Insyde Firmware for Nuvoton Embedded Keyboard Controller User Guide

Revision 1.9

August 2015

REVISION RECORD

REVISION	RELEASE DATE	SUMMARY OF CHANGES
1.0	1.0	December 2007 First release.
1.1	February 2008	Includes a more detailed CEIR section.
1.2	October 2009	Nuvoton version. New sections: - 3.11.3 - 3.12 to 3.15
1.3	June 2010	- Updated OEM directory structure - Removed CPK SUPPORT - Updated 32K clock calibration (Section 3.14)
1.4	June 2011	Added the following sections: - PECI Interface (Section 3.16) - Running from MRAM Memory (Section 3.17) - Automatic HW Keyboard Scan (Section 3.18)
1.5	October 2011	Formatting changes. Added the following sections: - One-Wire Interface (Section 3.19) - Non-Volatile Memory (Section 3.20) Added NPCE885N chip
1.6	January 2012	Fixed Typos. Updated Section 3.20 (NON-VOLATILE MEMORY (NVM) INTERFACE)
1.7	August 2012	Fixed Typos. Updated Section
1.8	April 2014	Update SMBus transaction type, remove configuration tale.
1.9	August 2015	Update Host Command Hook paramaters.

PREFACE

This document gives a general overview of the Insyde Firmware for the Nuvoton Embedded Keyboard Controller. It also provides information on adding platform-specific code to the Insyde Firmware for the Nuvoton Embedded Keyboard Controller.

The information contained in this document is subject to change.

No part of this document may be reproduced in any form or by any means without the prior written consent of Nuvoton Technology Corporation.[1]

[1]All brand or product names are trademarks of their respective holders.

CONTENTS

Chapter 1
TERMINOLOGY AND ACRONYMS

The Insyde Firmware for the Nuvoton Embedded Keyboard Controller (EC) implements all the general functionality of the EC firmware. This functionality includes Internal Keyboard scan, External PS2 Keyboard and Mouse handling, host command handling and ACPI support.

When bringing up a new mobile platform with the Nuvoton EC, platform-specific code should be added to the Insyde Firmware for the Nuvoton EC. This code may contain the following platform-specific functionality: Power Sequence scheme, interrupt handling, Health (Thermal and Fan) control and Battery and Charger control.

This User Guide provides porting guidelines for EC firmware development and is intended for EC firmware developers.

1.1 USER GUIDE OVERVIEW

Chapter 2 - EC FIRMWARE OVERVIEW. Describes directory organization, boot/init sequence, firmware loop and services.

Chapter 3 - ADDING PLATFORM-SPECIFIC CODE TO EC FW. Describes how to add platform-specific functionality.

1.2 REFERENCES

- *Insyde EC FW* Reference *Manual*

Chapter 2
EC FIRMWARE OVERVIEW

The Insyde Firmware for the Nuvoton Embedded Keyboard Controller (EC) implements all the general functionality of the EC firmware. This functionality includes Internal Keyboard scan, External PS2 Keyboard and Mouse handling, host command handling and ACPI support.

2.1 DIRECTORY STRUCTURE

The EC firmware source files are divided into three main categories:

- CORE
- CHIP
- OEM

CORE and CHIP can be used for every platform. OEM is platform specific. CORE, CHIP and OEM parts each have their own directory.

2.1.1 Core

This is the main part of the EC firmware. This directory contains the logic and generic code of the EC firmware. It does not contain chip-related code (this is found in the CHIP directory) or platform-specific code (this is found in the OEM directory).

The Core part contains the following functionality:

- Main Loop
- Host Interface handling
- Host command parser and handler
- ACPI command handling
- External (PS2) Keyboard and Mouse handling
- Internal Keyboard scan

2.1.2 Chip

This is the chip-related part of the EC firmware. The code in this directory is the Hardware Abstraction Layer for the needed hardware modules of the Nuvoton EC chip.

The Chip part contains abstraction for the following chip hardware modules:

- Host Interface
- PS2
- Keyboard Scan
- I2C/SMB
- Interrupt Service Routines (IRQs)
- GPIO
- Timers
- Flash Interface Unit (FIU)
- CEIR

2.1.3 OEM

This is the platform-specific part of the EC firmware. Platform-specific code is implemented here.

The OEM part may contain the following functionality:

- Interrupt Control Unit (ICU)
- Hooks from CORE and CHIP
- Platform Power Sequence scheme
- Platform-specific interrupt handling
- Platform-specific Events handling
- Platform Health (Thermal and Fan) control
- Platform Battery and Charger control

OEM Directory Structure

The OEM directory structure is as follows:

- **OEM**
  - **PROJECT**

PROJECT Sub-Directory

The **PROJECT** sub-directory is under the **OEM** directory. This directory name can be changed to the target platform name. In this case, the following line (in **OEM\PROJECT\OEMBLD.MAK**) should also be changed to the new name:

OEMDIR=PROJECT

The PROJECT sub-directory contains the following sub-directories:

Sub-Directory	Description
CHIP	Contains Chip Register initialization and registers definition
INC	Contains OEM include files.
MOD	For Core and Chip file platform modifications. C files in **MOD\CORE** override the corresponding C files in the Core directory. H files in **MOD\CORE\INC** override H files in the **CORE\INC** directory. The same is the case for **MOD\CHIP**.
ROMSRC	Contains the scan table binary file (**SCANTAB1.BIN**) for the platform-specific internal keyboard.

2.2 MAIN FLOW

2.2.1 The Boot/Init Sequence

The EC firmware is composed of two major parts:

