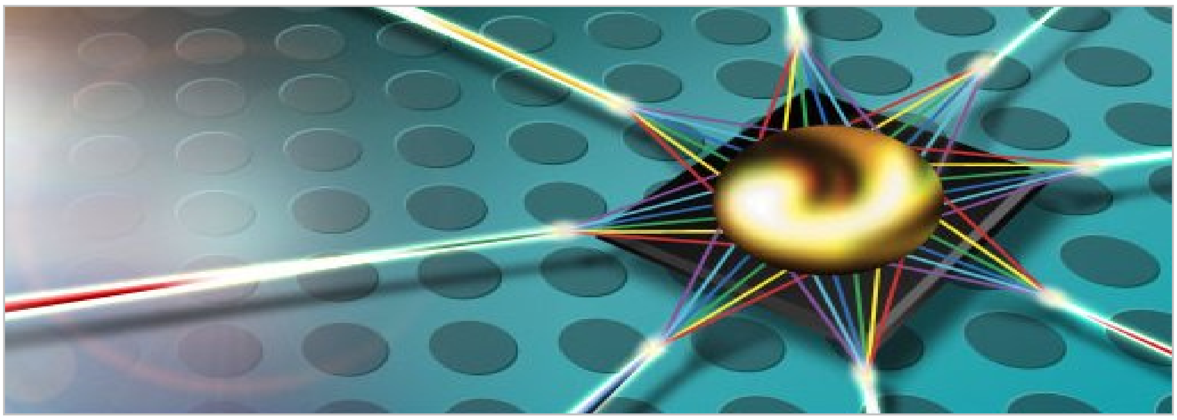
NPS-400
EZdev Reference Manual

Device Access Layer Library for NPS-400 Network Processors

Document Version 1.9



Document Number: 27-8216-04

EZchip Semiconductor Ltd.
1 Hatamar Street, P.O. Box 527
Yokneam 20692, Israel

EZchip Semiconductor Inc.
2700 Zanker Road, Suite 150
San Jose, CA 95134, USA

Preface

EZchip Semiconductor Inc.
2700 Zanker Road, Suite 150,
San Jose, CA 95134, USA
Tel: (408) 520-3700, Fax: (408) 520-3701

EZchip Semiconductor Ltd.
1 Hatamar Street, PO Box 527,
Yokneam 20692, Israel
Tel: +972-4-959-6666, Fax: +972-4-959-4166

Email: info@ezchip.com, Web: www.ezchip.com

EZchip welcomes your comments on this publication. Please address them to: supportNP@ezchip.com.

About this Manual

This document describes the EZchip Device Access Layer library (EZdev) and its related APIs. The EZdev library defines and implements the services required for accessing NPS devices, such as detecting the devices on the PCI Express bus, mapping the devices to the CPU address space, performing memory accesses to the devices and handling interrupt event notifications from the devices.

This manual is intended for software developers who plan to develop boards based on the EZchip NPS-400 network processor.

Related Documents

For additional information refer to:

DOCUMENT	*CONTENT*
EZcp Reference Manual	Describes the EZchip Control Plane Service library (EZcp) and its related APIs. The EZcp library provides an API for control-plane applications for the NPS network processor, abstracting the complexities of the underlying hardware.
EZenv Reference Manual	Describes the EZchip Environment library (EZenv) and its related APIs. The EZenv library provides a shared runtime infrastructure for all Control Plane Environment (CPE) libraries.

This Document

The following is a brief description of the contents of each section:

CHAPTER	*NAME*	*DESCRIPTION*
Section 1	**Overview**	Introduction to the EZdev library, its architecture and implementation.
Section 2	**Services**	Describes the services provided by the EZdev implementations.
Section 3	**Interface (EZdev.h)**	Describes the routines and calls relating to the EZdev functionality.
Section 4	**Reference Implementations**	Describes the various reverence implementations supplied.
Appendix A: NPS-400 PCI Identification		NPS-400 device identification in the PCI configuration space.

Revision History

REVISION	DATE	DESCRIPTION OF MODIFICATION
1.9	Sept. 8, 2015	Relates to EZdk version 1.9a. Modifications throughout the document. EZdev API routines were updated. REAL reference implementation for the device access layer replaced Split. Appendix A: NPS-400 PCI Identification added.
1.8	March 4, 2015	Relates to EZdk version 1.8a. EZvpci library was integrated into the EZdev library. The alternative reference implementations supplied for the device access layer were changed. EZdev API routines were updated.
1.7	Nov. 5, 2014	Relates to EZdk version 1.7a. No changes to the document.
1.6	July 17, 2014	Relates to EZdk version 1.6a. No changes to the document.
1.5	March 10, 2014	Initial release. Relates to EZdk version 1.5a.

