Host CPU SDK Programmer's Guide

For Brooklyn II Firmware v3.10.x | Ultimo Application Software v3.10.x | Host CPU SDK v2.0.x

Document version: 1.0

Document name: AUD-MAN-Host_CPU_SDK_Programmers_Guide-v1.0.pdf

Published: 6th July 2016

Please see Audinate OEM Home for change history.

Feedback: if you would like to suggest improvements to this information, please feel free to email us at documentation@audinate.com.

Contacts

Audinate Pty Ltd

Level 1, 458 Wattle Street
Ultimo NSW 2007
Australia
Tel. +61 2 8090 1000

Postal address

Audinate Pty Ltd
PO Box 855
Broadway NSW 2007
Australia

Audinate Inc

1732 NW Quimby Street
Suite 215
Portland, OR 97209
USA
Tel: +1.503.224.2998
Fax. +1.503.360.1155
info@audinate.com
www.audinate.com

European Office

Audinate Ltd
Suite 303
Brighton Media Centre
Friese-Greene House
15-17 Middle St
Brighton, BN1 1AL
United Kingdom
Tel. +44 (0) 1273 921695

Asia Pacific Office

Audinate Limited
Suite 1106-08, 11/F Tai Yau Building
No 181 Johnston Road
Wanchai, Hong Kong
澳迪耐特有限公司
香港灣仔莊士敦道181號
大有大廈11樓1106-8室
Tel. +(852)-3588 0030
+(852)-3588 0031
Fax. +(852)-2975 8042

Contents

About Audinate

Founded in 2006, Audinate revolutionizes how AV systems are connected so customers can thrive in a networked world. Audinate's Dante audio networking technology has been adopted by the professional audio industry's leading manufacturers. Dante is used extensively for live performance events, commercial installations, broadcast, recording and production, and communications systems. Audinate offices are located in US, United Kingdom, Hong Kong and Australia.

About Dante

Dante audio networking utilizes standard IP networks to transmit high-quality, uncompressed audio with near-zero latency. It's the most economical, versatile, and easy-to-use audio networking solution, and is scalable from simple installations to large-capacity networks running thousands of audio channels. Dante can replace multiple analog or multicore cables with a single affordable Ethernet cable to transmit high-quality multi-channel audio safely and reliably. With Dante software, the network can be easily expanded and reconfigured with just a few mouse clicks. Dante is the audio networking choice of nearly all professional audio manufacturers, with hundreds of Dante-enabled audio products now available.

For more information, please visit the Audinate website at www.audinate.com.

Introduction

This document is intended for OEMs that wish to interface an external processor to a Dante device module via a serial peripheral. The external processor can configure and monitor Dante over the serial interface as well as utilise the network interface of the Dante module to bridge network traffic to the external processor. The supported Dante devices are the Brooklyn II and Ultimo, and the supported serial interfaces are SPI and UART. This document refers only to the built in functionality that can be enabled on the modules via the capability file - on the Brooklyn II it is also possible to communicate with a host CPU by implementing a custom OEM application running from the user partition.

The external processor which communicates with a Dante device over a serial link is referred to as the Host CPU throughout this document.

High level physical information about interfacing a Dante device to a Host CPU is provided in Hardware Requirements of a Host CPU. Additional information about physical I/O interfaces on the Dante device should be obtained from the Brooklyn II Technical Datasheet and Ultimo Technical Datasheet. An overall description about the software features of the Brooklyn II is available in the Brooklyn II Programmer's Guide, and the software features of the Ultimo are available in the Ultimo Programmer's Guide.

The Host CPU SDK package provides a messaging API for communication between the Dante device and the Host CPU. The SDK implements functionality to process and prepare messages that are received or sent from the Dante device. Information on the Host CPU SDK package is described in Host CPU SDK Package.

The Brooklyn II Host Interface Protocol (BHIP) is used to carry the following types of messages:

- Packet Bridge (PB) messages
- Dante Device Protocol (DDP) messages

The Ultimo Host Interface Protocol (UHIP) is used to carry the following types of messages:

- Packet Bridge (PB) messages
- Dante Device Protocol (DDP) messages
- Dante Event messages

The Packet Bridge, Dante Event and DDP protocols are described in Packet Bridge, Dante Events and Dante Device Protocol.

Hardware Requirements of a Host CPU

This section describes the hardware requirements of a host CPU that communicates with a Brooklyn II or Ultimo Dante device.

Communicating with a Brooklyn II

Regardless of the communication method, 3.3V I/O signalling levels must be used. For in-depth information about the Brooklyn II I/O pins, signal timing, signal characteristics and pin muxes please refer to the relevant sections of the Brooklyn II Technical Datasheet.

SPI

Communication using SPI requires one SPI master peripheral on the Host CPU. This SPI peripheral will connect to the SPI slave on the Brooklyn II SPI slave interface. Data over SPI is transferred as full duplex. Please refer to the Brooklyn II Programmer’s Guidefor more information about the SPI slave protocol.

Figure 1 shows the hardware connection to realize communication over SPI between the Brooklyn II and the Host CPU. There is a pin called DATA_AVAILABLE on the Brooklyn II which the SPI slave module uses to signal the availability of data to be transmitted to an external SPI master. A Host CPU must utilise the DATA_AVAILABLE line to as an interrupt source or poll it on a periodic basis.

The Host CPU must have 3x 1508 bytes of RAM to send one maximum sized frame and receive up to two maximum sized frames.

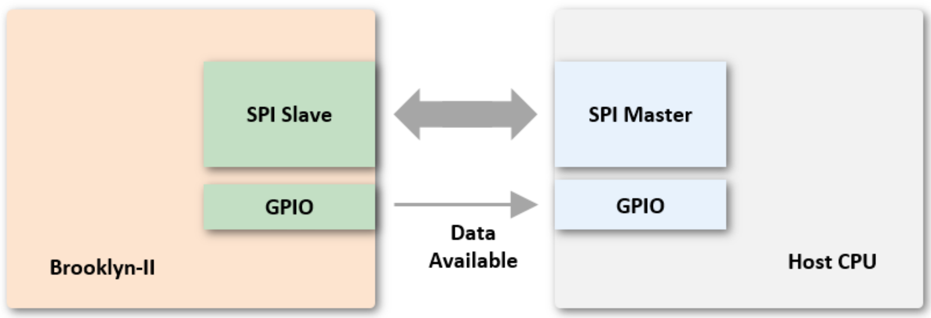

Figure 1 - Host CPU and Brooklyn II SPI communication

Table 1 shows the pin mapping between the Brooklyn II SPI slave and Host CPU SPI master.

Table 1 - Brooklyn II SPI slave and host CPU SPI master pins

Brooklyn II Pin (SPI Slave)	Host CPU Pin (SPI Master)	Usage
SPI_CLK_A / SPI_CLK_B	SPI CLK	SPI Clock

Brooklyn II Pin (SPI Slave)	Host CPU Pin (SPI Master)	Usage
SPI_MOSI_A / SPI_MOSI_B	SPI MOSI	SPI Master Out / Slave In
SPI_MISO_A / SPI_MISO_B	SPI MISO	SPI Master In / Slave Out
DATA_AVAILABLE (GPIO)	GPIO	Data is available to be read from Brooklyn II SPI slave

UART

A single UART peripheral is required to communicate with the Brooklyn II over a UART serial link. Data transferred over UART is full duplex. Figure 2 illustrates the hardware connections required to support communication over UART between the Brooklyn II and the host CPU.

The Host CPU must have 2x 1500 bytes of RAM to send and receive one max sized frame.

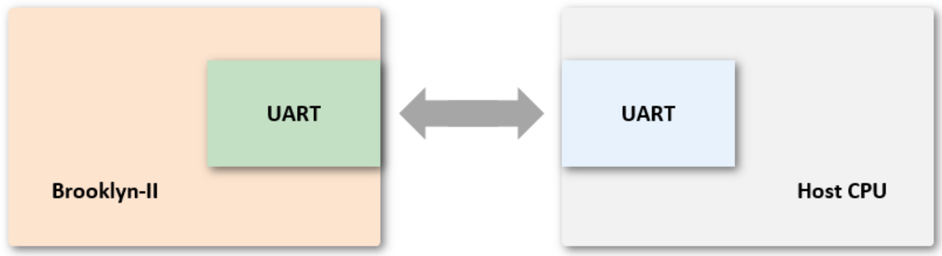

Figure 2 - Host CPU and Brooklyn II UART communication

Table 2 shows the pin mapping between the Brooklyn II UART and Host CPU UART.

Table 2 - Brooklyn II and host CPU UART pins

Brooklyn II Pin (UART)	Host CPU Pin (UART)	Usage
CMOS_RS_232_RX_A / CMOS_RS_232_RX_B	UART TX	Brooklyn-II receive, host transmit
CMOS_RS_232_TX_A / CMOS_RS_232_TX_B	UART RX	Brooklyn-II transmit, host receive

Communicating with an Ultimo

Regardless of the serial peripheral used for the communication the following requirements should be satisfied:

- 3.3V I/O signalling levels
- 2x 576 bytes RAM to send and receive a max sized packet over the serial link

SPI

Communication using SPI requires two SPI peripherals. One peripheral is a SPI master and the other peripheral is the SPI slave.

The SPI master on Ultimo must be connected to the SPI slave port on the host CPU. This unidirectional / simplex interface is used to send UHIP packets to / from the host CPU. The SPI slave on Ultimo must be connected to the SPI master port on the Host CPU. This unidirectional / simplex interface is used to send UHIP packets to / from the host CPU.

Figure 3 illustrates the hardware connections needed to support the SPI interface between Ultimo and a host CPU.

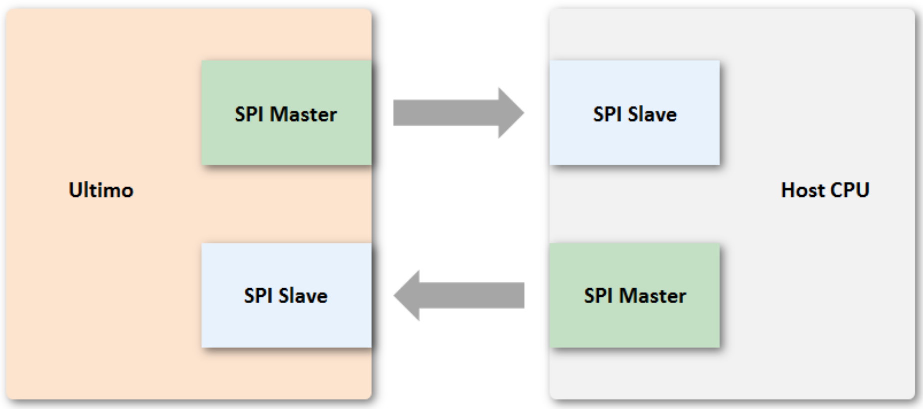

Figure 3 - Host CPU and Ultimo SPI communication

Table 3 - Ultimo SPI master and host CPU SPI slave pins

Ultimo Pin (SPI Master)	Host CPU Pin (SPI Slave)	Usage
SPI_CLK_A	SPI CLK	SPI Clock
SPI_MOSI_A	SPI MOSI	SPI Master Out / Slave In
SPI_MISO_A	SPI MISO	SPI Master In / Slave Out
nSPI_SEL0_A or nSPI_SEL1_A or nSPI_SEL2_A or nSPI_SEL3_A	SPI SELECT	SPI chip select

Table 4 - Ultimo SPI slave and host CPU SPI master pins

Ultimo Pin (SPI Slave)	Host CPU Pin (SPI Master)	Usage
SPI_CLK_B	SPI CLK	SPI Clock
SPI_MOSI_B	SPI MOSI	SPI Master Out / Slave In
SPI_MISO_B	SPI MISO	SPI Master In / Slave Out
nSPI_SEL_B	SPI SELECT	SPI chip select

For more details about the I/O pins, I/O characteristics, SPI master timing characteristics, SPI slave timing characteristics and supported SPI modes, please see the relevant sections of the Ultimo Technical Datasheet.

UART

Serial communication between an Ultimo and the host CPU requires one UART port.

The UART on Ultimo is a bi-directional full duplex interface used for sending UHIP datagrams to / from the host CPU.

Figure 4 illustrates the hardware connections needed to support the UART interface between Ultimo and the host CPU.

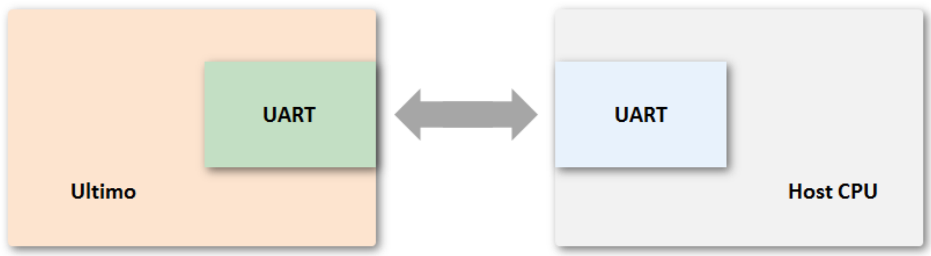

Figure 4 - Host CPU and Ultimo UART communication

Table 5 - Ultimo and host CPU UART pins

Ultimo Pin (UART)	Host CPU Pin (UART)	Usage
UART_RX_B	UART TX	Ultimo receive, host transmit
UART_TX_B	UART RX	Ultimo transmit, host receive

Ultimo Pin (UART)	Host CPU Pin (UART)	Usage

UART_CTS_B [optional]	UART RTS	Ultimo input / host output
		H = disable transmission from Ultimo
		L = enable transmission from Ultimo

UART_RTS_B [optional]	UART CTS	Ultimo output / host input H = disable transmission on host L = enable transmission on host

Host CPU Architecture

This section describes the architecture of the Host CPU and Dante device communication using a serial peripheral.

All functionality described in this section is fully implemented in the provided Host CPU SDK. The information contained in this section is for informational purposes only, and does not need to be implemented by the OEM.

The data transferred over the serial link is encoded using Consistent Overhead Byte stuffing (COBS) which provides packet framing. Framing ensures reliable resynchronisation after transmission or reception errors.

For more details about COBS see *Cheshire, S; Baker, M, Sept 1997, "Consistent Overhead Byte Stuffing", ACM*

All multi-byte values used for the protocols between the Host CPU and Brooklyn II or Ultimo are represented in network byte order. For example a 32-bit integer containing the value 305,419,896 (decimal) or 0x12345678 (hex) is sent as [0x12] followed by [0x34] followed by [0x56] followed by [0x78].

Key Features and Specifications

A very high level overview of the key features and specifications for serial communication between a Host CPU and a Brooklyn II or Ultimo is provided in Table 6. The protocols mentioned in Table 6 are discussed in the remaining sections of this document.

Table 6 - Host CPU to Dante device communication key features and specifications

Feature	Brooklyn II	Ultimo
Supported serial peripherals	SPI slave, UART	SPI master + SPI slave, UART
Supported high level protocols	DDP, Packet Bridge	DDP, Dante Events, Packet Bridge
Transport protocol	BHIP	UHIP
Maximum serial frame size	1508 bytes (SPI), 1500 bytes (UART)	576 bytes
Maximum transport protocol payload size	1484 bytes (for DDP), 1448 (for Packet Bridge)	500 bytes

Host CPU and Brooklyn II

Brooklyn II Host Interface Protocol (BHIP) is the protocol used to transport Dante Device Protocol (DDP) and Packet Bridge. The BHIP protocol does not have acknowledgements. Therefore, it will be up to clients of the DDP and/or Packet Bridge on the Host CPU retry if a response message was not received.

In typical deployments both DDP and Packet bridge will operate over the same serial peripheral. However it is also possible to configure the Brooklyn II to use one peripheral (e.g. SPI) for DDP and the other peripheral (e.g. UART) for the packet bridge. The intended use case for this is to connect two Host CPUs.

Framing and Padding Mechanisms

The framing of data differs when using SPI or UART. Figure 5 shows a single frame transferred on a SPI serial link and Figure 6 shows a single frame transferred on a UART serial link. The pipe number in the SPI header used for BHIP data is 0. Pipes 0-7 are reserved for Audinate use. If you wish to multiplex non-BHIP data over the SPI interface at the same time you may use any of pipes 8-15.

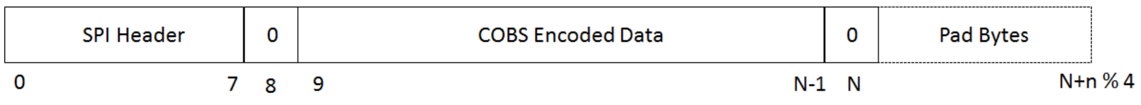

Figure 5 - Serial frame transferred between a host CPU and Brooklyn II over SPI

The frame transferred over a UART serial link does not use any padding so a Host CPU must not make any assumptions on the size of the frame.

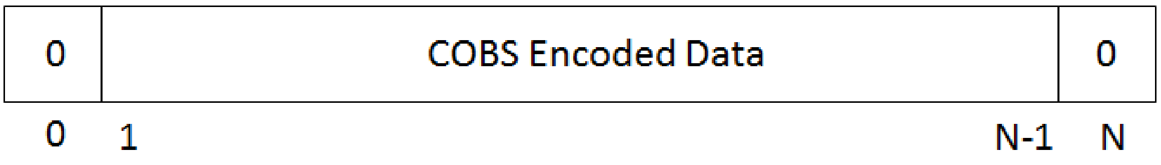

Figure 6 - Serial frame transferred between a host CPU and Brooklyn II over UART

Flow Control Mechanisms

Flow control is not provided by the BHIP protocol. However, the SPI slave on the Brooklyn II can be configured to provide flow control at the hardware level. The SPI slave on the Brooklyn II provides a flow control GPIO pin. This GPIO pin can be enabled via the module configuration tool. The flow control pin is asserted when the SPI slave buffer reaches the high water mark. For more information about the flow control pin please refer to the Brooklyn II Programmer's Guide. For UART mode there is no hardware level flow control from the Brooklyn II device.

