UM10913

PN7462AU Software user manual

Rev. 1.3 — 20 March 2017
336613

User manual
COMPANY PUBLIC

Document information

Info	Content
Keywords	PN7462AU/PN7360AU FW architecture, ROM FW, flash FW, HAL, examples
Abstract	This document describes the PN7462AU/PN7360AU FW architecture and how to use it.

Revision history

Rev	Date	Description
1.3	20172003	PSP example descriptions revisited
1.2	20161101	Fig 20 and Fig 25 updated
1.1	20160629	Section 9 PN7462AU critical sections in HAL added
1.0	20160330	initial version

Contact information

For more information, please visit: http://www.nxp.com

1. Introduction

This document describes the PN7462AU FW architecture. The PN7462AU FW consists of ROM boot, ROM services, flash boot, and hardware abstraction layers. It also includes NXP NFC contactless protocol library, NXP contact library, product support package examples, and RTOS abstraction.

This document is also valid for PN7360 derivative, in sections where the products differ characteristics of both products are described separately.

2. PN7462AU FW architecture

2.1 PN7462AU FW block diagram

The PN7462AU FW block diagram is shown in Fig 1.

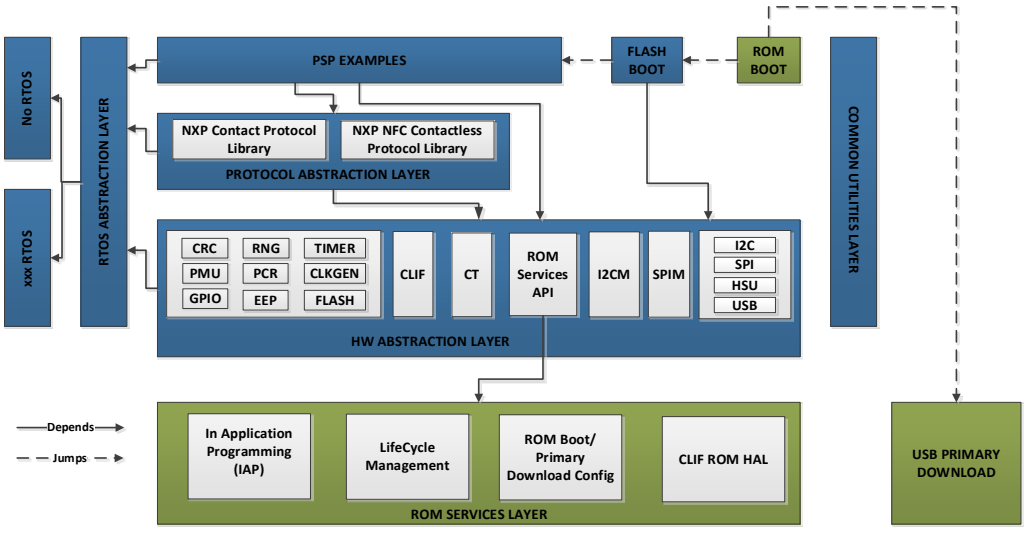

Fig 1. PN7462AU FW block diagram

The FW can be divided into NXP FW and user FW. The NXP FW is placed in ROM memory region and protected flash memory region. The user FW is placed in the user flash memory region. The protected flash memory region is primarily used to place ROM patches.

The NXP FW consists of ROM boot, ROM services and USB primary download. The user FW consists of flash boot, hardware abstraction layers, NXP NFC contactless protocol library, NXP contact library, product support package examples, and RTOS abstraction.

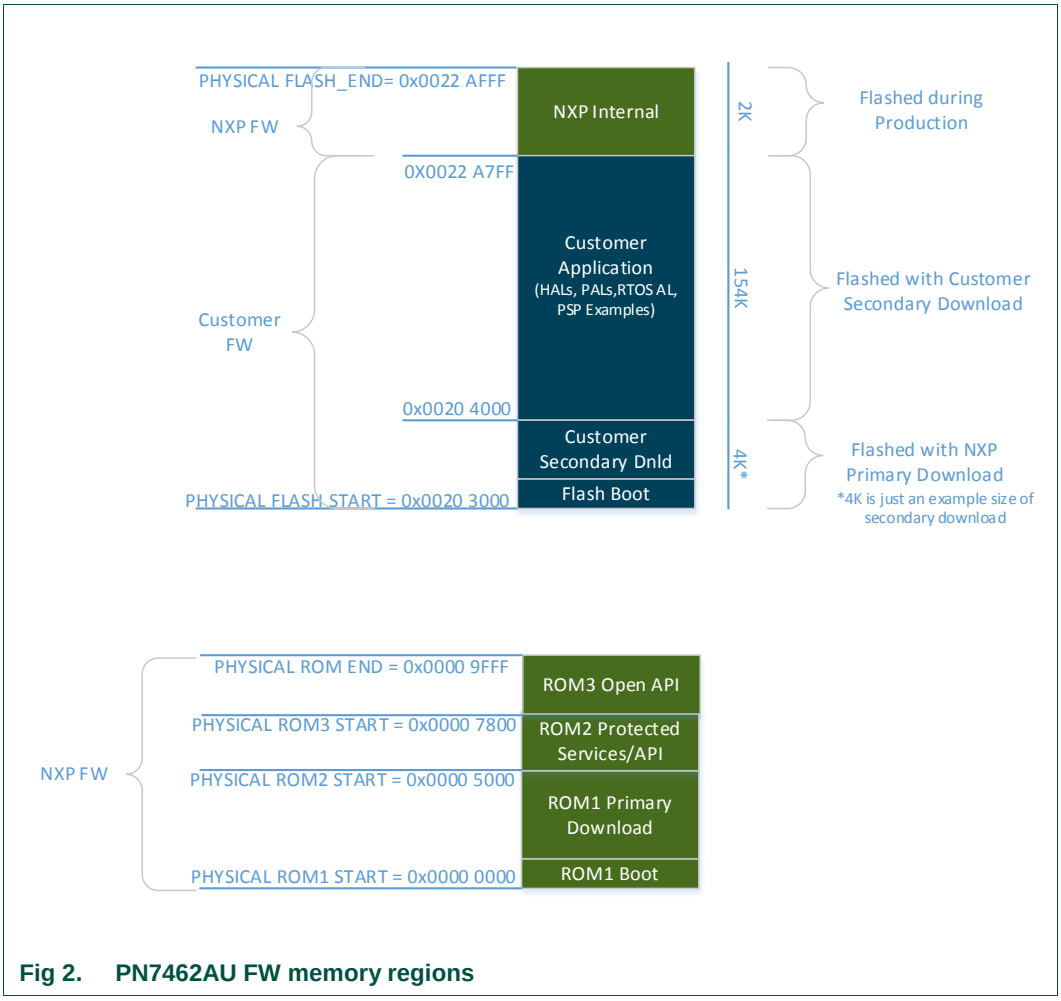

Fig 2. PN7462AU FW memory regions

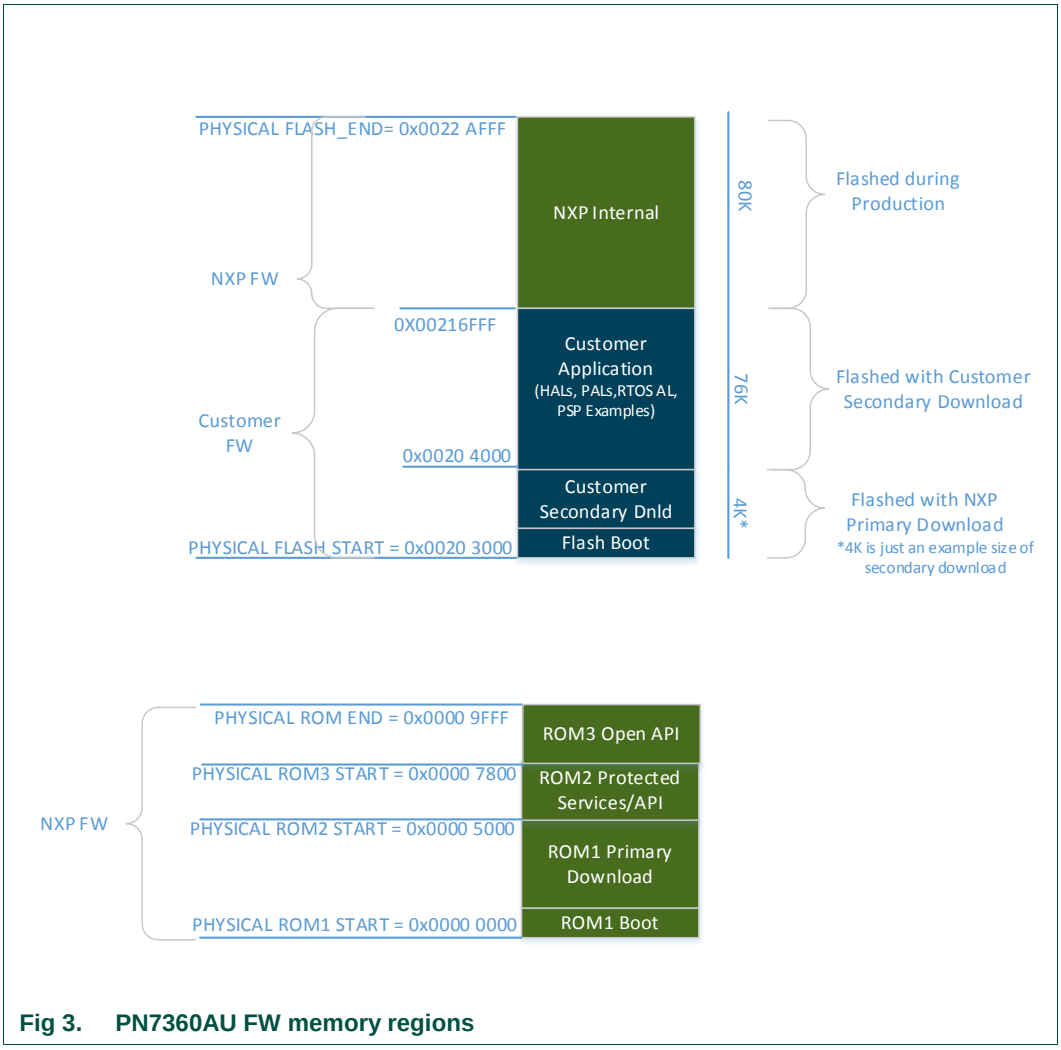

Fig 3. PN7360AU FW memory regions

Memory map of the PN7360AU derivative.

In the above diagrams, customer secondary download is shown only as a reference. Secondary download is not explained in this user manual.

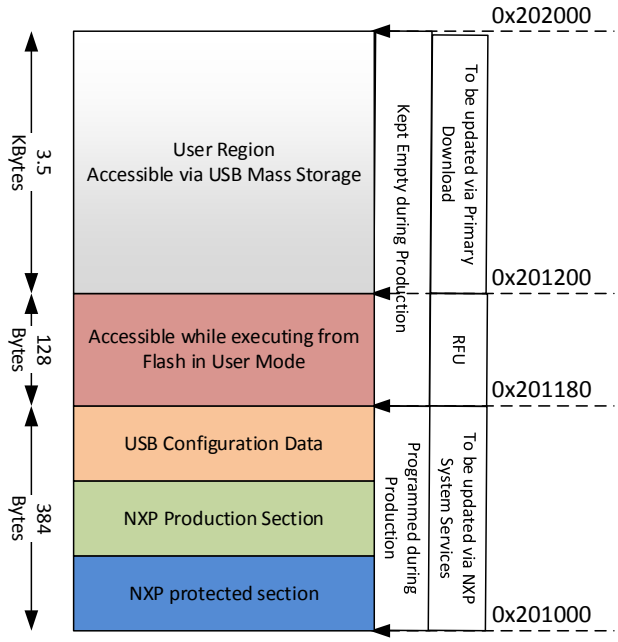

Fig 4. PN7360AU EEPROM memory regions

Note: The RFU section of 128 bytes is kept hidden from the aperture of USB mass storage. But this area is used by the HAL executing from the user flash for internal purposes. As of now, there is no special use case for this region and hence in the default examples of PN7462AU, this region remains unused.

2.2 PN7462AU FW layer dependencies view

Upon POR or wake-up from IC HPD state or IC standby state, the ROM boot is executed.

For primary download mode, USB primary download code follows the ROM boot.

For user application mode, flash boot follows the ROM boot. The flash boot routine is user accessible whereas the ROM boot routine is **not** user accessible. The flash boot uses HALs for device initialization and executes one of the PSP examples provided in the SW package.

The PSP examples use NXP contact protocol library, NXP NFC contactless protocol library, and HALs to demonstrate user application development. The PSP examples and the HALs use ROM services to execute protected functionality.

The PSP examples, PALs and the HALs use RTOS abstraction layer to execute. They can execute either with RTOS (such as FreeRTOS) or without RTOS, using single execution context in the thread mode of ARM.

Also, PSP examples, PALs and the HALs use common utilities layer for functions such as memcpy, delay loop, etc. For simplicity, the dependency link is not shown in the diagram.

2.3 PN7462AU FW modes

The PN7462AU can either be in USB primary download mode or user application mode, depending on the state of DWL_REQ pin and USB_VBUS pin at ROM boot execution.

Table 1. PN7462AU FW modes

DWL_REQ	USB_VBUS	FW mode
0	X	user application mode
1	0	undefined
1	1	USB primary download mode

3. PN7462AU ROM FW

3.1 PN7462AU ROM boot

The ROM boot is executed upon power-on reset, wake-up from IC hard power-down state and wake-up from IC standby state. The ROM boot performs the following functions:

1. Applies the trim values required for proper functioning of the IC.
2. Enable the HW blocks that are specified for the product part.
3. Start the PVDD LDO in case of internal PVDD configuration.
4. Sample the DWL_REQ pin and USB_VBUS pin in case pad voltage is available to determine FW mode.
5. Perform switch to the modes defined in Table 1.

The ROM boot also switches to user application mode in case no pad voltage is available. For switching to user application mode, the ROM boot performs a vector remapping and a CPU core reset.

The ROM boot communicates the boot result code to flash application through PCR_GPREG0_REG register. This result code is apart from HW boot reason which is present in PCR_BOOT_REG.BOOT_REASON and PCR_BOOT2_REG.

3.1.1 PN7462AU ROM boot EEPROM config

The PN7462AU ROM boot depends on the EEPROM parameters. These parameters are present in the NXP protected section of EEPROM memory, ranging from 0x201000 to 0x20117F. The EEPROM parameters are used by the user according to the system design using ROM Services. The parameters are as follows:

Table 2. ROM boot EEPROM parameters

Parameter	Possible value	Description	Default value	Max value
PVDD source	internal (0x55)	if voltage at VBUS pin is > 4 V, ROM boot assumes that PVDD_OUT pin is connected to PVDD_IN pin of the PN7462AU and starts the internal PVDD LDO	33 (Auto)	-
	external (0xAA)	ROM Boot assumes that PVDD_OUT pin is connected to GND and PVDD_IN is connected to external PVDD LDO	-	-
	auto (others)	it can be either internal or external HW configuration and the ROM boot detects the configuration	-	-
PVDD in time-out	Time-out in units of 100 µs	duration for which the ROM boot waits for external PVDD to arrive or internal PVDD output to stabilize on PVDD_IN pin	100 ms	200 ms
VBUS in time-out	Time-out in units of 100 µs	duration for which the ROM boot waits for VBUS voltage to become greater than 4 V in order to start internal PVDD LDO	100 ms	200 ms

3.1.2 PN7462AU ROM boot result code

The boot result codes are communicated to flash application through the first 16 bits of PCR_GPREG0_REG register. They primarily indicate if PVDD is available and the potential cause, if not available. The boot result codes are as follows:

Table 3. Boot result code

Boot result code	Description
0x0000, 0x0001, 0x0003	PVDD is available through internal PVDD LDO
0x0002, 0x0004	PVDD is available through external LDO
0x1000,0x1002,0x1003,0x1007,0x1008	PVDD is not available at PVDD IN even though internal LDO is turned on
0x1001,0x1009	VBUS is not greater than 4 V
0x1004,0x1006	PVDD is not available through external LDO
0x1005	IC woke up from standby because PVDD disappeared either from internal LDO or external LDO

3.2 PN7462AU ROM primary download

USB primary download is a feature available to the user to download code and data to user flash memory and user EEPROM memory using mass storage application respectively.

Based on the variant size, the user flash memory available in the IC from 0x203000 onwards. For example, for 80 k variants, user flash start: 0x00203000, user flash end: 0x00216FFF. For 154 k variants, user flash start: 0x00203000, user flash end: 0x002297FF. For 158 k variants, user flash start: 0x00203000, user flash end: 0x0022A7FF

The user EEPROM memory available in the IC is 3.5 K and is situated in the physical address range from 0x201200 to 0x201FFF.

The quick start guide provides information regarding the usage of USB primary download feature.

3.2.1 PN7462AU ROM primary download EEPROM config

The PN7462AU ROM primary download depends on the EEPROM parameters. These parameters are present in the NXP protected section of EEPROM memory ranging from 0x201000 to 0x20117F. The EEPROM parameters are used by the user according to their system design using the ROM services. The parameters are as follows:

Table 4. ROM primary download EEPROM parameter

Parameter	Possible value	Description	Default value	Max value
code read protection level	0	read, erase and write are allowed	0	-
	1	read and erase are not allowed; write is allowed		
	2	read is not allowed; erase and write are allowed		
	3	read, erase and write are not Allowed		
data read protection level	0	read, erase and write allowed	0	-
	1	read and erase not allowed; write is allowed		
	2	read is not allowed; erase and write are allowed		
	3	read, erase and write are not allowed		
primary download	any value other than 0x96	USB primary download feature is enabled	0	-
	0x96	USB primary download feature is disabled		

The ROM primary download uses USB interface for enumeration. The configuration required for USB interface can be categorized into two parts:

1. USB enumeration-specific configuration.
2. USB HW initialization-specific configuration.

The USB enumeration-specific configuration is fairly straightforward. Users may refer to USB specification and ROM services API for more details.

The USB HW initialization-specific configurations are described in Table 5.