Terminology and Conventions

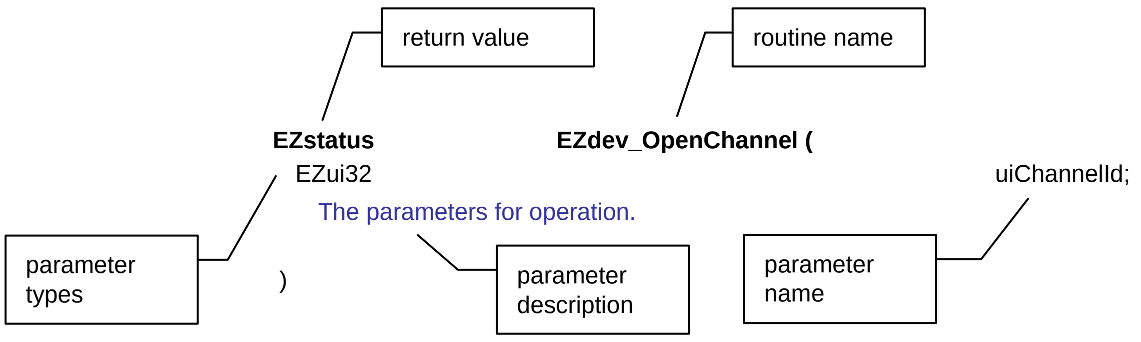
General

The following terminology is used throughout this document:

TERM	DESCRIPTION
NPS	Refers to the EZchip NPS-400 network processor device and/or software simulator.
Channel	Refers to an NPS-400 device and/or simulator in the system
Control Plane Application	Refers to a customer-developed application responsible for configuration and management of the NPS device.
Control Plane CPU	Refers to the CPU on which the control plane application resides. This may be an external host CPU or the NPS CTOPs.

Conventions Used for Routines

Routines, or functions/calls, are designated by the keyword EZstatus followed by the function name enclosed in parentheses. Parameters are listed in parentheses.



Typographical Conventions

The following typographical conventions are used in this manual. Routine (or function/call) names are written with a parenthesis, e.g. **EZdev_OpenChannel( )**.

📖 Refer to the section or document referenced here for additional information on the topic.

▸ *Notes provide additional information that is not necessarily mandatory.*

Important: Contains information that is mandatory for proper confirmation and/or operation that should not be overlooked.

Contents

List of Figures and Tables

1. Overview

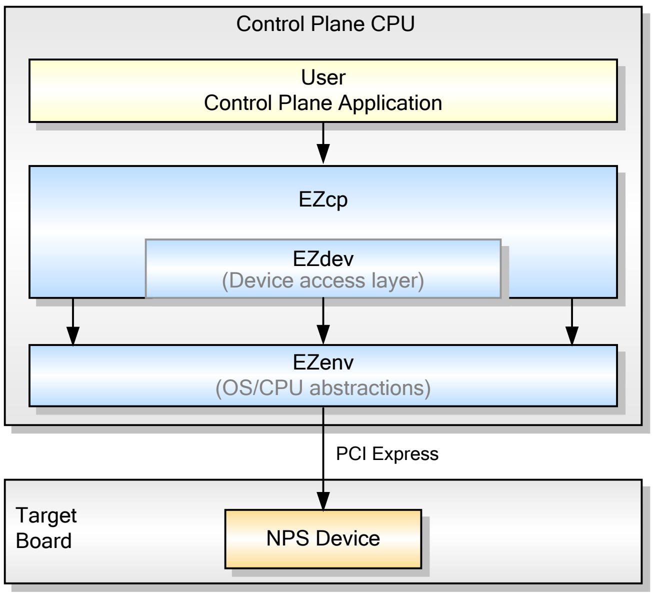
The EZdk Control Plane Environment (CPE) libraries are designed to be easily ported to a variety of target platforms.

All operating system functionality used by the EZdk Control Plane Environment libraries is encapsulated in a separate OS layer (within the EZenv library), allowing customers to easily port OS requirements to any operating system. In addition, the CPE libraries support compilation modes for both little-endian and big-endian CPUs, as well as for CPUs requiring aligned memory accesses.

The EZchip device access layer (EZdev) further extends the CPE libraries' portability by defining the services required for accessing NPS devices, such as detecting the devices on the PCI Express bus, mapping the devices to the CPU address space, performing memory accesses to the devices and handling interrupt event notifications from the devices.

The EZcp library is connected to the device access layer using a set of device access functions. While the EZcp library is responsible for issuing NPS device access requests, the device access layer provides the platform specific services to execute these requests.