Interaction Diagrams

Figure 7 shows a normal flow for BHIP regardless of the protocol that is being transported and the serial peripheral used for the communication. This interaction also applies when the Host CPU transmits a BHIP packet to the Brooklyn II.

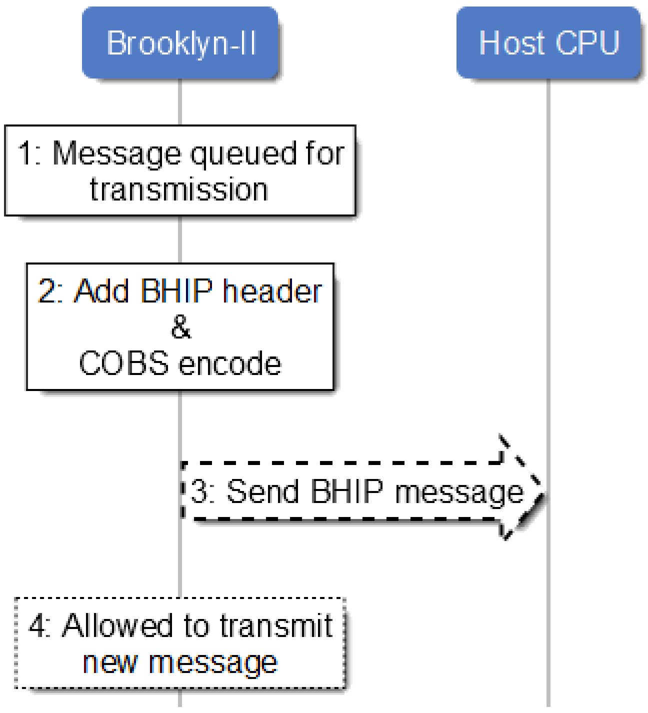
Brooklyn-II
Host CPU
1: Message queued for transmission
2: Add BHIP header & COBS encode
3: Send BHIP message
4: Allowed to transmit new message

Figure 7 - Normal flow behaviour for BHIP

Resynchronization Mechanisms

In rare situations where synchronization is lost on the receive path, the method to regain synchronization is based on the serial peripheral used for communication.

In SPI mode, data is transferred in data blocks which are a multiple of 4 bytes (minimum is 4 bytes). In addition, the SPI slave protocol has 4 sentinel bytes which always occur at the start of a 4 byte data block. Therefore, these sentinel bytes are used to re-acquire synchronisation.

If UART is used as the serial peripheral, data reception does not assume a data block size. Therefore, the COBS delimiters [0x00] are sufficient to regain synchronization.

Host CPU and Ultimo

Ultimo Host Interface Protocol (UHIP) is the protocol that is used to transport Dante Device Protocol (DDP), Dante Events and Packet Bridge packets. Each UHIP packet is encoded using Consistent Overhead Byte Stuffing (COBS). The UHIP protocol uses acknowledgements for flow control.

When the Ultimo is configured to communicate with a Host CPU all protocols transported by UHIP can only be transferred over a single serial peripheral. The Ultimo does not have the capability to transfer different UHIP protocols over different serial peripherals.

Framing and Padding Mechanisms

COBS is used to frame packets transferred over a serial link. A single frame sent over the serial link is shown in Figure 8.

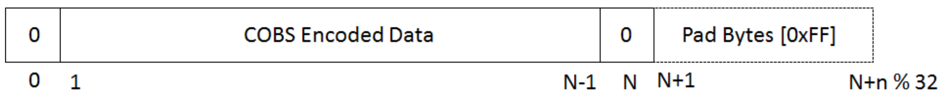

Figure 8 - Serial frame transferred between a host CPU and an Ultimo over SPI or UART

All packets transferred over the serial link are padded with [0xFF] bytes so that the packet size is a multiple of 32 bytes. This simplifies DMA driver implementations on the Ultimo and the host CPUs.

Flow Control Mechanisms

UHIP utilises a Stop-and-Wait mechanism for flow control on a per message basis. After a message has been sent, the sender must wait until either of the following occurs before sending another message:

1. A PROTOCOL_CONTROL acknowledgment has been received
2. No response is received for 1 second

Important: Due to limited RAM on the Ultimo, if a host CPU does not process UHIP messages quickly enough, UHIP messages will be dropped. As such it is important that received UHIP messages are processed in a timely manner.

Interaction Diagrams

Normal Acknowledgement

The interaction diagram shown in Figure 9 depicts a normal flow control behaviour when packets are acknowledged.

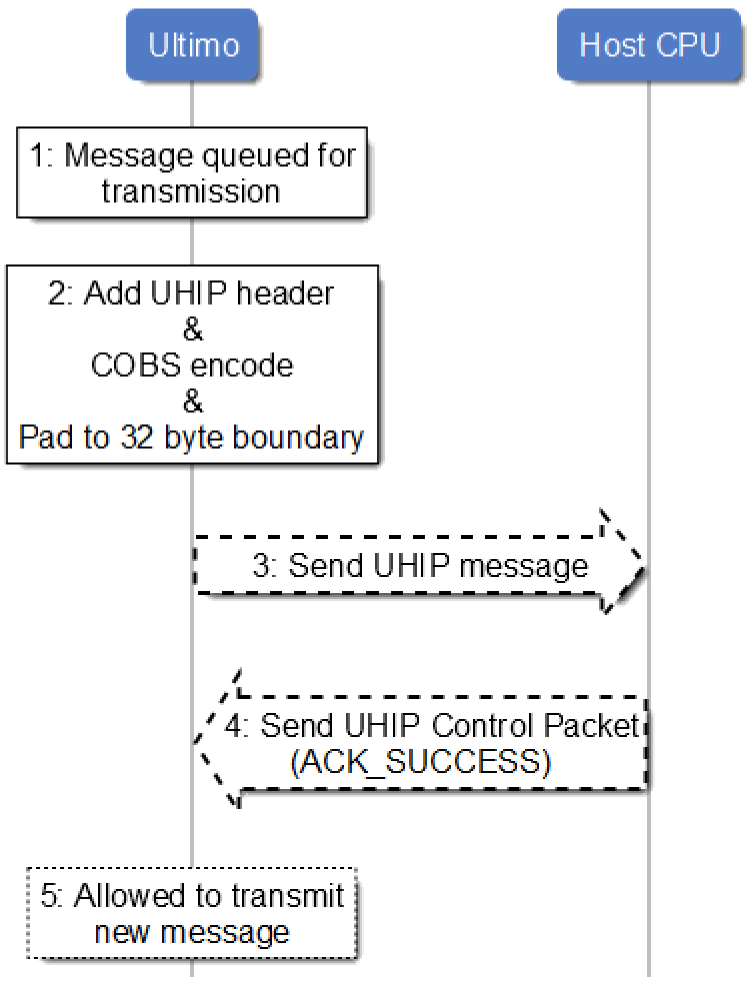
Ultimo
Host CPU
1: Message queued for transmission
2: Add UHIP header & COBS encode & Pad to 32 byte boundary
3: Send UHIP message
4: Send UHIP Control Packet (ACK_SUCCESS)
5: Allowed to transmit new message

Figure 9 - UHIP normal acknowledgement

Error Timeout (No Response from Host CPU)

The interaction diagram shown in Figure 10 depicts what occurs if a packet is transmitted from the Ultimo to the host CPU, but no acknowledgment is received.

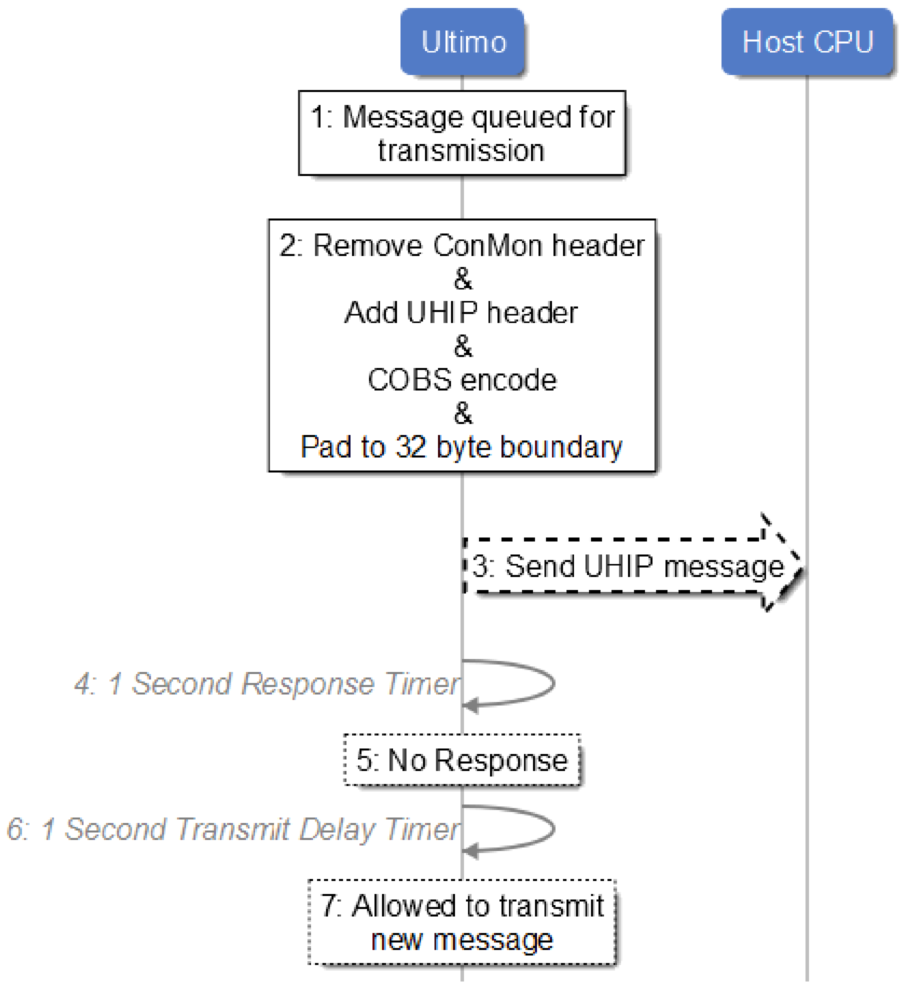
Ultimo
Host CPU
1: Message queued for transmission
2: Remove ConMon header & Add UHIP header & COBS encode & Pad to 32 byte boundary
3: Send UHIP message
4: 1 Second Response Timer
5: No Response
6: 1 Second Transmit Delay Timer
7: Allowed to transmit new message

Figure 10 - UHIP timeout error due to no response from host CPU

Error Timeout (No Response from Ultimo)

The interaction diagram shown in Figure 11 depicts what occurs if a packet is transmitted from the host CPU to the Ultimo, but no acknowledgment is received.

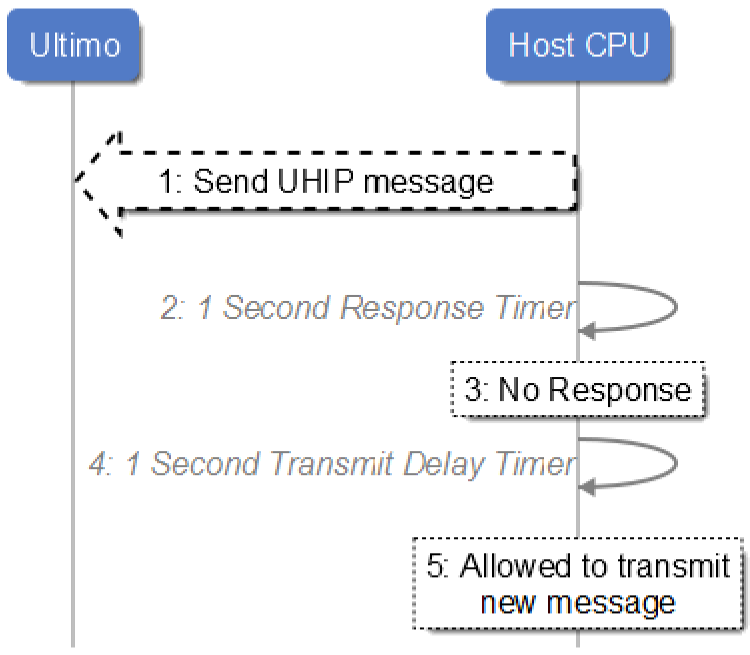
Ultimo
Host CPU
1: Send UHIP message
2: 1 Second Response Timer
3: No Response
4: 1 Second Transmit Delay Timer
5: Allowed to transmit new message

Figure 11 - UHIP timeout error due to no response from the Ultimo

Timing and Resynchronization Mechanisms

UHIP uses the following strategies to regain synchronisation after communication errors.

- Receive Path for COBS encoded, 32 byte padded packets:

1. Start/reset a 1 second timer every time a 32 byte chunk is received
2. If a full COBS encoded message is received, stop the timer
3. If the timer expires (1 second without receiving the end of a COBS packet), reset the receive buffer and/or DMA controller

- Transmit Path for COBS encoded, 32 byte padded packets:

1. When transmitting a UHIP message to the Host CPU.
2. Start a 1 second timer when the message is transmitted.
3. If the timer expires without a response (i.e. PROTOCOL_CONTROL) packet being received.
4. Wait another 1 second before sending another UHIP message to the Host CPU.

Packet Bridge

The Packet Bridge protocol allows a Host CPU to send and receive UDP/ IP network datagrams via the IP stack on a Brooklyn II or Ultimo device. This functionality will allow OEMs to quickly and easily implement custom network control, status and monitoring functionality on their Brooklyn II or Ultimo reference design product and PC software. The Packet Bridge should only be used for applications that have a low throughput and packet rate.

Key Features and Specifications

Table 7 summarizes the main features related to Packet Bridge for the Brooklyn II and the Ultimo.

Table 7 - Packet Bridge key features for Brooklyn II and Ultimo

Feature	Brooklyn II	Ultimo
Supported modes	Dante Control and Monitoring (ConMon: control and status) or standard UDP sockets	Dante Control and Monitoring (ConMon: control, status, and vendor broadcast) or standard UDP sockets
Maximum message payload	1448 bytes	500 bytes
ConMon Unicast subscriptions limit	16	5

Overall Architecture

The Packet Bridge uses UDP/IP messages to communicate across the network. The Packet Bridge can be configured in one of two modes - ConMon or UDP sockets; and can be enabled on the Dante device using the module configuration tool.

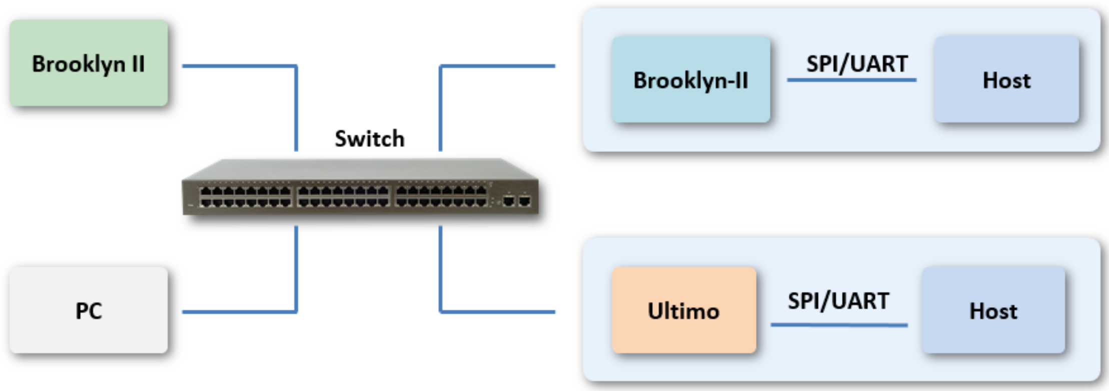

Figure 12 - Packet Bridge architecture applicable for a Brooklyn II or an Ultimo

ConMon Packet Bridge

When the Packet Bridge is configured in ConMon mode, packets are forwarded to devices as vendor specific ConMon messages. ConMon includes infrastructure that enables discovery of Brooklyn II or Ultimo Packet Bridge endpoints. Please refer to the Dante API Programmer's Guide for further details on ConMon.

The Packet Bridge endpoint forwards received messages to the Host CPU. Responses from the Host CPU forward back to the sending device. The Ultimo or Brooklyn II running the Packet Bridge cannot initiate messages. The Ultimo supports a limited method for transmitting message from the Packet Bridge endpoint using multicast.

Equivalent functionality is not implemented Brooklyn II, however it is possible to build a custom user application that is able to communicate using unicast messages with up to 16 devices. Please refer to the Brooklyn II Programmer's Guide on information how to build a custom user application.

The ConMon vendor ID in the ConMon header is the manufacturer-specific ID assigned to each OEM by Audinate. The ConMon packet bridge filters incoming packets based on this vendor ID to ensure that only the messages tagged with the specific vendor ID are sent over the packet bridge. If more precise filtering is required it is the responsibility of the vendor to implement this on the host processor.

The Brooklyn II Packet Bridge supports the control channel and status channel. The Ultimo Packet Bridge supports the control channel, status channel and vendor broadcast channel.