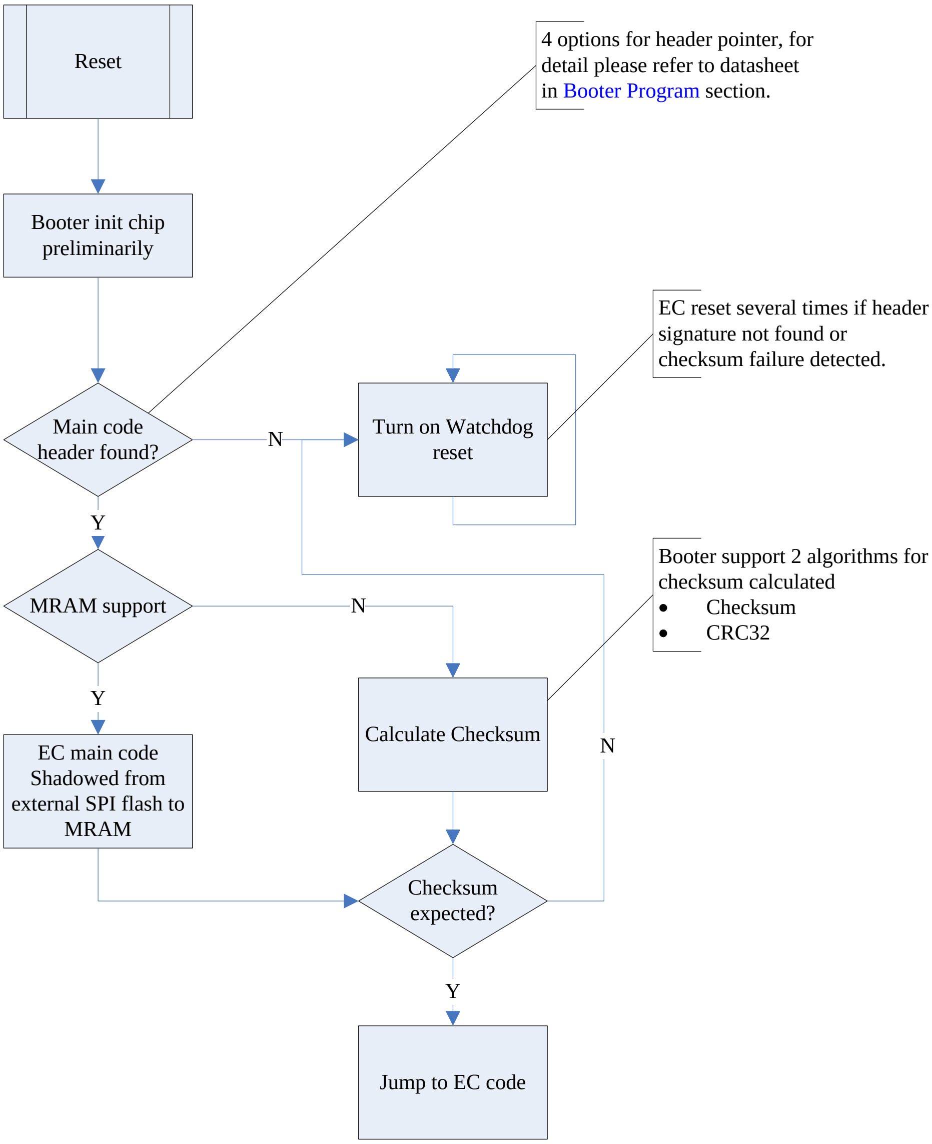
- **CRISIS** code - handles minimal functionality, to enable firmware flash update
- **MAIN Keyboard Controller (MAIN KBC)** code - handles all the regular functionality of the EC firmware

The Boot/Init sequence of the EC firmware starts with the Booter ROM code, which resides on the EC ROM, at the address zero. The sequence continues in the firmware's CRISIS code and then finishes in the firmware's MAIN KBC code. The firmware then enters the FW Main loop.

CRISIS code supported is a software switch (defined in OEM\PROJECT\OEMBLD.MAK) named CRISIS_PAGE. CRISIS code is enabled when "CRISIS_PAGE = 0" or disabled when "CRSIS_PAGE = none".

The Boot/Init sequence is described below in detail:

1. In ROM code
   a. Check the firmware flash header (defined in **CHIP\HEADER.C**).
   b. Perform basic hardware configuration, based on the parameters read from the firmware flash header.
   a. Perform checksum on the CRISIS code part of the firmware if CRISIS code is supported otherwise perform checksum on the main KBD code.
   d. If OK, jump to the beginning of the Crisis code part of the firmware when CRISIS code is enabled otherwise jump to main KBC code.



2. In CRISIS code
    a. Perform the CRISIS assembly start routine, defined in **CHIP\CRSISRST.S**.
    b. Perform the OEM hook function at the start of the CRISIS code (**OEM_Crisis_Reset()**, defined in **OEM\PROJECT\OEMCRSIS.C**).
    a. Perform the **Crisis_Reset()** routine, defined in **CHIP\CRISIS.C**:
        i. Initialize the FLASH UPDATE mechanism.
        ii. Perform checksum on the MAIN KBC code part of the firmware.
        iii. If OK, jump to the MAIN KBC code.
        iv. If it fails, remain in CRISIS code.
3. In MAIN KBC code
    a. Perform the MAIN KBC assembly start routine (defined in **CHIP\RESET.S**).
    b. Perform the **OEM_Reset()** routine, defined in **OEM\PROJECT\OEMRESET.C**:
        i. Initialize the RAM to zero.

ii. Fill the program stack with FFh.

a. Enter the **Main()** routine, defined in **CORE\PURMAIN.C**:

   i. Initialize chip registers by calling **Init_Regs() (from OEM\PROJECT\REGINIT.C)**. This function operates on the **Reg_Init[]** array. Each entry in this array holds a **REG_INIT_DESCRIPTOR** structure, which contains the register's address, size and initialization value. For each array entry, this function assigns the register initialization values to the register address, according to the register size.

   ii. Initialize the Interrupt Control Unit, **Icu_Init()**, defined in **OEM\PROJECT\ICU.C**.

   iii. Initialize the timers, **Timers_Init()**, defined in **CHIP\TIMERS.C**.

   iv. Initialize the host interface, **Host_If_Init()**, defined in **CHIP\HOST_IF.C**.

   v. Perform OEM initialization (**Hookc_Cold_Reset_Begin/End()**, defined in **OEM\PROJECT\OEMINI.C**).

   vi. Enter the firmware Main loop.

2.2.2 Firmware Main Loop

The EC firmware Main loop (**main_service()** in **CORE\PURMAIN.C**) continuously checks the 16-bit **Service** variable. Each bit in the **Service** variable represents a service. If a specific service bit is set, the Main loop passes control to the routine that handles this service and then continues checking the service bits; see Section 2.2.3 on page 6.***

The flow is passed to a service handling routine by the **service_table[]** function pointer array (defined in **CORE\PURMAIN.C**). For example, for service bit n, the corresponding service handling routine pointer is located at offset n of the **service_table[]** array. Therefore, when a service bit is set, it is used as the access key to the **service_table[]** array, to locate the appropriate service handling routine.

The services handled by the Main loop cover all the possible inputs that can influence the EC firmware. There are three main categories of inputs: host commands, interrupt events and periodic events.

Host Commands

These commands are sent to the firmware from the host BIOS or application, via the host interface channels. They instruct the firmware to perform actions and then send a response to the host. The relevant services for host commands are **PCI/2/3**. These services are set by the host interface interrupt handlers.

For complete details on the supported host commands, see the "*Insyde EC FW Reference Manual"* (*Insyde_EC_FW_Reference_Manual.pdf*).

Interrupt Events

Asynchronous events, for example, a keystroke on the scanned keyboard. In this case, the relevant service is **KEY**, which is set by the Keyboard Scan interrupt handler.

Periodic Events

Synchronous events, for example, a periodic scan of thermal sensors. In this case, the relevant service for periodic events is **MS_1**, which is set by the timer interrupt handler, and is used as a base system ticker for all periodic events in n * 1 millisecond.