Figure 1. High level overview of the Control Plane Environment

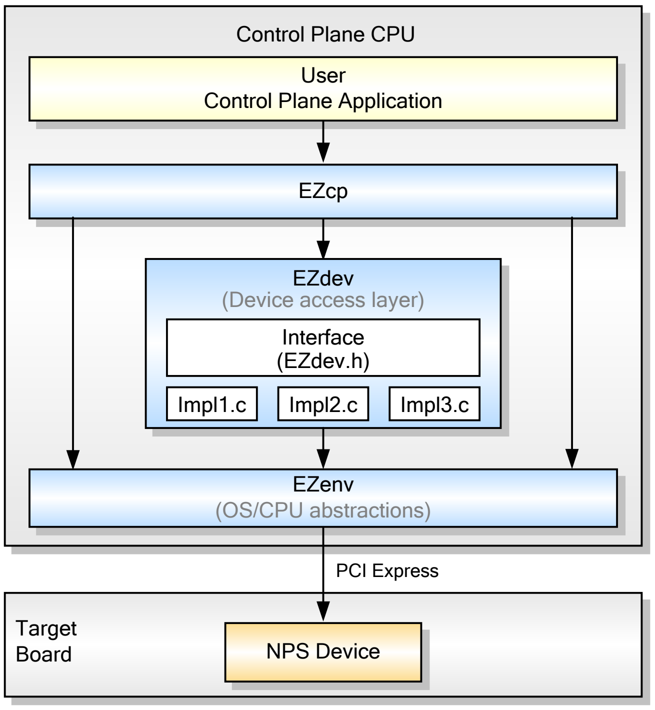


The services provided by the device access layer include:

- Device detection and address mapping.
- Device memory access (read/write).
- Device interrupt events handling and notification.

The device access library defines the interface required for device access layer implementations. Multiple device access layer implementations may be developed to match different target platforms (OSs, CPUs, and or target boards) with the actual implementation used selected at compilation time or at run time.

Figure 2. Device access library interface



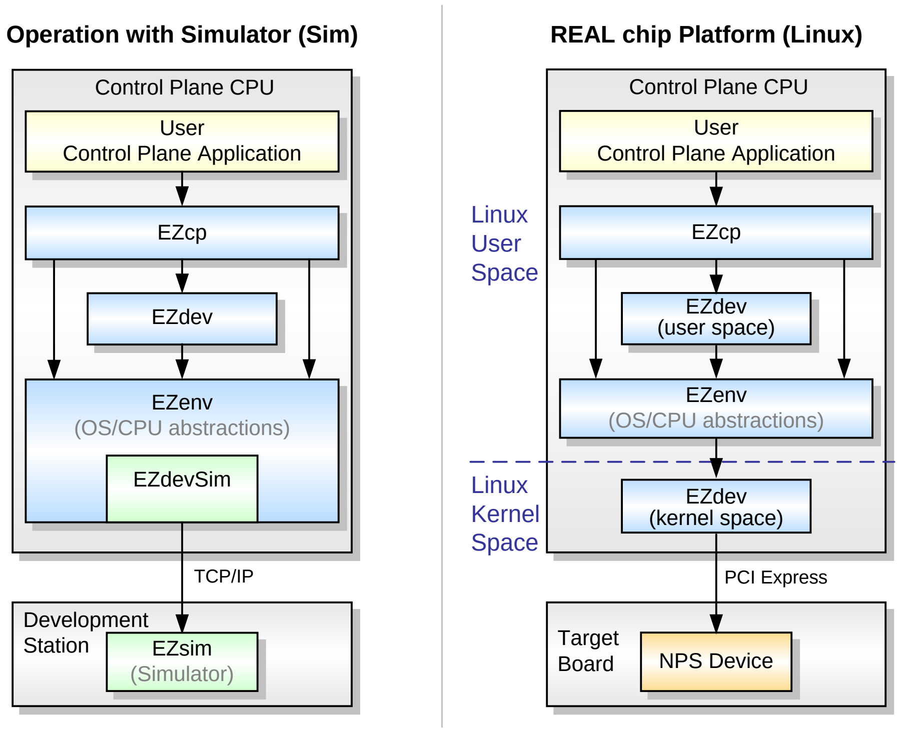
1.1 Device Layer Reference Implementations

The EZdev library supplied in the EZdk Software Development Kit provides the following alternative reference implementations for the device access layer (Figure 3).