- Control Channel - unicast receive channel. This channel is used to send control or query packets from a PC or Brooklyn-II based controller to a Brooklyn II or Ultimo packet bridge endpoint.
- Status Channel - multicast and unicast transmit channel. This channel is used to send status messages or query responses from a Brooklyn II or Ultimo packet bridge endpoint to PC or Brooklyn II based controllers. Remote devices subscribed to the Ultimo may receive messages via unicast. The maximum number of active unicast subscriptions supported by Brooklyn II is 16 whereas Ultimo is 5.
- Vendor Broadcast Channel - multicast transmit channel from the Ultimo, optional multicast receive channel from a remote device. This channel is typically used for Ultimo-to-Ultimo communication in small networks that don't have a larger "controller" device such as a Brooklyn-II or PC.

Note: Unicast Ultimo to Ultimo ConMon Packet Bridge messages are not supported.

UDP Sockets Packet Bridge

When the Packet Bridge is configured in UDP mode, the Dante device allows for UDP datagrams to be sent/received by the Host CPU via the Packet Bridge.

Note: If the Brooklyn II is configured in redundant mode, Packet Bridge in UDP sockets mode only operates over the primary interface.

The following types of channels are supported in UDP socket Packet Bridge mode:

- Unicast Channel - unicast receive and transmit channel. This channel is typically used to send control or query packets from a PC or Brooklyn-II based controller to Brooklyn II or Ultimo Packet Bridge endpoints. It is also used to send status messages or query responses from a Brooklyn II or Ultimo packet bridge endpoint to PC or Brooklyn-II based controllers.
- Multicast Channel - multicast transmit channel from the Brooklyn II or Ultimo Packet Bridge endpoint and optional multicast receive channel from a remote device. This channel should only be used in small networks. It is preferable to use a Brooklyn II or PC running a custom application as a controller.

Comparison of ConMon and UDP Sockets Packet Bridge

Table 8 - Comparing Features of ConMon and UDP Sockets Packet Bridge Modes

Feature	ConMon Packet Bridge	UDP Sockets Packet Bridge
Device Discovery	Automatic	Manual (OEM implemented)
Handling IP address change	Automatic	Manual (OEM implemented)
Packet Filtering	Automatic	Manual (OEM implemented)
Unicast Subscriptions	Yes (up to 16 for Brooklyn II and up to 5 for Ultimo)	Manual (OEM implemented)
Redundancy (Brooklyn II only)	Yes	No

Communication with a Dante Device Packet Bridge Endpoint from PC / Brooklyn II Devices

Message Format

ConMon Mode

All the messages to and from the Brooklyn II or Ultimo Packet Bridge endpoint are encapsulated in a ConMon message header. The ConMon vendor ID will be set to the manufacturer specific vendor ID assigned by Audinate. The Dante device Packet Bridge endpoint compares this ID to the manufacturer ID set in the capability and forwards the packets over the packet bridge if they match.

UDP Mode

When configured in UDP Packet Bridge mode, it is the responsibility of the OEM to add any necessary OEM message headers to the UDP payload. In UDP mode, the Brooklyn II or Ultimo Packet Bridge endpoint forwards the received UDP payload over the serial interface to the Host CPU. Messages received from the Host CPU are sent onto the network as UDP packets.

Discovering Packet Bridge Endpoints

ConMon Mode

A ConMon "controller" application on a Brooklyn II or PC can use the ConMon Manufacturer Versions message to identify devices matching a particular vendor ID. The Dante browsing API provides the following additional device information:

- Vendor ID - vid, allows the controlling application to select a device matching a particular vendor
- Vendor broadcast address - vba, The multicast address used by the device for the Vendor Broadcast Channel. The vendor broadcast address is only applicable for Ultimo Packet Bridge endpoints

Any ConMon messages with a matching Vendor ID will be forwarded over the Brooklyn II or Ultimo Packet Bridge. Audinate Vendor ID tagged messages will be handled by Dante application threads inside Brooklyn II or Ultimo as usual.

UDP Mode

When the Packet Bridge is configured in UDP mode, mDNS is used to discover the UDP Packet Bridge endpoints. The Brooklyn II and Ultimo provides configurable SRV, TXT and PTR mDNS records for use by the OEM and it is configurable via the module configuration tool.

A mDNS advert consists of a PTR, SRV, and TXT record. It allows for a "named service" on a particular port to be advertised and discovered. It consists of:

- PTR or Pointer Record - to enables the discovery of the "named" service on a particular device
- SRV or Service Record - to provide information about the "host" and "port" for the service
- TXT or Text Record - to provide additional information about the service

In the module configuration tool the OEM can set the "service name" and "port" as well as multiple keys in the Text Record.

- The "service name" is the OEM assigned name for the UDP packet bridge service
- The "port" is the OEM assigned listening UDP port for the unicast channel
- The "text keys" are extra information that the OEM wants to provide about the UDP packet bridge service on the Ultimo. For example it may be useful to provide protocol versions, multicast IP addresses or multicast port information in the Text Record

For example:

- UDP Packet Bridge service is called "oem-pb"
- The Unicast receive socket is bound to port "39030"
- The Multicast receive socket is bound to 239.254.50.123 / port 9000
- The protocol version is 1.2.3.4
- An Ultimo device is used and it is called Ultimo-0712b3

This will result in the following mDNS records:

- `Ultimo-0712b3._oempb._udp.local. 120 IN SRV 0 0 39030 Ultimo-0712b3.-local.`
- `_oempb._udp.local. 4500 IN PTR Ultimo-0712b3._oempb._udp.local.`
- `Ultimo-0712b3._oembp._udp.local. 4500 IN TXT ver=1.2.3.4 mc_ip=239.254.50.123 mc_port=9000`

Please refer to Dante API Programmer's Guide - DNS Service Discovery API section for information on how to discover the packet bridge endpoint via mDNS.

For further information about mDNS please refer to http://www.multicastdns.org/

For an Apple implementation of mDNS that is commonly used on both OS X and Windows please refer to http://www.apple.com/au/support/bonjour/

ConMon Control Channel Usage

The Control Channel carries control and query messages inbound to the Host CPU . The controller application on a PC or Brooklyn- II can use this channel to send control or query messages to the host CPU via the Packet Bridge. The Brooklyn II or Ultimo packet bridge endpoint cannot send messages using this channel.

ConMon Status Channel Usage

The Status Channel is designed for one-to-many communication. This channel type is always a multicast channel, with support for unicast transmission to a limited list of subscribed devices. Any message received from the Packet Bridge tagged as a ConMon status message is multicast on the Status Channel.

A PC can use a 'global' subscription to receive multicast status messages. A Brooklyn II controller can use unicast to subscribe to up to 32 devices. A Brooklyn II packet bridge endpoint can support up to 16 remote "controllers" subscribed via unicast. An Ultimo endpoint can support 5 remote subscribers.

ConMon Vendor Broadcast Channel Usage

The Vendor Broadcast channel is typically used for Ultimo-to-Ultimo communication. It is designed for one-to-many communication. This channel type is enabled/configured locally on each device that transmits or receives on this channel.

The ConMon advertisement can be used to discover devices with a specified vendor ID and Vendor Broadcast Channel multicast address. This channel can be used to both transmit and receive messages. Care must be taken by the vendor to select multicast IP addresses that are not currently assigned. The range recommended by Audinate is 239.254.0.0/16.

For more information about multicast IP address assignment see RFC5771 & RFC2365.

Important: When opening a multicast receive channel on an Ultimo, all packets sent by other devices will be received and must be processed by both the Ultimo and the host processor. Care must be taken in designing the transmission side of the vendor protocol (specifically the number of devices and how often they transmit) not to overwhelm or overburden the Ultimo and host processor with receive packets!

Note: Because multicast is a one-to-many communication medium, it is the responsibility of the vendor to add information identifying the sender to the transmitted packets. A MAC address or other unique device identifier may be appropriate.

Example Use Cases and Recommended ConMon Configurations

This section describes Packet Bridge use cases and appropriate ConMon channel type selection.

Host CPU Firmware Update by a PC Controller

Figure 13 shows a PC controller being used to update the firmware on the host processor via the Packet Bridge.

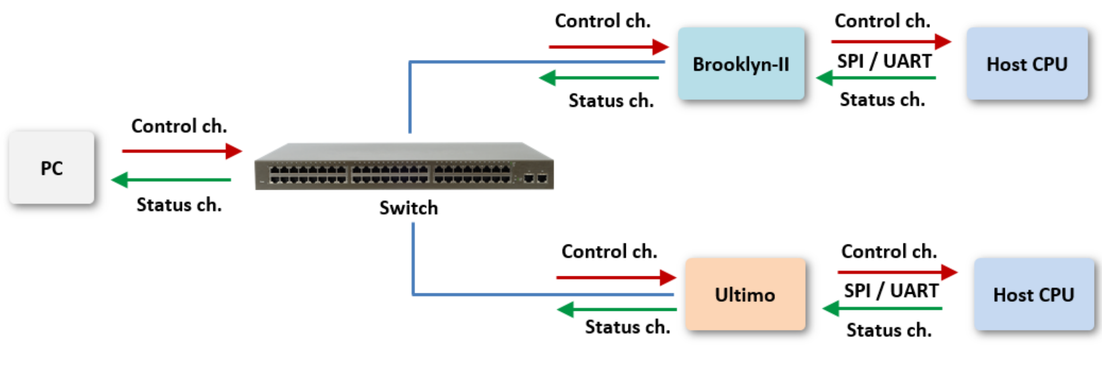

Figure 13 - Host CPU firmware upgrade by a PC controller

In this example, the PC controller sends vendor defined firmware update packets via the unicast ConMon Control Channel to the Brooklyn II or Ultimo which are forwarded to the host CPU according to the packet bridge configuration. The host CPU (via the Brooklyn II or Ultimo) then sends back status update and response messages via the multicast ConMon Status Channel to the PC.

Control of Host CPU by a Brooklyn II Based Controller

Figure 14 shows a Brooklyn II based controller sending commands and/or queries to the host processor via Packet Bridge.

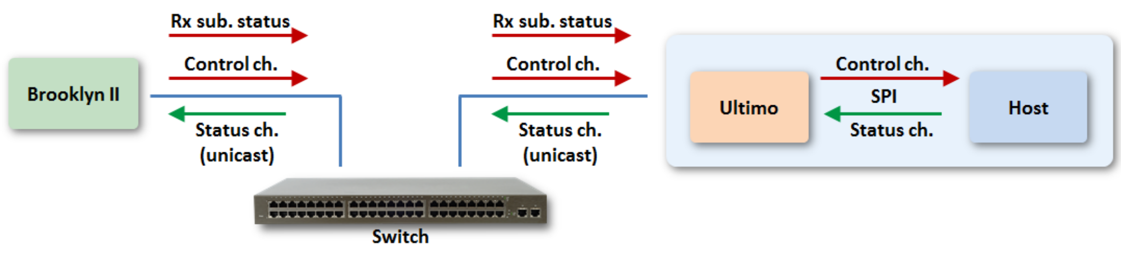

Figure 14 - Host CPU controlled by Brooklyn II controller

In this example, the Brooklyn II based controller sends control packets via the unicast Control Channel to the Ultimo, which is configured as a Packet Bridge to the host CPU. The Brooklyn II based controller also opens a remote subscription to the Status Channel on the Ultimo, which causes Ultimo to transmit unicast ConMon Status Channel messages from the host CPU to the Brooklyn II controller. The host CPU sends status messages and responses to the Brooklyn II controller via the Ultimo Packet Bridge and the unicast subscription to the Status Channel.

Status Reporting by Host CPU to PC Controllers

Figure 15 shows multiple host CPUs reporting status information to one or more PC controllers via Packet Bridge.

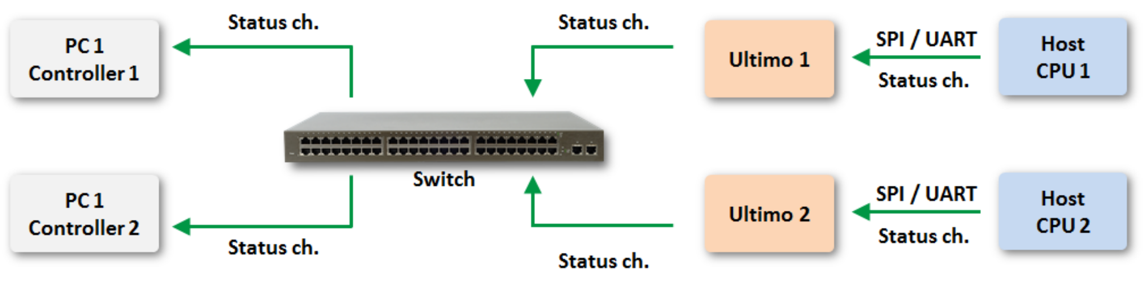

Figure 15 - Host CPU status reporting to PC controllers

In this example, all Ultimo and host CPUs transmit status information packets via the multicast Status Channel to one or more PC based controller applications. The host CPU on each Ultimo device sends periodic and/or event driven status messages via Packet Bridge to the multicast ConMon Status Channel. The PC controller listens to these status messages using the multicast ConMon Status Channel.

Host CPU to Host CPU Communications

Note: This use case can only be realized with Ultimo packet bridge endpoints.

Figure 16 shows inter-device communication between host processors connected via Packet Bridge.

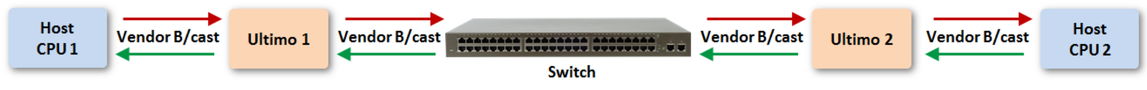

Figure 16 - Host CPU to host CPU communications

In this example, each Ultimo has been configured with the same Vendor Broadcast Channel using the same multicast IP address, which is used to transmit messages from the host CPU to the network.

Each Ultimo has also been configured to receive messages for the Packet Bridge from the Vendor Broadcast Channel. This allows the Vendor Broadcast Channel to be bridged from the network to the host CPU.

Hosts CPUs can transmit messages to other host CPUs via the Ultimo Packet Bridge and the multicast ConMon Vendor Broadcast Channel. Ultimo devices configured with the same vendor allocated multicast address will receive messages.

Important: When opening a multicast receive channel on an Ultimo, all packets sent by other devices will be received and must be processed by both the Ultimo and the host CPU. Care must be taken in designing the transmission side of the vendor protocol (specifically the number of devices and how often they transmit) not to overwhelm or overburden the Ultimo and host CPU with receive packets!

Important: Because multicast is a one-to-many communication medium, it is the responsibility of the vendor to add information identifying the sender to the transmitted packets. A MAC address or other unique device identifier may be appropriate.

Recommendations for OEM Protocol Design

Managing Device State

A device designed to control other devices typically needs to track the state of the devices it manages. To track device state, a controller subscribes to Status Channels and receives messages as parameters change or events occur. There may also be multiple controllers for a given device.

How should device state and control messages be structured? The paragraphs below discuss various possibilities open to the designer.

Self-Description

Devices that can describe their own features and capabilities can greatly simplify the task of writing general-purpose control software. The simplest approach is to define a single message type supported by all devices that describes the capabilities of the device. Controllers can use this information to determine which behaviours and features are applicable for the device.

For example, the ConMon VERSIONS_STATUS message provides a range of information about the types of ConMon messages available for a given Dante device.

Bulk State Summary

When a controller starts up, it does not know the state of the devices it is controlling. In combination with a control channel message to trigger it, a bulk state summary message provides an efficient means of bootstrapping the controller. When sent on the Status Channel, all other controllers will receive this message.

The bulk state summary may also be used as a control message for restoring a device to a particular configuration.

Small State Changes

Generally, a human can adjust one or only a small number of parameters at once. Therefore, support for small, single parameter state change messages appears sensible.

Groups of Changes

Metering information is a classic example of status information that ought to be collected together for transmission. Meter values change periodically and grouping several meter values together will result in more efficient network transmission.

If several configuration or status parameters always change together, they should be transmitted together. If parameters are transmitted in different messages, there is the chance that one of the messages will not be successfully received. Grouping parameters together results in a set of parameter changes to be received or lost as a group.

Stateless Protocol Design

In general, control or monitoring messages should be understandable on their own. If message can be interpreted properly without additional state, there is no burden on the receiver to track any extra state.

As an example, a control or status message that indicates:

`PhantomPower = On` or `PreampGain = -17.1dB`

Is better than a message like:

`PhantomPower = Toggle` or `PreampGain += 5.6dB`

If the second style of message is retransmitted, or received more than once because of redundancy, there is potential confusion about the final device state, which can only be resolved by accurately tracking every change.

Stateless messages have a well-defined meaning or result. With care, a stateless protocol design can be robust against retransmission and loss without requiring complex state tracking.

Avoiding Synchronization

A network containing devices that periodically send messages can self-synchronise, resulting in large periodic bursts of network traffic. Consider randomly jittering the time at which periodic messages are sent using the techniques described in Sally Floyd, Van Jacobson, “The synchronization of periodic routing messages in Computer Networks”, 1994, IEEE/ACM Transactions on Networking.

Byte Ordering

Processors can arrange bytes in memory in different ways and this must be considered when copying data (e.g. a 32 bit integer) into a buffer for transmission. A control protocol should clearly define the transmission order of bytes to avoid confusion. It is strongly recommended that you use “network byte order” or “big endian” transmission, as it is commonly used for protocols.