Table 5. Configuration for USB interface

Parameter	Possible value	Description	Default value	Max value
USB discharge	0	disable fast discharge of USB	0	-
	1	enable fast discharge of USB		

Parameter	Possible value	Description	Default value	Max value
XTAL HW activation time-out	time-out in units of 1 ms	duration the primary download waits for XTAL oscillator to be up after HW activation	2 ms	255 ms
XTAL SW activation time-out	time-out in units of 1 ms	duration the primary download waits for XTAL oscillator to be up after SW activation	2 ms	255 ms
USB PLL detection window length	0-255	window length to detect 27.12 MHz PLL input clock	13	255
USB PLL CLK edges	0-255	number of clock edges to detect 27.12 MHz PLL input clock	141	255

3.3 PN7462AU ROM services

The ROM services are accessible via flash APIs present in *root_dir/PN7462AU/phCommon/inc/phhalSysSer.h* and with detailed description in API documentation (.chm file). The PN7462AU provides ROM services for performing the functions described in sections below.

3.3.1 PN7462AU IC lifecycle management services

There are four lifecycle parameters that are used by users at various stages of product development.

3.3.1.1 ROM primary download disable

phhalSysSer_USB_PrimaryDnldConfig() is used to irreversibly disable the ROM primary download feature. On subsequent boots, the ROM boot never enters ROM primary download mode, even if DWL_REQ pin and USB_VBUS pin is high.

This feature is typically used after development and flashing of secondary downloader in the flash memory, for subsequent code/data upgrades.

3.3.1.2 SWD access permissions

When the PN7462AU IC is delivered from production to user, the default SWD access level enables the user to view and debug user flash memory, user EEPROM memory, user RAM memory, and peripheral registers. The access level can be irreversibly changed to prevent view/debug access to any memory region or peripheral registers, before deploying the IC to the field. *phhalSysSer_OTP_SecrowConfig()* can be used to lock the SWD against any further access.

Once SECROW functionality is locked, this feature cannot be used anymore.

3.3.1.3 Code write protection

It is required to lock flash memory from write at HW level. It is locked possibly at a stage when secure secondary upgrade is **not** planned for the remaining lifecycle of the product. For such use cases, *phhalSysSer_OTP_SecrowConfig()* is used to lock flash memory from any further write. Any flash programming after locking the flash results in hard fault.

Once SECROW functionality is locked, this feature cannot be used anymore.

3.3.1.4 RST_N pin behavior

The SecRow contains the bits that control the behavior of HW related to the RST_N pin when pad voltage is not available. Two parameters define the RST_N pin behavior, RST_N pull-down and RST_N value.

The *phhalSysSer_OTP_SecrowConfig()* is used to control the RST_N pin behavior.

Table 6. RST_N pin parameters

RST_N pull down	RST_N value	HW operation
0	X[1]	pad voltage availability is always assumed in this system; IC checks the status of RST_N pin at POR and enters either HPD or starts ROM booting
1	1	pad voltage availability is not assumed in this system; IC does not check the status of RST_N Signal and starts ROM boot normally upon POR

[1] X means “any value”.

Once SECROW functionality is locked, this feature cannot be used anymore.

3.3.1.5 SecRow lock

The HW SecRow contains the SWD access bits, code write-protection bits and RST_N pin behavior bits. For blocking any further writes to SECROW register, the phhalSysSer_OTP_SetSecrowLock() is used. It prevents further usage of phhalSysSer_OTP_SecrowConfig().

Note: phhalSysSer_OTP_SecrowConfig() ROM service API should be used considering EEPROM erase/write limitations. If power fails during an EEPROM write, then the state of the location being written is undefined. EEPROM corruption of SECROW register can compromise boot process, since following the POR the bootup sequence is automatically launched to fetch the 32-bit SECROW located in EEPROM protected area. It is recommended to program SECROW register at the production time only once.

3.3.2 PN7462AU ROM boot configuration

phhalSysSer_USB_PVDD_Config() is used to configure the ROM boot EEPROM configuration; see Section 3.1.1.

3.3.3 PN7462AU ROM primary download configuration

phhalSysSer_USB_PrimaryDnldConfig() and *phhalSysSer_USB_Config()* is used to configure the ROM primary download EEPROM configuration; see Section 3.2.1.

3.3.4 PN7462AU in-application programming

phhalSysSer_SetFlashProgram() is used during secondary FW upgrade developed by the user. Since the secondary downloader of the user executes in flash memory, it cannot write (programmed) at the same time. Hence the programming of flash page is performed from ROM memory and hence this API.

This API returns an error if code write protection is enabled in SECROW register.

3.3.5 PN7462AU CLIF ROM HAL

The CLIF ROM services contain a number of HAL functions. These functions are internally used by the CLIF flash HAL and are not supposed to be directly used by the user application code. Hence, these services are not described in this user manual.

3.3.6 Utilities

The ROM Services provide 2 utilities to customer.

3.3.6.1 CPU reset from flash boot

PN7462AU IC reset always results in ROM booting. However, if the booting has to be done directly from the flash (for example, perform ARM core reset only), the API *phhalSysSer_SetCPU_Reset()* is used. The digital peripherals are **not** reset and may contain residual state.

3.3.6.2 Get die ID

It is a unique IC-specific ID stored in the NXP protected section of EEPROM. User can use *phhalSysSer_GetDieID()* to read this value and further use for their security algorithms (for example, key diversification).

4. PN7462AU user FW

4.1 PN7462AU flash boot

The flash boot is executed when ROM boot performs a vector remapping and the CPU is reset. The relevant functions are present in */root_dir/PN7462AU/phBoot/src.*

The functions performed by flash boot (before jumping to one of the PSP examples) and the sequence of flow is shown in Fig 5. The total time taken for flash boot in NXP provided reference package is ~900 µs.

At the end of flash boot, execution always calls *AppMain().*

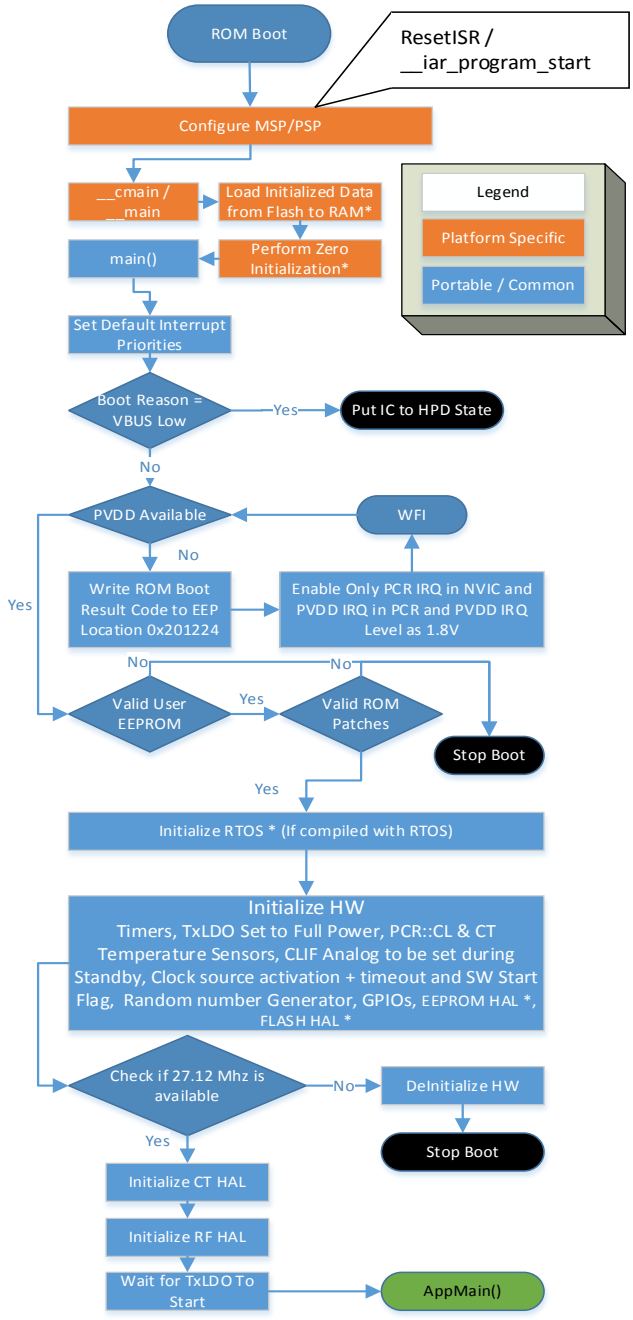

Fig 5. PN7462AU flash boot flow

4.1.1 ARM CPU and RAM regions initialization

The ARM is configured to execute from thread mode. The Main Stack Pointer (MSP) is initialized to the top of user SRAM. The user SRAM region is between 0x102000 to 0x102BFF. The process stack pointer is initialized after end of MSP. The process stack and the heap grow towards each other. The user has to ensure that the process stack and heap do not corrupt each other. The PSP/MSP initialization is done in "phFlashBoot_GCC.c" in LPCXpresso project and in "phFlashBoot_IAR.s" in iAR project. In the Keil project, the "startup_PN74xxxx.s" (part of Keil MDK PN7xxxx installer pack) initializes the PSP/MSP. The user SRAM is limited to 12000 bytes and 256 bytes is used by NXP FW executed in ROM.

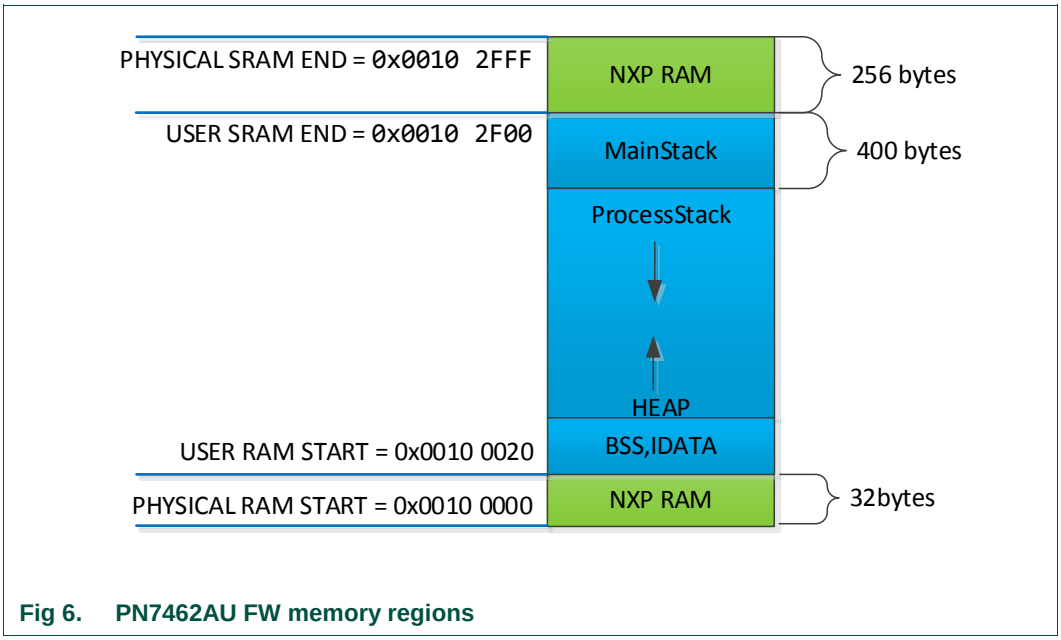

Fig 6. PN7462AU FW memory regions

4.1.2 BSS and IDATA initialization

Due to real-time target mode requirements (response within 5 ms of ROM boot), the flash boot time shall be as low as possible. Hence, the FW shall not contain any IDATA and shall also work with BSS **not** initialized to zero.

At 20 MHz, it takes approximately 171 us to zero initialize 976 bytes of data. This value represents combined ZI data for the Application + Protocol Library + RTOS + HALs. Its size varies based on the application and the configuration of the application (Used HALs, RTOS, Protocols, etc.) itself.

Ideally, the FW may potentially work even with BSS non-initialized to zero. But with integration of many commercially available RTOS or protocol libraries such as NxpNfcReaderLibrary, there is bound to be few bytes of initialized data and a requirement to initialize BSS with zero. Hence to keep the zero initialization to a minimum, `PH_NOINIT` is used to exclude variables from zero-init.

4.1.3 Default interrupt priorities

The interrupt priorities are application-dependent. Hence, there are few restrictions on the user with regards to interrupt priority setting. SVC is used for application purposes. However, ROM services shall **not** be called from SVC handler since ROM uses SVC handler for patch mechanism. Similarly, SVC handler must be set to highest priority before calling system services. The typical interrupt priority settings are shown below and are set in *phFlashBoot_SetIntrptsPrio().*

Table 7. Default interrupt priorities

Exception/IRQ	Priority	Description
RESET	-3 (highest)	reset ISR
NMI	-2	upon watchdog timer expiry
HardFault	-1	upon invalid memory accesses
SVC	PH_HAL_INTRPT_PRIO_REALTIME	used by system services function; is used by user application, but system services shall not be called from an SVC handler
CLIF, HIF, CTIF, PMU, PCR	PH_HAL_INTRPT_PRIO_HIGH	interfaces ISR and system event ISRs
Timer, EECNTRL, SPIM, I^2CM	PH_HAL_INTRPT_PRIO_HIGH	utility ISRs and master interfaces
PendSV, SysTick	PH_HAL_INTRPT_PRIO_LOW	context switching ISRs

4.1.4 Boot reason and result code handling

The flash boot checks PCR registers for exceptions in the boot reason and boot result code provided by the HW and ROM boot respectively; see Section 3.1.2.

If the HW boot reason is VBUS_LOW, it means that the IC wakes up from standby because the VBUS dropped below threshold. In such cases, the flash boot configures the IC to enter hard power down state. If there is any other boot reason, the flash boot checks for pad voltage availability. If the pad voltage is not available, it writes the boot result code to EEPROM location 0x201224 and waits for PVDD to arrive, by executing WFI (ARM SLEEP instruction).

The exception checks are performed using *phFlashBoot_PreCheck().*

4.2 PN7462AU HALs initialization at boot UP

The following initialization happens during default boot up of PN7462AU.

4.2.1 Temperature sensor initialization and RF standby configuration

The PN7462AU consists of two temperature sensors, one for contactless front end and another for contact front end. In this initialization, the temperature sensors are enabled and the lower and upper thresholds are configured.

The CLIF transmitter analog configuration required during standby/suspend are also initialized.

During USB suspend, a set of registers are configured to reduce the power configuration. This set of registers are provided during initialization.

It is recommended not to change the value different from the default value in EEPROM.

Table 8. EEPROM parameters for temperature sensors - PcrPwrTempConfig

Power clock reset temperature configuration related to phhalPcr_PwrTempConfig_t

Type	Field name	Default value	Description
u8	bUseTempSensor0	0	flag to indicate to use temperature sensor 0 or not 0: Disabled 1: Enabled
u8	bUseTempSensor1	0	flag to indicate to use temperature sensor 1 or not 0: Disabled 1: Enabled
u8	bLowTempTarget0	3	0 : 135 1 : 130 2 : 125 3 : 120
u8	bLowTempTarget1	3	0 : 135 1 : 130 2 : 125 3 : 120
u8	bHighTempTarget0	0	0 : 135 1 : 130 2 : 125 3 : 120
u8	bHighTempTarget1	0	0 : 135 1 : 130 2 : 125

For a detailed parameter description, and parameter addresses in the EEPROM refer to the EEPROM description [2] file.

Table 9. EEPROM parameters for power down settings - PcrPwrDown

It is a 32-bit value bit-file created by ORing enums of type phhalPcr_PwrDown_Setting_t used to select which settings must be applied to reduce power consumption during suspend.

Type	Field name	Default value	Description
u32	dwPwrDownSettings	0x7FFFFFFF	0x7FFFFFFF: E_APPLY_ALL_SETTNGS. i.e. all power reduction settings are applied during suspend

Table 10. EEPROM parameters for temperature sensors - TxAnaStandByConfing

TxAna register settings for standby; See phhalPcr_TxAnaStandByConfig_t

Type	Field name	Default value	Description
u32	dwAna Tx StandBy Value	0x0F	to hold CLIF standby GSN value selection
u32	dwAna Tx Pro tStandBy Value	0x03	to hold the CLIF configuration related to power-down

For a detailed parameter description and parameter addresses in the EEPROM refer to the EEPROM description [2] file.

4.2.2 CLKGEN initialization

The CLKGEN HW can have two sources: XTAL or external clock. Depending on the crystal characteristics, the activation time-out can be different. If HW activation fails, the HAL and HW provides a mechanism to perform SW activation. All these options are initialized during the flash boot.

The user shall modify the EEPROM according to the board/system design.

Table 11. EEPROM parameters for CLKGEN - Clkgen

Clock generator. *See phhalClkGen_Init_*

Type	Field name	Default value	Description
u16	wXtalActivationTimeOut	2000	dwXtalActivationTimeOut activation time-out value
u8	eSource	0x00	eSource clock source selection, See phhalClkGen_Source_t
u8	bKickOnError	0x00	bKickOnError kick on error.

For a detailed parameter description and parameter addresses in the EEPROM, refer to the EEPROM description [2] file.

4.2.3 CLIF transmitter TxLDO initialization

The TxLDO used for CLIF transmitter is initialized as part of the boot. The parameters such as whether internal TxLDO is used or external TxLDO is used, the power configuration for full power (used in reader mode and SL-ALM card mode if internal TxLDO is used), low power configuration (for PLM card mode), TxLDO start-up time and over current enable are configured.

The user shall modify the EEPROM according to the board/system design.

Table 12. EEPROM parameters for PMU – CLIF transmitter TxLDO

Power management unit. (See phhalPmu_TxLdoInit and phhalPmu_TxLdoParams_t)

Type	Field name	Default value	Description
u8	bUseTxLdo	0x01	parameter to use internal TxLDO or to use external TxLDO 0: do not use internal TxLDO 1: use internal TxLDO
u8	eFullPowerTvddSel	0x04	TVDD Sel for reader mode. See phhalPmu_TvddSel_t 0 : 3 V 1 : 3.3 V 2 : 3.6 V 3 : 4.5 V 4 : 4.7 V other: invalid
u8	eLowPowerTvddSrc	0x01	Source for the TVDD See phhalPmu_LowPower_TvddSrc_t 0: source is TVDD In 1: source is VUP 2: source is VBUS other: invalid
u8	bOverCurrentEnable	0x00	Over current interrupt 0: Disabled others: Enable
u16	wWaitTime	250	waiting time after the TxLDO is started 250us is typical value. Maximum 500us

For a detailed parameter description and parameter addresses in the EEPROM, refer to the EEPROM description [2] file.

4.2.4 RNG HW Initialization

The time-out for true random number generation is initialized. It is recommended not to change the value different from the default value in EEPROM.

Table 13. EEPROM parameters for RNG HW - RNG

Random Number Generator. See phhalRng_Init

Type	Field name	Default value	Description
u16	wXtalActivationTimeOut	2000	dwXtalActivationTimeOut activation time-out value

For a detailed parameter description and parameter addresses in the EEPROM, refer to the EEPROM description [2] file.