Embedded Host Block –	This implementation is used to operate directly with the embedded host block – a library supporting minimal NPS simulation capabilities for functionality related to host interaction as well as for dump generation. ▸ *For more information see Embedded Host Block section.*
Null Host Block –	This implementation is used to operate with a stub/dummy host block implementation – a set of routines performing no actual functionality other than returning hardcoded values allowing to successfully complete the configuration sequence to the EZcp library. ▸ *For more information see Null Host Block section.*
Sim –	This implementation is used to interface with the EZchip software simulator (EZsim). This is done using a TCP/IP connection to the software simulator, and includes services for performing memory read/write transactions to the software simulator as well as services for receiving virtual interrupts from the software simulator. ▸ *For more information see Sim (EZdevSim) section.*

REAL –	This implementation is used to interface with the NPS-400 real device. In this case, the device access layer implementation is based on the UIO framework. UIO requires a tiny kernel-space code module to set up the device, interfaces to the PCIe bus, and register an interrupt handler, but most of the device related operations are implemented in the user space code. The EZcp library functional code runs in user space, and accesses the user space portion of the platform layer. *For more information see REAL (EZdevREAL) section.*
CTOP –	This implementation is used to interface between a control plane application running internally on a CTOP core and the underlying hardware.

Figure 3. Various reference implementations offered



In many cases, the implementations provided in the EZdk Software Development Kit can be used as is. Alternatively, the provided implementation may be used as a reference to assist customers in their development of implementations optimized for specific target platforms.

1.2 Endianness

The device access layer supports both little and big endian format. The device access layer provides services to read from and write to the device, and distinguishes between 32 bit word access and byte stream access. In case of 32 bit word access, the endianness is like the CPU, whereas in case of byte stream access, byte[0] holds the first byte and byte[n] the nth byte. For additional flexibility, two compilation switches are provided to control the swapping of 32 bit words and byte streams

- EZ_DEV_VAL_SWAP – swap a 32 bit word after reading from the device, or before writing to the device.
- EZ_DEV_BUF_SWAP – swap the 4 bytes after reading a 32 bit word from the device, or before writing a 32 bit word to the device.

2. Services

The following sections describe the services provided by the EZdev implementations as well as the underlying implementations and requirements.

2.1 Device Detection & Mapping

The device access layer provides services for detecting NPS devices on the PCIe bus and mapping the NPS device registers to the CPU memory space.

The device detection part is handled by the kernel-space module. The detection is achieved by matching the vendor ID, device ID, vendor sub system ID and device sub system ID. Once there is a match, a uioX directory is added with the detected device's properties.

When the NPS CP is created, the device access layer user-space portion scans the UIO directory tree, detects the NPS devices and generates the database with the NPS devices. This includes the mapping between the logical device ID (e.g. channel ID) and the physical PCIe device (uioX device). When a channel device is created, the device access layer tries to open the already detected device for read/write access and maps the device to the CPU memory space (using mmap). From that point, the device is accessed from the user-space portion.

2.2 Memory Access

The device access layer provides services for accessing the NPS device registers via the PCIe interface.

When a channel device is created, the device's configuration space is mapped to the CPU memory using mmap(). Once mmap succeeds, all accesses to the device are performed from the user space portion.

2.3 Interrupt Events

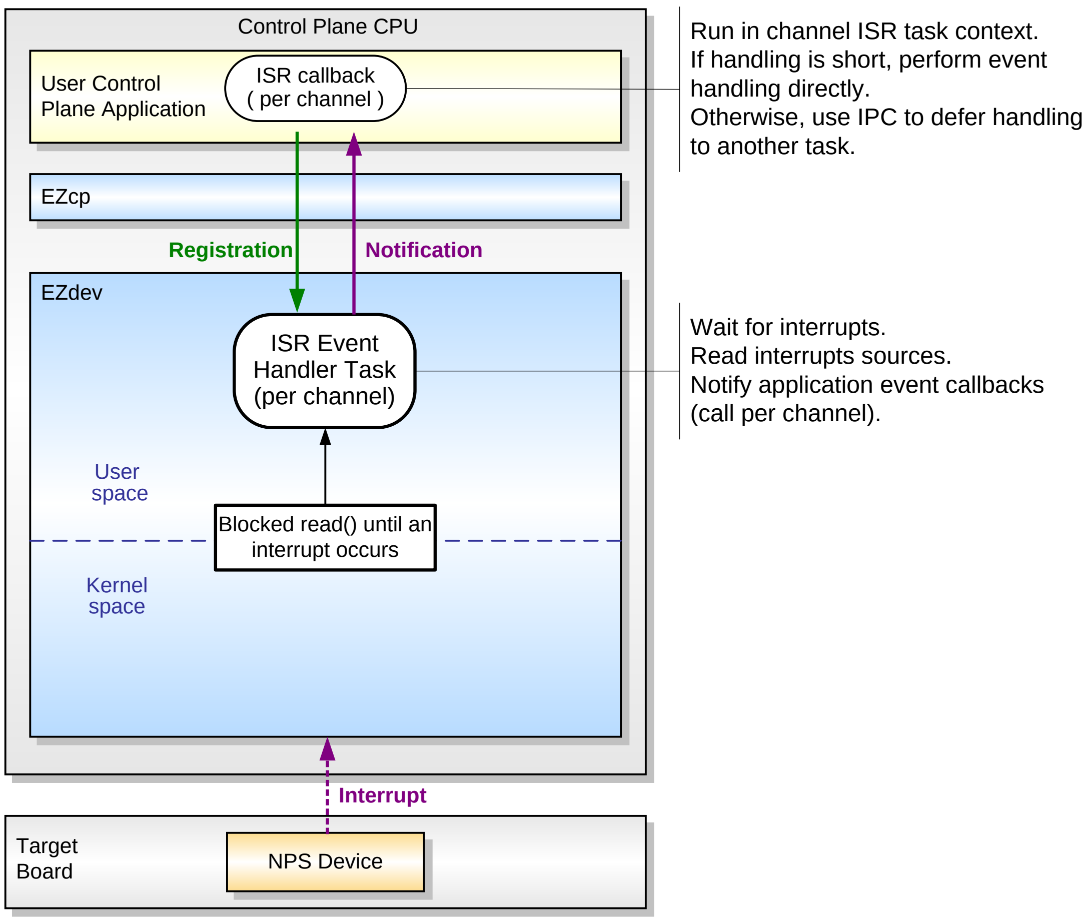
The device access layer provides services for handling NPS interrupt events. The kernel-space module enables PCIe MSI interrupts. All other interrupt related services are done in the user-space context.