Byte Alignment

Processing misaligned data is inefficient on some processors and not supported on others. Ensuring that all 16 and 32 bit message fields are correctly aligned with the start of the message makes it easier to process a message once it has been copied into application memory.

Detecting Message Loss

Incrementing sequence numbers on status and metering messages allow the receiver to tell if it has missed a message. In the case of metering, missing the odd message may be unimportant. For Status Channels, recovery may involve provoking the transmission of a full or partial state summary.

A periodic, low rate, keepalive status message can be used to bound the time taken to detect that an infrequent status messages is missing. A keepalive may also be useful if a controller is required to detect device disappearance in a timely manner - however, but note that the regular transmission of keepalives consumes network bandwidth.

Control messages may be acknowledged using a status message. Again, stateless protocol design pays off, as an acknowledgement may be lost even when the original message has been successfully received.

Power-On Events

Protocol designers must carefully consider the behaviour of many devices powering on at the same (or nearly the same) time. If several control devices start up at the same time and all want to acquire the current state for the devices on the network, this can result in a flood of network traffic.

To mitigate the effects of power-on events, consider the following strategies. Note: It is assumed there are vendor defined control messages describing the “device state summary” and for triggering its transmission on the Status Channel.

Status channel receiver behaviour:

1. Subscribe to Status Channel for the device.
2. Randomly wait (e.g. uniformly distributed between 0 and 3 seconds) whilst receiving Status Channel messages.

If a full “device state summary” is received before timeout, another device requested the same information, and there is no need to request it again.

If a full “device state summary” is not received before timeout, send a control message to trigger the “device state summary” transmission on the Status Channel.

Status channel sender behaviour:

- Rate limit “status summary” messages. Is sending more than one per second worthwhile?

Interaction Diagrams

Transmit a Packet from a Host CPU to the Network [Successful]

Brooklyn II

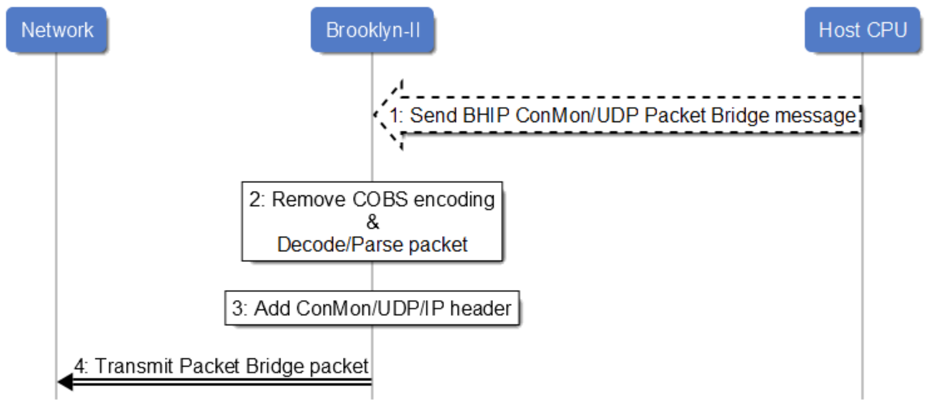
Network
Brooklyn-II
Host CPU
1: Send BHIP ConMon/UDP Packet Bridge message
2: Remove COBS encoding & Decode/Parse packet
3: Add ConMon/UDP/IP header
4: Transmit Packet Bridge packet

Figure 17 - BHIP ConMon/UDP packet bridge transmit success

Ultimo

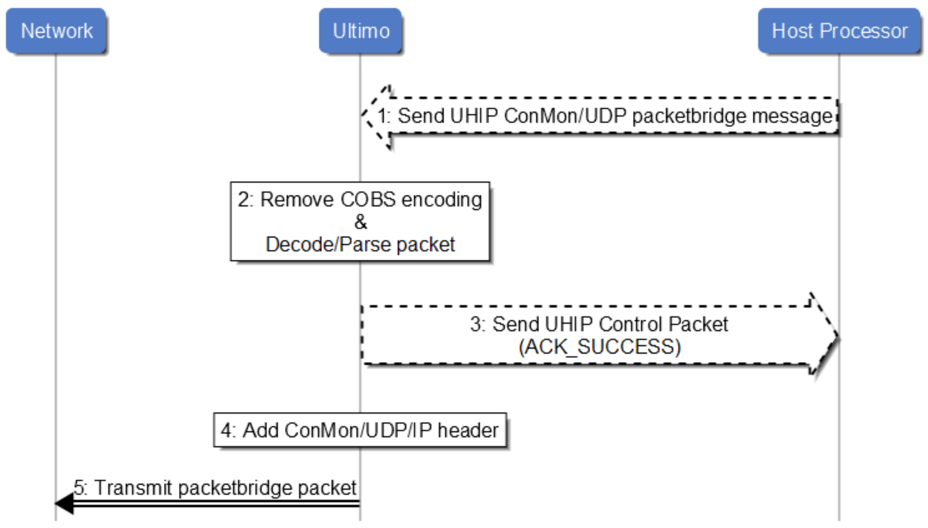
Network
Ultimo
Host Processor
1: Send UHIP ConMon/UDP packetbridge message
2: Remove COBS encoding & Decode/Parse packet
3: Send UHIP Control Packet (ACK_SUCCESS)
4: Add ConMon/UDP/IP header
5: Transmit packetbridge packet

Figure 18 - UHIP ConMon/UDP packet bridge transmit success

Transmit a Packet from a Host CPU to the Network [Failure - No Network Connection]

Brooklyn II

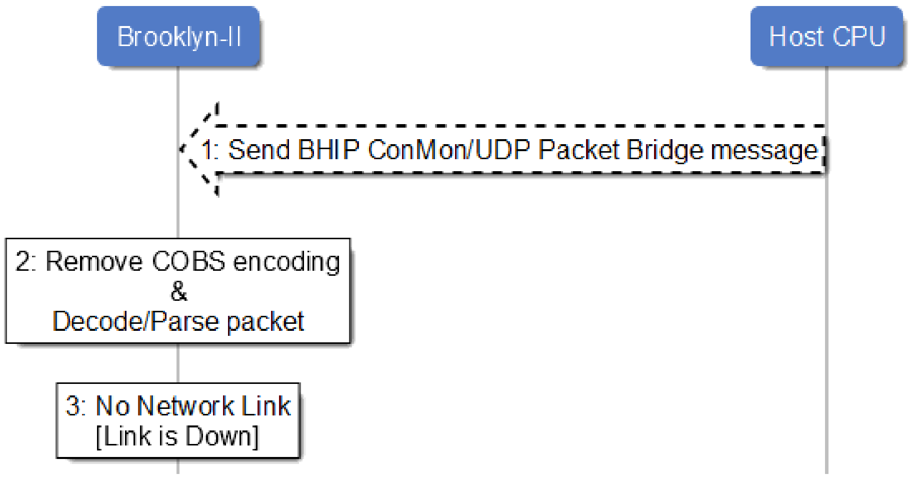
Brooklyn-II
Host CPU
1: Send BHIP ConMon/UDP Packet Bridge message
2: Remove COBS encoding & Decode/Parse packet
3: No Network Link [Link is Down]

Figure 19 - BHIP ConMon/UDP packet bridge transmit failure due to no network connection

Ultimo

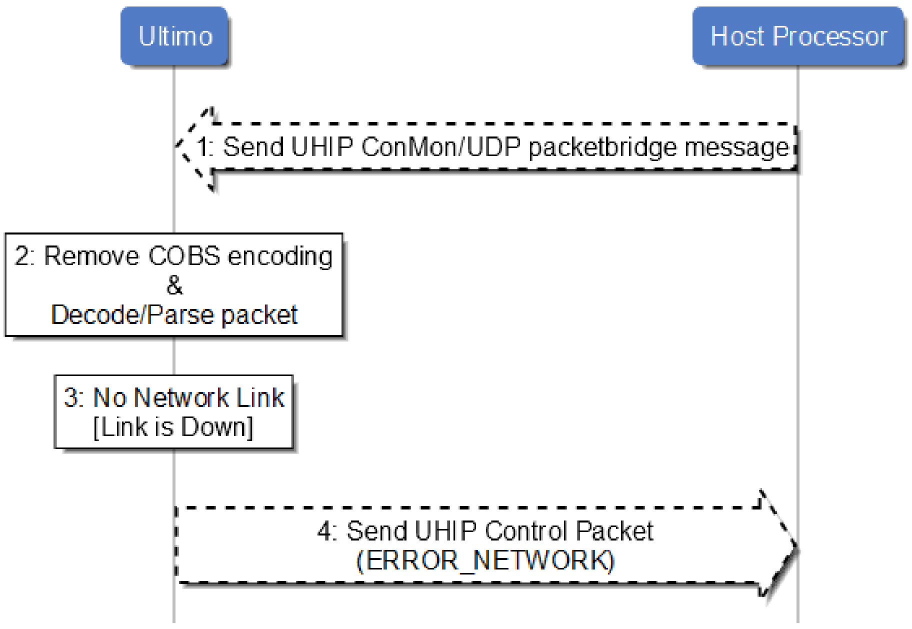
Ultimo
Host Processor
1: Send UHIP ConMon/UDP packetbridge message
2: Remove COBS encoding & Decode/Parse packet
3: No Network Link [Link is Down]
4: Send UHIP Control Packet (ERROR_NETWORK)

Figure 20 - UHIP ConMon/UDP packet bridge no network connection transmit failure

Transmit a Packet from a Host CPU to the Network [Failure - Message Malformed]

Brooklyn II

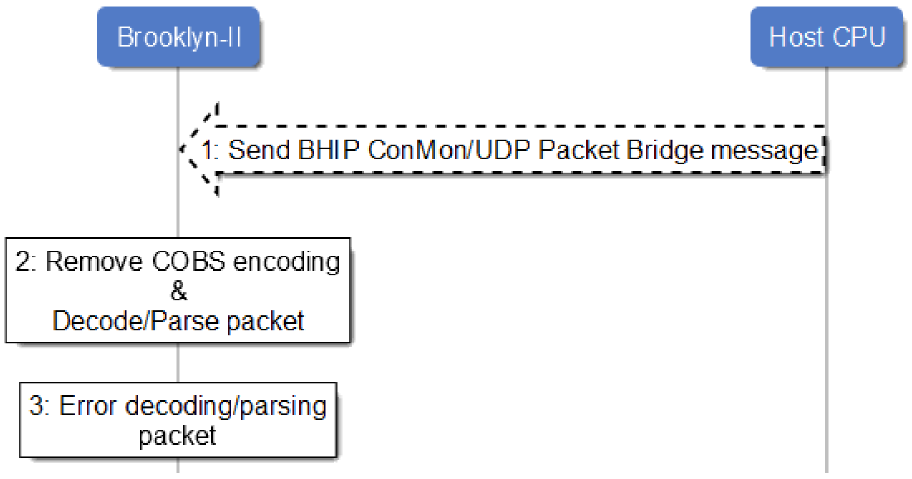
Brooklyn-II
Host CPU
1: Send BHIP ConMon/UDP Packet Bridge message
2: Remove COBS encoding & Decode/Parse packet
3: Error decoding/parsing packet

Figure 21 - BHIP ConMon/UDP packet bridge transmit failure due to malformed message

Ultimo

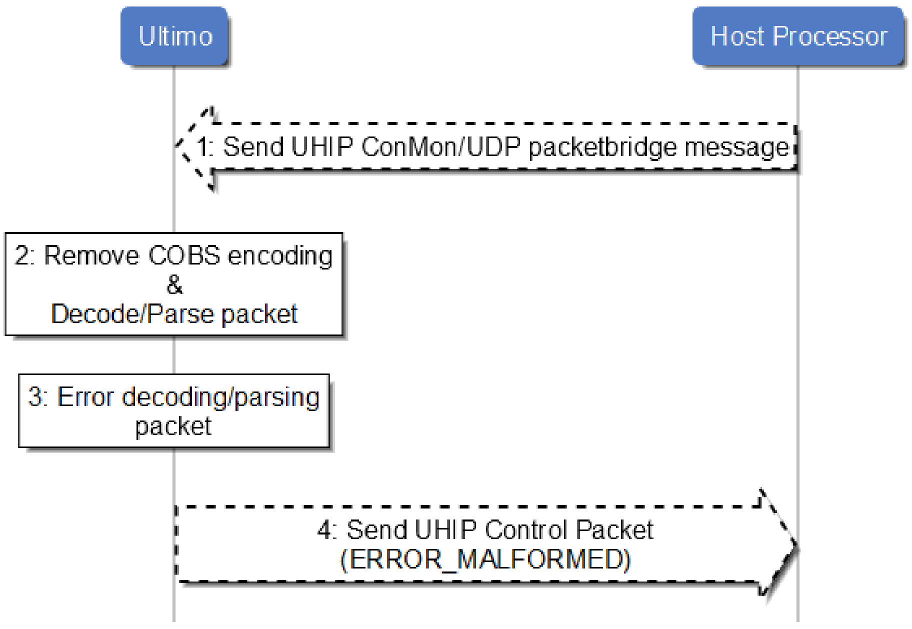
Ultimo
Host Processor
1: Send UHIP ConMon/UDP packetbridge message
2: Remove COBS encoding & Decode/Parse packet
3: Error decoding/parsing packet
4: Send UHIP Control Packet (ERROR_MALFORMED)

Figure 22 - UHIP ConMon/UDP packet bridge transmit failure due to malformed message

Receive a Packet from the Network and Forward to Host CPU [Successful]

Brooklyn II

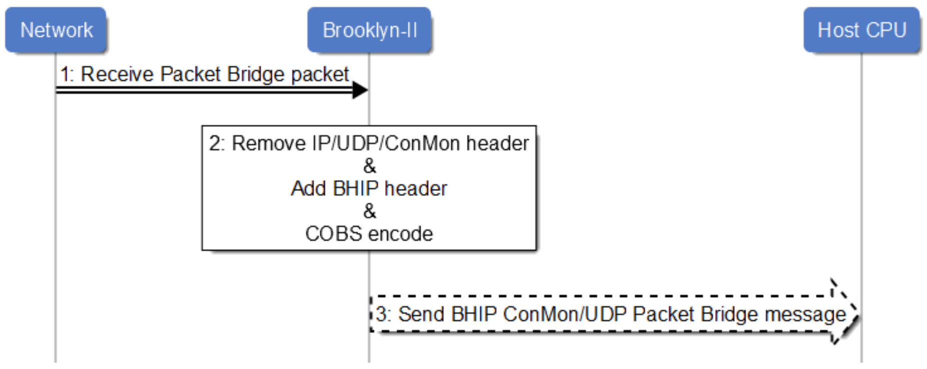
Network
Brooklyn-II
Host CPU
1: Receive Packet Bridge packet
2: Remove IP/UDP/ConMon header & Add BHIP header & COBS encode
3: Send BHIP ConMon/UDP Packet Bridge message

Figure 23 - BHIP ConMon/UDP packet bridge receive success

Ultimo

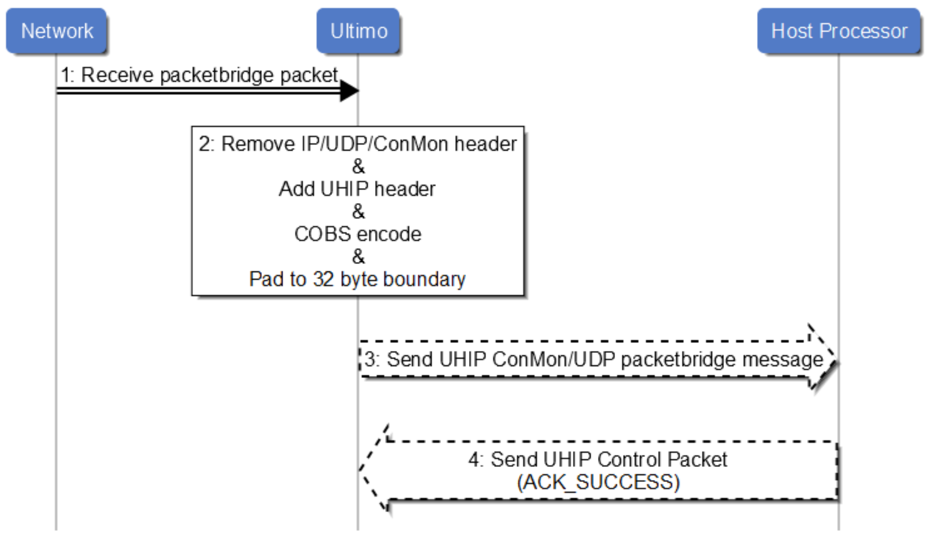
Network
Ultimo
Host Processor
1: Receive packetbridge packet
2: Remove IP/UDP/ConMon header & Add UHIP header & COBS encode & Pad to 32 byte boundary
3: Send UHIP ConMon/UDP packetbridge message
4: Send UHIP Control Packet (ACK_SUCCESS)

Figure 24 - UHIP ConMon/UDP packet bridge receive success

Dante Device Configuration

SPI and UART port parameters including baud rates, clock polarity, etc. for Brooklyn II and Ultimo are configured using the module configuration tool. Configuration and selection of the ConMon channels used with Packet Bridge or UDP sockets is also specified using the module configuration tool.

Dante Events

The Dante Event messages are sent by the Ultimo processor when the sample rate or pull up is changed. The Brooklyn II does not support Dante Events. These events are used to inform the Host CPU that the codec(s) need to be reconfigured for the new sample rate / pull-up. For more information see Sample Rate Changes and Pull-Up / Down Changes sections in the Ultimo Programmer's Guide.