2.2.3 Services

The following table shows the 16 service bits. They are defined in **CORE\INC\PURDAT.H**:

Bit #	Define	Service	Description
0	#define **UNLOCK**	Device Unlock	Is used to unlock PS2 devices (Keyboard, Mouse) and enable them to transmit data.
1	#define **PCI**	Main Host Interface	Is set by the Keyboard and Mouse (main) host interface interrupt handler, on an input buffer full event.
2	#define **AUX_PORT_SND**	Auxiliary Port Send to Host	Is set by the PS2 interrupt handler, when PS2 data arrives and must be sent to the host.
3	#define **SEND**	Scanned Keyboard Send to Host	Is set when scanned keyboard data arrives and must be sent to the host.
4	#define **KEY**	Scanner	Is set by the Keyboard Scan interrupt handler, on any keystroke event (in scanned keyboard).
5	#define **MS_1**	1 Millisecond Elapsed	Is set by the Timer interrupt handler, every 1 millisecond.
6	#define **PCI3**	Third Host Interface	Is set by the Power Management 2 (third) host interface interrupt handler, on an input buffer full event.
7	#define **CORE_7**	Core Service 7	Can be used for adding a Core service.
8	#define **PCI2**	Second Host Interface	Is set by the Power Management 1 (second) Host Interface interrupt handler, on input buffer full event.
9	#define **EXT_IRQ_SVC**	External Interrupt Function	Is set by an external IRQ that triggers a system control function.
10	#define **UPD_FLASH**	Flash Update	Is set by the Shared Memory interrupt handler, on a Flash Update event.
11	#define **CORE_11**	Core Service 11	Can be used for adding a Core service.
12	#define **CHK_EXT_M**	Check External Mouse	Is used to check if an external PS2 Mouse is connected.
13	#define **OEM_SRVC_0**	OEM Service 0	Can be used for adding an OEM (platform-specific) service.
14	#define **OEM_SRVC_1**	OEM Service 1	Can be used for adding an OEM (platform-specific) service.
15	#define **OEM_SRVC_2**	OEM Service 2	Can be used for adding an OEM (platform-specific) service.

Chapter 3
ADDING PLATFORM-SPECIFIC CODE TO EC FW

3.1 GENERAL CONSIDERATIONS

When adding new platform functionality, consider if this functionality is interrupt driven or periodic driven; see "Interrupt Events" and "Periodic Events", in Section 2.2.2 on page 5.

If it is interrupt driven, define a new OEM service bit. This service bit is set by the relevant new interrupt handler. When a new service bit is set, the main flow passes control to the new service handling routine.

If it is periodic driven, add the code handling this functionality to one of the n * 1millisecond hook functions.

3.1.1 Adding Interrupt-Driven Platform Functionality

Adding a New OEM Service

Choose an available OEM service bit: **OEM_SRVC_0/1/2,** defined in **CORE\INC\PURDAT.H**.

The implementation of the new service handling function should be done in: **Hookc_Service_OEM_0/1/2**, defined in **OEM\PROJECT\OEMMAIN.C.**

Adding a New Interrupt Handler

In **OEM\PROJECT\ICU.C**:

The pointer to the relevant interrupt handler should be inserted in the right place in the **dispatch_table[].** The interrupt handler should perform the needed hardware control activities, and set the new service bit. Setting the new service bit ensures that the main firmware flow passes control to the new service handling function.

The ISR (Interrupt Service Routine) should be declared as:

#pragma interrupt(My_ISR)

#pragma interrupt preprocessor is used to inform to compiler the specific subroutine is dedicate for ISR. Compiler is using **RETX** in the end of the ISR as return from Exception.

3.1.2 Adding Periodic-Driven Platform Functionality

In **OEM\PROJECT\OEMMAIN.C**:

As explained in "Periodic Events", in Section 2.2.2 on page 5, a **service_1mS()** is called every 1 millisecond. This Core service calls the OEM 1 millisecond hook function, **Hookc_Service_1mS()** (in **OEM\PROJECT\OEMMAIN.C**). This hook function calls more periodic OEM functions: **Service_OEM_10mS()**, **Service_OEM_100mS()**, **Service_OEM_500mS()**, and **Service_OEM_1000mS()**.

Add a call to the new functionality handling routine to one of the above OEM periodic functions, according to the period the new functionality requires.

3.2 THE BUILD PROCESS

In the root directory of the EC firmware source code, there are two build batch files, which are used to build the firmware code:

- **BUILD.BAT** (build all)
- **REBUILD.BAT** (build clean, *build* all)

These batch files call **OEM\PROJECT\BUILD.BAT**, which invokes the **NMAKER** build utility in the firmware makefile, **OEMBLD.MAK** (also located in **OEM\PROJECT\OEMPROJ**).

This makefile first defines a list of Build Variables, which control the firmware build process. Later on, the makefile defines a list of Build Targets, which instruct how to build the firmware.

3.2.1 Build Variables

The first part of the **OEMBLD.MAK** makefile defines many variables that control the build process.

During the build process, some of these variables are imported from the makefile to a header file, **swtchs.h** (in **OEM\PROJECT\INC**), and therefore can be referenced from C code.

3.2.2 Build Targets

Build Targets determine how the firmware is built:

Build Target Name	Description
all	Builds the firmware by invoking the following main targets, and then creates the firmware ROM binary file, **WPC.ROM** (in **OEM\PROJECT\OBJ**): **Core:** Builds all files in the Core directory according to the **CORE\LEVEL.MAK** file, which lists all the files that need to be built in the Core directory, their building rules and dependencies. **Chip:** Builds all files in the Chip directory according to the **CHIP\LEVEL.MAK** file, which lists all the files that need to be build in the Chip directory, their building rules and dependencies. **oemxxx:** Builds all files in the specified **OEM\PROJECT\OEMxxx** directory according to **OEM\PROJECT\OEMxxx\LEVEL.MAK** file, which lists all the files that need to be build in the **OEM\PROJECT\OEMxxx** directory, their building rules and dependencies. For a list of OEMxxx directories, see Section 2.1.3 on page 3.
clean	Cleans all build generated files in **OEM\PROJECT\OBJ**.

3.2.3 Adding a New C File to the Compilation when Adding New OEM Code

- Add the new C file to the **OEM\PROJECT\OEMxxx** directory.
- Add the new C file building rules and dependencies to the **OEM\PROJECT\OEMxxx\LEVEL.MAK** file.
- Add the **following** line to makefile **OEMBLD.MAK** (in the **linker.cmd** *generation* section, search for "linker.cmd:"):

  echo obj\<NEWFILE>.O>>obj\linker.cmd

This line adds the new object file to the **linker.cmd** file, which lists all the object files that should be linked to generate **EC_MAIN.BIN**.

- Add the following line to makefile **OEMBLD.MAK** (in the **$(EC_NAME).bin** *generation* section, search for "$(EC_NAME).bin:"**)**:

**obj\<NEWFILE>.O **

This adds the new object file to the **EC_MAIN.BIN** object dependencies list.

3.3 POWER SEQUENCE

The following flow is suggested (in **OEM\PROJECT\PWRSEQ.C**) to implement the Platform Power Sequence scheme

Each step in the Power Sequence should be implemented as a dedicated function. Each Power Sequence step (function) performs the needed action (typically, toggling relevant GPIOs) and then determines the next step (function) to be called and the millisecond delay until that next step.

Sample Power Sequence Step (Function)

```
void Seq_Step1(void)
{
    // toggle relevant GPIOs
    . . .

    // set next Power Sequence step
    pwrseq_handle = Seq_Step2;

    // set the delay (in milliseconds) until the next step
    PS_Ticker = 5; // e.g., 5 millisecond delay until Seq_Step2
}
```

Periodic Handling

Every 1 millisecond, **HandlePwrSeq()** is called from **Hookc_Service_1mS()**. This routine calls the next Power Sequence step routine if the required delay has expired.