4.2.5 GPIO initialization

The GPIOs are initialized for I/O and pullup/pulldown in case of input and slew rate in case of output.

The customer shall modify the EEPROM from where the GPIO initialization configuration is taken in accordance to the board/system design.

Table 14. EEPROM parameters for GPIO

GPIO Bootup configuration. Each byte represents a GPIO configuration starting from GPIO 1 to 12. See phCfg_EE_Boot_GPIO_t.

Type	Field name	Default value	Description
u8[12]	OutputPUPD	00 00 00 00 00 00 03 03 07 03 03 03 (hex)	lower nibble - related to output configuration upper nibble - related to pull-up/pull-down configuration Bit0 = 0: skip configuration as output on boot Bit0 = 1: configure GPIO as output Bit1 = 1: enable slew-rate Bit2 = 1: drive the output high Bit2 = 0: drive the output low Bit5 = 1: apply pull-up Bit6 = 1: apply pull-down
u8[12]	InputISR	00 00 00 00 00 00 00 00 00 00 00 00 (hex)	ALL = 0: skip configuration on boot Bit0 = 0: unconfigure as input Bit0 = 1: configure/SET as input Bit1 = 1: GPIO is a wake-up source Bit2 = 1: GPIO is an interrupt source Bit4 = 1: level sensitive interrupt Bit5 = 1: interrupt on active low or falling edge

For a detailed parameter description and parameter addresses in the EEPROM refer to the EEPROM description [2] file.

4.2.6 General-purpose timers initialization

The PN7462AU consists of four general-purpose timers. The HAL context to manage the timer requests are initialized during the flash boot.

4.2.7 Clock 27.12 MHz check

In PN7462AU, the FW always assumes the availability of 27.12 MHz clock sourced from either external crystal or external clock. This clock is required for all communication interfaces and flash programming. If a crystal is used, PN7462AU HW has a crystal oscillator that is activated by default and it takes maximum of 2 ms to activate and generate stable 27.12 MHz.

For this purpose, the flash boot performs most HW initialization to utilize the XTAL activation time. Hence the flash boot checks if the 27.12 MHz is available and if not available de-configures all HALs and stops booting. If available, flash boot proceeds to initialization of HALs that require 27.12 MHz clock.

4.2.8 EEPROM/flash HAL initialization

These HALs are not used in a typical contactless or contact application. They are used during secondary downloader application. By default, these HALs are disabled. The compile-time directives, *NXPBUILD__PHHAL_EEPROM* and *NXPBUILD__PHHAL_FLASH* enable the HALs.

After the common HALs initialization, the flash boot jumps to user application program.

Note: The secondary bootloader is placed directly after the flash boot code and provides the functionality to download a user application program; see Fig 2. The HAL API provides functions that enable to read and write user data on the flash or EEPROM. All the functions are described in PN7462AU FW API Guide document under “Hardware Abstraction Layer - Generic HALs - FLASH HAL / EEPROM HAL” and in the files *“root_dir/PN7462AU/phCommon/inc/ phhalFlash.h” and “root_dir/PN7462AU/phCommon/inc/phhalEeprom.h”*. Use *NXPBUILD__PHHAL_EEPROM* and *NXPBUILD__PHHAL_FLASH* compile-time directives to enable them. By default, both directives are disabled and are available in “*ph_NxpBuild_Default.h”.*

During data reading and writing, best time is achieved with flash HIGH perf ON and ramp clock ON.

4.2.9 RF HAL initialization

The CLIF IP registers are set to pre-defined values from EEPROM and few registers are reset to default values. Event mechanisms are initialized to act as IPC between ISR and HAL. This initialization prepares the RF HAL to transition to either target mode if external RF field is present or reader mode if application wishes.

By default, the operating mode of CLIF HAL is set to NFC Forum mode. It means that the guard times for various technologies are applied according to NFC Forum. All spurious interrupts are cleared and interrupt enabled at ARM level (NVIC interrupt).

Also, the DPC feature for reader mode and APC feature for SL-ALM card mode are initialized.

The EEPROM required for CLIF is split into 2 memory regions: One common region and one protocol-specific region.

4.2.10 CT HAL initialization

The CLIF IP registers are set to pre-defined values from EEPROM and few registers are reset to default values. Event mechanisms are initialized to act as IPC between ISR and HAL. Primarily the connector type (open/closed), pull-up/pull-down configuration, automatic CT deactivation and slew rate of the contact pads are configured.

If the pull-up/pull-down *configuration is mismatched between the EEPROM and the actual setting on board, spurious interrupt may occur.*

The customer shall modify the EEPROM in accordance to board/system design.

Table 15. EEPROM parameters for CT

Initial settings for CT interface. See also phhalCt_InitParam_t

Type	Field name	Default value	Description
u8	bPullUp	1	pull up configuration 0: pull-down 1: pull-up others: undefined behavior
u8	bConnectorType	1	connector type 0: normally closed others: normally open
u8	bAuto CT Deactivation Enable	1	auto deactivation 0: Disabled others: Enabled
u8	bSlewRate	0x38	CLK,IO,VCC slew rate 0: CLK,IO,VCC slew rate others: This value is directly mapped to ct_srr_reg to give enough options

For a detailed parameter description and parameter addresses in the EEPROM, refer to the EEPROM description [2] file.

4.2.11 HAL deinitialization

If any of the initializations fail, all the HALs are de-initialized in the phFlashBoot_HwTearDown() API.

4.3 PN7462AU generic HALs

For details of HAL functions and their description, refer to the API guide (CHM document). In the sections discussed below, functional usage of the HALs are described. The usage activity diagrams show a representation of usage and parameters.

To provide more memory for customer application, some of the below HAL functions are completely or partially moved to ROM3 region. The “PN7462AU_ROM3.h” contains the ROM3 functions that are used internally by the flash HAL implementation or flash HAL API.

4.3.1 Timer HAL

The PN7462AU IC provides four general-purpose timers, watchdog timer and system tick timer.

The SysTick timer is used for RTOS scheduler and is initialized in the RTOS-specific configuration file.

The general-purpose timer 0 and timer 1 are 12-bit timers at 3 kHz frequency. Timer 2 and Timer 3 are 32-bit timers at 20 MHz frequency. The Timer 0 and Timer 1 can be concatenated as a single timer with Timer 1 incrementing a step upon Timer 0 completion.

The Timer HAL provides APIs to manage the lifecycle of an HW timer; see Fig 7.

Upon request of a timer for a time unit (microsecond/millisecond/second), the timer HAL allocates the HW timer according to Table 16.

Upon successful timer request, the allocated timer may be configured for different timeouts and callbacks after one or multiple start-stop cycles.

The timer IRQ executes the callback in the ISR context.

The flash HAL implementation provides a wrapper for context management and the core register functions are present in ROM3.

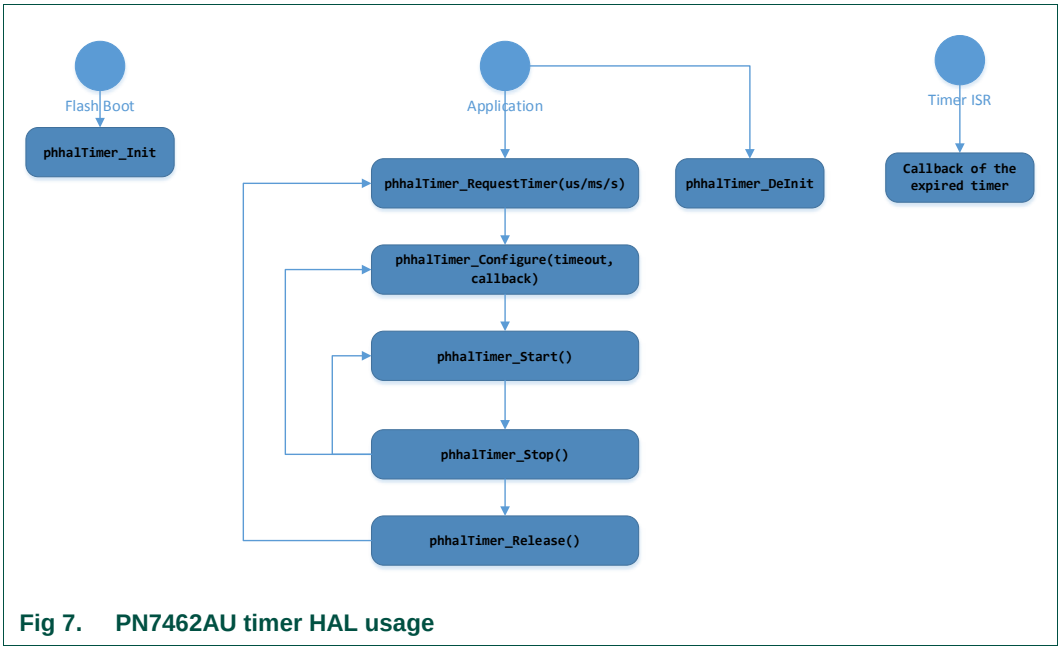

Fig 7. PN7462AU timer HAL usage

Table 16. HAL timer allocation

Timer	Width	Frequency	Min to	Max to	Recommended use
Timer 0	12-bit	3 kHz	330 µs	1351 ms	millisecond timer
Timer 1	12-bit	3 kHz	330 µs	1351 ms	millisecond timer
Timer 0 & Timer 1		3 kHz		4095 × 1351 ms = 5532 s	first priority: seconds timer second priority: millisecond timer
Timer 2	32-bit	20 MHz	50 ns	215 s	first priority: microsecond timer second priority: seconds timer
Timer 3	32-bit	20 MHz	50 ns	215 s	
Watchdog	10-bit	45 Hz	21.5 ms	22 s	recovery from HW or SW hangs or loops

The watchdog timer is a 10-bit timer at an approximate frequency of 45 Hz. Upon watchdog timer expiry, the watchdog timer asserts an IC reset. In order to perform recovery or cleanup tasks, a watchdog threshold is available. Upon watchdog threshold, FW is interrupted with an NMI and the FW can choose to re-initialize the watchdog timer or perform cleanup (before IC is reset). The activity flow is shown in Fig 8.

The WDT HAL is implemented in flash HAL since customer may modify it according to the application.

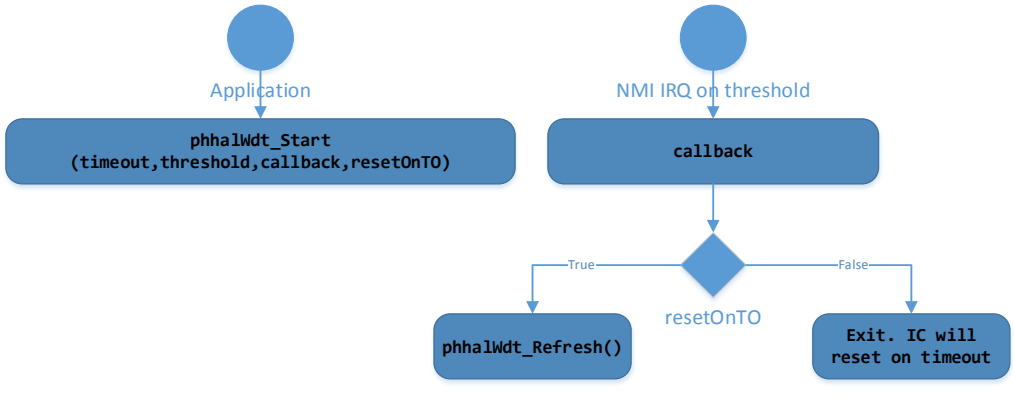

Fig 8. PN7462AU WDT HAL usage

4.3.2 CRC HAL

The PN7462AU IC provides a CRC co-processor to compute 16/32-bit CRC for a 32/16/8-bit input data. The co-processor computes the 16/32-bit CRC in parallel, providing the output in one clock cycle. The CRC computation can be done with MSB first or LSB first of input data stream. The CRC HAL provides the following functions:

- Calculate the CRC over a buffer of arbitrary length.

- Check the CRC of a buffer of arbitrary length with supplied CRC. The supplied CRC shall be last 2 bytes or 4 bytes of the buffer.

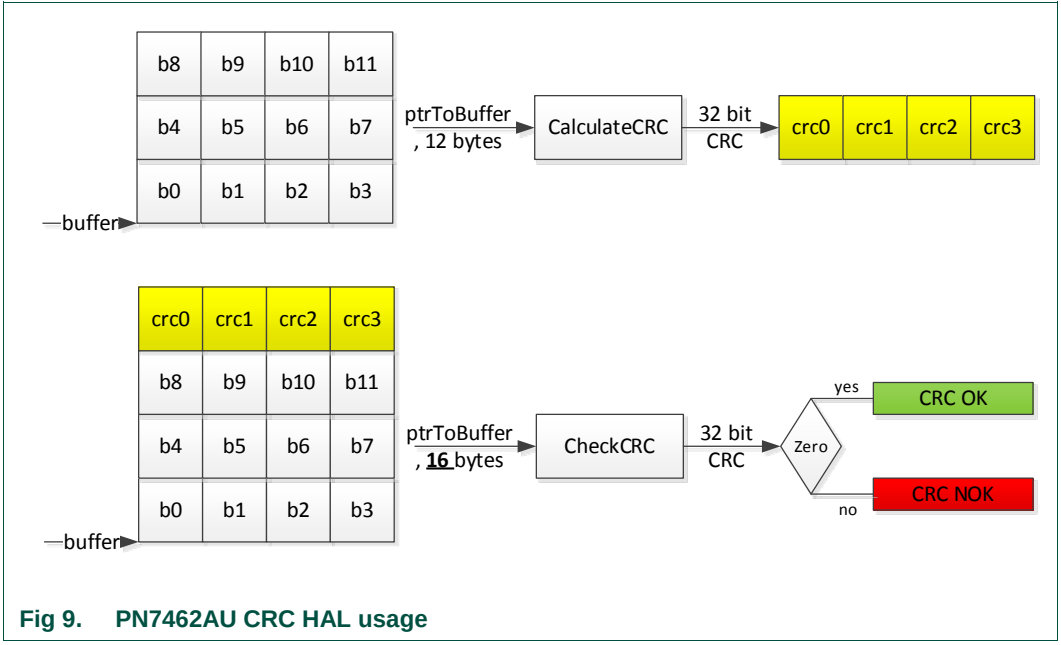

Fig 9. PN7462AU CRC HAL usage

4.3.3 RNG HAL

The PN7462AU IC provides an RNG co-processor that generates pseudo-random numbers. The RNG HAL provides APIs to generate one or more random numbers. The *phhalRng_GenerateRng* returns time-out error if random number is **not** generated within the initialized time-out.

This HAL is implemented in ROM3.

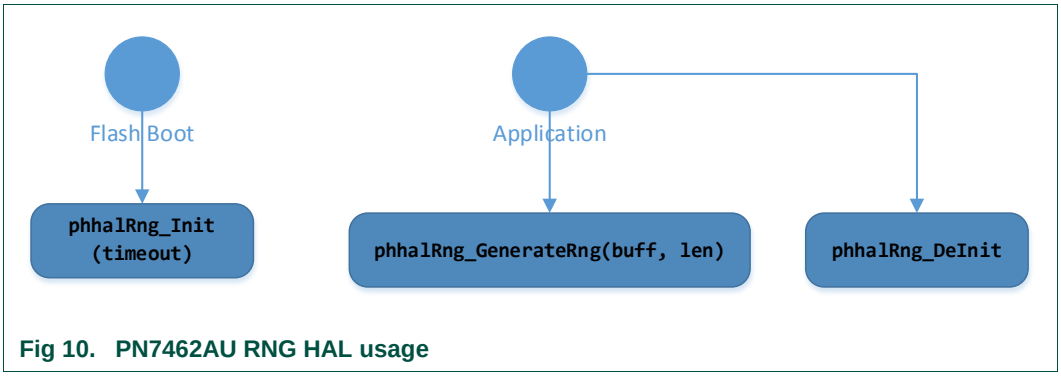

Fig 10. PN7462AU RNG HAL usage

4.4 PN7462AU master interface HALs

4.4.1 I²CM HAL

The I²CM HAL provides the following features.

4.4.1.1 Device-specific configuration

1. TX/RX FIFO threshold.
2. TX/RX completion time-out since the HALs are blocking.
3. Retry count of any transaction.

- The *phhalII2CM_Init() and phhalI2CM_DeInit()* used to set/reset this configuration.

4.4.1.2 Slave specific configuration

- Baud rate
- The SDA hold time
- 7-bit or 10-bit Address type of the slave

The baud rate is used to calculate the SCL frequency based in the equation. Users shall calculate the baud rate field according to their required SCL frequency

$$SCL_{Frequency} = \frac{27.12\ MHz}{(27+Boudrate)}$$

- The phhalI2CM_Config() is used for configuration.

4.4.1.3 Slave presence check

The *phhalI2CM_SlaveCheck()* is used to perform this check.

4.4.1.4 I²C-bus reset

- General call reset addressing is used to reset the I²C-bus that resets all attached slaves.
- The *phhalI2CM_GenCallReset()* is used to for configuration.

4.4.1.5 Single transaction

- This feature is used when the length of the transaction is greater than 32 bytes.
- The *phhalI2CM_Transmit() and phhalI2CM_Recieve()* are used to perform this transaction.

4.4.1.6 Multiple transactions

- This feature is used when multiple short (less than 4 bytes, 8 bytes, or 16 bytes) transactions are done to a single slave such as TDA registers read or write

- The *phhalI2CM_QueueTx(), phhalI2CM_QueueRx() and phhalI2CM_Start()* are used to perform these transactions.
- This features are by compile time macro *NXPBUILD__PHHAL_I2CM_MULTI_TRANSACTION*

4.4.1.7 Device reset

- This feature is used to reset the I²CM HW in cases when the bus is idle due to HW stuck in an erroneous bus condition. This is **not** I²C bus reset explained in item (4)
- The *phhalI2CM_Reset()* is used to perform this feature.

4.4.1.8 I²CM HAL usage overview

The I²CM Core register functions are implemented in ROM3 and logical functions are implemented in flash HAL.