The application uses the device layer services to register a user-defined ISR events handler for each NPS device.

When an interrupt triggers, the device access layer handles the interrupt and notifies the application of the detected interrupt events. The device access layer interrupt handling is divided into two portions:

- A dedicated task which is blocked until an interrupt occurs. When an interrupt is triggered, the task scans for any pending interrupt events and notifies the application event-handlers registered for the matching device.
- Device access layer services to enable, disable and acknowledge interrupt events.

Figure 4. Device access layer interrupt handling


Control Plane CPU
User Control Plane Application
ISR callback (per channel)
Run in channel ISR task context. If handling is short, perform event handling directly. Otherwise, use IPC to defer handling to another task.
EZcp
Registration
Notification
EZdev
ISR Event Handler Task (per channel)
Wait for interrupts. Read interrupts sources. Notify application event callbacks (call per channel).
User space
Blocked read() until an interrupt occurs
Kernel space
Interrupt
Target Board
NPS Device

3. Interface (EZdev.h)

3.1 Overview

The device access layer interface (as defined in the EZdev.h file) defines the services required to be provided by device access layer implementations.

The device layer interface defines a single common device, used for general control operations and interrupt event notifications, as well as a dedicated channel device for each NPS device, used for device memory access and device-specific control operations.

The following table summarizes the device layer interface:

Table 1. Summary of EZdev API routines

Interface Level	
EZdev_Create( )	Create the device access layer object.
EZdev_Delete( )	Delete the device access layer object.
EZdev_GetVersionInfo( )	Return structure with version information.
Common Device	
EZdev_OpenCommon( )	Creates/opens the common device
EZdev_CloseCommon( )	Deletes/closes the common device.
EZdev_ControlCommon( )	Perform one of the following operations: ▪ Connect – Connect/start the common device. ▪ Disconnect – Disconnect/stop the common device.
Channel Devices	
EZdev_OpenChannel( )	Creates/opens the channel device.
EZdev_CloseChannel( )	Deletes/closes the channel device.
EZdev_ControlChannel( )	Perform one of the following operations: ▪ Connect – Connect/start the channel device. ▪ Disconnect – Disconnect/stop the channel device. ▪ Set interrupt related parameters, control and acknowledge interrupts. ▪ Get PCI info – Get PCI configuration space information for the channel (revision ID).
EZdev_ReadValChannel( )	Read a 32-bit value from a channel device.
EZdev_WriteValChannel( )	Write a 32-bit value to a channel device.
EZdev_ReadBufChannel( )	Read a burst of bytes from a channel device.
EZdev_WriteBufChannel( )	Write a burst of bytes to a channel device.

