writing directly to the EEPROM, ***write a whole block of 256 bytes (page size)***. Do not attempt to write just one Global/Preset Object via an FCMD packet – for that situation, use the GOBJ/POBJ write packet. Normally the approach should be to use FCMD packets when writing all parameters (sending a full configuration to a speaker) or writing an FIR filter. For all other conditions, use the GOBJ/POBJ write packets.

Also, keep the 256 bytes to an EEPROM page on the modulo of 256 – i.e. the start addr % (modulo) 256 must equal 0. This prevents writing across page boundaries in the EEPROM which can result in erratic conditions.

For **global objects**, memory address space in the EEPROM starts at 0x000800.

| EEPROM ADDR (HEX) | GOBJ ID | Data Type | Description |
|---|---|---|---|
| 0x000800 | 1 | Global Object Name String | GOBJ_NAME_SPEAKERBOX<br>The name of the speaker in an array (row/col). |
| 0x000820 | 2 | Global Object Name String | GOBJ_NAME_INPUT_ANALOG<br>The name to assign to the analog input. |
| 0x000840 | 3 | Global Object Name String | GOBJ_NAME_INPUT_AESLEFT<br>The name to assign to the AES Left input. |
| 0x000860 | 4 | Global Object Name String | GOBJ_NAME_INPUT_AESRIGHT<br>The name to assign to the AES Right input. |
| 0x000880 | 5 | Global Object Boolean Flag | GOBJ_STANDBY_ALLOWED<br>If true, the speaker can enter standby. |
| 0x000882 | 6 | Global Object Boolean Flag | GOBJ_FORCE_STANDBY<br>If true, the speaker enters standby. |
| 0x000884 | 7 | Global Object Boolean Flag | GOBJ_GLOBAL_MUTE_ALLOWED<br>If true, the speaker can globally mute. |
| 0x000886 | 8 | Global Object Boolean Flag | GOBJ_FORCE_GLOBAL_MUTE<br>If true, the speakers is globally muted. |
| 0x000888 | 9 | Global Object Boolean Flag | GOBJ_PRESET_IN_USE1<br>Not used in typical systems. Set false. |
| 0x00088A | 10 | Global Object Name String | GOBJ_NAME_PRESET1<br>Not used in typical systems. Set blank. |
| 0x0008AA | 11 | Global Object Dsp Volume | GOBJ_SENSE_LF_LEVEL<br>The Sense Mic LF Signal Level. |
| 0x0008AF | 12 | Global Object Dsp Volume | GOBJ_SENSE_HF_LEVEL<br>The Sense Mic HF Signal Level. |
| 0x0008B4 | 13 | Global Object Dsp Volume | GOBJ_SENSE_LF_THRESHOLD<br>The Sense Mic LF Pickup Threshold. |
| 0x0008B9 | 14 | Global Object Dsp Volume | GOBJ_SENSE_HF_THRESHOLD<br>The Sense Mic HF Pickup Threshold. |
| 0x0008BE | 15 | Global Object Boolean Flag | GOBJ_SPEAKER_TEST_ENABLED<br>If true, the speaker will test with the sense microphone. |
| 0x0008C0 | 16 | Global Object Standby Record | GOBJ_AUTO_STANDBY_SETUP<br>These parameters determine if the speaker will go into auto standby. |

| | | | |
|---|---|---|---|
| 0x0008CB | 17 | Global Object Name String | GOBJ_NAME_BEAM1<br>The name to assign to Beam 1. |
| 0x0008EB | 18 | Global Object Name String | GOBJ_NAME_BEAM2<br>The name to assign to Beam 2. |
| 0x00090B | 19 | Global Object Name String | GOBJ_NAME_BEAM3<br>The name to assign to Beam 3. |
| 0x00092B | 20 | Global Object Name String | GOBJ_NAME_BEAM4<br>The name to assign to Beam 4. |

For **preset objects**, memory address space in the EEPROM starts at 0x000C00.