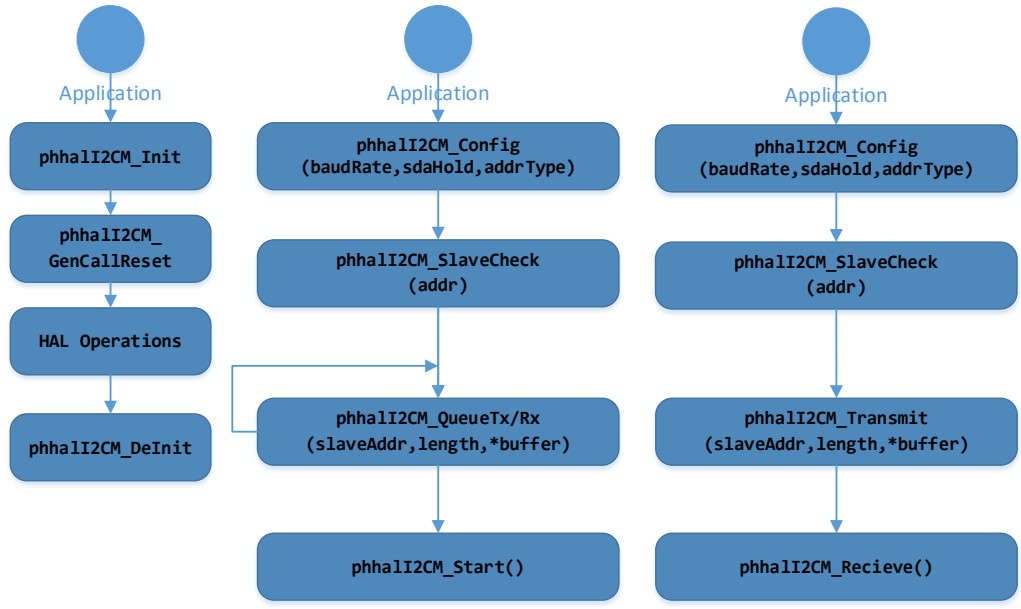

Fig 11. PN7462AU I²CM HAL usage

4.4.2 SPIM HAL

The SPIM HAL provides the following features.

4.4.2.1 Device-specific configuration

- TX/RX completion time-out since the HALs are blocking.
- phhalSPIM_Init() and phhalSPIM_DeInit() are used to set or reset this configuration.

4.4.2.2 Slave specific configuration

- Slave to be configured
- MSB or LSB first transmission
- CPOL/CPHA modes
- Baud rate
- NSS-specific configuration
- *phhalSPIM_Configure()* used to perform this configuration

4.4.2.3 TX/RX transactions

Since SPIM can be used for SD card use case, CRC configuration is required for some transactions and **not** required for some transactions to the same slave. Hence, every transaction has CRC configuration which can be enabled or disabled. *phhalSPIM_Transmit() / Receive() / Transmit_Continue() / Receive_Continue()* are used for transactions.

4.4.2.4 Water level configuration

Since SPIM HW is DMA-based, water level interrupt can be used to detect early reception complete or transmission complete and synchronize dependent functions. *phhalSPIM_SetWaterLevel()* is used to set the water level and the function callback is called in case there is a water level interrupt.

The SPIM Core register functions are implemented in ROM3 and logical functions are implemented in flash HAL.

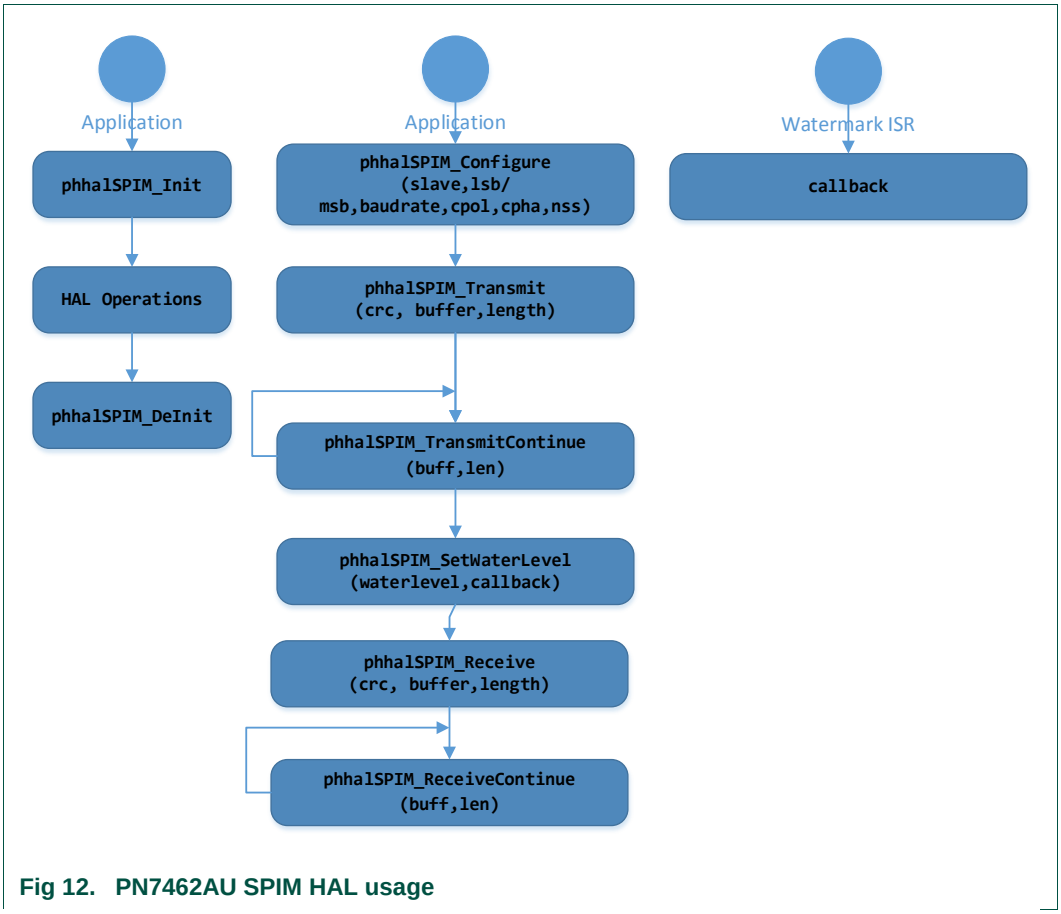

Fig 12. PN7462AU SPIM HAL usage

4.5 Host interface HAL

The PN7462AU provides four host interfaces to communicate with a host processor. The sections described below explain the functions of the host interface. At any instance, only one host interface shall be used. Within a boot session of PN7462AU, only one host interface is assumed and dynamic switching between host interfaces without an intermediate reset is **not** assumed.

The host interface can be I²C, SPI, HSU or USB.

4.5.1 I²C

- The HIF HAL initializes the I²C physical interface with 7-bit slave address and enables or disables HW response to device ID request from external I²CM. It also configures whether I²C core should reset the complete IC when receiving general call address for I²C-bus reset. It also configures if the I²C slave should switch to HS mode upon request from I²CM master.

4.5.2 SPI

- The HIF HAL initializes the SPI physical interface with either of the four modes of SPI operation.

Table 17. SPI operation modes

CPOL	CPHA
Active low	sampling @ even edge
Active low	sampling @ odd edge
Active high	sampling @ even edge
Active high	sampling @ odd edge

4.5.3 HSU

- The HIF HAL initializes the HSU physical interface with the EOF size, baud rate and number of stop bits. The interface can also be initialized with auto baud rate estimator. If the baud rate estimator is enabled, the HIF HAL ensures that no transmission can take place without first reception.

4.5.3.1 HSU standby scenario

- When a host sends some frames over HSU during the time PN7462AU IC is in standby, one to three bytes of frame are lost. They are characterized and the host shall always send dummy one to three bytes before actual frame. To make buffer management simple, HIF HAL always reserves the dummy bytes in the buffer. It is done so that the received frame is stored at the same offset every time.

The host interface HAL provides initialization API to configure above HW features – *phhalHif_Init()* API.

4.5.4 USB

The USB device controller enables USB 2.0 full-speed (12 Mbit per second) data exchange with a USB host controller and USB 3.0 hub connection capability.

4.5.5 Frame interfaces

The PN7462AU provides three different frame interfaces as described below. The frame interface to be chosen is initialized using *phhalHif_Init()* API.

4.5.5.1 Fixed-format frame interface

In this frame interface, the host processor (e.g.: LPC) and the HIF HW of PN7462AU agrees that the frames to be exchanged shall have a header containing the length and a trailer containing the 16-bit CRC. The header shall be minimum 2 bytes and maximum of 4 bytes. The length field can be maximum of 10 bits and can be positioned anywhere within the header.

When the HIF HAL is configured for fixed-format frame interface,

- At reception, the HW shall retrieve the length from the first 2/3/4 bytes of received data (e.g.: header) and shall count that many bytes of payload of further reception. After the payload, the HW checks the CRC of the received payload.
- Similarly, at transmission, the HW shall retrieve length from the first 2/3/4 bytes of transmit buffer and shall transmit the bytes as payload from the transmit buffer. After that, the HW appends the 16-bit CRC.

Maximum payload that can be transmitted or received is 1024 bytes.

4.5.5.2 Free format frame interface

In this frame interface, the host processor and the HIF HW of PN7462AU do not agree on any fixed format of the frame. Hence the HIF HW cannot parse the header and know the length of payload. Hence, the HIF HW cannot count the number of transmitted or received bytes and also cannot perform CRC checking or generation. Hence,

- At reception, the HIF HW uses the physical interface start and stop conditions to determine the length of reception.
- At transmission, a TX length register to determine the length of transmission.

The HIF application has to perform CRC checking/generation. The format that is configured is applicable to both TX and RX.

Maximum payload that can be transmitted or received is 250 bytes.

4.5.5.3 Native format frame interface

This format is same as free format with the exception that the maximum payload that can be transmitted or received is 1024 bytes.

4.5.6 Buffer interface HAL

The HIF HW in PN7462AU IC provides four RX buffers and one TX buffer. The HIF HAL provides APIs for HIF application to request RX buffer and release RX buffer. The HAL also provides APIs to send TX buffer. The access to these APIs is allowed in single task context only. The HAL manages the error handling in case of fixed-format mode. The HAL configures the HW to either discard erroneous buffer or retain the buffer and pass it to the application.

The ISR calls the callback function upon reception complete or transmission complete.

The HIF Core register functions are implemented in ROM3 and logical functions are implemented in flash HAL.

4.6 PN7462AU PCR HAL

The PN7462AU IC provides two low-power modes:

1. Standby mode

2. Suspend mode (only when USB interface is used)

The PCR HAL is primarily used to put the IC into one of these modes.

The *phhalPcr_Init()* is used by the flash boot to initialize the temperature sensors and CLIF analog characteristics during standby mode. The *phhalPcr_Init()* also takes as input a bitmap that indicates power down settings during IC suspend (USB suspend). The USB suspend performs these power-down settings as well as restores the settings when resuming.

4.6.1 Wake-up sources and prevention reason

The FW configures the sources that would wake up the IC from standby or suspend state. The sources that could wake up the IC are tabulated below. Refer PCR_WAKEUP_CFG_REG.

Table 18. Wake-up source

Wake-up source	FW configuration
WUC (WUC_VALUE)	The FW configures the WUC_VALUE and enable this source so that the IC wakes up every WUC_VALUE time period and performing polling operations
RFLDT	The FW enables this source so that the IC wakes up when an external RF is detected. If listen mode is **not** supported, as in the case of EMVCo reader, this source is disabled
TEMP0 (CL sensor)	The FW enables it during flash boot initialization
TEMP1 (CT sensor)	The FW enables it during flash boot initialization
GPIO	The FW enables this source if any peripheral is connected to GPIO that wakes up the IC; for example, Keypad
PVDD_LIMITER	If internal LDO is used to generate PVDD, the FW enables this source. While in standby, if there is a fault causing more current draw from PVDD LDO, then IC is woken up. But no functionality is possible because pad voltage is **not** available.
CT_PR	The FW enables this source if a CT card has to be detected while in standby. If WUC is also enabled, there is a possibility that IC always wakes up because of WUC and CT presence is never detected.
INT_AUX	The FW enables this source if a CT card connected to a TDA (through IO-AUX) has to be detected while in standby
TVDD_MON	If internal LDO is used to generate TVDD, the FW enables this source. While in standby, if there is a fault causing more current draw from TX LDO, then IC is woken up. No RF functionality may be possible.
VBUS_LOW	The FW enables this source if for example, the device is a battery-operated one and a drop in VBUS is detected while in standby so that indication can be provided to users.

The FW uses the EEPROM structure *phhalPcr_WakeUpConfig_t* `to` configure required wake-up sources depending on the requirement.

Table 19. EEPROM parameters for PCR HAL – Wake-up config

Wake-up Sources see also phhalPcr_WakeUpConfig_t

Type	Field name	Default value	Description
u16	wWakeUpTimerVal	300	Timer value for the wake-up in milliseconds
u8	bEnableHIFWakeup	0	Flag to know the host interface wake-up 0: Disabled 1: Enabled
u8	bI2CAddr	0x28	I^2C address if the wake-up is configured for HIF
u8	bWakeUpTimer	1	Flag to enable the wake-up timer as wake-up source 0: Disabled 1: Enabled
u8	bWakeUpRfLdt	0	Flag to enable the RfLdt as wake-up source 0: Disabled 1: Enabled
u8	bWakeUpPvddLim	1	Flag to enable PVDD current limitation as wake-up source when it goes below the lower threshold 0: Disabled 1: Enabled
u8	bWakeUpCtPr	1	Flag to enable CT presence as wake-up source when it goes below the lower threshold 0: Disabled 1: Enabled
u8	bWakeUpIntAux	0	Flag to enable PVDD Auxiliary interrupt as wake-up source when it goes below the lower threshold 0: Disabled 1: Enabled
u8	bWakeUpTvddMon	0	Flag to enable TVDD Monitoring as wake-up source when it goes below the lower threshold 0: Disabled 1: Enabled
u8	bWakeUpGpio	0	Flag to enable GPIO as wake-up source when it goes below the lower threshold 0: Disabled 1: Enabled

For a detailed parameter description, and parameter addresses in the EEPROM refer to the EEPROM description [2] file.

If the PN7462AU is **not** able to go to standby or suspend mode due to some reasons such as ongoing transaction in the interfaces, the HW provides the reason.

The *phhalPcr_ApplyLowPower()* is used to make the PN7462AU to enter standby or suspend mode. The first parameter in the *phhalPcr_ApplyLowPower()* is used to configure the wake-up sources and the second parameter gets the reason for prevention, if any.

Note: If standby is successful, this API will never back. If suspend is successful, this API returns back when the IC resumes because of any wake-up source (including USB host initiated resume).

4.6.2 Context saving

The PCR HW in PN7462AU provides eight general-purpose registers which are reserved during standby (PCR_GPREG0_REG - PCR_GPREG7_REG). An application may use *phhalPcr_SaveContext()* to save seven DWORDs before entering standby. After waking up, it may use *phhalPcr_RestoreContext()* to retrieve the saved DWORDs. The API allows to save seven DWORDs only since PCR_GPREG0_REG is internally used by the HALs.

4.6.3 Power down settings during USB suspend mode

The customer can add or delete this configuration according to the specification and board connections. An application may use phhalPcr_ReducePowerConsumption() API to apply this power down settings. Refer Table 9

Table 20. PWD settings during USB suspend mode

Bit	Description
Bit 0	pull down GPIO pad 1
Bit 1	pull down GPIO pad 2
Bit 2	pull down GPIO pad 3
Bit 3	pull down GPIO pad 4
Bit 4	pull down GPIO pad 5
Bit 5	pull down GPIO pad 6
Bit 6	pull down GPIO pad 7
Bit 7	pull down GPIO pad 8
Bit 8	pull down GPIO pad 9
Bit 9	pull down GPIO pad 10
Bit 10	pull down GPIO pad 11
Bit 11	pull down GPIO pad 12
Bit 12	pull down ATX A pad
Bit 13	pull down ATX B pad
Bit 14	pull down ATX C pad
Bit 15	pull down ATX D pad
Bit 16	pull down INT_AUX pad
Bit 17	pull down IO_AUX pad
Bit 18	pull down CLK_AUX pad
Bit 19	pull down all SPIM pads(SCK, NSS, MOSI, MISO)
Bit 20	pull down all IICM pads(SCL, SDA)
Bit 21	Disable CT clock
Bit 22	Disable CLIF modules and clocks (disable CLIF_PLL, power down sub-modules)
Bit 23	GSN value in standby mode set to standby value

Note: If an GPIO pin is used as wake-up source from USB suspend mode, then GPIO pin should be configured as input and pulled down. The corresponding GPIO bit position in dwPwrDownSettings in EEPROM should be set to 0 to prevent overwriting GPIO pad configuration in phhalPcr_ReducePowerConsumption() API.

4.6.4 Register IRQ callback

The PCR HW events are asynchronous. An application may wish to register for specific asynchronous events such as VBUSP (for DCDC LDO) monitor status, PVDD LDO current limiter irq take necessary action.

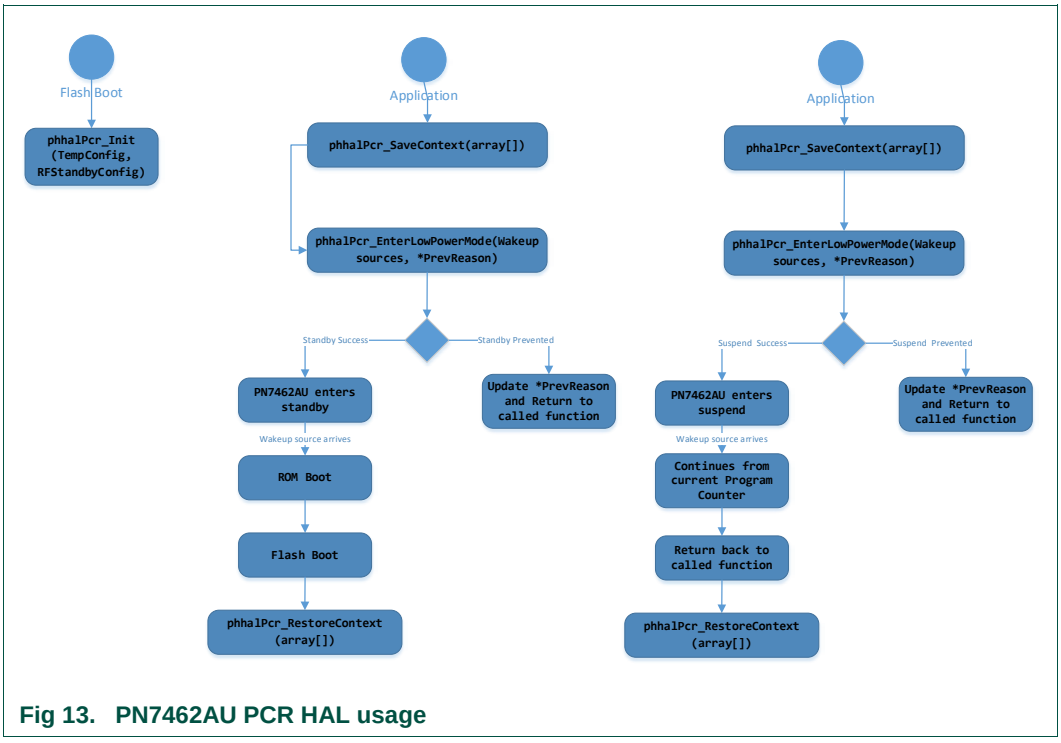

Fig 13. PN7462AU PCR HAL usage

4.7 PN7462AU PMU HAL

The PN7462AU provides three LDOs:

1. TXLDO for CLIF transmitter
2. PVDD LDO for generating 3.3 V pad voltages (requirement for USB)
3. DC-DC LDO for CT communication

The PMU HAL provides interfaces for application to configure these LDOs depending on the application use case.