The following tables summarize when each of the device layer services are invoked by the EZcp library (CP API Group routines and commands are in yellow):

Table 2. Invocation of common device layer services by the CP API Group

FUNCTION	*ROUTINE / COMMAND*	*NOTES*
CP Create	EZapiCP_Create()	
No affect		
CP Go	EZapiCP_Go()	
Open common device	EZdev_OpenCommon()	
Connect common device	EZdev_ControlCommon() EZdev_ControlChannelCmd_Connect	
CP Delete	EZapiCP_Delete()	
Disconnect common device	EZdev_ControlCommon() EZdev_ControlCommonCmd_Disconnect	
Close common device	EZdev_CloseCommon()	

Table 3. Invocation of channel device layer services by the Channel API Group

FUNCTION	*ROUTINE / COMMAND*	*NOTES*
Channel Create	EZapiChannel_Create()	
Open channel device	EZdev_OpenChannel()	
Connect channel device	EZdev_ControlChannel() EZdev_ControlChannelCmd_Connect	
Get PCI info for channel device	EZdev_ControlChannel() EZdev_ControlChannelCmd_GetPCIInfo	Read updated values to CP library cache after device layer detection process.
Channel Get PCI Info	EZapiChannel_Status() EZapiChannel_StatCmd_GetPCIInfo	
No affect		Read from CP library cache.
Channel Interrupt Handling	EZapiChannel_Config()	
Set interrupt parameters	EZdev_ControlChannel() EZdev_ControlChannelCmd_SetInterrupts Params	
Control interrupts	EZdev_ControlChannel() EZdev_ControlChannelCmd_SetInterrupts Control	
Acknowledge interrupts	EZdev_ControlChannel() EZdev_ControlChannelCmd_SetInterrupts Acknowledge	
Channel Delete	**EZapiChannel_Delete()**	
Disconnect channel device	EZdev_ControlChannel() EZdev_ControlChannelCmd_Disconnect	
Close channel device	EZdev_CloseChannel()	

3.2 Reference

This section lists each API routine in the group followed by its structures and enumerations.

3.2.1 Interface Level

3.2.1.1 EZdev_Create( )

Description

Create the device access layer object.

Synopsis

```
EZstatus        EZdev_Create (
  EZdev_PlatformParams*                         psPlatformParams
    Platform implementation.
)
```

Associated Structures

```
struct          EZdev_PlatformParams {
  EZdev_Platform                                ePlatform;
    EZdev target platform.
  EZui32                                        uiChipPhase;
    Phase/step of the NPS device in use.
}
```

```
enum            EZdev_Platform {
  EZdev_Platform_INVALID
    Invalid platform.
  EZdev_Platform_NULL_HB
    NULL Host Block platform, run on external host.
  EZdev_Platform_EMB_HB
    Embedded Host Block platform, run on external host, connect to EZsimHB library.
  EZdev_Platform_SIM
    Simulator platform, run on external host, connect to EZsim application.
  EZdev_Platform_REAL
    REAL platform, run on external host, connect to NPS chip thru PCIe interface.
  EZdev_Platform_CTOP
    CTOP platform, run internally on CTOP, connect to NPS chip thru internal interface.
}
```

3.2.1.2 EZdev_Delete( )

Description

Delete the device access layer object.

Synopsis

EZstatus EZdev_Delete (void)

3.2.1.3 EZdev_GetVersionInfo( )

Description

Return structure with version information.

Synopsis

EZversionInfo *EZdev_GetVersionInfo (void)

3.2.2 Implementation Level

3.2.2.1 EZdev_OpenCommon( )

Description

Open the common device.

Synopsis

EZstatus EZdev_OpenCommon (void)

Function Pointer Prototype

typedef EZstatus (*EZdev_OpenCommonDev_FuncPtr)(void);

3.2.2.2 EZdev_CloseCommon( )

Description

Close the common device.

Synopsis

EZstatus EZdev_CloseCommon (void)

Function Pointer Prototype

typedef EZstatus (*EZdev_CloseCommonDev_FuncPtr)(void);

3.2.2.3 EZdev_ControlCommon( )

Description

Control the common device.

Synopsis

EZstatus EZdev_ControlCommon (

EZdev_ControlCommonCmd eCmd,

Control command.

EZptr pData

Pointer to command data.

)

Function Pointer Prototype

typedef EZstatus (*EZdev_ControlCommonDev_FuncPtr)
(EZdev_ControlCommonCmd eCmd, EZptr pData);

Commands

Common device control commands and associated structures.

enum EZdev_ControlCommonCmd {

EZdev_ControlCommonCmd_Connect

Connects the common device.
The associated structure is NULL.

EZdev_ControlCommonCmd_Disconnect

Disconnects the common device.
The associated structure is NULL.

}

3.2.2.4 EZdev_OpenChannel( )

Description

Open a channel device.

Synopsis

```
EZstatus        EZdev_OpenChannel (
  EZui32                                      uiChannelId
    Channel identification number.
)
```

Function Pointer Prototype

typedef EZstatus (*EZdev_OpenChannelDev_FuncPtr)(EZui32 uiChannelId);

3.2.2.5 EZdev_CloseChannel( )

Description

Close a channel device.