| EEPROM ADDR (HEX) | POBJ ID | Data Type | Description |
|---|---|---|---|
| 0x000C00 | 81 | Preset Object FIR 2111 | POBJ_BEAM1_FIR_LF<br>The low frequency FIR filter for Beam 1. |
| 0x002D00 | 82 | Preset Object FIR 511 | POBJ_BEAM1_FIR_HF<br>The high frequency FIR filter for Beam 1. |
| 0x003500 | 89 | Preset Object FIR 2111 | POBJ_BEAM2_FIR_LF<br>The low frequency FIR filter for Beam 1. |
| 0x005600 | 90 | Preset Object FIR 511 | POBJ_BEAM2_FIR_HF<br>The high frequency FIR filter for Beam 1. |
| 0x005E00 | 97 | Preset Object FIR 2111 | POBJ_BEAM3_FIR_LF<br>The low frequency FIR filter for Beam 1. |
| 0x007F00 | 98 | Preset Object FIR 511 | POBJ_BEAM3_FIR_HF<br>The high frequency FIR filter for Beam 1. |
| 0x008700 | 105 | Preset Object FIR 2111 | POBJ_BEAM4_FIR_LF<br>The low frequency FIR filter for Beam 1. |
| 0x00A800 | 106 | Preset Object FIR 511 | POBJ_BEAM4_FIR_HF<br>The high frequency FIR filter for Beam 1. |
| 0x00B000 | 1 | Preset Object Volume | POBJ_ANALOG_INPUT_TRIM<br>The input gain trim of the analog input. |
| 0x00B006 | 2 | Preset Object Boolean Flag | POBJ_ANALOG_FILTER_BYPASS<br>If true bypasses all 15 analog filters. |
| 0x00B008 | 3 | Preset Object Filter | POBJ_ANALOG_FILTER1<br>Analog input biquad filter 1. |
| 0x00B024 | 4 | Preset Object Filter | POBJ_ANALOG_FILTER2<br>Analog input biquad filter 2. |
| 0x00B040 | 5 | Preset Object Filter | POBJ_ANALOG_FILTER3<br>Analog input biquad filter 3. |
| 0x00B05C | 6 | Preset Object Filter | POBJ_ANALOG_FILTER4<br>Analog input biquad filter 4. |
| 0x00B078 | 7 | Preset Object Filter | POBJ_ANALOG_FILTER5<br>Analog input biquad filter 5. |
| 0x00B094 | 8 | Preset Object Filter | POBJ_ANALOG_FILTER6<br>Analog input biquad filter 6. |
| 0x00B0B0 | 9 | Preset Object Filter | POBJ_ANALOG_FILTER7<br>Analog input biquad filter 7. |
| 0x00B0CC | 10 | Preset Object Filter | POBJ_ANALOG_FILTER8<br>Analog input biquad filter 8. |
| 0x00B0E8 | 11 | Preset Object Filter | POBJ_ANALOG_FILTER9<br>Analog input biquad filter 9. |
| 0x00B104 | 12 | Preset Object | POBJ_ANALOG_FILTER10 |