Every 1 millisecond, **HandlePwrSeq()** decrements **PS_Ticker** by 1. When **PS_Ticker** is zero, **HandlePwrSeq()** calls **pwrseq_handle()** to perform the next Power Sequence step.

Initiating Power Sequence Step Function

The following function initiates Power Sequence steps:

void PwrSeq_Init(FUNCT_PTR_V_V power_sequence_handle, BYTE ticker);

where:

power_sequence_handle is the function pointer argument assigned to the global Power Sequence next step function pointer, **pwrseq_handle**. It specifies the first step routine of the Power Sequence steps.

ticker is the argument assigned to the global Power Sequence Ticker, **PS_Ticker**. It specifies the delay (in milliseconds) until the first step (**power_sequence_handle**) of the next Power Sequence.

On first Power Button stroke detection, a Boot Power Sequence should be initiated by a call to **PwrSeq_Init()** with the first Boot step function as the first argument, and the millisecond delay until that first step as the second argument. For example:

PwrSeq_Init(Seq_Step1, 1);

Power State Monitoring

The last step function of a Boot Power Sequence specifies a Power State monitoring function as the next Power Sequence step function. This Power State monitoring function should be called every 1 millisecond and should poll the EC's relevant power input pins to detect a change in the system power state. If a change in the system power state is detected (e.g., from S0 to S3), the appropriate Power Sequence should be initiated by calling **PwrSeq_Init()** with the first relevant step function as the first argument, and the millisecond delay until that first step as the second argument.

3.4 ADDING A NEW PLATFORM/OEM HOST COMMANDS

Most of opcodes are reserved for adding proprietary Platform/OEM host commands except standard commands for 8042 and ACPI EC interface. The following hook functions, defined in **OEM\PROJECT\HOOK.C**, are used for adding new Platform/OEM host commands. OEM command hook uses **hif** as parameter to identify the interface command/data from.

WORD Hookc_Pmx_Cmd0(HIF_VAR* hvp, BYTE command_num, BYTE hif)

This function is a command parser and handler for the 0x0x OEM host command opcodes. It implements the necessary handling for the added new platform host command, where:

hvp is pointer of host interface variables

command_num is the host command opcode.

hif is host interface command sent from

Return value:

– The data byte (in lower byte of the returned word) sent to the host

– 0xFFFF, if no data is sent to the host

WORD Hookc_Pmx_Dat0(HIF_VAR* hvp, BYTE command_num, BYTE data, BYTE hif)

This function is a command parser and handler for the 0x0x OEM host command opcodes that have accompanying data. It should implement the necessary handling for an added new Platform/OEM host command that has accompanying data, where:

hvp is pointer of host interface variables

command_num is the host command opcode.

data is the data byte of the host command.

hif is host interface command sent from

Return value:

– The data byte (in lower byte of the returned word) sent to the host

– 0xFFFF, if no data is sent to the host

Similar hook functions are defined for handling 0x4x and 0x5x new OEM host commands:

WORD Hookc_Pc_Cmd1/F(HIF_VAR* hvp, BYTE command_num, BYTE hif);
WORD Hookc_Pc_Dat1/F(HIF_VAR* hvp, BYTE command_num, BYTE data, BYTE hif);

Example for Adding a Host Command Without Data

If no data is needed for the new command, use **Hookc_Pc_CmdX()** only. There is no need to use **Hookc_Pc**_DatX**()**.

```
WORD Hookc_Pc_Cmd1(HIF_VAR* hvp, BYTE command_num, BYTE hif){
    if (command_num == MY_COMMAND)
    {
                    // handle the command
            return MY_RESPONSE;
```

```
        }
        else
        {
                return 0xFFFF;
        }
    }
```

Example for Adding a Host Command Without Data

If data is needed for the new command, use both **Hookc_Pc_CmdX()** and **Hookc_Pc_DatX()**.

```
WORD Hookc_Pc_Cmd1(HIF_VAR* hvp, BYTE command_num, BYTE hif)
{
        if (command_num == MY_COMMAND)
        {
                // save the command and signal the core to wait for data from host
                hvp->Cmd_Byte = command_num;
        }
        else
        {
                return 0xFFFF;
        }
}
WORD Hookc_Pc_Dat1(HIF_VAR* hvp, BYTE command, BYTE data, BYTE hif)
{
        if (command_num == MY_COMMAND)
        {
                switch (data)
                {
                        case DATA1:
                                // handle the command
                                return RES1;
                                break;
                                case DATA2:
                                // handle the command
                                return RES2;
                                break;
                }
        }
        else
        {
                        return 0xFFFF;
        }
}
```

3.5 SENDING A MULTI-BYTE ARRAY AS A RESPONSE TO A HOST COMMAND

The following code can be used while handling a new or modified host command in case multi-data bytes must be sent to the host (as a response to the host command):

```
HIF_Response[hif].byte = respARRAY;
Hif_Var[].Tmp_Pntr    = Array;              // Pointer to the byte Array
Hif_Var[].Tmp_Load    = bytes_num; // Number of bytes in Array
```

where:

HIF_Response[].byte is a global variable defined in **CORE\PURDAT.C**:

BITS_8   HIF_Response[];

respARRAY is defined in **CORE\PURDAT.H**:

#define respARRAY   0x82 /* Sends bytes in an array. */

Hif_Var[].Tmp_Pntr is a global variable defined in **CORE\PURDAT.C**:

***Tmp_Pntr** is an element of Hif_Var[] structure.

Array is a pointer to the byte array

Tmp_Load is a global variable defined in **CORE\PURDAT.C**:

Tmp_Load is an element of Hif_Var[] structure

bytes_num is the number of bytes in the byte array

Example

Assuming the response to host command 0x50 is two bytes, which represent the EC FW version:

```
const BYTE EcFw_Version[] = {0x01,0x15};

WORD Hookc_Pc_Cmd5(HIF_VAR* hvp, BYTE command_num, BYTE hif){
   if (command_num == 0x50)
   {
                    HIF_Response[hif].byte = respARRAY;

                    hvp->Tmp_Pntr = EcFw_Version;

                    hvp->Tmp_Load = 2;

   }
   return 0xFFFF;
}
```

3.6 ACPI/EC SPACE

3.6.1 Read/Write EC Space Hook Functions

Part of this ACPI/EC space (ACPI offset 0x60-0x87) is reserved for the ACPI commands that manipulate the EC SMBus. For complete details on the Configuration Table, refer to the "Insyde EC FW Reference Manual", Sections 3 and 9.

The ACPI/EC space can be used for storing platform parameters, such as platform Thermal and Fan parameters.

These ACPI/EC space platform parameters are read and written by host commands 0x80 and 0x81.

Sometimes, it is more convenient to hold such platform parameters in data structures outside the ACPI/EC space.