Synopsis

```
EZstatus        EZdev_CloseChannel (
  EZui32*                                     uiChannelId
    Channel identification number.
)
```

Function Pointer Prototype

typedef EZstatus (*EZdev_CloseChannelDev_FuncPtr)(EZui32 uiChannelId);

3.2.2.6 EZdev_ControlChannel( )

Description

Control a channel device.

Synopsis

```
EZstatus        EZdev_ControlChannel (
  EZui32                                   uiChannelId,
    Channel identification number.
  EZdev_ControlChannelCmd                  eCmd,
    Control command.
  EZptr                                    pData
    Pointer to command data.
)
```

Function Pointer Prototype

```
typedef EZstatus (*EZdev_ControlChannelDev_FuncPtr)
                  ( EZui32 uiChannelId, EZdev_ControlChannelCmd eCmd, EZptr pData );
```

Commands

Channel device control commands and associated structures.

```
enum            EZdev_ControlChannelCmd {
  EZdev_ControlChannelCmd_Connect
    Connects the channel device. This command detects the device on the configuration space,
    maps it to the CPU memory space, and registers an ISR for the device.
    The associated structure is NULL.
  EZdev_ControlChannelCmd_Disconnect
    Disconnects the channel device. This command unmaps the device from the CPU memory
    space and unregisters the ISR for the device.
    The associated structure is NULL.
  EZdev_ControlChannelCmd_SetInterruptsParams
    Set the ISR callback to receive interrupts notifications.
    The associated structure is EZdev_InterruptsParams (=EZapiChannel_InterruptsParams).
  EZdev_ControlChannelCmd_SetInterruptsControl
    Control interrupts sources.
    The associated structure is EZdev_InterruptsControl (=EZapiChannel_InterruptsControl).
  EZdev_ControlChannelCmd_SetInterruptsAcknowledge
    Acknowledge interrupts sources.
    The associated structure is EZdev_InterruptsAcknowledge
    (=EZapiChannel_InterruptsAcknowledge).
  EZdev_ControlChannelCmd_GetPCIInfo
    Get the device's PCI configuration space information.
    The associated structure is EZdev_PCIInfo (=EZapiChannel_PCIInfo).
}
```

Associated Structures

```
struct          EZdev_PCIInfo {
  EZapiCP_Version                            eVersion;
    The version of the structure.
  EZui32                                     uiRevisionId;
    PCI configuration space revision ID.
}
```

3.2.2.7 EZdev_ReadValChannel( )

Description

Read a 32-bit value from a channel device.

Synopsis

```
EZstatus        EZdev_ReadValChannel (
  EZui32                                          uiChannelId,
    Channel identification number.
  EZui32                                          uiOffset,
    Data offset.
  EZui32*                                         puiData
    Data.
)
```

Function Pointer Prototype

```
typedef EZstatus (*EZdev_ReadValChannelDev_FuncPtr)
           ( EZui32 uiChannelId, EZui32 uiOffset, EZui32 *puiData );
```

3.2.2.8 EZdev_WriteValChannel( )

Description

Write a 32-bit value to a channel device.

Synopsis

```
EZstatus        EZdev_WriteValChannel (
  EZui32                                          uiChannelId,
    Channel identification number.
  EZui32                                          uiOffset,
    Data offset.
  EZui32                                          uiData
    Data.
)
```

Function Pointer Prototype

```
typedef EZstatus (*EZdev_WriteValChannelDev_FuncPtr)
           ( EZui32 uiChannelId, EZui32 uiOffset, EZui32 uiData );
```

3.2.2.9 EZdev_ReadBufChannel( )

Description

Read bytes from a channel device.

Synopsis

```
EZstatus        EZdev_ReadBufChannel (
  EZui32                                        uiChannelId,
    Channel identification number.
  EZui32                                        uiOffset,
    Data offset.
  EZui32                                        uiSize,
    Size of data in bytes.
  EZuc8*                                        pucData
    Stream of data to print (in 8 bit segments).
)
```

Function Pointer Prototype

```
typedef EZstatus (*EZdev_ReadBufChannelDev_FuncPtr)
          ( EZui32 uiChannelId, EZui32 uiOffset, EZui32 uiSize, EZuc8 *pucData );
```

3.2.2.10 EZdev_WriteBufChannel( )

Description

Write bytes to a channel device.