| | | | |
|---|---|---|---|
| | | Filter | Analog input biquad filter 10. |
| 0x00B120 | 13 | Preset Object Filter | POBJ_ANALOG_FILTER11 Analog input biquad filter 11. |
| 0x00B13C | 14 | Preset Object Filter | POBJ_ANALOG_FILTER12 Analog input biquad filter 12. |
| 0x00B158 | 15 | Preset Object Filter | POBJ_ANALOG_FILTER13 Analog input biquad filter 13. |
| 0x00B174 | 16 | Preset Object Filter | POBJ_ANALOG_FILTER14 Analog input biquad filter 14. |
| 0x00B190 | 17 | Preset Object Filter | POBJ_ANALOG_FILTER15 Analog input biquad filter 15. |
| 0x00B1AC | 18 | Preset Object Invert | POBJ_ANALOG_INVERT Analog input signal inversion |
| 0x00B1AE | 19 | Preset Object DRC | POBJ_ANALOG_DRC Analog input dynamic range control. |
| 0x00B228 | 20 | Preset Object Volume | POBJ_AES_LEFT_INPUT_TRIM The input gain trim of the AES left input. |
| 0x00B22E | 21 | Preset Object Boolean Flag | POBJ_ AES_LEFT_FILTER_BYPASS If true bypasses all 15 AES left filters. |
| 0x00B230 | 22 | Preset Object Filter | POBJ_AES_LEFT_FILTER1 AES left input biquad filter 1. |
| 0x00B24C | 23 | Preset Object Filter | POBJ_AES_LEFT_FILTER2 AES left input biquad filter 2. |
| 0x00B268 | 24 | Preset Object Filter | POBJ_AES_LEFT_FILTER3 AES left input biquad filter 3. |
| 0x00B284 | 25 | Preset Object Filter | POBJ_AES_LEFT_FILTER4 AES left input biquad filter 4. |
| 0x00B2A0 | 26 | Preset Object Filter | POBJ_AES_LEFT_FILTER5 AES left input biquad filter 5. |
| 0x00B2BC | 27 | Preset Object Filter | POBJ_AES_LEFT_FILTER6 AES left input biquad filter 6. |
| 0x00B2D8 | 28 | Preset Object Filter | POBJ_AES_LEFT_FILTER7 AES left input biquad filter 7. |
| 0x00B2F4 | 29 | Preset Object Filter | POBJ_AES_LEFT_FILTER8 AES left input biquad filter 8. |
| 0x00B310 | 30 | Preset Object Filter | POBJ_AES_LEFT_FILTER9 AES left input biquad filter 9. |
| 0x00B32C | 31 | Preset Object Filter | POBJ_AES_LEFT_FILTER10 AES left input biquad filter 10. |
| 0x00B348 | 32 | Preset Object Filter | POBJ_AES_LEFT_FILTER11 AES left input biquad filter 11. |
| 0x00B364 | 33 | Preset Object Filter | POBJ_AES_LEFT_FILTER12 AES left input biquad filter 12. |
| 0x00B380 | 34 | Preset Object Filter | POBJ_AES_LEFT_FILTER13 AES left input biquad filter 13. |
| 0x00B39C | 35 | Preset Object Filter | POBJ_AES_LEFT_FILTER14 AES left input biquad filter 14. |
| 0x00B3B8 | 36 | Preset Object Filter | POBJ_AES_LEFT_FILTER15 AES left input biquad filter 15. |
| 0x00B3D4 | 37 | Preset Object Invert | POBJ_AES_LEFT_INVERT AES left input signal inversion |
| 0x00B3D6 | 38 | Preset Object DRC | POBJ_AES_LEFT_DRC AES left input dynamic range control. |
| 0x00B450 | 39 | Preset Object | POBJ_AES_RIGHT_INPUT_TRIM |