Interaction Diagrams

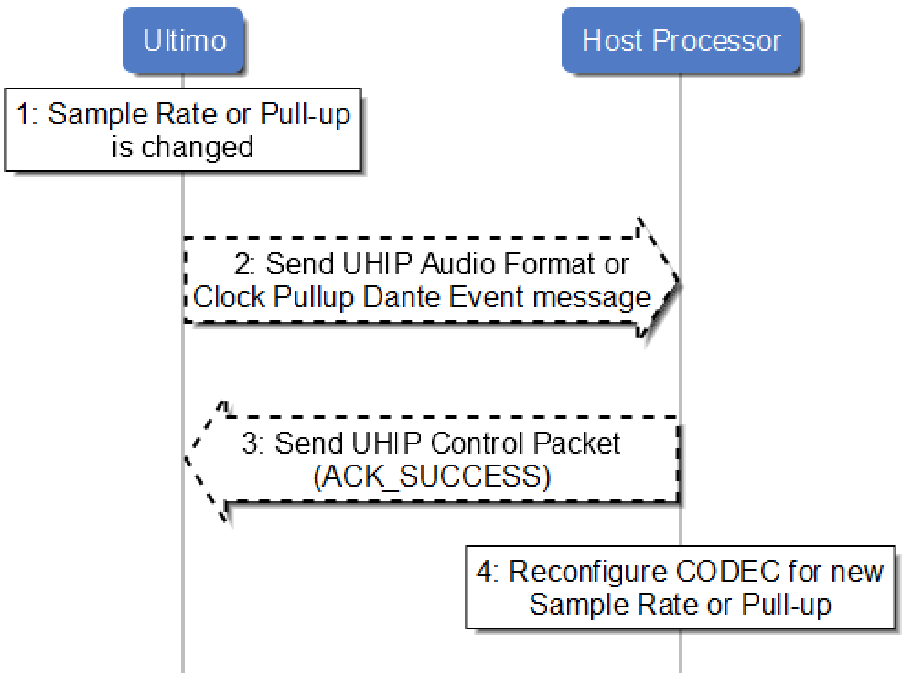
Ultimo
Host Processor
1: Sample Rate or Pull-up is changed
2: Send UHIP Audio Format or Clock Pullup Dante Event message
3: Send UHIP Control Packet (ACK_SUCCESS)
4: Reconfigure CODEC for new Sample Rate or Pull-up

Figure 25 - Dante Events sent as a result of sample rate or pullup changes

Ultimo Configuration

To enable Dante Events on the Ultimo the following options should be configured in module config web Host CPU section:

1. Host CPU checkbox should be ticked
2. Mode should be selected as either set to SPI or UART. Based on the selection the serial peripheral should be configured as appropriate

Dante Device Protocol

The Dante Device Protocol (DDP) allows a Host CPU to control and monitor the Dante functionality on a Dante device. This functionality will allow OEMs to provide Dante status information or control the Dante device via a user interface locally on the device.

Message Classes

There are several classes of messages used by the Dante Device Protocol:

1. Request messages - Request messages are sent to a Dante device and are used to either query for information from a device or to control or change parameters on a device. Every request message has:

- A message opcode
- A sequence number used to identify the request. This must be a non-zero value. The reply to this request will always use the sequence number from the request. This allows for a request/response pair to be linked.

2. Response messages - Response messages are sent from a Dante device in response to a request. Every response message has:

- An opcode that is the same as the corresponding request opcode
- A non-zero sequence number matching the request sequence number
- A status that indicates whether the request was successful or an error occurred

3. Event messages - Event messages are pushed or sent gratuitously from a Dante device when there is a change in status. Event messages are identical in format and opcode to a response event with the following key differences:

- A zero sequence number

Protocol Functionality

This section provides a summary of the functionality provided by Dante Device Protocol.

Basic Information

Mechanisms: Query/Response & Asynchronous Notification (on power on)

- Model ID
- Model ID string
- Software Version & Build
- Firmware Version & Build
- Bootloader Version & Build
- Configuration & Memory Error flags (capability partition error / user partition error / configuration store error)

Manufacturer Information

Mechanisms: Query/Response & Asynchronous Notification (on power on)

- Manufacturer ID
- Manufacturer Name
- Model ID
- Model Name
- Model Version
- Software Version & Build
- Firmware Version & Build
- Bootloader Version & Build
- Configuration & Memory Error flags (capability partition error / user partition error / configuration store error)

Upgrade

Mechanisms: Command/Response & Asynchronous Notification (after an upgrade)

- Start an upgrade via TFTP
- Start an upgrade via XMODEM via SPI/UART (Ultimo only)
- Receive progress notifications after the upgrade has completed

Erase Configuration

Mechanisms: Command/Response & Asynchronous Notification (during an erase)

- Erase to factory defaults
- Erase to factory defaults but keep static IP information
- Receive a notification when a erase is externally triggered

Device Reboot

Mechanisms: Command/Response & Asynchronous Notification (before reboot)

- Trigger a software reset
- Receive a notification when a reboot is externally triggered

Identity

Mechanisms: Command/Query/Response & Asynchronous Notification (on name changes)

- Device ID & Process ID (uniquely identify a device)
- Default name
- Friendly name
- Advertised name
- Change the friendly name of the device
- Receive a notification on a name change

Device Lock Information

Mechanisms: Query/Response & Asynchronous Notification (on lock/unlock state changes)

- Device lock/unlock state
- Receive a notification on lock/unlock state change

AES67 (Brooklyn II Only)

Mechanisms: Command/Query/Response & Asynchronous Notification (on AES67 enable/disable changes)

- Current AES67 enable/disable state
- Enable/disable AES67
- Receive a notification on enable/disable changes

VLANs

Mechanisms: Command/Query/Response & Asynchronous Notification (on VLAN configuration selection changes)

- Simple mode: Change between switched and redundant modes (Brooklyn II Only)
- Custom VLANs: Change between different custom VLAN configurations set in the device capability
- Receive a notification on selecting a different VLAN configuration

Metering (Brooklyn II Only)

Mechanisms: Command/Query/Response & Asynchronous Notification (on metering update rate changes)

- Current metering update rate
- Set the metering update rate to 10Hz or 30Hz
- Receive a notification on changes to the metering rate

UART Configuration (Brooklyn II Only)

Mechanisms: Command/Query/Response & Asynchronous Notification (on changes to configuration changes to a UART port)

- Current configuration parameters of a UART port: number of bits, stop bits, parity, baud rate, mode of the port, and whether the port can be configured
- Configure parameters (number of bits, stop bits, parity, and baud rate) of a UART port only if the port is marked as configurable in the capability
- Receive a notification on configuration changes to a UART port

Network

Mechanisms: Command/Query/Response & Asynchronous Notification (on network changes)

- MAC address
- Current Interface state (Link up or down)
- Current Link Speed
- Current IP Address / Netmask / Gateway
- Set Static IP Address / Netmask / Gateway
- Receive a notification if the network state changes

Clocking / PTP

Mechanisms: Command/Query/Response & Asynchronous Notification (on clock changes)

- Current clock state (master / slave / etc)
- Current mute state
- Current preferred state
- Current frequency offset
- PTP logging state
- Current pullup
- Current subdomain
- Set preferred state
- Enable / disable PTP logging
- Set clock pullup mode & subdomain
- Enable / disable unicast delay requests for PTP v1 and/or v2 protocols
- Enable / disable multicast
- Enable / disable slave only
- Enable / disable PTP ports
- Enable / disable PTP v1 and v2 protocols
- Receive a notification on any clock state changes

Audio

Mechanisms: Command/Query/Response & Asynchronous Notification (on audio changes)

- Default sample rate
- Current & Reboot sample rate
- Supported sample rates
- Default encoding
- Current & Reboot encoding
- Supported encodings
- Number of RX channels
- Number of TX channels
- Change audio sample rate
- Change audio encoding
- TDM interface information (Brooklyn II only)
- Receive a notification on any audio sample rate change
- Receive a notification on any audio encoding change
- Receive signal presence over DDP (Ultimo only)

Routing

Mechanisms: Command/Query/Response & Asynchronous Notification (on changes)

- Routing ready state
- Max Rx & Tx flows
- Max Rx & Tx slots per flow
- Min & max latency
- Min & max FPP
- Rx port range
- Channel Labels
- Set latency & FPP
- RX & TX Flow state and status
- RX & TX Channel state and status
- RX & TX channel labels
- Subscribe & unsubscribe RX channels
- Create and delete multicast TX flows
- Set channel labels
- Receive a notification on any flow state or status change
- Receive a notification on any channel state or status change
- Receive a notification on any channel label change

Interaction Diagrams

Sending a DDP Command/Query Message and Receiving a DDP Response Message

Interaction diagrams shown in Figure 26 and Figure 27 depict a Host CPU sending a DDP control message either to change (i.e. control) parameters on the Brooklyn II and Ultimo or to send a query for the current state of the Brooklyn II and Ultimo respectively.

Brooklyn II

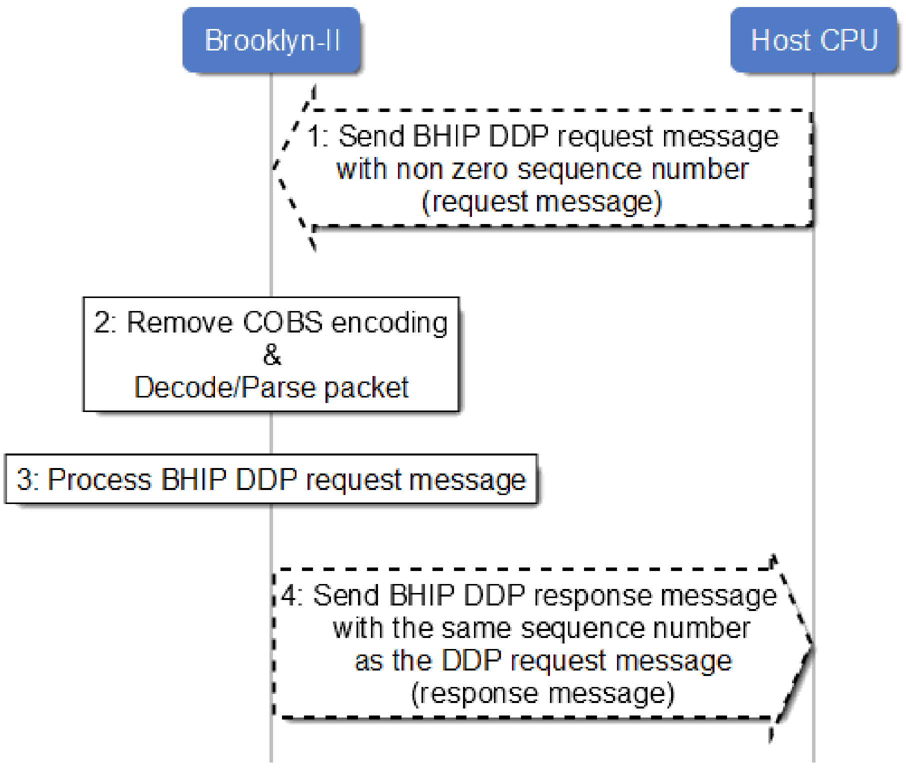
Brooklyn-II
Host CPU
1: Send BHIP DDP request message with non zero sequence number (request message)
2: Remove COBS encoding & Decode/Parse packet
3: Process BHIP DDP request message
4: Send BHIP DDP response message with the same sequence number as the DDP request message (response message)

Figure 26 - BHIP DDP command/query response

Ultimo

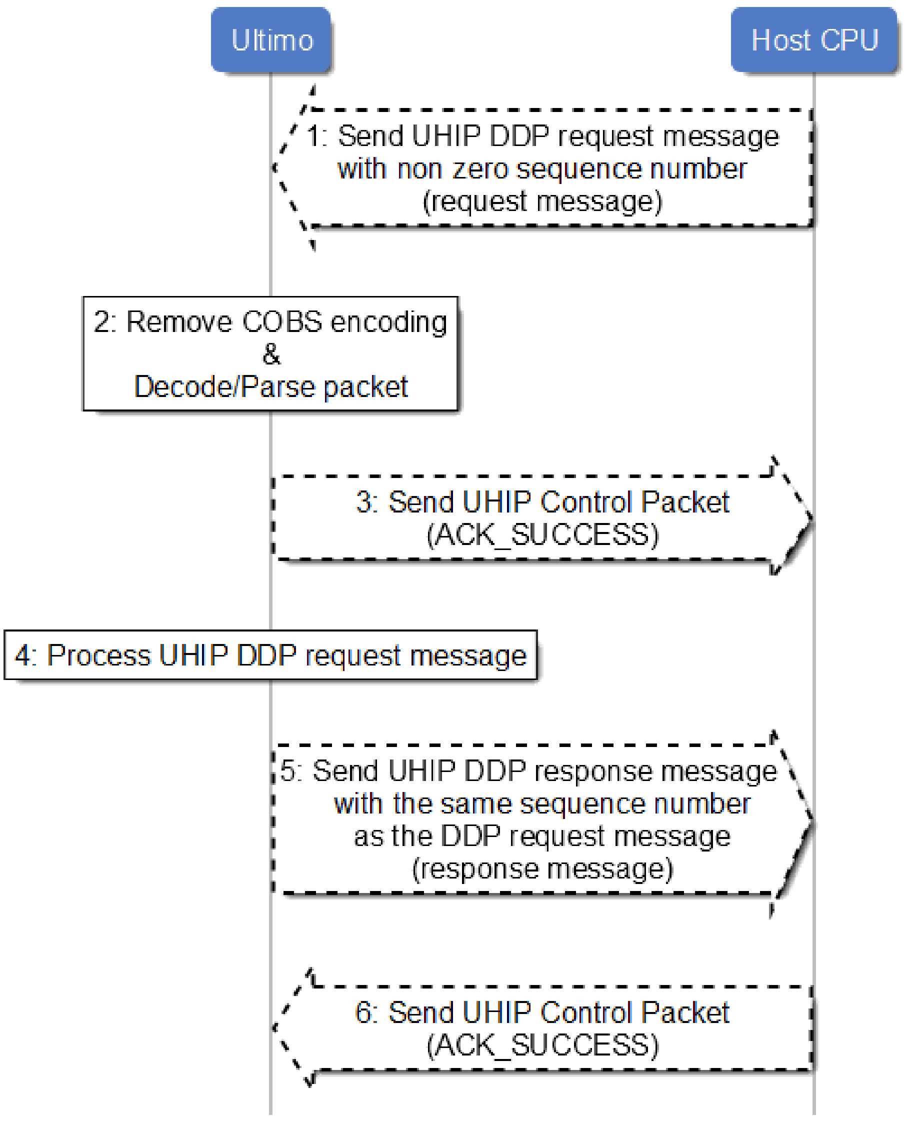
Ultimo
Host CPU
1: Send UHIP DDP request message with non zero sequence number (request message)
2: Remove COBS encoding & Decode/Parse packet
3: Send UHIP Control Packet (ACK_SUCCESS)
4: Process UHIP DDP request message
5: Send UHIP DDP response message with the same sequence number as the DDP request message (response message)
6: Send UHIP Control Packet (ACK_SUCCESS)

Figure 27 - UHIP DDP command/query response message

Receiving a DDP Event Message

Brooklyn II

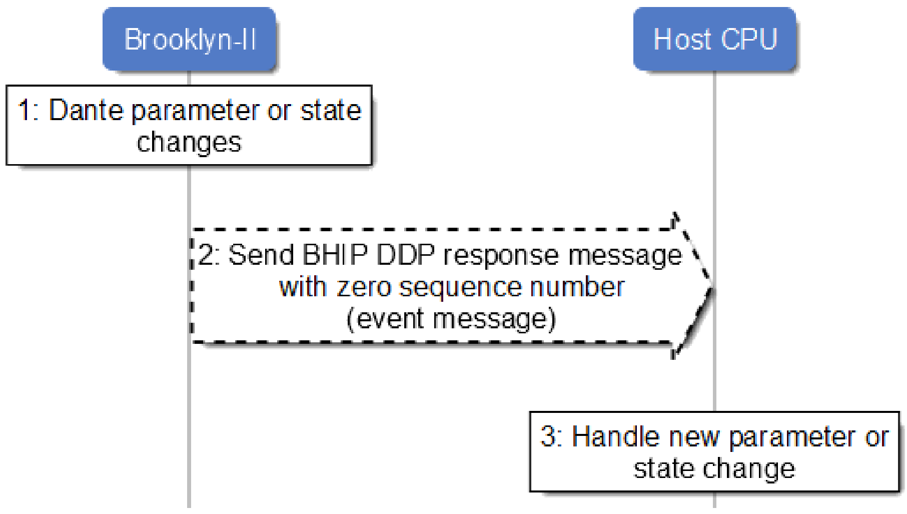
Brooklyn-II
Host CPU
1: Dante parameter or state changes
2: Send BHIP DDP response message with zero sequence number (event message)
3: Handle new parameter or state change