4.7.1 TXLDO HAL

The TXLDO is started in full power mode.

In full power mode, the TXLDO can be configured to provide an output voltage of 3.0 V/3.3 V/3.6 V/4.5 V/4.75 V.

Optionally, overcurrent detector can be enabled during its on-period.

The default boot from flash uses *phhalPmu_TxLdoInit()* to configure these parameters (by reading values from the EEPROM). The API *phhalPmu_TxLdoStart()* is used to start the TXLDO. Whenever the CLIF HW is **not** used (to reduce power consumption), the TXLDO can be stopped using the *phhalPmu_TxLdoStop()* API.

For the use cases where the TXLDO output requirement is 4.5 V or 4.75 V, it is required to have 5 V at the input of TXLDO. It is checked by the application using *phhalPmu_TxLdo5VMonitor()*.

In the reference boot code and standby/suspend APIs provided by NXP uses the *phhalPmu_TxLdoStart()* on boot up and before entering standby/suspend mode.

For system configuration, the system does not use internal TXLDO for TVDD. It uses an external LDO to connect to TVDD. The application shall set the information during initialization.

Note: If TXLDO is **not** used, HAL does not check if TVDD is available.

In case of external TVDD, TVDD_IN supply must be stable before turning on the RF field. User application should ensure that TVDD_IN supply is stable before turning ON RF field. The PN7462 includes two levels (4 V and 3.3 V) voltage monitor for monitoring the voltage on the TVDD_IN or VUP_TX pins. Voltage Monitor can be configured to monitor voltage on either TVDD_IN or VUP_TX pins.

Two APIs *"phhalPmu_TxLdoMonitorEnable"* and "*phhalPmu_TxLdoMonitorCheck*" are provided to configure and Check Monitor. These APIs should be use by User application to ensure TVDD_IN Supply is stable before turning ON RF field.

4.7.2 DC-to-DC LDO HAL

The DC-to-DC LDO is primarily used to drive the VCC LDO either in the follower mode or in doubler mode, depending on the configuration discussed in Table 21.

Table 21. DC-to-DC LDO mode configuration

Class of card	Voltage @VBUSP	DC-to-DC LDO mode
Class A	> 3 V	doubler
Class B	< 3.9 V	doubler
	> 3.9 V	follower
Class C	> 2.7 V	follower

The DC-to-DC LDO API *phhalPmu_DcdcLdoStart()* is used to perform the above configuration, depending on the class of card being used for activation. The CT HAL internally uses this API and the application does not need to use this API directly.

The *phhalPmu_DcdcLdoStart()* also enables the voltage monitor at VBUSP pin, depending on the class of card and the mode of operation. If the VBUSP drops below the

threshold voltage, an interrupt is generated. The interrupt performs asynchronous deactivation of the CT and notifies the application.

In FW deactivates CT, the function *phhalPmu_DcdcLdoStop()* is called automatically.

Table 22. VBUSP threshold

Class of card	VBUSP monitor threshold	Typical action on reaching below threshold
class A doubler mode	3 V	SW deactivates CT
class B follower mode	3.9 V	SW deactivates CT and reactivate in doubler mode
class B doubler mode	2.7 V	SW deactivates CT
class C follower mode	2.7 V	SW deactivates CT

4.7.3 PVDD LDO HAL

The PN7462AU provides an internal LDO for generating 3.3 V pad voltage. According to the system requirement of using internal or external LDO, the HW configuration is different. Ideally, if an internal LDO is used, the ROM boot identifies the configuration and starts the PVDD LDO.

The PVDD LDO APIs are typically used during USB suspend scenario. Here, the PVDD LDO is put to low power using *phhalPmu_PvddLdoLowPower()* and reverted to full power using *phhalPmu_PvddLdoStart()* .

The PVDD LDO HAL API also provides *phhalPmu_PvddLdoStop()* even though it is **not** used.

4.7.4 Register IRQ callback

The PMU HW events are asynchronous. An application may wish to register for specific asynchronous events such as DC-to-DC Overload, TxLDO overcurrent etc. and take necessary action.

4.8 PN7462AU CLKGEN HAL

In PN7462AU, two clock frequencies are required:

1. 27.12 MHz for CLIF, CT, I^2CM, SPIM, and HSU
2. 48 MHz for USB

The system design allows two clock sources to be connected to the crystal pins of the PN7462AU. They are:

1. 27.12 MHz crystal
2. 27.12 MHz external clock source such as digital clock or on-board resonator

For the clock source of 27.12 MHz crystal, PN7462AU contains a crystal oscillator that starts automatically upon booting. Depending on the crystal, there is some delay until the oscillator is stable. If the automatic activation fails, the oscillator can be SW activated.

Application configures the above options with the *phhalClkGen_Init()* API.

For the external clock source of 27.12 MHz, ideally the crystal oscillator can be shut-down.

4.8.1 CLIF clock

The PN7462AU also contains a PLL that generates a stable 27.12 MHz clock. The PLL can be used for the following two reasons:

1. The PLL is used if the external digital/resonator clock source generates a frequency other than 27.12 MHz. It is **not** supported in this product.
2. For single-loop ALM use case, DPLL is used to lock to RF clock. It is supported in this product.

Hence for reader modes, active modes and card mode (PLM), PLL is not started and in card mode (SL-ALM), PLL is started and is locked to RF clock.

This functionality is achieved by calling *phhalClkGen_ClifClockStart()* API. The RF HAL internally uses these APIs and the application use them.

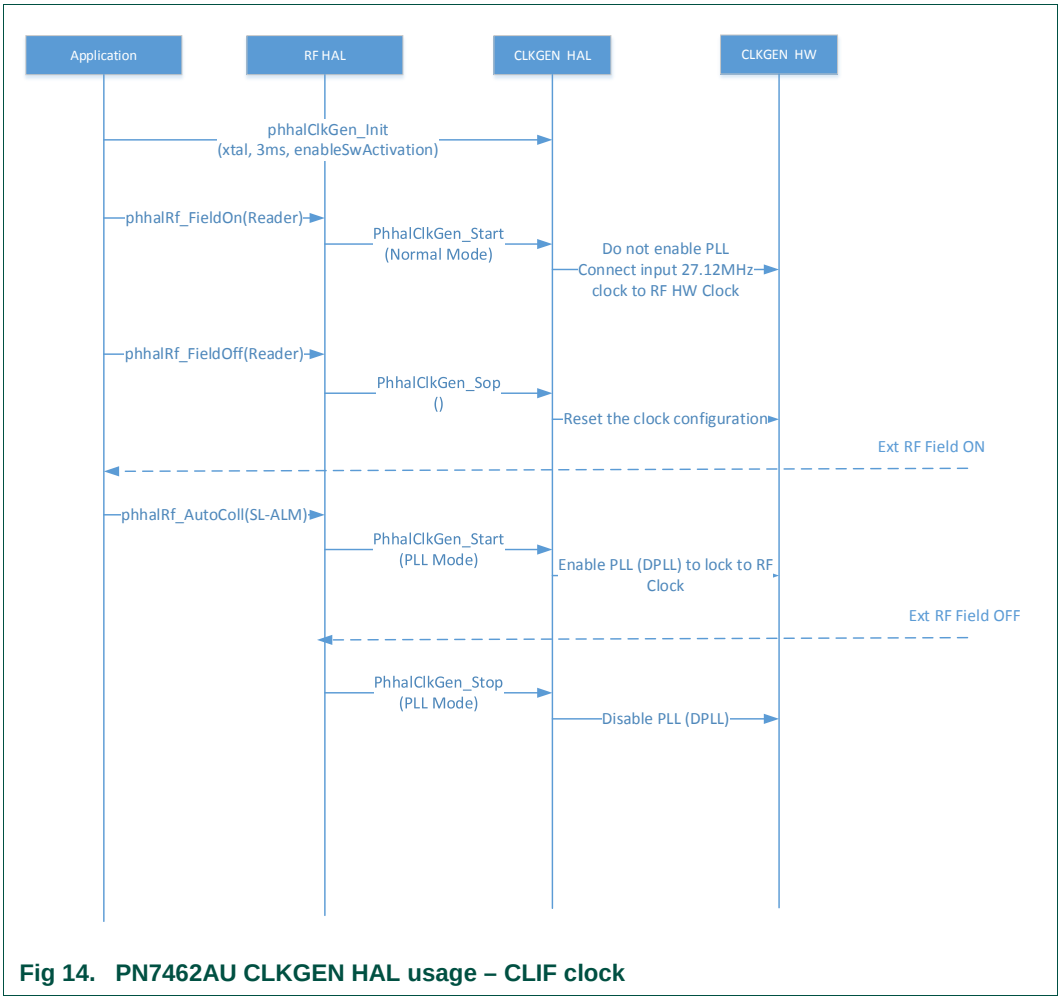

Fig 14. PN7462AU CLKGEN HAL usage – CLIF clock

4.8.2 USB clock

The USB requires a bus clock of 48 MHz. This clock is derived from the input source of 27.12 MHz, by appropriate configuration of USB PLL. During USB HAL initialization, the *phhalClkGen_UsbClockStart()* is called to lock the PLL to 48 MHz. Similarly, during USB suspend or USB HAL de-initialization, *phhalClkGen_UsbClockStop()* is called.

4.9 PN7462AU CT HAL

The CT HAL provides the following features.

Note: PN7360AU derivative does not provide CT HAL functionality.

4.9.1 Profiles

The CT protocol library can be configured with two different profiles, namely ISO7816 or EMVCo profile. The configuration can be done during the protocol initialization. The card

activation and the transactions are performed according to the selected profile. It is not recommended to switch the profile in between card activation or transaction.

4.9.2 Set Config

In this version of FW, the profile can be changed at run-time using SetConfig API that enables/disables the EMV profile.

4.9.3 Card presence check

The card presence in the slot can be checked using this feature. This feature provides the card presence or absence in the main slot.

4.9.4 Cold activation

1. Cold activation in class A or class B or class C
2. ATR reception and on-the-fly ATR parsing according to EMVCo or ISO7816

- phhalCt_CardActivate() API is used to carry out cold activation

4.9.5 Warm reset

1. Warm reset in class A or class B or class C
2. ATR reception and on the fly ATR parsing according to EMVCo or ISO7816
3. Cold activation is mandatory before warm activation

- phhalCt_WarmReset() API is used to do the warm activation

4.9.6 PPS exchange

1. If the card supports the negotiable mode, PPS exchange is used
2. Baud rates supported according to EMVCo or ISO7816

- phhalCt_PPSRequestHandling() API is used to do the PPS exchange

4.9.7 Set baud rate

1. Set baud rate API sets the baud rate for different FiDi values and calculates the different timing values (WWT, BWT and CWT)
2. If the card supports specific mode with higher baud rates, set baud rate is used

- phhalCt_SetBaudRate() API is used set the baud rate.

4.9.8 Set timer

- Different modes of the timer can be set using this API
- Different modes possible are
  - PHHAL_CT_APDUMODE_BWT: It sets the timer mode to BWT mode and sets the BWT value
  - PHHAL_CT_APDUMODE_WWT: It sets the timer mode to WWT mode and sets the WWTW value
- The timer values are set before any transaction
- phhalCt_SetTimer() API is used to set the timer values

4.9.9 Set protocol

- Set protocol sets the protocol information in the CT hardware
- phhalCt_SetTransmissionProtocol API is used for setting the protocol
- This API is mandatory before doing any transaction, which sets either the T = 0 or T = 1 protocol in CT hardware

4.9.10 Transceive

- The phhalCt_Transmit() API is used to transmit the bytes to the card
- The phhalCt_Receive() API is used to receive the bytes from the card
- The transmit and receive APIs uses the 32 bytes FIFO internally
- These APIs send and receive the raw bytes, without knowing any protocol
- The user can build T = 0 or T = 1 protocol using these APIs to perform transactions on the card
- If these APIs fail to transmit or receive the bytes from the card, appropriate error codes are returned to the user

4.9.11 Card deactivation

- The card can be deactivated using the phhalCt_CardDeactivate() API
- User can call this API once the transactions are over, which saves power to the reader
- After deactivation, activation of the card is done before carrying out any new transaction

4.9.12 Switch slot

- The CT slot can be switched from main slot to auxiliary slot and vice versa
- Auxiliary slot can be used to connect TDA ICs to PN7462AU
- phhalCt_SwitchSlot() API is used to perform the switch

Note: Connecting TDA IC and performing the transactions are not within the scope of this document.

4.9.13 Async shutdown

During emergency shutdown cases, where a system ISR has to unblock an ongoing CT transaction, this API can be used asynchronously.

4.9.14 Register IRQ callback

The CT HAL Library is a blocking HAL and IRQs and HAL interface through event mechanism. If an application has to perform some extra functionality during an IRQ, it could register a callback to that IRQ.

For example, if a CT application has to inform the host upon card removal, it could register a callback to that IRQ.

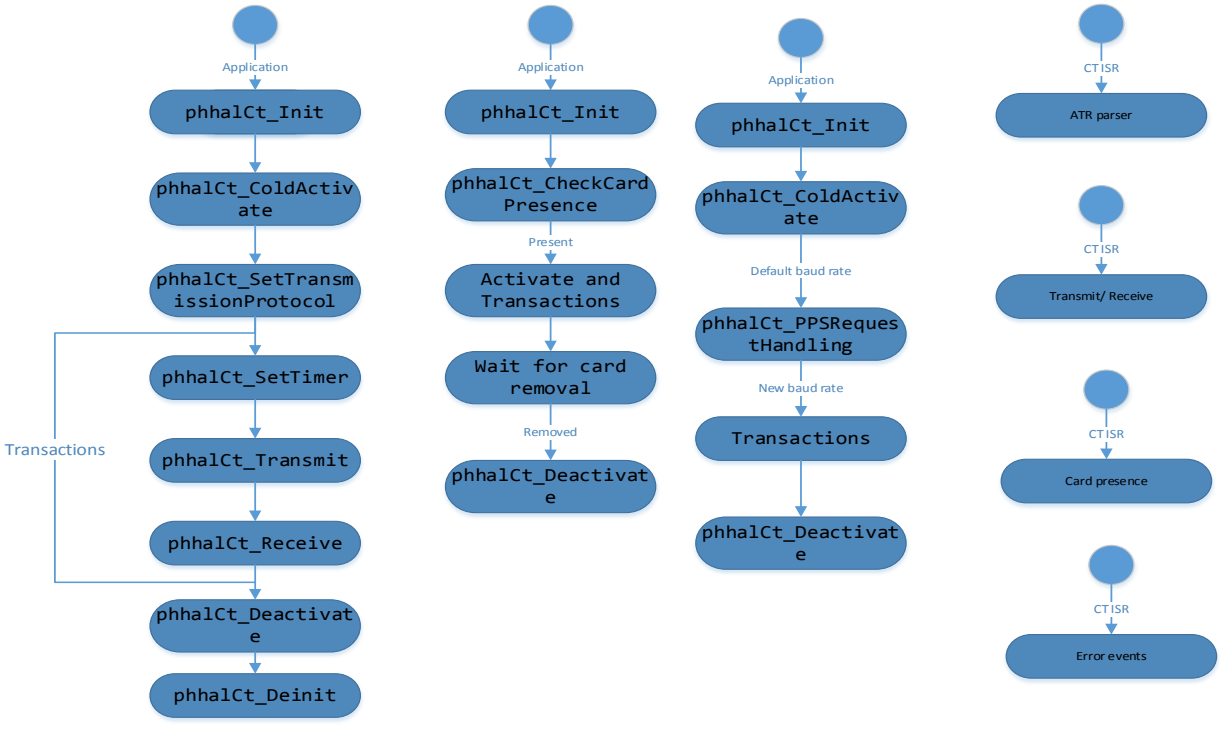

Fig 15. PN7462AU CT HAL usage

4.10 PN7462AU RF HAL

The RF HAL provides the following 5 function categories.

4.10.1 Common functions

These functions are common to both initiator and target modes.

- phhalRf_Init() is called by the application before using the RF interface. Similarly phhalRf_DeInit() is called by the application if the RF interface is no longer required.
- phhalRf_RegCallBack() is used by the application that performs application-specific function upon an interrupt arrival. The ISR shall perform HAL-specific functions and then call the application-specific function using the callback. This API does not enable any interrupt.
- phhalRf_SetMinGuardTime() is used to configure TX/RX guard time in the HW. This API should not be used when transmit/receive is in progress. This API is typically used in reader or initiator mode to change. For example, the API is used to change the TX guard time for the first frame after RATS based on start-up frame guard time.
- *phhalRf_SetIdleState()* is used to stop the communication (transmit/receive) at any time. This API can be used in a synchronous way (for example, after transmit/receive is complete) or asynchronous way (for example, when transmit/receive is in progress). For asynchronous usage, the *bIrqEnable* parameter shall be set to 1.
- *phhalRf_SetConfig()* is used to configure specific HW registers automatically at run time. The configurations are specific to the functions such as RF field, initiator modes, target modes and exchange functions.
  - *CRC* / *Parity* / *LastBitsSkip* / *FirstBitsSkip* are configurations common to both initiator and target.
- *phhalRf_GetConfig()* is provided for symmetry and is used to refer to some configuration set by the application using the *phhalRf_SetConfig()* API.

4.10.2 RF field

These functions are used to control the RF field generation.