Synopsis

```
EZstatus        EZdev_WriteBufChannel (
  EZui32                                        uiChannelId,
    Channel identification number.
  EZui32                                        uiOffset,
    Data offset.
  EZui32                                        uiSize,
    Size of data in bytes.
  EZuc8*                                        pucData
    Stream of data to write (in 8 bit segments).
)
```

Function Pointer Prototype

```
typedef EZstatus (*EZdev_WriteBufChannelDev_FuncPtr)
          ( EZui32 uiChannelId, EZui32 uiOffset, EZui32 uiSize, EZuc8 *pucData );
```

4. Reference Implementations

This section provides details on the various device layer reference implementations supplied in the EZdk Software Development Kit and their underlying data structures.

4.1 Sim (EZdevSim)

The Sim implementation (EZdevSim) implements the EZdev interface for operating with the EZsim software simulator. The Sim implementation supports connectivity of up to 32 network processors (EZsim software simulator instances).

The Sim implementation uses two tasks/threads:

- Server Thread – The server thread continually listens for incoming EZsim socket connections, and is responsible for setting up the connections. Two parallel socket connections are generated for each EZsim: one for data operations and the other for virtual interrupt events. When the server thread receives new EZsim connections, it registers the connections in the related data.
- ISR Thread – The ISR thread continually listens for incoming virtual ISR events on any of the connected channels. When the thread detects a virtual ISR event, it identifies the source channel, reads the ISR message, and then calls the ISR callback function to notify the application of the virtual interrupt event. The ISR thread also monitors the EZsim connections. If it detects an aborted connection, it closes the matching sockets and removes the connection from the related data structures.

4.2 Embedded Host Block

The Embedded Host Block implementation is used to operate with the embedded host block library. This library is a minimized EZsim component that may be linked with the control plane application to supply in-process host interaction and dump generation capabilities. A stub function implementation is provided to enable linking of the embedded host block implementation without the host block library.

When working in this mode:

- Only a single device/instance is supported (channel ID 0).
- Read/write operations are performed using direct function calls to the embedded host block library.
- The connections array is not used. Thus, no TCP connections and/or device mappings are required.

4.3 Null Host Block

The Null Host Block implementation is used to operate with a stub/dummy host block implementation – a set of routines perform no actual functionality other than returning hardcoded values allowing to successfully complete the configuration sequence to the EZcp library.

When working in this mode:

- All 32 channels are connected.
- Write data operations do nothing.
- Read data operations return hardcoded values allowing to successfully complete the configuration sequence. This is comprised mainly of setting the various NPS register ready bits to 1 and the empty/fail bits to 0.

- The connections array is not used. Thus, no TCP connections and/or device mappings are required.

Limitations:

- Not supported for applications using entries operation for a hash search structure with static learn mode.

4.4 REAL (EZdevREAL)

The REAL implementation (EZdevREAL) implements the EZdev interface for platforms with memory protection isolating user-space applications from OS kernel and hardware resources (e.g. Unix/ Linux variants).

The REAL implementation is divided between the user space and kernel space. The implementation internally handles the required communications between the user space portion and the kernel space portion which performs the actual hardware accesses.

4.4.1 Kernel Space

The kernel space access layer module is based on the UIO framework. The kernel module is tiny, and provides only the necessary implementation in the kernel mode, allowing the majority of the access layer driver to be implemented in the user space.

Device Detection

The kernel module implements the probe() function which is called when a device is detected, according to the device ID table, filled with the NPS-400 vendor ID and device ID. Once an NPS-400 device is detected, BAR4 is mapped to the CPU memory space, allowing access to the NPS-400 configuration spaces.

Interrupt Handling

The kernel module implements a blocking read() function which returns control to the user space once an interrupt occurs.

4.4.2 User Space

Memory Mapping

Once a NPS-400 device is detected and registered, the access layer user space portion can map BAR4 to the CPU memory space using the mmap() service. Then, all accesses to the NPS-400 configuration spaces are performed from the user space context.

Interrupt Handling

The user space portion spawns a dedicated ISR task per channel. The ISR task is blocked on the call to read(). Once an interrupt triggers, the blocking is released, the ISR task scans for the interrupt(s) cause and calls the already registered ISR callback, notifying about the triggered interrupt(s).

Logging

The user space logging is performed using the EZlog infrastructure in the EZenv library to the device layer component (EZlog_COMP_DEV). Two sub-components are defined: common part logging (EZlog_SUB_COMP_DEV_COMMON) and REAL device logging (EZlog_SUB_COMP_DEV_REAL).

5. Appendix A: NPS-400 PCI Identification

When performing auto detection, the device layer implementation must check for the various possible/ expected combinations of the NPS-400 device identification in the PCI configuration space (e.g. vendor ID and device ID). The following table provides the NPS-400 device identification in the PCI configuration space:

PRODUCT	VENDOR ID	DEVICE ID	REVISION ID
NPS-400	0x1719	0x801	1