Figure 28 - BHIP DDP event message

Ultimo

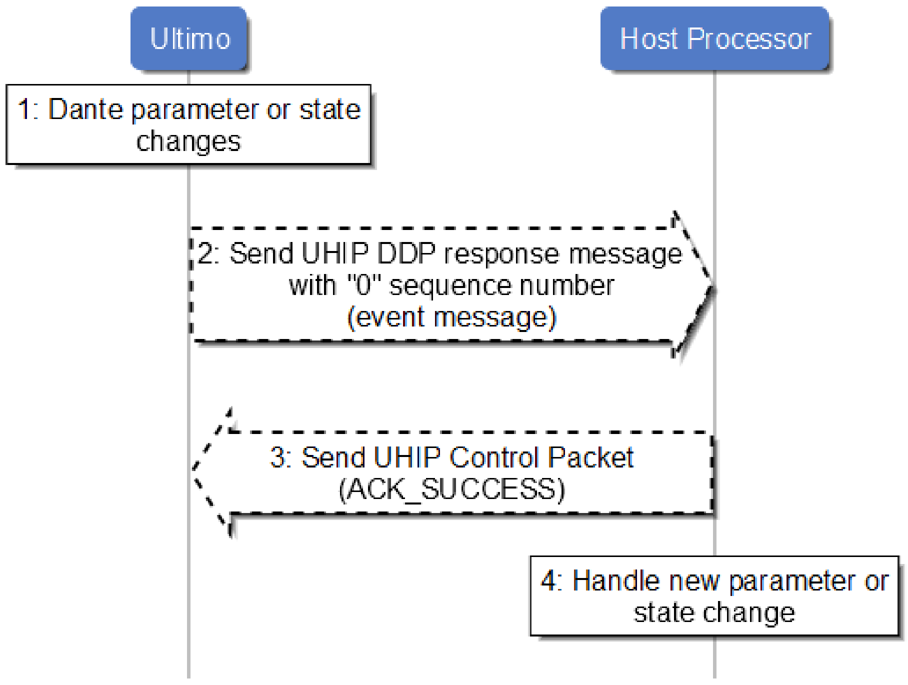
Ultimo
Host Processor
1: Dante parameter or state changes
2: Send UHIP DDP response message with "0" sequence number (event message)
3: Send UHIP Control Packet (ACK_SUCCESS)
4: Handle new parameter or state change

Figure 29 - UHIP DDP event message

Sending a Command to a Locked Dante Device

Brooklyn II

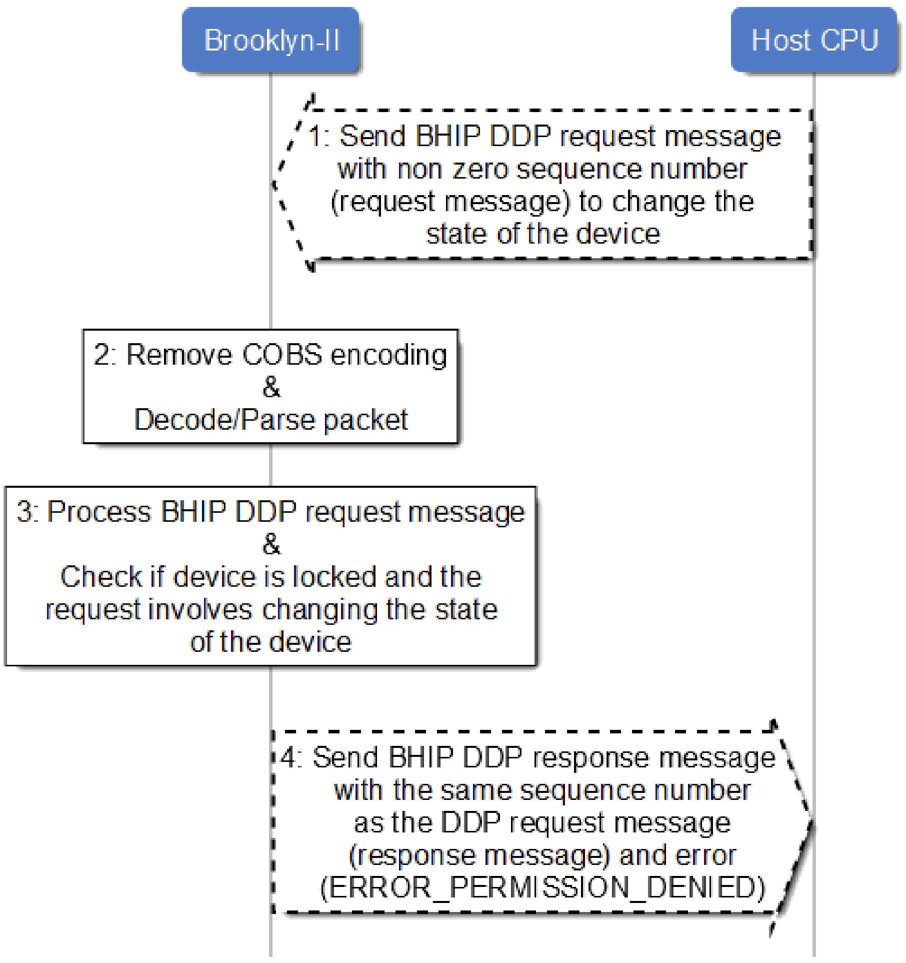
Brooklyn-II
Host CPU
1: Send BHIP DDP request message with non zero sequence number (request message) to change the state of the device
2: Remove COBS encoding & Decode/Parse packet
3: Process BHIP DDP request message & Check if device is locked and the request involves changing the state of the device
4: Send BHIP DDP response message with the same sequence number as the DDP request message (response message) and error (ERROR_PERMISSION_DENIED)

Figure 30 - BHIP DDP command send to a locked Brooklyn II

Ultimo

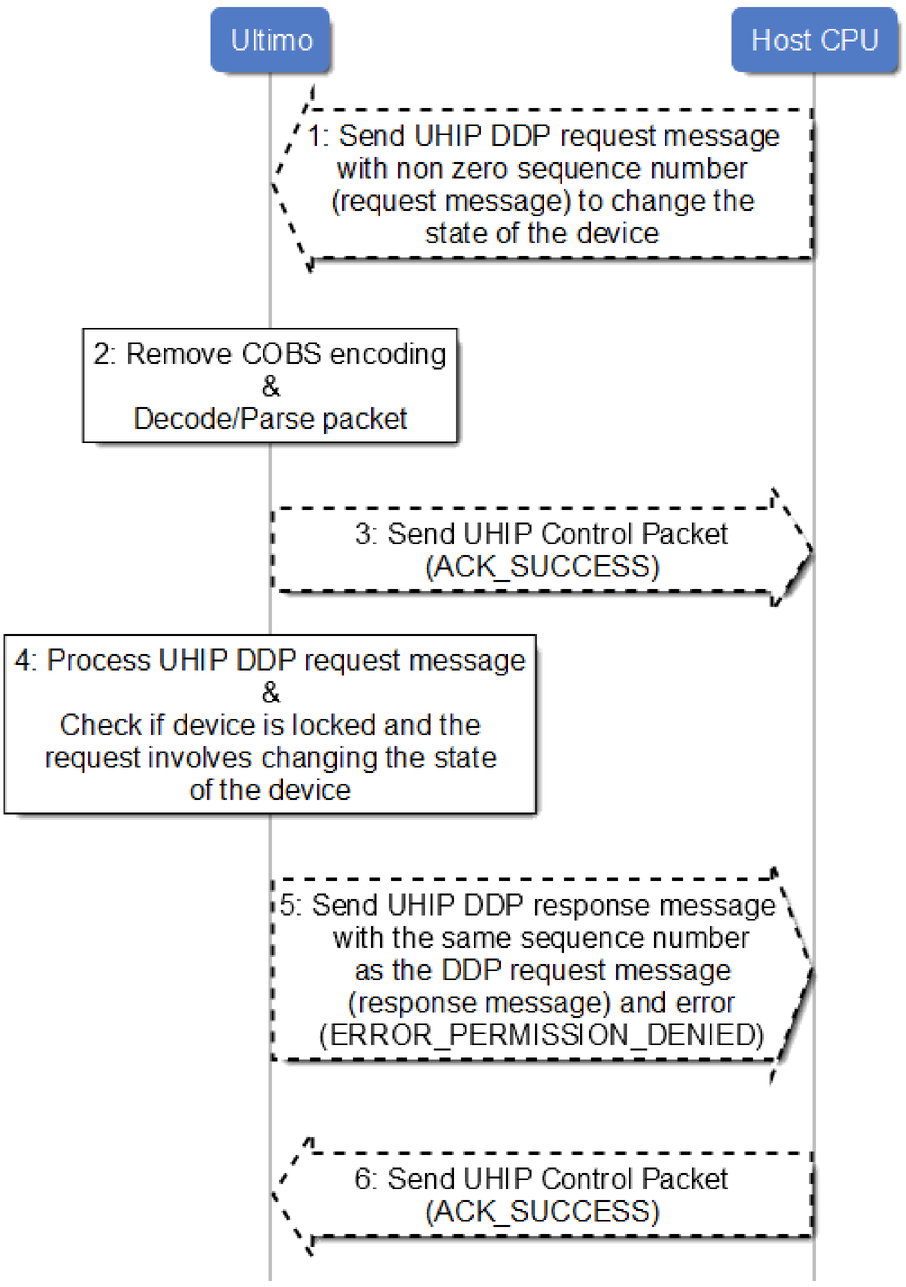
Ultimo
Host CPU
1: Send UHIP DDP request message with non zero sequence number (request message) to change the state of the device
2: Remove COBS encoding & Decode/Parse packet
3: Send UHIP Control Packet (ACK_SUCCESS)
4: Process UHIP DDP request message & Check if device is locked and the request involves changing the state of the device
5: Send UHIP DDP response message with the same sequence number as the DDP request message (response message) and error (ERROR_PERMISSION_DENIED)
6: Send UHIP Control Packet (ACK_SUCCESS)

Figure 31 - UHIP DDP command send to a locked Ultimo

Protocol Messages

Device Basic Information

Summary: Provides basic device information such as model, software version and device errors.

Mechanisms: Query → Response; Asynchronous Notification (on power on)

Message OPCODE / Type: DDP_DEVICE_GENERAL

API Functions: Device General

Provided Functionality / Information:

- Model ID
- Model ID string
- Software Version & Build
- Firmware Version & Build
- Bootloader Version & Build
- Configuration & Memory Error flags (capability partition error / user partition error / configuration store error)

Device Manufacturer Information

Summary: Provides manufacturer specified device information such as name, model, model versions, etc.

Mechanisms: Query → Response; Asynchronous Notification (on power on)

Message OPCODE / Type: DDP_DEVICE_MANUFACTURER

API Functions: Device Manufacturer

Provided Functionality / Information:

- Manufacturer ID
- Manufacturer Name
- Model ID & Model ID string
- Model Name
- Model Version
- Software Version & Build
- Firmware Version & Build
- Model Version & Model Version string

Device Upgrade

Summary: Start a TFTP or XMODEM upgrade, and receive notifications when an upgrade is externally started

Note: XMODEM via SPI/UART is only supported by the Ultimo

Mechanisms: Query/Command → Response; Asynchronous Notification (after an upgrade)

Message OPCODE / Type: DDP_DEVICE_UPRGADE

API Functions: Device Upgrade

Provided Functionality / Information:

- Start an upgrade via TFTP or XMODEM via SPI/UART
- Receive progress notifications after the upgrade has started

Device Erase Configuration

Summary: Erase all Dante configuration data from the device; receive a notification if an erase occurs

Mechanisms: Command → Response; Asynchronous Notification (during an erase)

Message OPCODE / Type: DDP_DEVICE_ERASE_CONFIG

API Functions: Device Erase Configuration

Provided Functionality / Information:

- Erase to factory defaults
- Erase to factory defaults but keep static IP information
- Receive a notification when a erase is externally triggered

Device Reboot

Summary: Reboot the device; receive a notification if a reboot is triggered

Mechanisms: Command → Response; Asynchronous Notification (before reboot)

Message OPCODE / Type: DDP_DEVICE_REBOOT

API Functions: Device Reboot

Provided Functionality / Information:

- Trigger a software reset
- Receive a notification when a reboot is externally triggered before the reboot occurs

Device Identity

Summary: Provide friendly / default name and unique ID information; notify on name changes

Mechanisms: Query/Command → Response; Asynchronous Notification (on name changes)

Message OPCODE / Type: DDP_DEVICE_IDENTITY

API Functions: Device Identity

Provided Functionality / Information:

- Device ID & Process ID (uniquely identify a device)
- Default name
- Friendly name
- Advertised name
- Change the friendly name of the device
- Receive a notification on a name change

Device Identify

Summary: Notify when a ConMon identify message is received

Mechanisms: Asynchronous Notification (on ConMon identify messages)

Message OPCODE / Type: DDP_DEVICE_IDENTIFY

API Functions: Device Identify

Provided Functionality / Information:

- Receive a notification when the device receives a valid ConMon identify message

Device GPIO

Note: This message is only supported on Ultimo

Summary: Control and Query for GPIO state; notify on GPIO state changes

Mechanisms: Query/Command → Response; Asynchronous Notification (on GPIO state changes)

Message OPCODE / Type: DDP_DEVICE_GPIO

API Functions: Device GPIO

Provided Functionality / Information:

- Control GPIO output state
- Query for current GPIO input or output state
- Receive a notification when the GPIO input or output value changes

Device Switch LED

Note: This message is only supported on Ultimo

Summary: Control the external RJ45 LEDs on a switch

Mechanisms: Command → Response

Message OPCODE / Type: DDP_DEVICE_SWITCH_LED

API Functions: Device Switch LED

Provided Functionality / Information:

- Control LEDs on the switch

Device Lock/Unlock

Summary: Provide the current lock/unlock state of the device; notify on device lock state changes

Mechanisms: Query → Response; Asynchronous Notification (on device lock state changes)

Message OPCODE / Type: DDP_DEVICE_TYPE_LOCK_UNLOCK

API Functions: Device Lock/Unlock

Provided Functionality / Information:

- Query current lock/unlock state
- Receive a notification when the lock/unlock state changes

Device Switch Redundancy

Note: This message is only supported on Brooklyn II

Summary: Change the device VLAN configuration between switched and redundant; notify on switch redundancy changes

Mechanisms: Query/Command → Response; Asynchronous Notification (on switch redundancy changes)

Message OPCODE / Type: DDP_DEVICE_TYPE_SWITCH_REDUNDANCY

API Functions: Device Switch Redundancy

Provided Functionality / Information:

- Change between switched and redundant VLAN configurations
- Receive a notification when the current VLAN configuration has been changed to a different state

Device UART Configuration

Note: This message is only supported on Brooklyn II

Summary: Change UART port parameters for UART ports designated as being configurable in the capability

Mechanisms: Query/Command → Response; Asynchronous Notification (on UART port parameter changes)

Message OPCODE / Type: DDP_DEVICE_TYPE_UART_CONFIG

API Functions: Device UART Configuration

Provided Functionality / Information:

- Change the number of bits, stop bits, baud rate, parity of a UART which can be configured
- Receive a notification when the configuration of a configurable UART port changes

Device AES67

Note: This message is only supported on Brooklyn II

Summary: Enable / disable AES67

Mechanisms: Query/Command → Response; Asynchronous Notification (on AES67 enable/disable state changes)

Message OPCODE / Type: DDP_DEVICE_TYPE_AES67

API Functions: Device AES67

Provided Functionality / Information:

- Change the AES67 enable/disable state
- Receive a notification when the AES67 enable/disable state changes

Device VLAN Configuration

Summary: Change between one of the custom VLAN configurations set in the capability

Mechanisms: Query/Command → Response; Asynchronous Notification (on changing to a different VLAN configuration)

Message OPCODE / Type: DDP_DEVICE_TYPE_VLAN_CONFIG

API Functions: Device VLAN Configuration

Provided Functionality / Information:

- Switch between up to four different custom VLAN configurations set in the capability
- Receive a notification when changing to a different VLAN configuration

Device Meter Configuration

Note: This message is only supported on Brooklyn II

Summary: Change the metering update rate

Mechanisms: Query/Command → Response; Asynchronous Notification (on metering update rate changes)

Message OPCODE / Type: DDP_DEVICE_TYPE_METER_CONFIG

API Functions: Device Meter Configuration

Provided Functionality / Information:

- Change the metering update rate between 10Hz and 30Hz
- Receive a notification when the meter update rate has changed

Network Basic

Summary: Provides basic network information such as MAC address, IP address, link state, etc. Notify on network state changes

Mechanisms: Query → Response; Asynchronous Notification (on network state changes)

Message OPCODE / Type: DDP_DEVICE_NETWORK_BASIC

API Functions: Network Basic

Provided Functionality / Information:

- MAC address
- Current Interface state (Link up or down)
- Current Link Speed
- Current IP Address / Netmask / Gateway
- Receive a notification if the network state changes

Network Configuration

Summary: Change between DHCP+LinkLocal and static IP address configurations. Notify on network configuration changes

Mechanisms: Command/Query → Response; Asynchronous Notification (on network configuration changes)

Message OPCODE / Type: DDP_DEVICE_NETWORK_CONFIG

API Functions: Network Configuration

Provided Functionality / Information:

- Change between static IP address and DHCP / LinkLocal network configurations
- Set a static IP Address / Netmask / Gateway address configuration
- Receive a notification if the network configuration changes

Clock Basic Legacy

Note: This message was formerly known as Clock Basic and it is now deprecated. It is only supported on Ultimo. Please migrate to Clock Basic 2.