| | | | |
|---|---|---|---|
| | | Volume | The input gain trim of the AES right input. |
| 0x00B456 | 40 | Preset Object Boolean Flag | POBJ_ AES_RIGHT_FILTER_BYPASS<br>If true bypasses all 15 AES right filters. |
| 0x00B458 | 41 | Preset Object Filter | POBJ_AES_RIGHT_FILTER1<br>AES right input biquad filter 1. |
| 0x00B474 | 42 | Preset Object Filter | POBJ_AES_RIGHT_FILTER2<br>AES right input biquad filter 2. |
| 0x00B490 | 43 | Preset Object Filter | POBJ_AES_RIGHT_FILTER3<br>AES right input biquad filter 3. |
| 0x00B4AC | 44 | Preset Object Filter | POBJ_AES_RIGHT_FILTER4<br>AES right input biquad filter 4. |
| 0x00B4C8 | 45 | Preset Object Filter | POBJ_AES_RIGHT_FILTER5<br>AES right input biquad filter 5. |
| 0x00B4E4 | 46 | Preset Object Filter | POBJ_AES_RIGHT_FILTER6<br>AES right input biquad filter 6. |
| 0x00B500 | 47 | Preset Object Filter | POBJ_AES_RIGHT_FILTER7<br>AES right input biquad filter 7. |
| 0x00B51C | 48 | Preset Object Filter | POBJ_AES_RIGHT_FILTER8<br>AES right input biquad filter 8. |
| 0x00B538 | 49 | Preset Object Filter | POBJ_AES_RIGHT_FILTER9<br>AES right input biquad filter 9. |
| 0x00B554 | 50 | Preset Object Filter | POBJ_AES_RIGHT_FILTER10<br>AES right input biquad filter 10. |
| 0x00B570 | 51 | Preset Object Filter | POBJ_AES_RIGHT_FILTER11<br>AES right input biquad filter 11. |
| 0x00B58C | 52 | Preset Object Filter | POBJ_AES_RIGHT_FILTER12<br>AES right input biquad filter 12. |
| 0x00B5A8 | 53 | Preset Object Filter | POBJ_AES_RIGHT_FILTER13<br>AES right input biquad filter 13. |
| 0x00B5C4 | 54 | Preset Object Filter | POBJ_AES_RIGHT_FILTER14<br>AES right input biquad filter 14. |
| 0x00B5E0 | 55 | Preset Object Filter | POBJ_AES_RIGHT_FILTER15<br>AES right input biquad filter 15. |
| 0x00B5FC | 56 | Preset Object Invert | POBJ_AES_RIGHT_INVERT<br>AES right input signal inversion |
| 0x00B5FE | 57 | Preset Object DRC | POBJ_AES_RIGHT_DRC<br>AES right input dynamic range control. |
| 0x00B678 | 58 | Preset Object Boolean Flag | POBJ_PINK_NOISE_MASTER_MUTE<br>Master mute for the pink noise generator. |
| 0x00B67A | 59 | Preset Obj Volume LUT | POBJ_BEAM1_INPUTMIX_ANALOG<br>Analog input to Beam 1 mix level. |
| 0x00B67D | 60 | Preset Obj Volume LUT | POBJ_BEAM1_INPUTMIX_AES_LEFT<br>AES left input to Beam 1 mix level. |
| 0x00B680 | 61 | Preset Obj Volume LUT | POBJ_BEAM1_INPUTMIX_AES_RIGHT<br>AES right input to Beam 1 mix level. |
| 0x00B683 | 62 | Preset Obj Volume LUT | POBJ_BEAM1_INPUTMIX_PINK_NOISE<br>Pink noise to Beam 1 mix level. |
| 0x00B686 | 63 | Preset Obj Volume LUT | POBJ_BEAM1_INPUTMIX_MASTER<br>Beam 1 master level. |
| 0x00B689 | 64 | Preset Obj Volume LUT | POBJ_BEAM2_INPUTMIX_ANALOG<br>Analog input to Beam 2 mix level. |
| 0x00B68C | 65 | Preset Obj Volume LUT | POBJ_BEAM2_INPUTMIX_AES_LEFT<br>AES left input to Beam 2 mix level. |
| 0x00B68F | 66 | Preset Obj | POBJ_BEAM2_INPUTMIX_AES_RIGHT |