The following hook functions, defined in **OEM\PROJECT\HOOK.C**, are used to modify the standard read/write ACPI/EC space host commands. These hook functions are used to read and write platform parameters outside the ACPI/EC space as if they were stored in the ACPI/EC space:

WORD Hookc_Read_EC(BYTE index);

This function is called before data is read from the EC space (host command 0x80), where:

index is the offset to the EC space

Return value:

- A value with all bits set to 1 to allow the Core to handle reading the EC space.
- Otherwise, returns the data byte (in lower part of the word) sent to the host (skip Core handling).

Read EC Space Hook Example

```
WORD Hookc_Read_EC(BYTE index)
{
	WORD rval = ~0;

	if (index == 0x5A)
	{
		/* Brightness Level in ACPI/EC Space offset 0x5A */
			rval = (WORD) BrightCtrlByte.field.BrightLevel;
	}
	return (rval);
}
```

FLAG Hookc_Write_EC(BYTE index, BYTE data);

This function is called before data is written to the EC space (host command 0x81), where:

index is the offset to the EC space

data is the byte to write into the EC_Space array

Return value:

- 0, to allow the Core to handle writing to the EC space.
- Otherwise, returns 1 (skip Core handling).

Write EC Space Hook Example

```
FLAG Hookc_Write_EC(BYTE index, BYTE data)
{
    FLAG rval = 0;


    if (index == 0x5A)
    {
        /* Brightness Level in ACPI/EC Space offset 0x5A */
        BrightCtrlByte.field.BrightLevel = data;
        rval = 1;
    }
    return(rval);
}
```

3.7 SMBUS INTERFACE

The following SMBus API functions are defined in **OEM\PROJECT\SMB_TRAN.C**

3.7.1 SMBus Control Functions

FLAG Transfer_Finished(BYTE Channel);

Is called to check if the SMBus transaction on the specified channel, initiated by one of the SMB_xxx functions listed below, is finished. If it returns TRUE, the SMBus transaction is finished; otherwise, it is not yet finished.

BYTE *Get_Result(BYTE Channel);

Is called after the SMBus transaction on the specified channel is finished, to get the returned data in the case of a read SMBus transaction. This function returns a pointer to the data received from the SMBus transaction. When reading a block of data, the first byte holds the number of bytes that are read, and the rest of the bytes contain the data.

void OEM_SMB_callback(BITS_8 i2c_state, BYTE Channel)

Is called after the SMBus transaction on the specified channel is finished from I2C low level driver. Flags are set/clear indicate I2C transaction status. OEM code need to call **Transfer_Finished(Channel)** to check the transaction done or not and call **Get_Result(Channel)** to check when transaction status indicated done.

3.7.2 SMBus Protocol Functions

void SMB_rdBYTE(BYTE Channel, BYTE SMB_SLAVE, BYTE SMB_COMMAND, FLAG Prtcl_Type);
void SMB_wrBYTE(BYTE Channel, BYTE SMB_SLAVE, BYTE SMB_COMMAND, BYTE wData, FLAG Prtcl_Type);

These functions initiate a read/write byte SMBus protocol transaction, where:

channel is the SMBus module (0,1,2,3)

SMB_SLAVE is the slave device address

SMB_COMMAND is the slave device read /write byte command code

wData is the data byte to be written to the slave device

Prtcl_Type is the SMBus transaction type for interrupt/polling mode

1: Interrupt mode

0: Polling mode

void SMB_rdWORD(BYTE Channel, BYTE SMB_SLAVE, BYTE SMB_COMMAND, FLAG Prtcl_Type);
void SMB_wrWORD(BYTE Channel, BYTE SMB_SLAVE, BYTE SMB_COMMAND, WORD wData, FLAG Prtcl_Type);

These functions initiate a read/write word SMBus protocol transaction, where:

channel is the SMBus module (0,1,2,3)

SMB_SLAVE is the slave device address

SMB_COMMAND is the slave device read /write word command code

wData is the data word to be written to the slave device

Prtcl_Type is the SMBus transaction type for interrupt/polling mode

1: Interrupt mode

0: Polling mode

void SMB_RECEIVE(BYTE Channel, BYTE SMB_SLAVE, FLAG Prtcl_Type);
void SMB_SEND (BYTE Channel, BYTE SMB_SLAVE, BYTE wData, FLAG Prtcl_Type);

These functions initiate a receive/send byte SMBus protocol transaction, where:

channel is the SMBus module (0,1,2,3)

SMB_SLAVE is the slave device address

wData is the data byte to be written to the slave device

Prtcl_Type is the SMBus transaction type for interrupt/polling mode

1: Interrupt mode

0: Polling mode

void SMB_rdBLOCK(BYTE Channel,BYTE SMB_SLAVE,BYTE SMB_COMMAND, FLAG Prtcl_Type);
void SMB_wrBLOCK(BYTE Channel,BYTE SMB_SLAVE,BYTE SMB_COMMAND,BYTE Send_CNT, BYTE * trans_pntr, FLAG Prtcl_Type);

These functions initiate a read/write Block SMBus protocol transaction, where:

channel is the SMBus module (0,1,2,3)

SMB_SLAVE is the slave device address

SMB_COMMAND is the slave device read /write Block command code

Send_CNT is the number of data bytes to be written to the slave device.

trans_pntr is transfer buffer pointer. Prior to the call to **SMB_wrBLOCK**, data bytes should be copied to the global **transfer buffer** array from **buffer[0]** to **buffer[Send_CNT]**

Prtcl_Type is the SMBus transaction type for interrupt/polling mode

1: Interrupt mode

0: Polling mode

3.8 GPIO INTERFACE

The following GPIO macros, defined in **OEM\PROJECT\INC\PINDEF.H**, are used to access GPIO pins.

PORT_PIN(port, pin)

This macro is used to create an 8-bit port-pin combination (e.g., 32 means pin 2 in port 3), where:

port specifies the I/O port number.

pin specifies the pin number of the I/O port (0-7).

Return value: The port-pin combination byte

This macro should be used to define input and output pins for a specific platform. For example, assuming battery LED is connected to GPIO04:

#define EC_BATTERY_LED1 PORT_PIN(0, 4)

READ_GPIO(port, pin)

This macro is used to read IO input pin, where:

port specifies the I/O port number.

pin specifies the pin number of the I/O port (0-7).

Return value:

- 0: I/O pin is low
- 1: I/O pin is high

WRITE_GPIO(port, pin, data)

This macro is used to set IO pin to a specified data, where:

port specifies the I/O port number.

pin specifies the pin number of the I/O port (0-7).

Return value:

- 0: Set I/O pin low
- 1: Set I/O pin high

SET_PIN(port_pin, data)

This macro is used to set the I/O pin, specified by the port-pin combination byte, to a specified data, where:

port_pin is the I/O port-pin combination byte.

data

- 0: Set I/O pin low
- 1: Set I/O pin high

READ_INP_PIN(port_pin)

This macro is used to read the I/O input pin defined by the port-pin combination byte, where:

port_pin is the I/O port-pin combination byte.

Return value:

- 0: I/O pin is low
- 1: I/O pin is high

READ_OUTP_PIN(port_pin)

This macro is used to read the I/O output pin, defined by the port-pin combination byte, where:

port_pin is the I/O port-pin combination byte.