Summary: Provides legacy basic clock information; notify on clock state changes

Mechanisms: Query → Response; Asynchronous Notification (on clock changes)

Message OPCODE / Type: DDP_CLOCK_BASIC

API Functions: Clock Basic Legacy

Provided Functionality / Information:

- Current clock state (master / slave / etc)
- Current mute state
- Current preferred state
- Current frequency offset
- Receive a notification on any clock state changes

Clock Basic 2

Summary: Provides basic clock information; notify on clock state changes

Mechanisms: Query → Response; Asynchronous Notification (on clocking related changes)

Message OPCODE / Type: DDP_CLOCK_TYPE_BASIC2

API Functions: Clock Basic 2

Provided Functionality / Information:

- Current clock source
- Current clock state (master / slave / etc)
- Current mute state
- Current preferred state
- Current frequency offset
- Current external word clock state
- Current clock stratum
- Current UUID, master UUID, and grand master UUID
- PTP port parameters such as ID, protocol supported, state, unicast/multicast, interface index
- Receive a notification on any clock state changes

Clock Configuration

Summary: Change clock configuration parameters; notify on clock configuration changes

Mechanisms: Command/Query → Response; Asynchronous Notification (on clock configuration changes)

Message OPCODE / Type: DDP_CLOCK_CONFIG

API Functions: Clock Configuration

Provided Functionality / Information:

- PTP logging state
- Set preferred state
- Enable / disable PTP logging
- Enable / disable unicast delay requests for PTP v1 and/or v2 protocols
- Enable / disable multicast
- Enable / disable slave only
- Enable / disable PTP V1 and V2 protocols
- Enable / disable PTP ports
- Receive a notification on any clock configuration changes

Clock Pull-up

Summary: Change clock pull-up or clock subdomain configurations; notify on clock pull-up or subdomain changes

Mechanisms: Command/Query → Response; Asynchronous Notification (on clock pullup changes)

Message OPCODE / Type: DDP_CLOCK_PULLUP

API Functions: Clock Pullup

Provided Functionality / Information:

- Current pullup
- Current subdomain
- List of supported pullups
- Set clock pullup mode & subdomain
- Receive a notification on any clock pullup / subdomain changes

Audio Basic

Summary: Provides basic audio information such as number of RX and TX channels, default sample rate and default encoding

Mechanisms: Query → Response; Asynchronous Notification (on Rx and Tx channel count changes due to setting a sample rate which causes this condition)

Message OPCODE / Type: DDP_AUDIO_BASIC

API Functions: Audio Basic

Provided Functionality / Information:

- Default sample rate
- Default encoding
- Number of RX channels
- Number of TX channels

Audio Sample Rate Configuration

Summary: Change audio sample rate configuration and query for supported sample rates. Notify on audio sample rate changes.

Mechanisms: Command/Query → Response; Asynchronous Notification (on audio sample rate changes)

Message OPCODE / Type: DDP_SRATE_CONFIG

API Functions: Audio Sample Rate Configuration

Provided Functionality / Information:

- Current & Reboot sample rate
- Supported sample rates
- Change audio sample rate
- Receive a notification on any audio sample rate change

Audio Encoding Configuration

Summary: Change audio encoding configuration and query for supported encodings. Notify on audio encoding changes.

Mechanisms: Command/Query → Response; Asynchronous Notification (on audio encoding changes)

Message OPCODE / Type: DDP_AUDIO_ENC_CONFIG

API Functions: Audio Encoding Configuration

Provided Functionality / Information:

- Current & Reboot encoding
- Supported encoding
- Change audio encoding
- Receive a notification on any audio encoding change

Audio Interface

Note: This message is only supported on Brooklyn II

Summary: Provides information about the audio TDM interface of the Dante device

Mechanisms: Query → Response; Asynchronous Notification (on boot up)

Message OPCODE / Type: DDP_AUDIO_TYPE_INTERFACE

API Functions: Audio Interface

Provided Functionality / Information:

- Audio channels per TDM interface
- TDM clock framing type
- TDM sample alignment type
- Alignment of first channel on a TDM line with respect to the LRCLK
- A notification is sent during boot up of the Dante device

Audio Signal Presence Configuration

Note: This message is only supported on Ultimo

Summary: Enable/disable the Ultimo from pushing signal presence data over DDP periodically

Mechanisms: Command → Response

Message OPCODE / Type: DDP_AUDIO_TYPE_SIGNAL_PRESENCE_CONFIG

API Functions: Audio Signal Presence Configuration

Provided Functionality / Information:

- Enable/disable the Ultimo from sending signal presence over DDP

Audio Signal Presence Data

Note: This message is only supported on Ultimo

Summary: Signal presence information both Rx and Tx channels

Mechanisms: Asynchronous Notification (on availability of new signal presence data)

Message OPCODE / Type: DDP_AUDIO_TYPE_SIGNAL_PRESENCE_DATA

API Functions: Audio Signal Presence Data

Provided Functionality / Information:

- Number of Rx and Tx channels that have signal presence information
- The signal presence which indicates audio clip, audio signal, and no audio signal

Routing Basic

Summary: Provides basic routing information

Mechanisms: Query → Response

Message OPCODE / Type: DDP_ROUTING_BASIC

API Functions: Routing Basic

Provided Functionality / Information:

- Max Rx & Tx flows
- Max Rx & Tx slots per flow
- Min & max latency
- Min & max FPP
- Rx port range
- Number of Rx/Tx channels contained in a single Rx/Tx channel config state DDP message
- Number of Rx/Tx flows contained in a single Rx/Tx flow config state DDP message
- Number of Rx channels/flows contained in a single Rx channel/flow status DDP message

Routing Ready

Summary: Provides status information on whether the Dante device is ready to support routing commands.

Mechanisms: Query → Response; Asynchronous Notification (on routing ready state changes)

Message OPCODE / Type: DDP_ROUTING_READY_STATE

API Functions: Routing Ready State

Provided Functionality / Information:

- Routing ready state
- Receive a notification when the routing ready state changes

Routing Performance Configuration

Summary: Provides current latency/FPP configuration, change current latency/FPP settings, notification on latency/FPP changes.

Mechanisms: Command/Query → Response; Asynchronous Notification (on changes)

Message OPCODE / Type: DDP_ROUTING_PERFORMANCE_CONFIG

API Functions: Routing Performance Config

Provided Functionality / Information:

- Set latency & FPP
- Receive a notification on routing performance configuration changes

Routing Rx Channel Configuration State

Summary: Provides RX channel configuration information such as labels, encodings, current subscriptions; Notification on RX channel configuration changes.

Note: The number of Rx channels contained in a single message is limited, and this number can be obtained from the Routing Basic DDP message

Mechanisms: Query → Response; Asynchronous Notification (on changes)

Message OPCODE / Type: DDP_ROUTING_RX_CHAN_CONFIG_STATE

API Functions: Routing RX Channel Configuration State

Provided Functionality / Information:

- Default Channel Label
- Current Channel Label
- Channel Encoding
- Channel Sample Rate
- Channel Supported PCM encodings
- Channel Supported custom encodings
- Subscriptions (subscribed device channel) and Subscription status
- Receive a notification on any channel state change
- Receive a notification on any channel label change

Routing Tx Channel Configuration State

Summary: Provides TX channel configuration information such as labels, encodings, etc; Notification on TX channel configuration changes.

Note: The number of Tx channels contained in a single message is limited, and this number can be obtained from the Routing Basic DDP message

Mechanisms: Query → Response; Asynchronous Notification (on changes)

Message OPCODE / Type: DDP_ROUTING_TX_CHAN_CONFIG_STATE

API Functions: Routing TX Channel Configuration State

Provided Functionality / Information:

- Default Channel Label
- Current Channel Label
- Channel Encoding
- Channel Sample Rate
- Channel Supported PCM encodings
- Channel Supported custom encodings
- Receive a notification on any channel state change
- Receive a notification on any channel label change

Routing Rx Channel Status

Summary: Provides a query mechanism for the current RX channel status (i.e. unresolved / dynamic)

Note: The number of Rx channels contained in a single message is limited, and this number can be obtained from the Routing Basic DDP message

Mechanisms: Query → Response

Message OPCODE / Type: DDP_ROUTING_RX_CHAN_STATUS

API Functions: Routing RX Channel Status

Provided Functionality / Information:

- Channel Status - this is the audio subscription status of this channel, for example DDP_RX_CHAN_STATUS_UNRESOLVED or DDP_RX_CHAN_STATUS_DYNAMIC

Routing Rx Flow Configuration State

Summary: Provides RX flow configuration information such as status, slots, channels, etc; Notification on RX flow configuration changes.

Note: The number of Rx flows contained in a single message is limited, and this number can be obtained from the Routing Basic DDP message

Mechanisms: Query → Response; Asynchronous Notification (on changes)

Message OPCODE / Type: DDP_ROUTING_RX_FLOW_CONFIG_STATE

API Functions: Routing RX Flow Configuration State

Provided Functionality / Information:

- Flow status - for example FLOW_STATUS_ACTIVE or FLOW_STATUS_BAD_ADDRESS
- Flow configuration - e.g. number of slots, flow IP address and port, RX channels, encoding, sample rate, latency, FPP
- Receive a notification on any flow configuration change (except flow status changes for flow status changes the “Rx Flow Status” notification will occur)

Routing Tx Flow Configuration State

Summary: Provides TX flow configuration information such as status, slots, channels, etc; Notification on TX flow configuration changes.

Note: The number of Tx flows contained in a single message is limited, and this number can be obtained from the Routing Basic DDP message

Mechanisms: Query → Response; Asynchronous Notification (on changes)

Message OPCODE / Type: DDP_ROUTING_TX_FLOW_CONFIG_STATE

API Functions: Routing TX Flow Configuration State

Provided Functionality / Information:

- Flow status - for example FLOW_STATUS_ACTIVE or FLOW_STATUS_BAD_ADDRESS
- Flow configuration - e.g. number of slots, flow IP address and port, TX channels, encoding, sample rate, latency, FPP
- Receive a notification on any flow configuration change

Routing Rx Flow Status

Summary: Provides a query mechanism for the current RX flow status (i.e. active)

Note: The number of Rx flows contained in a single message is limited, and this number can be obtained from the Routing Basic DDP message

Mechanisms: Query → Response

Message OPCODE / Type: DDP_ROUTING_RX_FLOW_STATUS

API Functions: Routing RX Flow Status

Provided Functionality / Information:

- Flow available - whether the flow is available on any interface
- Flow active - whether the flow is actively receiving audio
- Receive a notification on any flow status change

Routing Rx Channel Label Set

Summary: Set / change an RX channel label

Note: The number of Rx channels that can have their labels updated by a single message is limited, and this number can be obtained from the Rx channel config state per DDP message field in the Routing Basic DDP message

Mechanisms: Command → Response

Message OPCODE / Type: DDP_ROUTING_RX_CHAN_LABEL_SET

API Functions: Routing RX Channel Label Set

Provided Functionality / Information:

- Change / Set the RX Channel Label on 1 or more channels
- Note: The response to a RX Channel Label Set message contains the same payload as a Rx Routing Channel Configuration State message

Routing Tx Channel Label Set

Summary: Set / change a TX channel label

Note: The number of Tx channels that can have their labels updated by a single message is limited, and this number can be obtained from the Tx channel config state per DDP message field in the Routing Basic DDP message

Mechanisms: Command → Response;

Message OPCODE / Type: DDP_ROUTING_TX_CHAN_LABEL_SET

API Functions: Routing TX Channel Label Set

Provided Functionality / Information:

- Change / Set the TX Channel Label on 1 or more channels
- Note: The response to a TX Channel Label Set message contains the same payload as a Tx Routing Channel Configuration State message

Routing RX Subscribe

Summary: Create a label based RX subscription on the Dante device

Note: The number of Rx channels subscribed using a single message is limited, and this number can be obtained from the Rx channel config state per DDP message field in the Routing Basic DDP message

Mechanisms: Command → Response;

Message OPCODE / Type: DDP_ROUTING_RX_SUBSCRIBE_SET

API Functions: Routing RX Subscribe

Provided Functionality / Information:

- Create RX subscriptions
- Note: The response to a RX Subscribe Set message contains the same payload as a RX Channel Configuration State message

Routing RX Unsubscribe

Summary: Remove RX subscriptions on the Dante device

Mechanisms: Command → Response;

Message OPCODE / Type: DDP_ROUTING_RX_UNSUB_CHAN

API Functions: Routing RX Unsubscribe

Provided Functionality / Information:

- Delete RX subscriptions

Routing Multicast TX Flow Configuration

Summary: Create a multicast TX flow

Mechanisms: Command → Response;

Message OPCODE / Type: DDP_ROUTING_MCAST_TX_FLOW_CONFIG_SET

API Functions: Routing TX Multicast Flow Configuration

Provided Functionality / Information:

- Create Multicast TX flows

Routing Flow Delete

Summary: Remove flows on the Ultimo

Mechanisms: Command → Response;

Message OPCODE / Type: DDP_ROUTING_FLOW_DELETE

API Functions: Routing Flow Delete

Provided Functionality / Information:

- Delete an existing flow, this command can be used to delete a Tx multicast flow

Host CPU SDK Package

The Host CPU SDK package can be integrated into an existing software project which enables communication between a Host CPU and a Dante device. It provides the core functionality required to implement the software for a Host CPU to communicate with a Brooklyn II or Ultimo. The core functionality is exposed via an API.

The package also includes an example application for Windows that uses the UART and SPI transport. The SPI transport in example Windows application uses the API from Total Phase Inc. and will work with their Aardvark Host Adapters. The implementer should add platform-specific code as required to port the functionality to their platform of choice.

Data Types

The data types used throughout the Host CPU SDK is specified in Table 9. These data type definitions are specified in the `stdint.h` C header file.

Table 9 - Host CPU SDK data types

Data Type	Description
uint8_t	Unsigned 8-bit integer
int8_t	Signed 8-bit integer
uint16_t	Unsigned 16-bit short integer
int16_t	Signed 16-bit short integer
uint32_t	Unsigned 32-bit long integer
int32_t	Signed 32-bit long integer

Provided Functionality

The Host CPU SDK package contains the following (organised by folder):

- `docs` - Doxygen documentation for the core functionality API. Open the `docs.html` to access the Doxygen documentation
- `src/common` - Dante types & helpers
- `src/libs` - Host CPU core functionality library
  - `src/libs/lib_cobs` - COBS encoding / decoding library used for BHIP and UHIP packet framing

- src/libs/lib_bhip - Brooklyn II Host Interface Protocol packet read / write library to create and parse BHIP messages
- src/libs/lib_uhip - Ultimo Host Interface Protocol packet read / write library to create and parse UHIP messages
- src/libs/lib_ddp - Dante Device Protocol packet read /write library to create and parse DDP messages
- src/libs/interface - Interface to UHIP API library
    - src/libs/interface/uhip_hostcpu_rx_timer.h - UHIP RX timer interface
    - src/libs/interface/uhip_hostcpu_tx_timer.h - UHIP TX timer interface
    - src/libs/interface/hostcpu_transport.h - Raw data (SPI / UART) transport layer interface
- src/libs/lib_serial - Utility functions to prepare and extract COBS encoded serial frames for Brooklyn II and Ultimo

- src/app - Example implementation of a Host CPU to Brooklyn II or Ultimo interface application
    - src/app/example_bhip:
        - src/app/example_bhip/example_bhip_main.[c|h] - “Core” state machine implementing BHIP protocol
        - src/app/example_bhip/example_bhip_common.[c|h] - Common functionality used across the BHIP HostCPU examples
        - src/app/example_bhip/example_rx_bhip_[spi|uart].[c|h] - BHIP Receive functionality specific to a serial peripheral
        - src/app/example_bhip/example_tx_bhip.[c|h] - Examples to handle transmitting BHIP packets
        - src/app/example_bhip/example_tx_bhip_pb.[c|h] - Examples to create and send ConMon and UDP mode BHIP packet bridge packets
    - src/app/example_uhip:
        - src/app/example_uhip/example_uhip_main.[c|h] - “Core” state machine implementing UHIP protocol
        - src/app/example_uhip/example_uhip_common.[c|h] - Common functionality used across the UHIP HostCPU examples
        - src/app/example_uhip/example_rx_uhip.[c|h] - UHIP Receive functionality
        - src/app/example_uhip/example_tx_uhip.[c|h] - Examples to handle transmitting UHIP packets
        - src/app/example_uhip/example_tx_pb.[c|h] - Examples to create and send ConMon and UDP mode UHIP packet bridge packets
    - src/app/example_ddp:
        - src/app/example_ddp/example_rx_ddp.[c|h] - Examples to handle received DDP and Dante Event messages
        - src/app/example_ddp/example_tx_ddp.[c|h] - Examples to create and send DDP messages
- src/app/win - A sample Windows implementation of the interface in src/libs/lib_interface
    - src/app/win/bhip/aud_platform.h - Global header file for BHIP Host CPU API
    - src/app/win/uhip/aud_platform.h - Global header file for UHIP Host CPU API

- src/app/win/uhip/rx_timer.c, src/app/win/uhip/tx_timer.c, src/app/win/uhip/timer_common.c - Implementation of timers conforming to timer interface for UHIP
- src/app/win/bhip_uart_transport.c - Implementation of UART transport conforming to transport interface for BHIP communication. In this file the user must specify the COM port number by changing the PC_COM_PORT manifest constant. The UART baud rate used for the communication is 115200.
- src/app/win/bhip_spi_transport.c - Implementation of SPI transport conforming to transport interface for BHIP communication. The SPI clock speed used for the communication is 4MHz.
- src/app/win/uhip_spi_transport.c - Implementation of SPI transport conforming to transport interface for UHIP communication. In this file the user must specify the TotalPhase Inc. Aardvark SPI host adapter SPI master and SPI slave serial numbers by changing the AARDVARK_SPI_MASTER_SERIAL and AARDVARK_SPI_SLAVE_SERIAL manifest constants. The SPI clock speed used for the communication is 4MHz.
- src/app/win/uart_transport.c - Implementation of UART transport conforming to transport interface for UHIP communication. In this file the user must specify the COM port number by changing PC_COM_PORT manifest constant. The UART baud rate used for the communication is 115200.