| | | | |
|---|---|---|---|
| | | Volume LUT | AES right input to Beam 2 mix level. |
| 0x00B692 | 67 | Preset Obj<br>Volume LUT | POBJ_BEAM2_INPUTMIX_PINK_NOISE<br>Pink noise to Beam 2 mix level. |
| 0x00B695 | 68 | Preset Obj<br>Volume LUT | POBJ_BEAM2_INPUTMIX_MASTER<br>Beam 2 master level. |
| 0x00B698 | 69 | Preset Obj<br>Volume LUT | POBJ_BEAM3_INPUTMIX_ANALOG<br>Analog input to Beam 3 mix level. |
| 0x00B69B | 70 | Preset Obj<br>Volume LUT | POBJ_BEAM3_INPUTMIX_AES_LEFT<br>AES left input to Beam 3 mix level. |
| 0x00B69E | 71 | Preset Obj<br>Volume LUT | POBJ_BEAM3_INPUTMIX_AES_RIGHT<br>AES right input to Beam 3 mix level. |
| 0x00B6A1 | 72 | Preset Obj<br>Volume LUT | POBJ_BEAM3_INPUTMIX_PINK_NOISE<br>Pink noise to Beam 3 mix level. |
| 0x00B6A4 | 73 | Preset Obj<br>Volume LUT | POBJ_BEAM3_INPUTMIX_MASTER<br>Beam 3 master level. |
| 0x00B6A7 | 74 | Preset Obj<br>Volume LUT | POBJ_BEAM4_INPUTMIX_ANALOG<br>Analog input to Beam 4 mix level. |
| 0x00B6AA | 75 | Preset Obj<br>Volume LUT | POBJ_BEAM4_INPUTMIX_AES_LEFT<br>AES left input to Beam 4 mix level. |
| 0x00B6AD | 76 | Preset Obj<br>Volume LUT | POBJ_BEAM4_INPUTMIX_AES_RIGHT<br>AES right input to Beam 4 mix level. |
| 0x00B6B0 | 77 | Preset Obj<br>Volume LUT | POBJ_BEAM4_INPUTMIX_PINK_NOISE<br>Pink noise to Beam 4 mix level. |
| 0x00B6B3 | 78 | Preset Obj<br>Volume LUT | POBJ_BEAM4_INPUTMIX_MASTER<br>Beam 4 master level. |
| 0x00B6B6 | 79 | Preset Obj<br>Delay | POBJ_BEAM1_PREFIR_DELAY<br>Beam 1 bulk delay inserted before FIRs. |
| 0x00B6BA | 80 | Preset Obj<br>Adv Crossover | POBJ_BEAM1_ADVCROSSOVER_HPF<br>Beam 1 HPF crossover– up to 8$^{th}$ order. |
| 0x00B70F | 83 | Preset Obj<br>DRC | POBJ_BEAM1_POSTFIR_LF_DRC<br>Beam 1 LF compression after FIRs. |
| 0x00B789 | 84 | Preset Obj<br>DRC | POBJ_BEAM1_POSTFIR_HF_DRC<br>Beam 1 HF compression after FIRs. |
| 0x00B803 | 85 | Preset Obj<br>Delay | POBJ_BEAM1_POSTFIR_LF_DELAY<br>Beam 1 LF delay after FIRs.<br>Use this for large arrays. |
| 0x00B807 | 86 | Preset Obj<br>Delay | POBJ_BEAM1_POSTFIR_HF_DELAY<br>Beam 1 HF delay after FIRs.<br>Use this for large arrays. |
| 0x00B80B | 87 | Preset Obj<br>Delay | POBJ_BEAM2_PREFIR_DELAY<br>Beam 2 bulk delay inserted before FIRs. |
| 0x00B80F | 88 | Preset Obj<br>Adv Crossover | POBJ_BEAM2_ADVCROSSOVER_HPF<br>Beam 2 HPF crossover– up to 8$^{th}$ order. |
| 0x00B864 | 91 | Preset Obj<br>DRC | POBJ_BEAM2_POSTFIR_LF_DRC<br>Beam 2 LF compression after FIRs. |
| 0x00B8DE | 92 | Preset Obj<br>DRC | POBJ_BEAM2_POSTFIR_HF_DRC<br>Beam 2 HF compression after FIRs. |
| 0x00B958 | 93 | Preset Obj<br>Delay | POBJ_BEAM2_POSTFIR_LF_DELAY<br>Beam 2 LF delay after FIRs.<br>Use this for large arrays. |
| 0x00B95C | 94 | Preset Obj<br>Delay | POBJ_BEAM2_POSTFIR_HF_DELAY<br>Beam 2 HF delay after FIRs.<br>Use this for large arrays. |
| 0x00B960 | 95 | Preset Obj<br>Delay | POBJ_BEAM3_PREFIR_DELAY<br>Beam 3 bulk delay inserted before FIRs. |