Return value:

– 0: I/O pin is low

– 1: I/O pin is high

3.9 SENDING A SCAN CODE TO THE HOST

On detection of some system events, the FW must send a scan code to the host (e.g., on detection of a Volume Up key pressed). Kernel code provides an API to allow OEM code call to send scan code to Host. Of course OEM code can send scan code to host by calling to Buffer_Key or Buffer_String directly.

The following is the API for sending scan code to the host:

void Send_OEM_Key(BYTE key, BYTE Event);

where:

Send_OEM_Key() is defined in **OEM\PROJECT\EVENTS.C**

key is the Insyde Key Number used as the access key to the Scan Code tables.

For the Insyde Key list, see the "Insyde EC FW Reference Manual", Section 12.4.

The following Key list completes the Insyde Key list:

Insyde Key Number	Description
0x0F	Next Track event
0x10	Previous Track event
0x11	Stop event
0x12	Play/Pause event
0x17	Mute event
0x18	Volume Up event
0x19	Volume Down event
0x37	Mail event
0x38	Search event
0x39	Web/Home event
0x56	Back event
0x57	Forward event
0x5E	Stop event
0x5F	Refresh event
0x60	Favorites event
0x61	Calculator event

0x62	My Computer event
0x63	Media event

FLAG Buffer_Key(BYTE row_column);
SMALL Buffer_String(const BYTE *pntr)

where:

Buffer_Key() is defined in **CORE\PURSCN.C**

row_column is the scan code to be send to Host

pntr is pointer of scan code buffer array

FLAG is indicator of overflow or not

Below two instructions need to be called to make sure the scan code is able to send to Host.

```
if (Check_Scan_Transmission())     /* Is data available? */
{
    Start_Scan_Transmission();     /* Yes, start new transmission. */
}
```

Event is one of the following, defined in **CORE\INC\PURXLT.H**:

#define MAKE_EVENT 0
#define BREAK_EVENT 1

3.10 GENERATING AN SMI/SCI INTERRUPT

On detection of some system events, the FW must generate an SMI/SCI interrupt to the host (e.g,. on detection of an LID close event).

The following is the API for generating an SMI/SCI Interrupt:

void System_Control_Function(WORD Event, SMALL make_event);

where:

System_Control_Function() is defined in **CORE\PURFUNCT.C**

Event is a 16-bit value, as follows:

The upper 8 bits - 0001 0tuv (can use **#define Y_SMI_SCI 0x10** from **CORE\INC\PURFUNCT.H**):

t: 1 - Generates an SMI in Legacy mode
u: 1 - Generates an SMI in ACPI mode
v: 1 - Generates an SCI in ACPI mode

The lower 8 bits are the cause code for the specific event.

For example, assuming the cause code for the LID event is 0x35:

– Event parameter with the value 0x1535 means an LID event generates an SCI in ACPI mode and generates SMI in Legacy mode
– **Event** parameter with the value 0x1135 means an LID event generate an SCI in ACPI mode

make_event

Use the following definition of **MAKE_EVENT**, which is defined in **CORE\INC\PURXLT.H**:

#define MAKE_EVENT 0

3.11 PWM INTERFACE

To enable PWM functionality in the firmware, set the PWM_SUPPORTED flag (in **OEM\PROJECT\OEMBLD.MAK**) to On.

The following PWM APIs are defined in **CHIP\PWM.C**:

void PWM_config(PWM_Module_t pwm_module,

uint16 prescaler_divider,

uint16 cycle_time_factor,

uint16 duty_cycle_factor)

where:

pwm_module is the PWM module to be configured, taken from the following enum:

```
typedef enum
{
    PWM_MODULE_A = 0,
    PWM_MODULE_B = 1,
    PWM_MODULE_C = 2,
    PWM_MODULE_D = 3,
    PWM_MODULE_E = 4,
    PWM_MODULE_F = 5,
    PWM_MODULE_G = 6,
    PWM_MODULE_H = 7,
    PWM_MODULE_LAST = 8
} PWM_Module_t;
```

prescaler_divider is the prescaler divider value (0000-FFFF).

cycle_time_factor is the cycle time factor value (0000-FFFF).

duty_cycle_factor is the duty cycle factor value (0000-FFFF).

This function configures the PWM module specified by the **pwm_module** parameter. It assigns the **prescaler**_divider parameter to PRSC register, the **cycle_time_factor** parameter to CTR register, and the **duty_cycle_factor** parameter to DCR register.

The PWM output signal cycle time is defined by:

(PRSC+1) x (CTR+1) x Tclk

where **Tclk** is the PWM input clock cycle time (Core clock or 32K clock).

The PWM output signal duty cycle (in %) is defined by:

(DCR+1) / (CTR+1) x 100

Note that to activate this PWM module, call the **PWM_ENABLE(pwm_module)** macro after calling the **PWM_config()** function:

PWM_ENABLE(pwm_module)

where:

pwm_module is the PWM module to be enabled.

This macro activates the PWM module specified by the **pwm_module** parameter.

PWM_DISABLE(pwm_module)

where:

pwm_module is the PWM module to be disabled.

This macro disables the PWM module specified by the **pwm_module** parameter.

PWM_SET_DUTYCYCLE(pwm_module, duty_cycle)

where:

pwm_module is the PWM module.

duty_cycle is the PWM duty cycle value.

This macro sets the PWM duty cycle specified by the **duty_cycle** parameter.

void PWM_config_heart_beat_mode(

PWM_Module_t pwm_module,

PWM_Heart_Beat_Mode_t heart_beat_max_dc,

PWM_Heart_Beat_Rate_Div_t rate_div)

where:

pwm_module is the PWM module to be configured

heart_beat_max_dc is the heartbeat max duty cycle value, taken from the following enum:

```
typedef enum
{
    NORMAL = 0,
    MAX_25_PERCENT = 1,
    MAX_50_PERCENT = 2,
    MAX_100_PERCENT = 3
} PWM_Heart_Beat_Mode_t;
```

rate_div is the heartbeat rate divider , taken from the following enum:

```
typedef enum
{
    DIVIDE_BY_1 = 0,
    DIVIDE_BY_4 = 1,
    DIVIDE_BY_8 = 2,
    DIVIDE_BY_16 = 3
} PWM_Heart_Beat_Rate_Div_t;
```

This function configures Heartbeat mode for the PWM module specified by the **pwm_module** parameter. Heartbeat mode uses the PWM generator to generate a modulation signal with a constantly changing duty cycle (from 0 to max duty cycle).