- vs_project - Windows Visual Studio project files for each serial peripheral for BHIP and UHIP and a solution file which groups the project files. Minimum supported version of Visual Studio is 2013.

Porting the Host CPU SDK Package

This section details how to port the Host CPU SDK package to your host CPU. Doing a trial build and run on the existing example platforms before starting implementation is strongly recommended.

Project Setup

The package will compile as-is as a Windows application. The first step in porting to a different platform will be to execute the create_ultimo_hostcpu.bat or create_brooklyn-ii_hostcpu.bat to get the source files to communicate with a specific Dante device. The next step is to create platform specific build files (i.e. Makefiles) or add the source code to an existing build project.

The implementer will also need to add folders for platform-specific code similar to src/win

The following header search paths should be included for the C compiler:

- (lib includes)
- Platform specific header file location

Implement the RX Timer interface

The RX timer interface specified in uhip_hostcpu_rx_timer.h needs to be implemented and this is only required for host CPUs interfacing with an Ultimo. This is a polled one-shot timer that is used for the UHIP protocol RX timeouts.

The implementation requirements are:

- A timer with a resolution and accuracy of approximately 100ms

Implement the TX Timer interface

The TX timer interface specified in `uhip_hostcpu_tx_timer.h` needs to be implemented and this is only required for host CPUs interfacing with an Ultimo. This is a polled one-shot timer that is used for the UHIP protocol TX timeouts.

The implementation requirements are:

- A timer with a resolution and accuracy of approximately 100ms

Implement the Host CPU Transport Interface

The Host CPU Transport interface specified in `hostcpu_transport.h` needs to be implemented. This is the platform specific SPI or UART driver for the Host CPU. This is used by the API to send / receive data to the Brooklyn II or Ultimo. The following sub-sections specify the requirements for each interface for each type of host CPU protocol.

BHIP Host CPU SPI Interface

- The SPI master is used to send data and while sending data, the Host CPU SPI master must receive data from the Brooklyn II SPI slave. The send data functionality should be implemented by the `hostcpu_transport_write()` function
- Receiving data can be performed while sending data, but if there is no data to be sent the SPI master should sent dummy data. The receive data functionality should be implemented by the `hostcpu_transport_read()` function. The host CPU should make use of the DATA_AVAILABLE line from the Brooklyn II which signals the availability of data to be read by the host CPU. See Communicating with a Brooklyn II for information about the DATA_AVAILABLE line.

UHIP Host CPU SPI Interface

- A SPI master peripheral driver used for sending data only (any RX data from the SPI master should be dropped). This should be implemented by the `hostcpu_transport_write()` function
- A SPI slave peripheral driver used for receiving data only (TX dummy bytes may need to be sent at the peripheral driver layer). This should be implemented by the `hostcpu_transport_read()` function. This implementation must use either DMA or interrupt driven I/O as the driver must be able to buffer up to 576 bytes of received data

Implementation requirements for a UART Host CPU interface:

- A UART TX driver used for sending data only. This should be implemented by the `hostcpu_transport_write()` function.
- A UART RX driver used for receiving data only. This should be implemented by the `hostcpu_transport_read()` function. This implementation must use either DMA or interrupt driven I/O as the driver must be able to buffer up to 576 bytes of received data.

Add required RX message handling

Some example stub functions for receiving different message types are implemented in the `example_rx_bhip.c/h, example_rx_uhip.c/h, example_rx_ddp.c/h` files. The customer will need to modify these stub functions and add additional receive functions as required for their application.

Add required TX message handling

Some example stub functions for building and transmitting different message types are implemented in the `example_tx_bhip.c/h, example_tx_uhip.c/h, example_tx_ddp.c/h` files. The customer will need to modify these stub functions and add additional transmit functions as required for their application.

Modify aud_platform.h

`uhip/aud_platform.h` and `bhip/aud_platform.h` include platform specific defines and includes, modify these to suit your platform.

Modify or Implement a Main Loop

`example_uhip_main.c` and `example_bhip_main.c` contain a simple implementation of a main loop that does not assume any special platform functionality. It is assumed that the sending of Tx messages from this application needs to be driven off a timer or via integration of an external trigger such as a push button press. Alternatively a more sophisticated main loop using `select()` and callbacks may be used.

Host CPU SDK Library API Functions

UHIP ConMon Packet Bridge Functions

Message Type	Build Function (for TX)	Parse Function (for RX)
UHIP ConMon Packet Bridge	`uhip_packet_write_cmc_packet_bridge()`	`uhip_packet_read_cmc_packet_bridge()`

BHIP ConMon Packet Bridge Functions

Message Type	Build Function (for TX)	Parse Function (for RX)
BHIP ConMon Packet Bridge	`bhip_packet_write_cmc_packet_bridge()`	`bhip_packet_read_cmc_packet_bridge()`

UHIP UDP Packet Bridge Functions

Message Type	Build Function (for TX)	Parse Function (for RX)
UHIP UDP Packet Bridge	`uhip_packet_write_udp_packet_bridge()`	`uhip_packet_read_udp_packet_bridge()`

BHIP UDP Packet Bridge Functions

Message Type	Build Function (for TX)	Parse Function (for RX)
BHIP UDP Packet Bridge	`bhip_packet_write_udp_packet_bridge()`	`bhip_packet_read_udp_packet_bridge()`

Dante Events Functions

Message Type	Build Function (for TX)	Parse Function (for RX)
Audio Format Change (sample rate)	N/A	`ddp_read_local_audio_format()`
Clock Pullup Change	N/A	`ddp_read_local_clock_pullup()`

Dante Device Protocol (DDP) Functions

Message Type	Build Function (for TX query/command)	Parse Function (for RX response/event)
Device General	`ddp_read_device_general_request()`	`ddp_read_device_general_response()`
Device Manufacturer	`ddp_add_device_manufacturer_request()`	`ddp_read_device_manufacturer_response()`
Device Upgrade	`ddp_add_device_upgrade_xmodem_request()` `ddp_add_device_upgrade_tftp_request()`	`ddp_read_device_upgrade_response()`
Device Erase Configuration	`ddp_add_device_erase_request()`	`ddp_read_device_erase_response()`
Device Reboot	`ddp_add_device_reboot_request()`	`ddp_read_device_reboot_response()`
Device Identity	`ddp_add_device_identity_request()`	`ddp_read_device_identity_response()`
Device Identify		`ddp_read_device_identify_response()` [event only]
Device GPIO	`ddp_add_device_gpio_request()`	`ddp_read_device_gpio_response()`
Device Switch LED	`ddp_add_device_switch_led_request()`	`ddp_read_device_switch_led_response()`

Message Type	Build Function (for TX query/command)	Parse Function (for RX response/event)
Device Lock/Unlock	ddp_add_device_lock_unlock_request()	ddp_read_device_lock_unlock_response()
Device Switch Redundancy	ddp_add_device_switch_redundancy_request()	ddp_read_device_switch_redundancy_response()

Device UART Configuration	ddp_add_device_uart_config_request()	ddp_read_uart_config_response_header()
		ddp_read_uart_config_response_uart_st_array()

Device AES67	ddp_add_device_aes67_request()	ddp_read_device_aes67_response()

Device VLAN Configuration	ddp_add_device_vlan_config_request()	ddp_read_device_vlan_config_response_header()
		ddp_read_device_vlan_config_response_vlan_st_array()
		ddp_read_device_vlan_config_response_name_string()

Device Meter Configuration	ddp_add_device_meter_config_request()	ddp_read_device_meter_config_response()

Network Basic	ddp_add_network_basic_request()	ddp_read_network_basic_response_header()
		ddp_read_network_basic_response_interface()
		ddp_read_network_basic_response_interface_address()
		ddp_read_network_basic_response_dns()
		ddp_read_network_basic_response_domain()

Network Configuration	ddp_add_network_config_request()	ddp_read_network_config_response()
Clock Basic Legacy	ddp_add_clock_basic_legacy_request()	ddp_read_clock_basic_legacy_response()

Message Type	Build Function (for TX query/command)	Parse Function (for RX response/event)

Clock Configuration	`ddp_add_clock_config_request()`	`ddp_read_clock_config_response()`
		`ddp_read_clock_config_response_port()`

Clock Pullup	`ddp_add_clock_pullup_request()`	`ddp_read_clock_pullup_response()`

Clock Basic 2	`ddp_add_clock_basic2_request()`	`ddp_read_clock_basic2_response_header()`
		`ddp_read_clock_basic2_response_port()`

Audio Basic	`ddp_add_audio_basic_request()`	`ddp_read_audio_basic_response()`

Audio Sample Rate Configuration	`ddp_add_audio_sample_rate_config_request()`	`ddp_read_audio_sample_rate_config_response()`
		`ddp_read_audio_sample_rate_config_supported_srate()`

Audio Encoding Configuration	`ddp_add_audio_encoding_config_request()`	`ddp_read_audio_encoding_config_response()`
		`ddp_read_audio_encoding_config_supported_encoding()`

Audio Interface	`ddp_add_audio_interface_request()`	`ddp_read_audio_interface_response()`
Audio Signal Presence Configuration	`ddp_add_audio_signal_presence_config_request()`	`ddp_read_audio_signal_presence_config_response()`

Audio Signal Presence Data		`ddp_read_audio_signal_presence_data_response()`
		`ddp_read_audio_signal_presence_data_tx_chan_value()`
		`ddp_read_audio_signal_presence_data_rx_chan_value()`

Routing Basic	`ddp_add_routing_basic_request()`	`ddp_read_routing_basic_response()`
Routing Ready State	`ddp_add_routing_ready_state_request()`	`ddp_read_routing_ready_state_response()`

Message Type	Build Function (for TX query/command)	Parse Function (for RX response/event)
Routing Performance Configuration	ddp_add_routing_performance_config_request()	ddp_read_routing_performance_config_response()

Routing RX Channel Configuration State	ddp_add_routing_rx_chan_config_state_request()	ddp_read_routing_rx_chan_config_state_response_header()
		ddp_read_routing_rx_chan_config_state_response_chan_params()
		ddp_read_routing_rx_chan_config_state_response_custom_encoding()

Routing TX Channel Configuration State	ddp_add_routing_tx_chan_config_state_request()	ddp_read_routing_tx_chan_config_state_response_header()
		ddp_read_routing_tx_chan_config_state_response_chan_params()
		ddp_read_routing_tx_chan_config_state_response_custom_encoding()

Routing RX Flow Configuration State	ddp_add_routing_rx_flow_config_state_request()	ddp_read_routing_rx_flow_config_state_response_header()
		ddp_read_routing_rx_flow_config_state_response_flow_params()
		ddp_read_routing_rx_flow_config_state_response_flow_slot()
		ddp_read_routing_rx_flow_config_state_response_flow_slot_chans()
		ddp_read_routing_rx_flow_config_state_response_flow_address_nw_ip()

Message Type	Build Function (for TX query/command)	Parse Function (for RX response/event)
Routing TX Flow Configuration State	`ddp_add_routing_tx_flow_config_state_request()`	`ddp_read_routing_tx_flow_config_state_response_header()` `ddp_read_routing_tx_flow_config_state_response_flow_params()` `ddp_read_routing_tx_flow_config_state_response_flow_slots()` `ddp_read_routing_tx_flow_config_state_response_flow_address_nw_ip()`
Routing RX Channel Status	`ddp_add_routing_rx_chan_status_request()`	`ddp_read_routing_rx_chan_status_response_header()` `ddp_read_routing_rx_chan_status_response_chan_params()`
Routing RX Flow Status	`ddp_add_routing_rx_flow_status_request()`	`ddp_read_routing_rx_flow_status_response_header()` `ddp_read_routing_rx_flow_status_response_flow_params()`
Routing RX Subscribe	`ddp_add_routing_rx_sub_set_request()`	`ddp_read_routing_rx_sub_set_response_header()` `ddp_read_routing_rx_sub_set_response_chan_params()` `ddp_read_routing_rx_sub_set_response_custom_encoding()`
Routing RX Unsubscribe	`ddp_add_routing_rx_unsub_chan_request()`	`ddp_read_routing_rx_unsub_chan_response()`
Routing RX Channel Label Set	`ddp_add_routing_rx_chan_label_set_request()`	`ddp_read_routing_rx_chan_label_set_response_header()` `ddp_read_routing_rx_chan_label_set_response_chan_params()` `ddp_read_routing_rx_chan_label_set_response_custom_encoding()`

Message Type	Build Function (for TX query/command)	Parse Function (for RX response/event)

Routing TX Channel Label Set	`ddp_add_routing_tx_chan_label_set_request()`	`ddp_read_routing_tx_chan_label_set_response_header()`
		`ddp_read_routing_tx_chan_label_set_response_chan_params()`
		`ddp_read_routing_tx_chan_label_set_response_custom_encoding()`

Routing Multicast TX Flow Configuration	`ddp_add_routing_multicast_tx_flow_config_request()`	`ddp_read_routing_multicast_tx_flow_config_response_header()`
		`ddp_read_routing_multicast_tx_flow_config_response_flow_params()`
		`ddp_read_routing_multicast_tx_flow_config_response_flow_slots()`
		`ddp_read_routing_multicast_tx_flow_config_state_response_flow_address_nw_ip()`
		`ddp_routing_multicast_tx_flow_config_add_slot_params()`

Routing Flow Delete	`ddp_add_routing_flow_delete_request()`	`ddp_read_routing_flow_delete_response()`

Getting Started

The purpose of this section is to provide information on how to get stared with the Host CPU SDK package.

1. **Configure the Dante device**: The initial step should be to configure the Brooklyn II or the Ultimo for communication with a host CPU. Refer to Packet Bridge, Dante Events (Ultimo only), and Dante Device Protocol.
2. **Connect the Dante device to the computer**:
   - If a Brooklyn II PDK is available then connect a UART cable (e.g. FTDI serial to USB cable) to the UART port configured for host CPU communication.
   - If an Ultimo PDK is available then connect a UART cable to UARTB. Alternatively, if TotalPhase Inc. Aardvark host adapters are available then connect them as appropriate.
   - Use Table 10 when connecting an Aardvark SPI master to a Brooklyn II SPI slave on port B. Use Table 11 and Table 12 when connecting an Aardvark SPI slave and Aardvark SPI master respectively to an Ultimo SPI master + SPI slave.

Table 10 - Brooklyn II PDK SPI slave to Aardvark SPI master pin mapping

SPI Function	Aardvark SPI Master Header	Brooklyn II PDK SPI Port B Header
Clock	SCK - Pin 7	CLK - Pin 3
Slave Select	SS - Pin 9	SEL - Pin 1
Ground	GND - Pin 2	GND - Pin 8
Master in Slave out	MISO - Pin 5	MISO - Pin 7
Master out Slave in	MOSI - Pin 8	MOSI - Pin 5
Data Available	GPIO - Pin 1	Data Available - Pin 2

Table 11 - Ultimo PDK SPI master to Aardvark SPI slave pin mapping

SPI Function	Aardvark SPI Slave Header	Ultimo PDK SPI Master Header
Clock	SCK - Pin 7	CKA - Pin 3
Slave Select	SS - Pin 9	S0A - Pin 1
Ground	GND - Pin 2	GND - Pin 8
Master in Slave out	MISO - Pin 5	DIA - Pin 7

SPI Function	Aardvark SPI Slave Header	Ultimo PDK SPI Master Header
Master out Slave in	MOSI - Pin 8	DOA - Pin 5

Table 12 - Ultimo PDK SPI slave to Aardvark SPI master pin mapping

SPI Function	Aardvark SPI Master Header	Ultimo PDK SPI Slave Header
Clock	SCK - Pin 7	CKB - Pin 18
Slave Select	SS - Pin 9	SLB - Pin 20
Ground	GND - Pin 2	GND - Pin 13
Master in Slave out	MISO - Pin 5	DOB - Pin 14
Master out Slave in	MOSI - Pin 8	DIB - Pin 16

3. **Open the Visual solution file**: Use Microsoft Visual Studio (currently 2015 known as Visual Studio Community, it can be obtained from Microsoft's website - minimum supported version is 2013) to open the `vs_project/hostcpu_api.sln` solution file. Build the example project of interest which is based on the transport protocol (BHIP or UHIP) and the serial peripheral.
4. **Run the examples**: Open a command line window and simply execute the executable built by Visual Studio. There are no command line options. Alternatively, the example can be run by Visual Studio.
5. **Test receiving DDP events**: If the Brooklyn II or Ultimo is configured for DDP, the reboot the Dante device to view boot up events. Otherwise, use Dante Controller, to change sample rate, encoding, etc. and view the corresponding DDP event.
6. **Test sending DDP messages**: Uncomment one of the `hostcpu_bhip_set_tx_flag()` function calls of interest in `src/app/example/example_bhip/example_bhip_main.c`, or `hostcpu_uhip_set_tx_flag()` in `src/app/example/example_bhip/example_uhip_main.c`, re-build the Visual Studio project and run the executable. After running the executable the DDP response can be viewed.
7. **Port the examples projects to a target platform**: The steps involved in porting the example code and library code contained in the Host CPU SDK package are discussed in detail in Porting the Host CPU SDK Package. To get started execute the `create_brooklyn-ii_hostcpu.bat` for a host CPU targeting communication with a Brooklyn II or execute `create_ultimo_hostcpu.bat` for a host CPU targeting communication with an Ultimo. These batch files will group the relevant source files without Visual Studio project files into a separate directory.
8. **Read the API documentation**: Doxygen-styled documentation is provided for low level functionality to process and build packets for the various serial protocols covered in this document. These docs can be accessed by opening the `docs/docs.html` file.

Index