- phhalRf_FieldOn() is used to switch on the RF field for passive initiator/reader mode, active initiator and active target mode. The HAL/HW handles initial RF collision avoidance and response RF collision avoidance (for NFC active mode). The application is not required to use this function for active target mode since it is internally used by AutoColl function of target mode. It is always recommended to peform a phhalRf_LoadProtocol_Initiator() before calling phhalRf_FieldOn() for passive reader or active initiator.
- phhalRf_FieldOff() is used to switch off the RF field for passive initiator/reader mode, active initiator and active target mode. This function can be used in a synchronous way (after transmit or receive is complete) or asynchronous way (when transmit or receive is in progress). For example, in a USB suspend ISR, phhalRf_FieldOff() can be called to switch off the field even if a TX/RX is ongoing. bIrqEnable parameter is set to 1 for asynchronous usage.

- phhalRf_FieldReset() is used to switch off and on the RF field. The function takes care of maintaining a guard time during the off period and recovery time after switching on.
- phhalRf_GetExtFieldStatus() and phhalRf_GetIntFieldStatus() are used to monitor external RF field status and internal RF field status.
- phhalRf_SetConfig()
  - EXT_FIELD_ON_IRQ is used to enable the external RF ON IRQ. This *SetConfig(),* along with *phhalRf_RegCallBack()* is used to asynchronously notify the application about external RF ON.

4.10.3 Initiator modes

These functions are used for protocol configuration for passive reader and active initiator modes, low-power card detection and MIFARE classic enabling or disabling.

- The *phhalRf_LoadProtocol_Initiator()* for various technologies and protocols is done internally in two phases through fixed settings and variable settings. The variable settings can be loaded from EEPROM/flash, described in section 4.10.6. An application has to use the TX/RX number provided in Table 23.

Table 23. Initiator modes

Protocol	TX	RX
Passive initiator NFC-A 106 kbps	2	2
Passive initiator NFC-A 212 kbps	3	3
Passive initiator NFC-A 424 kbps	4	4
Passive initiator NFC-A 848 kbps	5	5
Passive initiator NFC-B 106 kbps	7	7
Passive initiator NFC-B 212 kbps	8	8
Passive initiator NFC-B 424 kbps	9	9
Passive initiator NFC-B 848 kbps	10	10
Passive initiator NFC-F 212 kbps	12	12
Passive initiator NFC-F 424 kbps	13	13
Passive Initiator ISO15693_Tx_26_100ASK_Rx_26	15	15
Passive Initiator ISO15693_Tx_26_10ASK_Rx_53	16	16
Passive Initiator ISO18000p3m3_TARI_9_44_Rx_424_2MP	18	18
Passive Initiator ISO18000p3m3_TARI_9_44_Rx_424_4MP	18	19
Passive Initiator ISO18000p3m3_TARI_9_44_Rx_848_2MP	18	20
Passive Initiator ISO18000p3m3_TARI_9_44_Rx_848_4MP	18	21
Passive Initiator ISO18000p3m3_TARI_18_88_Rx_424_2MP	19	18
Passive Initiator ISO18000p3m3_TARI_18_88_Rx_424_4MP	19	19
Passive Initiator ISO18000p3m3_TARI_18_88_Rx_848_2MP	19	20
Passive Initiator ISO18000p3m3_TARI_18_88_Rx_848_4MP	19	21

Protocol	TX	RX
Active Initiator A 106 kbps	21	23
Active Initiator F 212 kbps	22	24
Active Initiator F 212 kbps	23	25

- The *phhalRf_LPCD()* is used to perform low-power card detection before actual polling. The application using the EEPROM parameter gpkphCfg_EE_HW_RfInitUserEE->dwLCPDDurations controls the duration of the LPCD. It is the responsibility of the application (user) to provide proper AGC reference value and threshold for card detection based on respective system design. If the customer passes 0xFF for the AGC reference value or for threshold, the HAL retrieves the value from the EEPROM.

Table 24. EEPROM parameter for RF_LPCD - RfInitUserEE

See phhalRf_InitUserEE_t

Type	Field name	Default value	Description
u32	dwAgcConfig1CMValue	0x0107FF7 (hex)	card mode AGC Config1 value
u32	dwAgcConfig0CMValue	0x44003 (hex)	card mode AGC Config0 value
u32	dwLCPDRefValue	0x000020AC (hex)	reference value of AGC for LPCD
u32	dwLCPDThreashold	0x00000005 (hex)	threshold value for LPCD
u32	dwLCPDDurations	0x00000028 (hex)	duration value for LPCD
u16	wAgcCMInputValue	0x00 (hex)	card mode most possible sensitive input value
u8	bAnaNFCLD	0x02 (hex)	NFC LD threshold value
u8	bAnaTxProt	0x09 (hex)	initial value for Ana TX Prot register

For a detailed parameter description and parameter addresses in the EEPROM refer to the EEPROM description [2] file.

- The detections reported by LPCD can be false (polling requests fail). Hence, the application has to identify false detections and use the current AGC value *(*pNewAgcValue)* as reference *(dwRef_Agc_Value)* to next LPCD cycle.
  - Typically, it is done by loading the new AGC value to GPREG and go to standby and upon wake-up, retrieve the GPREG as reference value to be provided to LPCD API

- Application shall use *phhalRf_MFC_Disable()* to disable usage of MIFARE classic crypto-HW after communication to an MFC card. The *phhalRf_MFC_Enable()* is used internally by MIFARE authenticate function and is typically **not** required for direct use from application.

- phhalRf_SetConfig()
  - JEWEL_MODE is used to configure jewel-specific transmission after identification of type 1 tag
  - RX_MULTIPLE is used to configure reception of multiple SENSF_RES for FeliCa activation. This SetConfig() is used before transmission of SENSF_REQ
  - EPC_TX_SYMBOL is used to change the SOF for EPCV2 preamble and FrameSync
  - HID configuration is used after LoadProtocol_Initiator for ISO15693 to communicate with HID cards

4.10.4 Target modes

These functions are used for protocol configuration for passive card mode and active target modes. In passive target mode, single loop active load modulation-specific configuration is also taken care of.

- The phhalRf_LoadProtocol_Target() for various technologies and protocols is done internally in two phases through fixed settings and variable settings. The variable settings can be loaded from EEPROM/flash, described in section 4.10.6. The configuration is further divided into TX and RX settings. An application shall use the TX/RX number provided in Table 25.
- The above API is used only during transitioning from base rate (106 kbps) to higher baud rate (> 212 kbps). The AutoColl function takes care of configuring the base rates for all the supported technologies.

Table 25. Target modes

Protocol	TX	RX
Passive target NFC-A 212 kbps	3	3
Passive target NFC-A 424 kbps	4	4
Passive target NFC-A 848 kbps	5	5
Passive target NFC-F 212 kbps	7	7
Passive target NFC-F 424 kbps	8	8
Active target A 106 kbps	10	10
Active target F 212 kbps	11	11
Active target F 212 kbps	12	12

- The AutoColl function performs the following functionalities:
  - Configuration of type A anti-collision HW feature: The type A anti-collision sequence until “SELECT” command is handled by the HW and the HAL function manages the HW state machine.

- Configures supported technologies such as Type A and Type F in both passive mode and/or active mode.
- The AutoColl returns back upon:

- Receiving a RATS/ATR_REQ/next frame after SENSF_REQ from reader
- External RF OFF is detected during the AutoColl function

4.10.5 Exchange functions

For passive reader mode usage, exchange functions are used to transmit a poll frame and receive a listen frame *(phhalRf_PCD_Exchg()).*

A number of specific abstractions for exchange are provided for reader mode:

- Short frame exchange
- Anti-collision frame exchange phhalRf_PCD_ExchgISO14443A_ACFrame())
- MIFARE classic 3-Way authentication exchange (*phhalRf_PCD_ExchgMFC_Auth())*

In the *phhalRf_PCD_Exchg(),* the application may or may not provide a DWORD aligned buffer. If the application provides DWORD aligned buffer, the HAL DMAs the incoming frame in the same buffer. If the application provides the DWORD an aligned buffer, the HAL DMAs the incoming frame to a local buffer and performs a memory copy from local buffer to application provided buffer. Hence it is necessary for application to provide a DWORD aligned buffer for exchange function.

The transmit function *phhalRf_Transmit()* and receive function *phhalRf_Receive()* are used in passive card mode or passive target mode due to asynchronous reception.

The *phhalRf_Receive()* function always DMAs the incoming frame to a local buffer and performs a memory copy to the application provided buffer through the receive function. It is designed in such a way because of the asynchronous nature of receive. In other words, *phhalRf_Receive()* may be called after a receive is complete.

In case of LLCP initiator, *phhalRf_Transmit() and phhalRf_Receive()* is used instead of exchange since the application starts the link loss timer after every transmit.

phhalRf_SetConfig()

1. TIMEOUT_VALUE_US, TIMEOUT_VALUE_MS is used to configure the FDT value of reader mode operation
2. NFCIP1 is used to configure the HW handling of NFCIP1 Sync byte for passive/active NFC-A 106 P2P. Application shall use this SetConfig after LoadProtocol_Initiator(active A 106) or after receiving a SAK that supports NFC-DEP protocol. In target mode, upon detecting active mode 106, the application shall use this SetConfig (NFCIP1)

4.10.6 RF register settings

During LoadProtocol, the register configuration can be divided into two parts, fixed configuration by the HAL and variable configuration by the user.

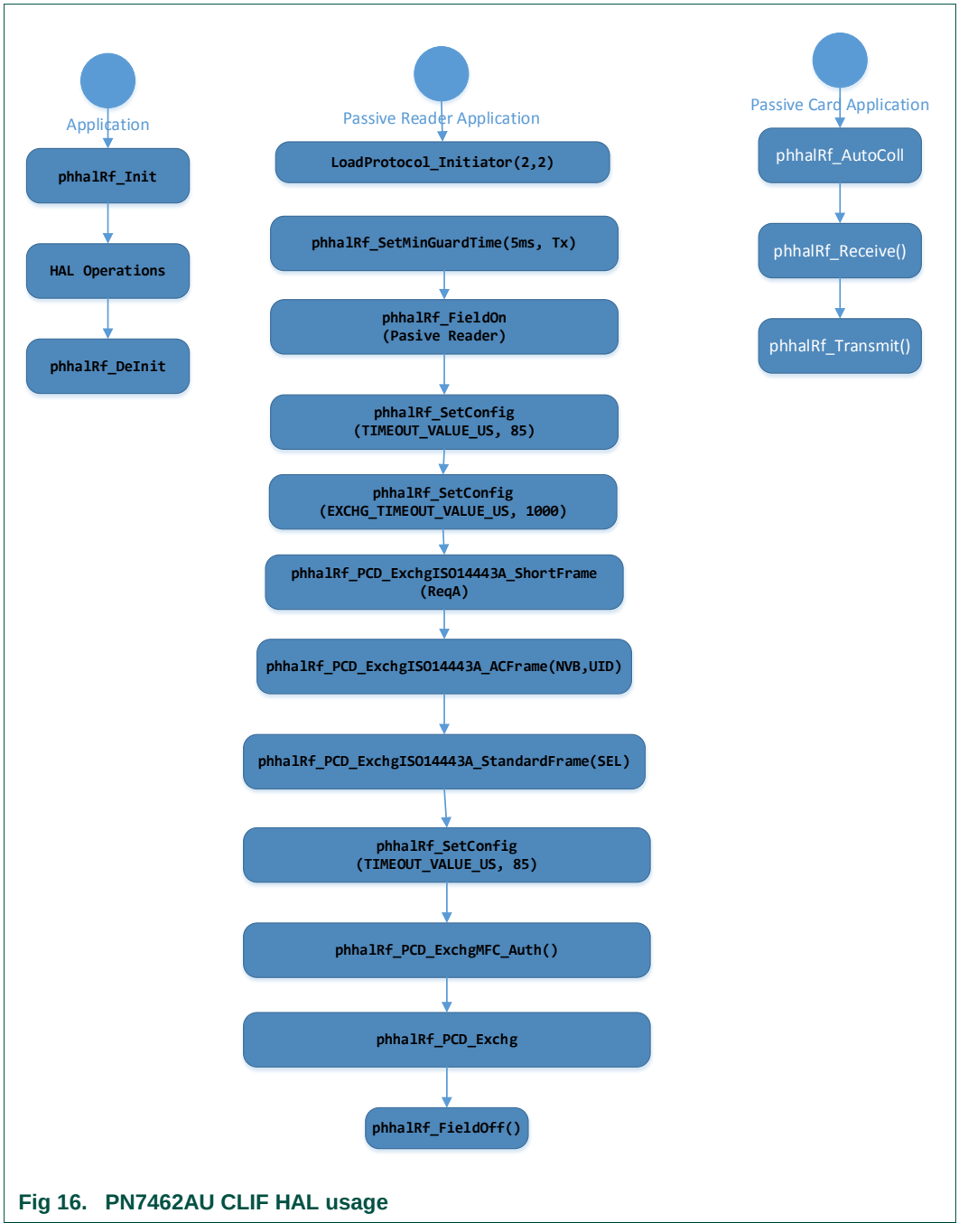

Fig 16. PN7462AU CLIF HAL usage

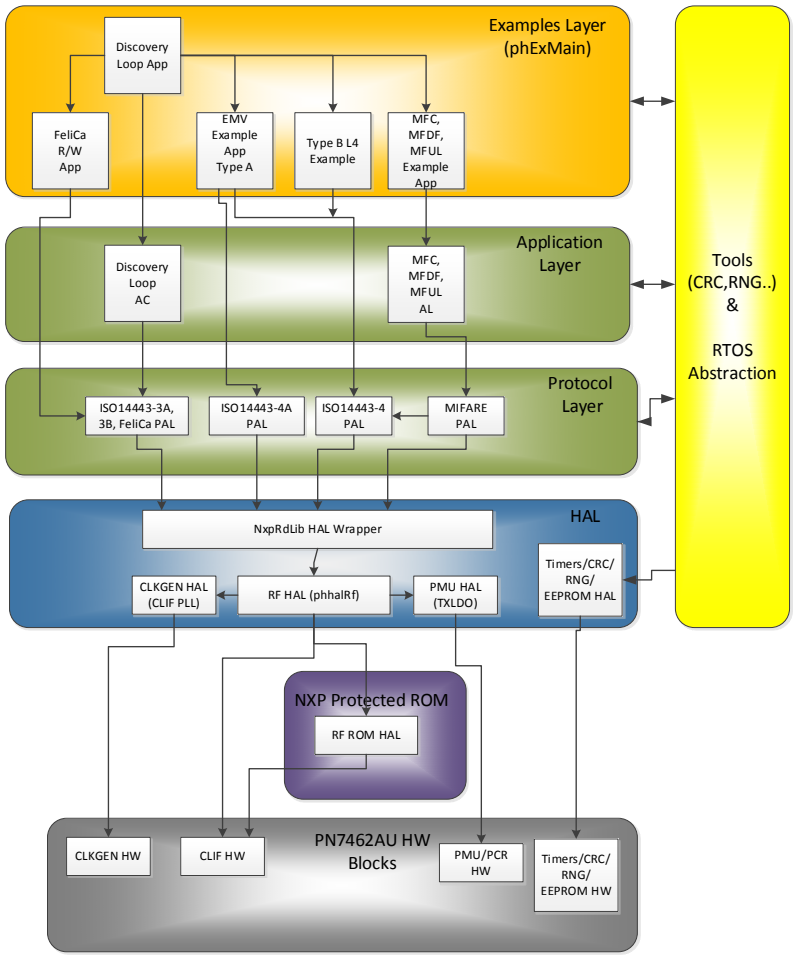

Fig 17. Contactless architecture view

4.11 PN7462AU NXP NFC contactless protocol library

Refer to the NxpRdLib CHM document for NxpRdLib.

The NxpNfcRdLib\comps\phhalHw\src\PN7462AU contains the reader library HAL API to RF HAL API wrapper.

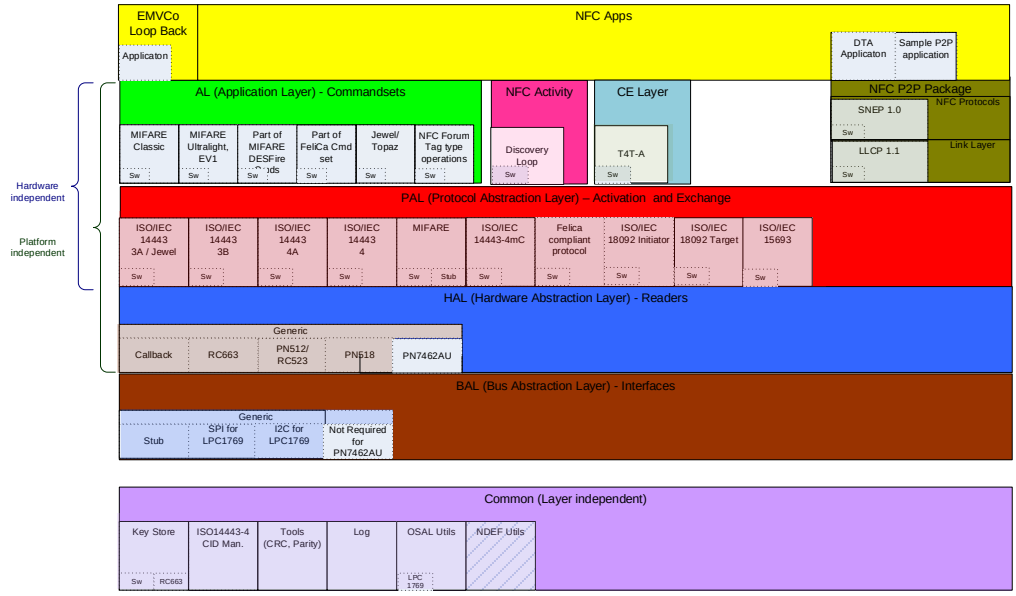

Fig 18. NxpNfcRdLib

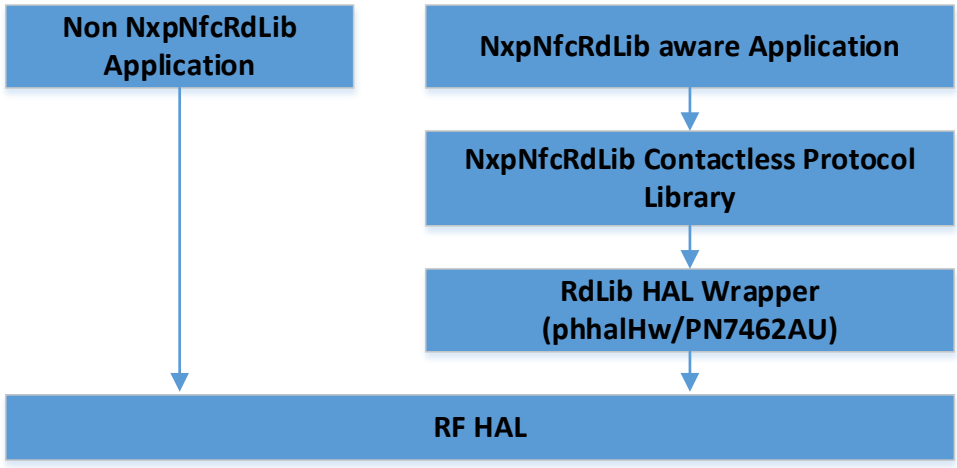

Fig 19. NxpNfcRdLib HAL wrapper

4.12 PN7462AU NXP CT protocol library

The CT protocol library uses the CT HAL APIs.