The duration of one heartbeat cycle is given by:

(PRSC+1) x (CTR+1) x rate_div x 64 x Tclk

For optimal Heartbeat mode operation, CTR register should be set (by the above **PWM_config()** function) to 0x7F.

void PWM_config_force_active(

PWM_Module_t pwm_module,

PWM_Force_Src_t source,

bool source_active_high)

where:

pwm_module is the PWM module to be configured,

source is the source of the force active action, taken from the following enum:

```
typedef enum
{
    DISABLED = 0,
    GPIO42 = 1,
    GPIO43 = 2
} PWM_Force_Src_t;
```

source_active_high TRUE - source active high

FALSE - source active low

This function is used to enable/disable force activation for the PWM module specified by the **pwm_module** parameter. It determines the source GPIO for the force activation (specified by the **source** parameter), and whether the source is active high or active low (specified by the **source_active_high** parameter). Force active functionality is supported by NPCE78nx and later devices.

void PWM_clock_select(

PWM_Module_t pwm_module,

PWM_Clock_Select_t clock_select)

where:

pwm_module is the PWM module to be configured.

clock_select selects the PWM input clock, taken from the following enum:

```
typedef enum
{
    PWM_CORE_CLOCK = 0,
    PWM_32K_CLOCK   = 1
} PWM_Clock_Select_t;
```

This function selects the input clock (Core clock or 32K clock) for the PWM module specified by the **pwm_module** parameter.

void PWM_custom_pattern(PWM_Module_t pwm_module, uint32 pattern)

where:

pwm_module is the PWM module to be configured.

pattern is the custom pattern.

This function configures the PWM module specified by the **pwm_module** parameter to operate according to a 32-bit custom pattern (specified by the **pattern** parameter). The pattern is assigned to a 32-bit shift register. Every 128 PWM cycles (i.e., one “time slot”), the pattern register is shifted by one bit. If the read bit of the pattern is 1, the PWM generates a normal PWM cycle for the current time slot. If the read bit is 0, the PWM is inactive for the current time slot. The pattern value default is 0xFFFFFFFF. Custom pattern functionality is supported by NPCE78nx and later devices.

3.12 PORT80 INTERFACE

To enable PORT80 functionality in the firmware, set the PORT80_SUPPORTED flag (in **OEM\PROJECT\OEMBLD.MAK**) to On.

The following PORT80 APIs are defined in **CHIP\PORT80.C**:

void DP80_init()

This function is called on EC FW initialization **(Hookc_Cold_Reset_End()** in **OEM\PROJECT\OEMINI.C**) to initialize port 80 hardware.

#pragma interrupt(DP80_int_handler)

void DP80_int_handler(void)

This interrupt routine is assigned to INT16 in the dispatch table and invoked on a host write to port 80. The routine sends the data from port 80 to OEM hook **(OEM_Get_Port80_Val()** in **OEM\PROJECT\HOOK.C**).

3.13 PECI INTERFACE

To enable PECI functionality in the firmware, set the **PECI_SUPPORTED** flag (in **OEM\PROJECT\OEMBLD.MAK**) to On.

The PECI APIs are designed to be non-blocking. Once a PECI transaction is initiated, control returns immediately to the EC FW. When the PECI transaction is finished and data is available, an application PECI callback function is called from the PECI interrupt handler, for PECI data processing.

The following PECI APIs are defined in **CHIP\PECI.C**:

void PECI_Init (PECI_CALLBACK_T callback, DWORD peci_freq)

where:

callback is the application callback function to be called from the PECI interrupt handler.

peci_freq is the PECI bit rate (in Hz).

This function initializes the PECI module.

void PECI_SetAddress (BYTE client_address)

where:

client_address is the PECI Client Address.

This function sets the value of the PECI Client Address frame of the next PECI transaction.

void PECI_SetDomain (PECI_DOMAIN_T domain)

where:

domain is the PECI domain number.

This function sets the value of the PECI domain to access in the next PECI transaction.

PECI_CC_T PECI_GetCompletionCode (void)

This function retrieves the Completion Code of the last PECI transaction.

void PECI_Ping (void)

This function sends a **Ping** command. The command is used to enumerate devices or determine if a device has been removed or powered-off.

void PECI_GetDIB (void)

This function sends a **GetDIB** command. The command provides information regarding the client revision number and the number of supported domains.

void PECI_GetTemp (void)

This function sends a **GetTemp** command. The command is used to retrieve the maximum temperature from a target PECI address.

void PECI_RdPkgConfig (PECI_DATA_SIZE_T read_data_size, BYTE host_id, bool retry, BYTE index, WORD parameter)

where:

read_data_size is the desired data return size (BYTE/WORD/DWORD).

host_id is the Host ID.

retry - TRUE: retry on failure; otherwise, FALSE.

index is the requested service.

parameter is the service parameter value.

This function sends a **RdPkgConfig** command. The command provides read access to the Package Configuration Space (PCS) within the processor, including various power and thermal management functions.

void PECI_WrPkgConfig (PECI_DATA_SIZE_T write_data_size, BYTE host_id, bool retry, BYTE index, WORD parameter, DWORD data)

where:

write_data_size is the desired write granularity (BYTE/WORD/DWORD).

host_id is the Host ID.

retry - TRUE: retry on failure; otherwise, FALSE.

index is the requested service.

parameter is the service parameter value.

data is the data to write to the processor Package Configuration Space.

This function sends a **WrPkgConfig** command. The command provides write access to the Package Configuration Space (PCS) within the processor, including various power and thermal management functions.

void PECI_RdIAMSR (PECI_DATA_SIZE_T read_data_size, BYTE host_id, bool retry, BYTE processor_id, WORD msr_address)

where:

read_data_size is the desired data return size (BYTE/WORD/DWORD/QWORD).

host_id is the Host ID.

retry - TRUE: retry on failure; otherwise, FALSE.

processor_id is the logical processor ID within the CPU.

msr_address is the Model Specific Register address.

This function sends a **RdIAMSR** command. The command provides read access to the Model Specific Registers (MSRs) defined in the processor's Intel Architecture (IA).

void PECI_RdPCIConfig (PECI_DATA_SIZE_T read_data_size, BYTE host_id, bool retry, DWORD pci_config_address)

where:

read_data_size is the desired data return size (BYTE/WORD/DWORD).

host_id is the Host ID.

retry - TRUE: retry on failure; otherwise, FALSE.

pci_config_address is the PCI configuration address (28-bit).

This function sends a **RdPCIConfig** command. The command provides read access to the PCI configuration space maintained in downstream devices external to the processor.

void PECI_RdPCIConfigLocal (PECI_DATA_SIZE_T read_data_size, BYTE host_id, bool retry, DWORD pci_config_address)

where:

read_data_size is the desired data return size (BYTE/WORD/DWORD).

host_id is the Host ID.

retry - TRUE: retry on failure; otherwise, FALSE.

pci_config_address is the PCI configuration address for local accesses (24-bit).

This function sends a **RdPCIConfigLocal** command. The command provides read access to the PCI configuration space that resides within the processor.

void PECI_WrPCIConfigLocal (PECI_DATA_SIZE_T write_data_size, BYTE host_id, bool retry, DWORD pci_config_address, DWORD data)

where:

write_data_size is the desired write granularity (BYTE/WORD/DWORD).

host_id is the Host ID.

retry - TRUE: retry on failure; otherwise, FALSE.

pci_config_address is the PCI configuration address for local accesses (24-bit).

data is the data to write to the PCI configuration Space within the processor.

This function sends a **WrPCIConfigLocal** command. The command provides write access to the PCI configuration space that resides within the processor.

void PECI_Enable_HwAWFCS(void)

This function enables HW generation of an Assured Write FCS byte.

void PECI_Disable_HwAWFCS(void)

This function disables HW generation of an Assured Write FCS byte.