| | | | |
|---|---|---|---|
| 0x00B964 | 96 | Preset Obj<br>Adv Crossover | POBJ_BEAM3_ADVCROSSOVER_HPF<br>Beam 3 HPF crossover– up to 8th order. |
| 0x00B9B9 | 99 | Preset Obj<br>DRC | POBJ_BEAM3_POSTFIR_LF_DRC<br>Beam 3 LF compression after FIRs. |
| 0x00BA33 | 100 | Preset Obj<br>DRC | POBJ_BEAM3_POSTFIR_HF_DRC<br>Beam 3 HF compression after FIRs. |
| 0x00BAAD | 101 | Preset Obj<br>Delay | POBJ_BEAM3_POSTFIR_LF_DELAY<br>Beam 3 LF delay after FIRs.<br>Use this for large arrays. |
| 0x00BAB1 | 102 | Preset Obj<br>Delay | POBJ_BEAM3_POSTFIR_HF_DELAY<br>Beam 3 HF delay after FIRs.<br>Use this for large arrays. |
| 0x00BAB5 | 103 | Preset Obj<br>Delay | POBJ_BEAM4_PREFIR_DELAY<br>Beam 4 bulk delay inserted before FIRs. |
| 0x00BAB9 | 104 | Preset Obj<br>Adv Crossover | POBJ_BEAM4_ADVCROSSOVER_HPF<br>Beam 4 HPF crossover– up to 8th order. |
| 0x00BB0E | 107 | Preset Obj<br>DRC | POBJ_BEAM4_POSTFIR_LF_DRC<br>Beam 4 LF compression after FIRs. |
| 0x00BB88 | 108 | Preset Obj<br>DRC | POBJ_BEAM4_POSTFIR_HF_DRC<br>Beam 4 HF compression after FIRs. |
| 0x00BC02 | 109 | Preset Obj<br>Delay | POBJ_BEAM4_POSTFIR_LF_DELAY<br>Beam 4 LF delay after FIRs.<br>Use this for large arrays. |
| 0x00BC06 | 110 | Preset Obj<br>Delay | POBJ_BEAM4_POSTFIR_HF_DELAY<br>Beam 4 HF delay after FIRs.<br>Use this for large arrays. |
| 0x00CACE | 201 | Preset Obj<br>Volume LUT | POBJ_ANA_OUT_INPUTMIX_ANALOG<br>Analog input to Analog output level. |
| 0x00CAD1 | 202 | Preset Obj<br>Volume LUT | POBJ_ANA_OUT_INPUTMIX_AES_LEFT<br>AES left input to Analog output level. |
| 0x00CAD4 | 203 | Preset Obj<br>Volume LUT | POBJ_ANA_OUT_INPUTMIX_AES_RIGHT<br>AES right input to Analog output level. |
| 0x00CAD7 | 204 | Preset Obj<br>Volume LUT | POBJ_ANA_OUT_INPUTMIX_PINK_NOISE<br>Pink noise to Analog output level. |

## COL/SQ SIGNAL MONITORING

For the SMON packet, 26 monitoring values are returned in the SMON packet (within the "Monitoring Data" section of the reply packet). As the values are 32-bit (big-endian), this equates to a length of 104 bytes.

In order, the monitoring values are:

| Byte Position | Monitoring Value Reference |
|---|---|
| 0 | Analog Input |
| 4 | AES Left Input |
| 8 | AES Right Input |
| 12 | Analog Pre DRC |
| 16 | AES Left Pre DRC |
| 20 | AES Right Pre DRC |
| 24 | Analog Post DRC |
| 28 | AES Left Post DRC |
| 32 | AES Right Post DRC |
| 36 | Beam 1 LF Pre DRC |
| 40 | Beam 2 LF Pre DRC |
| 44 | Beam 3 LF Pre DRC |
| 48 | Beam 4 LF Pre DRC |
| 52 | Beam 1 HF Pre DRC |
| 56 | Beam 2 HF Pre DRC |
| 60 | Beam 3 HF Pre DRC |
| 64 | Beam 4 HF Pre DRC |
| 68 | Beam 1 LF Post DRC |
| 72 | Beam 2 LF Post DRC |
| 76 | Beam 3 LF Post DRC |
| 80 | Beam 4 LF Post DRC |
| 84 | Beam 1 HF Post DRC |
| 88 | Beam 2 HF Post DRC |
| 92 | Beam 3 HF Post DRC |
| 96 | Beam 4 HF Post DRC |
| 100 | Sense Mic |