4.12.1 Profiles

The CT Protocol library can be configured with two different profiles namely ISO7816 or EMVCo profile. The configuration can be done during the protocol initialization. The card activation and the transactions are performed according to the selected profile. It is recommended **not** to switch the profile in between card activation or transaction.

4.12.2 Activation loop

The contact card activation loop is performed according to the ISO7816 or EMVCo specification as per the selected profile. The activation of any card always starts with class A, followed by class B if class A is **not** successful and then class C if class B is **not** successful. The ATR bytes are returned to the user after successful activation.

4.12.3 Protocol selection

User can select the protocol T = 0 or T = 1 according to the capability of the card. It is an optional feature. If the user does not select any protocol, the first protocol offered from the card is applied by default.

4.12.4 Transceive

The transceive feature is used to perform the transaction on the contact card. The transceive can be either on T = 0 or T = 1 protocol.

4.12.4.1 T = 0 protocol

User can carry out the card transaction using the T = 0 protocol, when the card supports the T = 0 protocol. The T = 0 protocol supports the following APDU types:

- Case 1 APDU
- Case 2 APDU
- Case 3 APDU
- Case 4 APDU
- Get response command
- Handling of wait extension bytes

Extended APDU is **not** supported in case of T=0 protocol.

4.12.4.2 T = 1 protocol

User can carry out the card transaction using the T = 1 protocol, when the card supports the T = 1 protocol. The T = 1 protocol supports the following features:

- IFSD: Interface device maximum packet size of 255 bytes
- Handling of I block, S block and R block
- Chaining from interface device and handling of chaining from card
- Handling of all error scenarios (mentioned in the Appendix A of ISO7816-3)
- Handling of wait extension block, abort block and resynchronization block

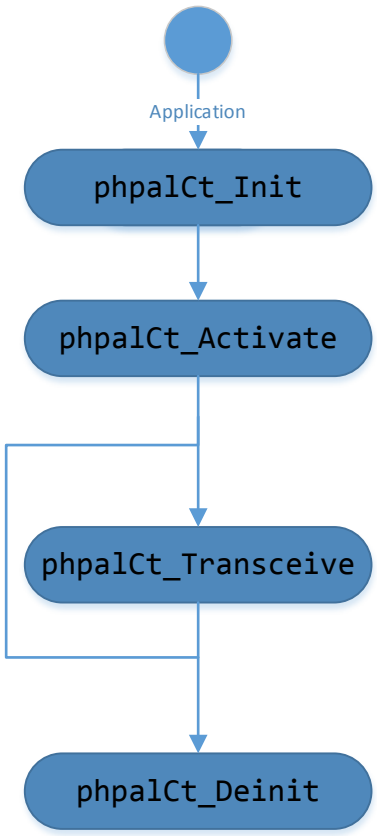

Fig 20. PN7462AU CT PAL usage

5. USB device stack architecture

The following figure shows the architectural block diagram of the USB device stack. The stack consists of three main layers:

- The Class layer
- The USB core layer
- The Device Controller Driver (DCD) layer

The bottom-most layer is Device Controller Driver (DCD) layer which is also referred as hardware layer in this document. The layer above is the USB core layer, which handles the USB protocol-specific code. Above that is the class layer which contains the class Function Drivers (FD). Finally, the application sits at the top of the stack.

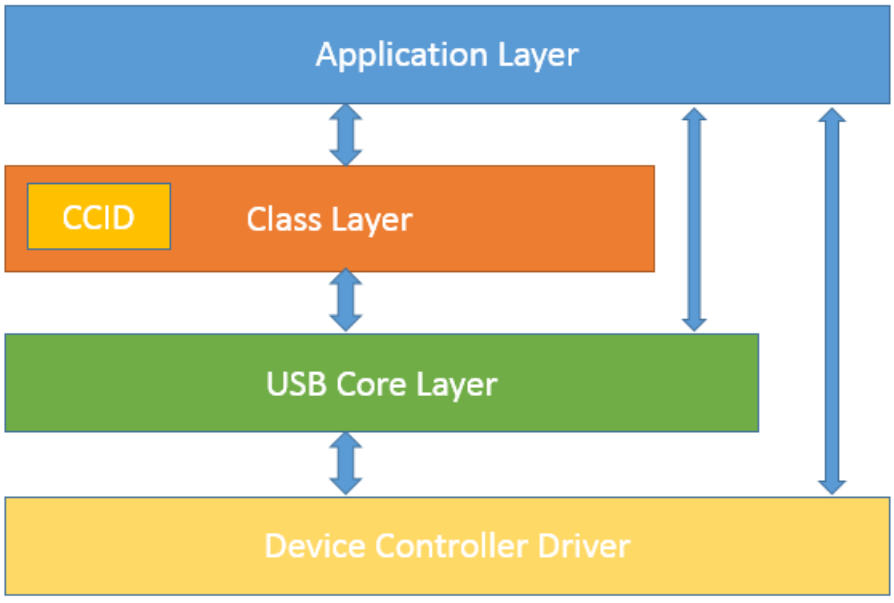

Fig 21. USB device stack architecture

Each layer has one or more components which are accessible to user application. Each component has two main structures *USBD_xxx_API_T* and *USBD_xxx_INIT_PARAM_T*. The *USBD_xxx_API_T* structure contains all the user callable routines to use the component. The *USBD_xxx_INIT_PARAM_T* structure is used by application to specify various initialization parameters along with pointers to call back routines. These call-back routines can be categorized as event callbacks and stack overrides

- Event callbacks: They are called by the stack in interrupt context when an appropriate event occurs.

- Stack overrides: If the default behavior/handling of the stack is not appropriate for the user application, these routines are defined to allow customization (override the default behavior) of the stack by user.

5.1 Developing with USB device stack

This section introduces you to application development using USB device stack. The topics in this section describe high-level USB concepts and provide step-by-step instructions on implementing a USB device using USB Device stack. For detailed information on USB concepts, see USB specifications at usb.org.

Prerequisites

- Step1: Define USB descriptors
- Step2: Initializing the USBD stack
- Step3: Connecting USB IRQ handler
- Step4: Initialize and attach class drivers
- Step5: Connect the device

5.1.1 Prerequisites

- Processor initialization
- Clocks initialization (oscillator, system PLL and USB PLL)
- Initialize other necessary configuration related for the board

5.1.2 Step 1: Define USB descriptors

A USB device provides information about itself in data structures called USB descriptors to a host during enumeration process. Hence user application should create these descriptors as per the application and provide it to the USB device stack. So that the stack takes care of responding to the standard USB requests generated by the host.

5.1.3 Step 2: Initializing the USBD stack

To initialize the USBD stack, configure the initialization parameters.

Create an instance of USBD_API_INIT_PARAM_T as a local variable.

USBD_API_INIT_PARAM_T usb_param;

Define the memory area USBD stack is be used for its global variables. The application has to provide memory area in which the stack can allocate its global variables and also the device controller driver buffers. This memory address should be accessible by USB bus master.

usb_param.mem_base = (uint32_t) gphExCcid_Usb_CORE_Buffer

usb_param.mem_size = USBHAL_CORE_MEM_SIZE

If the application is using dynamic memory allocator or wants to know exactly how much memory stack is required. It can call *USBD_HW_API_T::GetMemSize* routine to know the size of memory buffer stack needs. Set the USB register base address location and number of endpoints used by the application to optimize driver buffer space.

Setting_max_num_ep to more than the maximum number of endpoints available in hardware causes program crashes.

usb_param.usb_reg_base = PN7462AU_USB_BASE

usb_param.max_num_ep = USB_MAX_EP_NUM

Provide event callback routines for event in which application is interested in.

usb_param.USB_Configure_Event = USB_Configure_Event

usb_param.USB_Reset_Event = USB_Reset_Event

Define USB descriptors and initialize the descriptor parameter.

USB_CORE_DESCS_T DeviceDes;

DeviceDes.device_desc = (uint8_t)&gphExCcid_DeviceDescriptor*

DeviceDes.high_speed_desc = (uint8_t)&gphExCcid_FSConfigurationDescriptor*

DeviceDes.full_speed_desc = (uint8_t)&gphExCcid_FSConfigurationDescriptor*

DeviceDes.string_desc = (uint8_t)gphExCcid_FSStringDescriptor*

DeviceDes.device_qualifier = (uint8_t)0*

To initialize the stack, call Init() routine is used. If the routine returns anything other than LPC_OK, the parameters are **not** configured properly. If the initialization call is successful, the handle to the instance of USBD stack is returned in UsbHandle parameter. Application should store this handle in a global scope variable to access stack functions from USBD and IRQ contexts.

```
ret = hwUSB_Init(&UsbHandle, &DeviceDes, &usb_param)
/* Failed initialization */
while(ret != LPC_OK)
```

5.1.4 Step3: Connecting USB IRQ handler

USBD stack implements the interrupt handler for USB device controller and invokes the USBD stack routines according to the event type. Hence application should connect this handler to the application vector table. It is done by calling *USBD_HW_API_T::ISR* from the USB IRQ handler as shown in the code snippet below. The USBD handle passed to the ISR() routine is the one obtained in previous step.

```
void HIF_IRQHandler(void)
{
        hwUSB_ISR(UsbHandle);
}
```

5.1.5 Step4: Initialize and attach class drivers

In the following example, CCID class driver is attached to the only interface (interface 0, alt 0) present in the device configuration.

Configure initialization parameters of the CCID class driver.

USBD_CCID_INIT_PARAM_T ccid_param

Set the memory location where the CCID class driver can allocate buffers and global variables.

All the *init()* routines are written in such a way that the mem_base and mem_size members of the XXX_INIT_PARAM_T are updated with free location and size before returning. So that the next component XXX_INIT_PARAM_T can use the update values for its *init()* routine. Applications can cascade the component initialization this way without worrying about memory wastage/overlap issues.

```
.mem_base     = (uint32_t)gphExCcid_Usb_CCID_buffer
.mem_size     = USBHAL_CCID_MEM_SIZE
```

Now set the CCID specific parameters.

```
.CCID_EpBulkOut_Hdlr    = &CCID_Bulk_Out_hdlr
.CCID_EpBulkIn_Hdlr     = &CCID_Bulk_In_hdlr
.CCID_EpIntIn_Hdlr      = &CCID_Interrupt_In_hdlr
```

Now call the mwCCID_init routine of the CCID class.

5.1.6 Step5: Connect the device

Once the core, device controller and class drivers are initialized the stack is ready to receive the packets from host. Even if the device is physically connected to the host using a USB cable, the host does not recognize the presence of device until the D+ line

is pulled-up using 1.5 kΩ resistor. To enable the connection, the application software should call *hwUSB_Connect()* with enable set to 1. Before enabling the connection, the application should enable the USB interrupt.

```
void phExCcid_Usbd_Connect_Enable(void)
{
    hwUSB_Connect(UsbHandle, 1);
}
```

5.1.7 Defining USB descriptors

A USB device provides information about itself in data structures called USB descriptors. This section provides information about various descriptors that a USB device should provide to host during enumeration process.

The host obtains descriptors from an attached device by sending various standard control requests (GET_DESCRIPTOR requests) to the default endpoint. Those requests specify the type of descriptor to retrieve. In response to such requests, the device sends descriptors that include information about the device, its configurations, interfaces and the related endpoints. Device descriptors contain information about the whole device. Configuration descriptors contain information about each device configuration. String descriptors contain Unicode text strings. The USB device stack handles all the standard requests, eliminating the complexity from user application, as long as the user application provides the proper descriptor arrays to stack initialization routine.

Every USB device exposes a device descriptor that indicates the class information of the device, vendor and product identifiers, and number of configurations. Each configuration exposes its configuration descriptor that indicates number of interfaces and power characteristics. Each interface exposes an interface descriptor for each of its alternate settings that contain information about the class and the number of endpoints. Each endpoint within each interface exposes endpoint descriptors that indicate the endpoint type and the maximum packet size. User application should provide these descriptors to the USB device stack for proper handling of standard requests.

5.1.8 Defining USB device descriptor

The device descriptor contains information about a USB device as a whole. The application, through device_desc field of USB_CORE_DESCS_T structure, provides the pointer to memory containing device descriptor. The address is passed to stack initialization routine *hwUSB_Init*(). The user application can create an instance of _USB_DEVICE_DESCRIPTOR structure in global scope (defined in memory accessible by USB bus master) and pass the address of the instance in device_desc field. Or the application can define the descriptor as character array as shown below:

```
USB_DEVICE_DESCRIPTOR  gphExCcid_DeviceDescriptor =
{
   .bLength          = sizeof(USB_DEVICE_DESCRIPTOR),
```

```
    .bDescriptorType  = 0x1,     /* USB_DEVICE_DESCRIPTOR_TYPE       */
    .bcdUSB           = 0x0200,  /* USB Specification Release Number */
    .bDeviceClass     = 0x00,    /* Device Class                     */
    .bDeviceSubClass  = 0x00,    /* Device SubClass                  */
    .bDeviceProtocol  = 0x00,    /* Device Protocol                  */
    .bMaxPacketSize0  = 0x40,    /* Endpoint 0 Max Packet Size       */
    .idVendor         = 0x1FC9,  /* Vendor ID                        */
    .idProduct        = 0x0117,  /* Product ID                       */
    .bcdDevice        = 0x0101,  /* Device Release Number            */
    .iManufacturer    = 0x01,    /* Index of String Descriptor- Manufacturer  */
    .iProduct         = 0x02,    /* Index of String Descriptor - Product      */
    .iSerialNumber    = 0x03,    /* Index of String Descriptor - Serial No    */
    .bNumConfigurations = 0x01     /* Number of Configurations               */
};
```

- bLength: Size of this descriptor in bytes. Always set this field to 18.
- bDescriptorType: DEVICE descriptor type. Always set this field to 0x01.
- bcdUSB: Set to USB specification version the device and its descriptors are compliant. Setting the value to 0x0200 implies the device is compliant with USB specification version 2.00.
- bDeviceClass: Class code. If this field is reset to zero, each interface within a configuration specifies its own class information and the various interfaces operate independently. If this field is set to a value between 1 and FEH, the device supports different class specifications on different interfaces and the interfaces may not operate independently. This value identifies the class definition used for the aggregate interfaces. If this field is set to FFH, the device class is vendor-specific.
- bDeviceSubClass: Set the subclass code of the device as assigned by the USB specification group.
- bDeviceProtocol: Set the protocol code of the device as assigned by the USB specification group.
- bMaxPacketSize: Maximum packet size, in bytes, for endpoint zero of the device. Current USB stack implementation assumes 64 bytes, hence set this field to 64.
- idVendor: Set the USB Vendor ID assigned to the manufacturer of this product, by the USB-IF.
- idProduct: Set the product ID assigned by the manufacturer for this product.
- bcdDevice: Device release number in binary-coded decimal.
- iManufacturer: Set index of the string descriptor that provides a string containing the name of the manufacturer of this device. Check defining USB string descriptor for more details in defining string descriptors.
- iProduct: Set index of the string descriptor that provides a string that contains a description of the device. Check defining USB string descriptor for more details in defining string descriptors.
- iSerialNumber: Set index of the string descriptor that provides a string that contains a manufacturer-determined serial number for the device. Check defining USB string descriptor for more details in defining string descriptors.

- bNumConfigurations: Set the total number of possible configurations for the device.

5.1.9 Defining USB device qualifier descriptor

The device_qualifier descriptor describes information about a high-speed capable device that would change if the device were operating at the other speed (see _USB_DEVICE_QUALIFIER_DESCRIPTOR structure). For example, if the device is operating at full-speed, the device_qualifier returns information about how it would operate at high-speed and vice-versa. The pointer to the memory containing device qualifier descriptor is provided by application through device_qualifier field of USB_CORE_DESCS_T structure, whose address is passed to stack initialization routine *hwUSB_Init*().

Note: If application is implementing full-speed only device, this field should be set to 0. And also the field high_speed_desc and full_speed_desc should point to full-speed configuration descriptor array.

5.1.10 Defining USB configuration descriptors array

A USB device exposes its capabilities in the form of a series of interfaces called a USB configuration. A USB configuration is described in a configuration descriptor (see _USB_CONFIGURATION_DESCRIPTOR structure). A configuration descriptor contains information about the configuration and its interfaces, alternate settings, and their endpoints. Each interface descriptor or alternate setting is described in a _USB_INTERFACE_DESCRIPTOR structure. In a configuration, each interface descriptor is followed in memory by all of the endpoint descriptors for the interface and alternate setting. Each endpoint descriptor is stored in a _USB_ENDPOINT_DESCRIPTOR structure.

The USB stack assumes that all the descriptors associated with the configuration are arranged as a consecutive byte array in memory. The address to this array is passed to the stack through full_speed_desc and high_speed_desc field of USB_CORE_DESCS_T structure, whose address is passed to *hwUSB_Init*() routine. The following diagram illustrates how configuration information should be laid out in memory.