3.14 RUNNING FROM MRAM MEMORY

To enable support for running the FW from MRAM, set the **RUN_FROM_MRAM** flag (in **OEM\PROJECT\OEMBLD.MAK**) to On.

This flag instructs the FW to do the following:

Use the MRAM header signature in the flash header.

- Calculate the FW checksum on all ROM images (not just the CRISIS part).
- Disable the instruction cache.
- Disable the FSPI signals by setting SPI_TRIS bit in DEVCNT register.

3.15 AUTOMATIC HW KEYBOARD SCAN

Automatic HW Keyboard Scan mode enhances the performance of the keyboard scan operation.

To enable automatic HW keyboard scan, set the **HW_KB_SCN_SUPPORTED** flag (in **OEM\PROJECT\OEMBLD.MAK**) to On.

Changing the default configuration setting of the automatic HW keyboard scan can be done using OEM hook function **OEM_Change_Default_HW_KBS_CFG()** in **OEM\PROJECT\HOOK.C.**

This hook OEM function can be used to change the default value of the following configuration fields:

- **KBS_DLY1**
- **KBS_DLY2**
- **KBS_RTYTO**
- **KBS_CNUM**
- **KBS_CDIV**

For complete details about these fields, see the *NPCE8mnx Architectural Specification.*

3.16 ONE-WIRE INTERFACE (OWI)

To enable One-Wire functionality in the firmware, set the **OWI_SUPPORTED** flag (in **OEM\PROJECT\OEMBLD.MAK**) to On.

The following OWI APIs are defined in **CHIP\OWI.C**:

void OWI_Init(OWI_Callback_t owi_callbak)

This function Should be called on FW initialization. It initializes the OWI status to **OWI_STATUS_READY** and enables an OWI interrupt in the ICU.

owi_callback is a Pointer to a callback function called by the OWI interrupt when the OWI read/write operation is finished (successfully or unsuccessfully). If NULL, no function is called on completion of the operation, and the FW must poll the OWI status using the **OWI_Get_Status()** function.

void OWI_SetClockFrequency(OWI_CORE_CLK_DVSR dvsr)

This function sets the Core clock divisor.

dvsr can be:

OWI_CORE_CLK_DVSR_4,
OWI_CORE_CLK_DVSR_5,
OWI_CORE_CLK_DVSR_6,
OWI_CORE_CLK_DVSR_7,
OWI_CORE_CLK_DVSR_8,
OWI_CORE_CLK_DVSR_10,
OWI_CORE_CLK_DVSR_12,
OWI_CORE_CLK_DVSR_14,
OWI_CORE_CLK_DVSR_16,
OWI_CORE_CLK_DVSR_20, // should be used on core clock 20 MHz
OWI_CORE_CLK_DVSR_24,
OWI_CORE_CLK_DVSR_28,

OWI_CORE_CLK_DVSR_32,
OWI_CORE_CLK_DVSR_40,
OWI_CORE_CLK_DVSR_48,
OWI_CORE_CLK_DVSR_56

OWI_RESET_RESULT OWI_Reset()

This function initiates a One-Wire Reset command through the Command Register.

OWI_RESET_RESULT can be:

OWI_RESET_ERR_BUS_LOW,	// 1-wire bus turned low
	// when the master was idle
OWI_RESET_ERR_BUS_SHORT,	// 1-wire bus was low before
	// the master tried to reset the bus
OWI_RESET_ERR_DEVICE_NOT_DETECTED,	// 1-wire device was not found
OWI_RESET_ERR_DETECT_TIME_NOT_OK,	// 1-wire device detect time
	// was not O.K
OWI_RESET_OK	// 1-wire device detected O.K

OWI_STATUS OWI_Get_Status()

This function returns the OWI status.

OWI_STATUS can be:

OWI_STATUS_READY,
OWI_STATUS_IN_WRITE,
OWI_STATUS_IN_READ,
OWI_STATUS_ERROR

OWI_RETURN_CODE OWI_Start_Read(BYTE *ReadDataArray, int BytesToRead)

This function starts an OWI read transaction of **BytesToRead** bytes from the One-Wire device into **ReadDataArray.**

OWI_RETURN_CODE can be:

OWI_RETURN_OK	if OWI Read started
OWI_RETURN_WRONG_SIZE	if wrong size parameter
OWI_RETURN_BUSY	if OWI is busy with another transaction

After OWI Read started, the OWI status is **OWI_STATUS_IN_READ**.

FW should periodically poll **OWI_Get_Status()** until OWI Read ends.

If OWI Read ended O.K, the OWI status is **OWI_STATUS_READY**.

If NOT OK, the OWI status is **OWI_STATUS_ERROR**.

OWI_RETURN_CODE OWI_Start_Write(BYTE *WriteDataArray,int BytesToWrite)

This function starts an OWI write transaction of **BytesToWrite** bytes from **WriteDataArray** to the One-Wire device.

OWI_RETURN_CODE can be:

OWI_RETURN_OK	if OWI Write started
OWI_RETURN_WRONG_SIZE	if wrong size parameter
OWI_RETURN_BUSY	if OWI is busy with another transaction

After OWI Write started, the OWI status is **OWI_STATUS_IN_WRITE**.

FW should periodically poll **OWI_Get_Status()** until OWI Write ends.

If OWI Write ended O.K, the OWI status is **OWI_STATUS_READY**.

If NOT OK, the OWI status is **OWI_STATUS_ERROR**.

#pragma interrupt(OWI_Handler)

void OWI_Handler()

The function is the OWI module interrupt handler. It is used to manage the OWI Read and Write operations. This OWI interrupt handler should be assigned to the INT1 entry of the FW dispatch table.

Nuvoton provides comprehensive service and support.
For product information and technical assistance, contact the nearest Nuvoton center.

Headquarters
No. 4, Creation Rd. 3,
Science-Based Industrial Park,
Hsinchu, Taiwan, R.O.C
TEL: 886-3-5770066
FAX: 886-3-5665577
http://www.nuvoton.com.tw

Taipei Office
1F, No.192, Jingye 1st Rd.,
Zhongshan District
Taipei 104,
Taiwan, R.O.C.
TEL: 886-2-2658-8066
FAX: 886-2-8751-3579

Nuvoton Technology Corporation America
2727 North First Street,
San Jose, CA 95134, U.S.A.
TEL: 1-408-9436666
FAX: 1-408-5441798

Winbond Electronics Corporation Japan
NO. 2 Ueno-Bldg., 7-18, 3-chome
Shinyokohama Kohoku-ku,
Yokohama, 222-0033
TEL: 81-45-4781881
FAX: 81-45-4781800

Nuvoton Technology (Shanghai) Ltd.
27F, 2299 Yan An W. Rd.
Shanghai, 200336 China
TEL: 86-21-62365999
FAX: 86-21-62365998

Nuvoton Technology (H.K.) Ltd.
Unit 9-15, 22F, Millennium City 2,
378 Kwun Tong Rd.,
Kowloon, Hong Kong
TEL: 852-27513100
FAX: 852-27552064

For Advanced PC Product Line information contact: APC.Support@nuvoton.com

www.nuvoton.com