## COL/SQ STATUS DATA

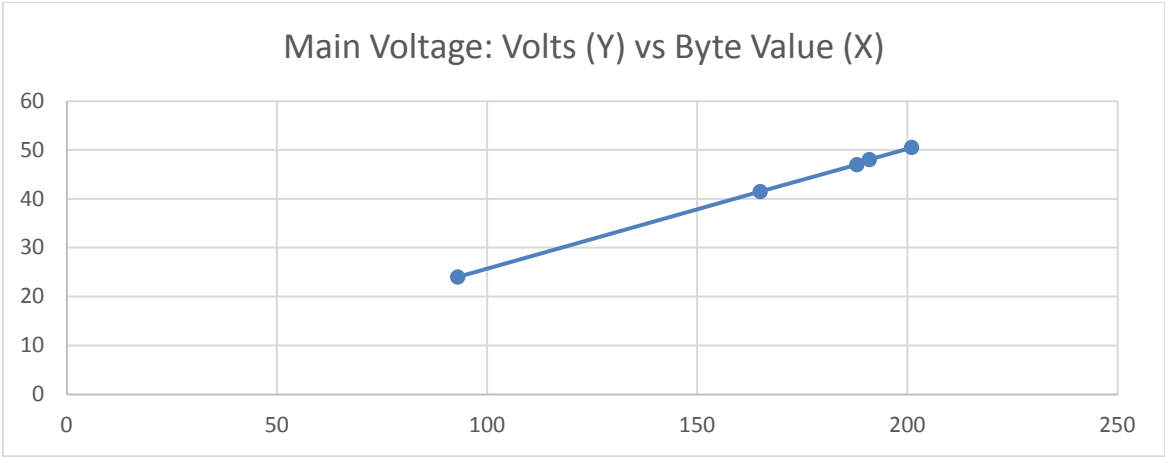
There are two types of STAT packets. A basic reply and an advanced reply. For a basic status packet, the data sent back is of the form:

| Byte Position | Status Data (Basic Packet) |
|---|---|
| 0 | 0x11 (Basic Status Indicator) |
| 1 | Summary Byte 1<br>Bit 0,1 = Status of Prime Amp Board<br>(00 = no fault, 01 = thermal warning, 02 = fault, 03 = thermal cut-out)<br>Bit 2,3 = Status of Slave Port #1 Board<br>(00 = no fault, 01 = thermal warning, 02 = fault, 03 = thermal cut-out)<br>Bit 4,5 = Status of Slave Port #2 Board<br>(00 = no fault, 01 = thermal warning, 02 = fault, 03 = thermal cut-out)<br>Bit 6 = 1 if DSP failed to boot, 0 if DSP running<br>Bit 7 = Unused |
| 2 | Summary Byte 2<br>Bit 0,1 = Temperature Indicator of Prime Amp Board<br>(00 = Temperature OK, 01 = Critical Temperature, 02 = Board Not Found)<br>Bit 2,3 = Temperature Indicator of Slave Port #1 Board<br>(00 = Temperature OK, 01 = Critical Temperature, 02 = Board Not Found)<br>Bit 4,5 = Temperature Indicator of Slave Port #2 Board<br>(00 = Temperature OK, 01 = Critical Temperature, 02 = Board Not Found)<br>Bit 6 = LF Driver Error Flag (1 = Errors, 0 = OK)<br>Bit 7 = HF Driver Error Flag (1 = Errors, 0 = OK) |

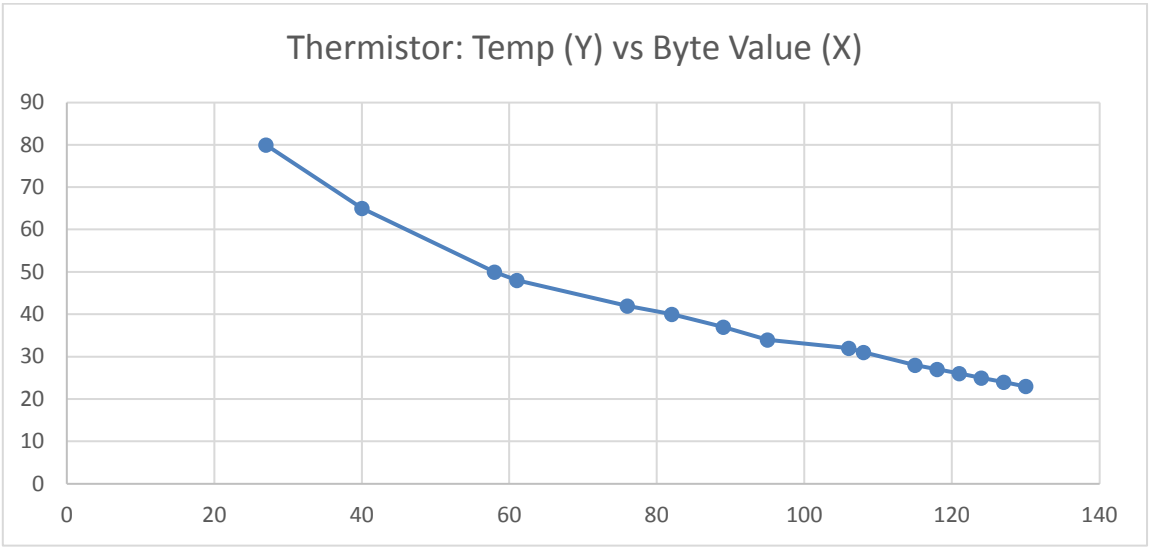
For the advanced status packet, the data sent back is of the form:

| Byte Position | Status Data (Basic Packet) |
|---|---|
| 0 | 0x22 (Advanced Status Indicator) |
| 1 | DSP Booted OK Flag (1 = Success, 0 = Failed) |
| 2 | Main Voltage Rail Value |
| 3 | Prime Amp Board Temperature |
| 4 | Slave Port #1 Amp Board Temperature (if fitted)<br>0xFF indicates not fitted |
| 5 | Slave Port #2 Amp Board Temperature (if fitted)<br>0xFF indicates not fitted |
| 6 | Standby Flag (1 = in Standby, 0 = Active) |
| 7..33 | Amp IC States (bits 0,1)<br>00 = Amp IC OK or not in use<br>01 = Amp in Thermal Fault<br>02 = Amp in Fault |
| 34..42 | LF Driver (1-9) Check Results<br>0x00 = Driver OK<br>0x01 = Driver Fault |
| 43..123 | HF Driver (1-81) Check Results<br>0x00 = Driver OK<br>0x01 = Driver Fault |

The main voltage rail value will be returned as indicated on the graph. A nominal value of around 93 (decimal) translates to 24V.




The temperatures are read from thermistors. The graph shows temperature (Celsius) on the Y vs the reading of the byte value (X).




In tabulated form, this data is:

| PSU ADC Reading (Byte Value, Decimal) | Main PSU Voltage Rail |
|---|---|
| 93 | 24 |
| 165 | 41.5 |
| 188 | 47 |
| 191 | 48 |
| 201 | 50.5 |

| Thermistor Reading (Byte Value, Decimal) | Approximate Amp Board Temperature |
|---|---|
| 27 | 80 |
| 40 | 65 |
| 58 | 50 |

| | |
|---|---|
| 61 | 48 |
| 76 | 42 |
| 82 | 40 |
| 89 | 37 |
| 95 | 34 |
| 106 | 32 |
| 108 | 31 |
| 115 | 28 |
| 118 | 27 |
| 121 | 26 |
| 124 | 25 |
| 127 | 24 |
| 130 | 23 |

# COL/SQ FIR FILTER DEPLOYMENT

Due to the size of FIR filters, they are deployed differently to a speaker as compared to smaller standard DSP blocks which will normally be deployed with POBJ packets.

The recommended approach for deployment is:

1) Assemble the payload data of the FIR for the firmware as indicated in the FIR Filter DSP Block section of this document. Each FIR type (2111 or 511 coefficients) has been defined to be broken into 256 page-size chunks.
2) Using the 256-byte size chunks, use the FCMD packet to issue a write command to the EEPROM. The start address of the first chunk should be the address listed in the POBJ index table for a COL/SQ speaker type as listed earlier in this section.
3) Wait for the speaker to send an ACKN packet back in response to the FCMD packet.
4) Repeat until all packets making up the FIR filter have been sent. For a 2111 coefficient FIR, this will be 33 FCMD packets, and for a 511 coefficient FIR, this will be 8 FCMD packets.
5) Finally, issue a PSET command packet with the ID type of 0xFE to issue a force reload of the live preset.
6) The new FIR filter will now be active.

EDC has developed a method to formulate FIR coefficients using data from Room EQ Wizard software. This is not documented here but if required, can be obtained from EDC. Note that an FFT library is required which will have a different API depending on the target platform.