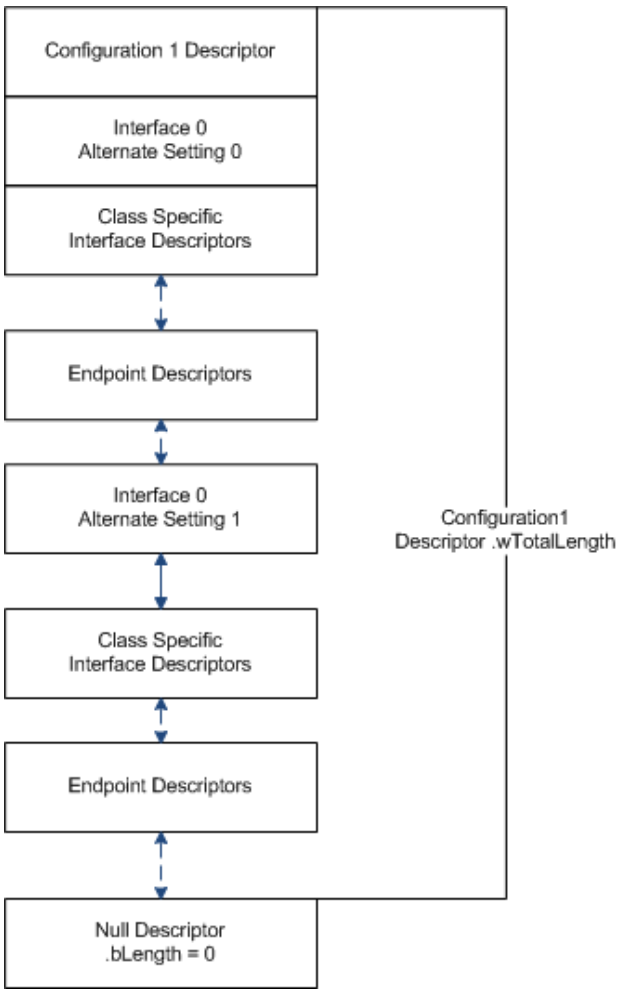

Fig 22. PN7462AU USB configuration descriptors

The following example shows the configuration descriptor for the USB CCID class device:

```
phExCcid_Descriptors_USB_Descriptor_Configuration_t gphExCcid_FSConfigurationDescriptor =
{
    /* Configuration Descriptor */
    .Config =
    {
        .bLength             = sizeof(USB_CONFIGURATION_DESCRIPTOR),                    /* Length of Descriptor     */
        .bDescriptorType     = 0x2,                                                      /* Descriptor Type          */
        .wTotalLength        = sizeof(phExCcid_Descriptors_USB_Descriptor_Configuration_t) - 1,/* Total Length           */
        .bNumInterfaces       = 1,                                                      /* Number of Interfaces     */
        .bConfigurationValue = 1,                                                       /* Configuration Value      */
        .iConfiguration      = 0,                                          /* Index of String Descriptor - Configuration */
#if (PH_EXCCID_USB_IF_USB_REMOTE_WAKEUP_FTR == 1)
        .bmAttributes         = 0xA0,                                     /* Configuration Characteristics              */
#else
        .bmAttributes         = 0x80,                                     /* Configuration Characteristics              */
#endif
        .bMaxPower            = 0x7D                                      /* Maximum Power Consumption                  */
    },
    /* Interface Descriptor */
    .Ccid_Interface =
```

```
    {
        .bLength             = sizeof(USB_INTERFACE_DESCRIPTOR),        /* Length of Descriptor      */
        .bDescriptorType     = 0x04,                                    /* Interface Descriptor Type */
        .bInterfaceNumber    = 0x00,                                    /* Interface Number          */
        .bAlternateSetting   = 0x00,                                    /* Alternate Settings        */
        .bNumEndpoints       = 0x03,                                    /* Total Number of Endpoints */
        .bInterfaceClass     = 0x0B,                                    /* Smart Card Class          */
        .bInterfaceSubClass  = 0x00,                                    /* Interface Sub Class       */
        .bInterfaceProtocol  = 0x00,                                    /* Interface Protocol        */
        .iInterface          = 0x00                                     /* Index of String Descriptor - Interface */
    },
    .Ccid_Descriptor =
    {
        .bLength                = sizeof(USB_SMARTCARD_DESCRIPTOR),     /* Length of Descriptor      */
        .bDescriptorType        = 0x21,                                 /* CCID Descriptor Type      */
        .bcdCCID                = 0x0110,                               /* CCID Specification Number */
        .bMaxSlotIndex          = 0,                                    /* Maximum Slot Index        */
        .bVoltageSupport        = 0x7,                                  /* Voltage Support           */
        .dwProtocols            = 0x03,                                 /* Protocols Support         */
        .dwDefaultClock         = 0xE65,                                /* Default Clock             */
        .dwMaximumClock         = 0x37F0,                               /* Maximum Clock             */
        .bNumClockSupported     = 0x00,                                 /* Num of Clock Supported    */
        .dwDataRate             = 0x26B5,                               /* Data Rate                 */
        .dwMaxDataRate          = 0xCF080,                              /* Maximum Data Rate         */
        .bNumDataRatesSupported = 0x00,                                 /* Number of Data Rates Sup  */
        .dwMaxIFSD              = 0xFE,                                 /* Maximum IFSD              */
        .dwSynchProtocols       = 0x0,                                  /* Synch Protocols           */
        .dwMechanical           = 0x0,                                  /* Mechanical Features       */
        .dwFeatures             = 0x204BE,                              /* Features Supported        */
        .dwMaxCCIDMessageLength = 0x10F,                                /* CCID Msg Length           */
        .bClassGetResponse      = 0x00,                                 /* Get Response              */
        .bClassEnvelope         = 0x00,                                 /* Class Envelope            */
        .wLcdLayout             = 0x0,                                  /* LCD Support               */
        .bPinSupport            = 0,                                    /* PIN Support               */
        .bMaxCCIDBusySlots      = 1                                     /* Busy CCID Slots           */
    },
    /* Bulk IN End Point Descriptor */
    .Ccid_DataInEndpoint =
    {
        .bLength             = sizeof(USB_ENDPOINT_DESCRIPTOR),         /* Length of the Descriptor    */
        .bDescriptorType     = 0x05,                                    /* Endpoint Descriptor Type    */
        .bEndpointAddress    = 0x81,                                    /* Bulk IN Endpoint Address    */
        .bmAttributes        = 0x02,                                    /* Attributes                  */
        .wMaxPacketSize      = 0x40,                                    /* Maximum Packet Size         */
        .bInterval           = 0x00                                     /* Interval Period             */
    },
    /* Bulk OUT End Point Descriptor */
    .Ccid_DataOutEndpoint =
    {
        .bLength             = sizeof(USB_ENDPOINT_DESCRIPTOR),         /* Length of the Descriptor    */
        .bDescriptorType     = 0x05,                                    /* Endpoint Descriptor Type    */
        .bEndpointAddress    = 0x01,                                    /* Bulk Out Endpoint Address   */
        .bmAttributes        = 0x02,                                    /* Attributes                  */
        .wMaxPacketSize      = 0x40,                                    /* Maximum Packet Size         */
        .bInterval           = 0x00                                     /* Interval Period             */
    },
    .Ccid_InterruptInEndpoint =
    {
        .bLength             = sizeof(USB_ENDPOINT_DESCRIPTOR),         /* Length of Descriptor       */
        .bDescriptorType     = 0x05,                                    /* Endpoint Descriptor Type   */
        .bEndpointAddress    = 0x82,                                    /* Interrupt Endpoint Address */
        .bmAttributes        = 0x03,                                    /* Attributes                 */
        .wMaxPacketSize      = 0x40,                                    /* Maximum Packet Size        */
        .bInterval           = 0x04                                       /* Interval Period            */
     },
    .Ccid_Termination = 0x00

};
```

Note: For devices implementing multiple configurations, the second configuration descriptors should follow immediately after the first configuration descriptors with NULL descriptor at the end of the second configuration array.

5.1.11 Defining USB string descriptor

Device, configuration, and interface descriptors may contain references to string descriptors. String descriptors are referenced by their one-based index number. A string descriptor contains one or more Unicode strings; each string is a translation of the others into another language.

A string descriptor contains:

- bLength: Size of this descriptor in bytes. This field is at offset 0 and occupies 1 byte. The size of the descriptor is size of Unicode string + 2.
- bDescriptorType: STRING descriptor Type. Always set this field to 0x03. This field is at offset 1 and occupies 1 byte.
- bString: UNICODE encoded string.

String index zero for all languages should return a string descriptor that contains an array of two-byte LANGID codes supported by the device. Current implementation of USB device stack assumes single language support. For applications using US English strings this descriptor should be:

```
0x04,               /* bLength */
0x03,               /* bDescriptorType */
WBVAL (0x0409),     /* wLANGID: US English */
```

The USB Device stack assumes that all USB strings referenced in various descriptors are provided to stack as a single character array containing multiple string descriptors. USB Device stack traverses to the next descriptor in array by adding the value of bLength field to current index. Hence it is important to construct this descriptor array properly with bLength fields reflecting the exact size of its string descriptor.

```
uint8_t gphExCcid_FSStringDescriptor[] =
{
    /* Index 0x00: LANGID Codes */
    0x04,
    0x03,
    WBVAL (LANGUAGE_ID_ENG),

    /* Index 0x01: Manufacturer */
    0x08,
    0x03,
    'N',0,'X',0,'P',0,

    /* Index 0x02: Product */
    0x1C,
    0x03,
    'P',0,'N',0,'7',0,'4',0,'6',0,'2',0,'A',0,'U',0,'
',0,'C',0,'C',0,'I',0,'D',0,

    0x0A,
```

```
    0x03,
    '1',0,'.',0,'0',0,'0',0
};
```

Reference:

Refer the following files in the PN7462AU_ex_phExCcid example:

1. phExCcid_UsbUser.c
2. phExCcid_Descriptors.c
3. phExCcid_UsbCcid.c (Callback functions for CCID class)
4. phExCcid_Usb_If.c (PN7462AU system specific initialization)

5.2 Porting existing LPC USB Virtual Keyboard implementation to PN7462AU

In order to implement USB HID device class, there is also possibility to port existing LPCXpresso USB Virtual keyboard example implementation to the PN7462AU. Suitable USB HID device implementation is Virtual keyboard HID example for LPC11u14 MCU *nxp_lpcxpresso_11u14_usbd_lib_hid_keyboard* example project located in *"LPCXpresso_install_dir\lpcxpresso\Examples\LPCOpen\ lpcopen_v2_00a_lpcxpresso_nxp_lpcxpresso_11u14.zip"* project archive.

Refer to PN7462AU_ex_phExVCom example in PSP to create needed project structure and to adjust PN7462AU_USB_BASE address.

Specific HID USB class handler is implemented in USBD ROMSTACK and it is available upon request from NXP.

6. PN7462AU PSP examples

The PN7462AU PSP provides seventeen examples in export controlled version, available through NXP DocStore, and fifteen examples in full version available through the product page, to demonstrate various features of HALs, NxpNfcReaderLibrary and contact protocol library. The structure of all the PSP examples are as follows:

- The *src* directory contains "*pspexample.c*" which contains the *main()* which is the entry point.
  - For example, *phExMain* example *src* directory contains *phExMain.c* and *phExNFCForum* example *src* directory contains *phExNFCForum.c.*
- The individual features within the example are present in independent C source files.
- The *inc* directory contains *APP_NxpBuild.h* which contains all the build configuration required for example such as NxpNfcReaderLibrary components, HAL components and feature diversity such as standby enable/disable, RTOS/No-RTOS, etc.
- The logging control for the examples is provided in *phCommon/inc/ph_Log.h.*

The logging is disabled if the project is built in release mode or if standby is enabled.

6.1 PN7462AU_ex_phExMain

The phExMain example is provided to demonstrate the following features of PN7462AU FW. This example is a standalone example executing inside PN7462AU without any host interface.

6.1.1 NFC forum + discovery loop

This example polls the technologies listed below. Before polling, the discovery loop can perform low-power card detection (depending on run-time configuration).

- Active NFC-A 106
- Active NFC-F 212
- Active NFC-F 424
- Passive NFC-A 106
- Passive NFC-B 106
- Passive NFC-F 212
- Passive NFC-F 424
- Passive ISO15693
- Passive ISO18000p3m3
- Listen for passive & active NFC-A and NFC-F from peer

The listen duration is taken from EEPROM location:

wWakeUpTimerVal @ PH_CFG_EE_WAKEUPCONFIG_START_ADDRESS (0x2012c0).

6.1.2 Standby

During the listen duration, the PN7462AU can be in either of the following modes:

- Active power mode waiting for external RF ON for the duration or CT presence interrupt: In this mode, a GP Timer is used for the listen duration.
- Standby power mode waiting for external RF ON for the duration or CT presence wake-up event: In this mode, PCR wake-up counter is used for the listen duration.

The build macro *PHFL_ENABLE_STANDBY* controls this behavior.

6.1.3 MIFARE classic reader

If a type A card is detected with SAK of 0x8 (1 K MFC card) or 0x18 (4 K MFC card), then the MF Classic example is executed. The MF classic example assumes an MF classic card with a predefined key for the sectors to be accessed. The example performs initial authentication of block X and then reads/write to this sector. Further, the example performs authentication of another block (say X +1) using the session key established during initial authentication (it is called reauthentication). It then performs read/write to

this block (X+1). See the function *phExMain_MiFareClassic()* and *phExMain_MifareOperations().*

Supported Functionalities:

- Authentication & Re-Authentication
- Read/Write block

Implementation in the *"phExMain_MiFareClassic.c"* file. The default key 0xFFFFFFFFFFFF is used for both keys Key A and Key B.

6.1.4 MIFARE Ultralight reader

If a type A card is detected with SAK 0x00, then MF ultralight example is executed.

The MF ultralight example assumes a non-secure MF ultralight card. The example performs read and write to predefined pages of the card.

Supported functionalities:

- Read
- Write
- Since the stack also supports Type 2 tag, a check is performed to see if the card is NDEF tag or ultralight tag.

Implementation in the *"phExMain_MiFareUltralight.c"* file.

6.1.5 Jewel reader

If the ATQA of a Type A card denotes jewel card, the jewel example is executed.

The jewel example assumes a non-secure jewel card. The example performs read and write to predefined blocks of the card.

Supported functionalities:

- Read
- Write
- Since the stack also supports Type 1 tag, a check is performed to see if the card is an NDEF tag or jewel card.

Implementation in the *"phExMain_Jewel.c"* file.

6.1.6 ISO14443-4 Type A reader (MFDF card)

If the detected SAK is 0x20 (expected card is MFDF EV1), then ISO14443-4 Type A reader example is executed. This example performs L4 exchange of "GetVersion" command at 106, 212, 424, and 848 kbps. No other commands are currently exchanged as they require authentication and crypto-operations (which are currently **not** supported in the release).

Supported functionalities:

- Get version command
- 106/212/424/848 kbps
- No encryption

Implementation in the *"phExMain_TypeA_L4Exchange.c"* file.

6.1.7 ISO14443-4 Type B reader (EzLink/SLE Card)

If the detected technology is Type B, then ISO14443-4 Type B reader is executed. This example performs L4 exchange of "get challenge" command at 106, 212, 424, and 848 kbps. No other commands are currently exchanged as they require authentication and crypto-operations (which are **not** currently supported in the release).

Supported functionalities:

- Get challenge command
- 106/212/424/848 kbps
- No encryption

Implementation in the *"phExMain_TypeB_L4Exchange.c"* file.

6.1.8 FeliCa reader

If the detected technology is NFC –F, FeliCa reader example is executed. This example is used to read and write FeliCa frames to FeliCa card at 212/424 kbps. Since the stack supports Type 3 tags, check is performed to see if the card is an NDEF tag or FeliCa card.

Supported functionalities:

- CHECK
- UPDATE
- 212/424 kbps

Implementation in the *"phExMain_FeliCa.c" file.*

6.1.9 ISO15693 reader (ICODE SLIX card)

If the detected technology is ISO15693, then this example performs read and write to predefined blocks of the card.

Supported functionalities:

- Read single block
- Write single block
- 26 kb/s TX (1 out of 4 coding) and 26 kb/s RX

Implementation in the *"phExMain_ISO15693.c"* file.

6.1.10 EPCV2 (ISO18000p3m3) reader – (ICODE ILT card)

The example performs read and write to predefined blocks of the card.

Supported functionalities:

- Read block
- Write word
- TX TARI = 9.44 and RX 424_2 manchester period

Implementation in the *"phExMain_ISO18000p3m3.c"* file.

6.1.11 NFC forum tag reader/writer

If the card is detected in type A or type F technology and if the card is NDEF compliant, this example performs read and write of NDEF message to the card. The write message is always of URI RTD, denoting *"nxp.com".*

Implementation is in *phExMain_Type1Tag.c, phExMain_Type2Tag.c, phExMain_Type3Tag.c, and phExMain_Type4Tag.c.*

6.1.12 ISO14443-4 card mode (until activation)

During listen, the example can be configured to either act as ISO14443-4 card emulator (SAK = 0x20) or NFC-DEP Target (SAK = 0x40). This configuration is done via #define macro in *phExMain_Clif.h.*

If the SAK is 0x20, discovery loop detects peer ISO14443A reader, the phExMain_CardMode is executed, that responds to RATS from the reader. This example does not demonstrate L4 APDUs exchange (it is done in phExHCE and phExNFCForum).

6.1.13 Passive and active ISO18092 initiator (until activation)

During active poll mode, if ATR_REQ is received or during passive poll mode, if SAK denotes 0x40, then the example implemented in phExMain_PasIni.c/ phExMain_ActIni.c is executed. These examples simply transmit a DEP_REQ command with arbitrary payload to peer target. The purpose of this example is only to demonstrate integration of ISO18092.

6.1.14 Passive ISO18092 target (until activation)

During listen, the example can be configured to either act as ISO14443-4 card emulator (SAK = 0x20) or NFC-DEP Target (SAK = 0x40). This configuration is done via #define macro in *phExMain_Clif.h.*

If the SAK is 0x40, discovery loop detects peer ISO18092 initiator, the phExMain_PasTgt is executed, that responds to ATR_REQ from the initiator. This example further waits for a NFC-DEP frame from the initiator.

6.1.15 Contact 7816 reader

If the IC boots because of CT presence wake-up or if the CT presence interrupt is generated, the phExMain_Ct example is executed. This application activates the card, and determines whether the card is of EMVCo payment card or nonpayment card. If payment card is detected, the application further communicates with the card to know the type of card (master card, visa or amex card), and prints the information.

ISO7816 supported functionalities:

- ATR parsing
- Create MF
- Create EF
- Select EF
- Write binary
- Read binary
- Delete EF
- SCOSTA card
- TA1 = 97
- Class A (DC-to-DC converter always in double mode)

6.1.16 RTOS task management

The phExMain example can be executed in both RTOS. In RTOS environment, three tasks are created.

1. System task
   a. Creates CLIF task
   b. Creates CT task
   c. Waits for PMU/PCR exception events
   d. If standby is enabled, waits for CLIF Task completion
   e. If standby is enabled, Waits for CT Task completion
   f. Enter low-power mode (standby)
2. CLIF task
   a. If standby is **not** enabled, starts GP timer for listen duration and wait for GP timer expiry
   b. Configure external RF on detection
   c. If boot reason is WUC counter or GP timer expiry, perform polling mode of discovery loop
   d. If boot reason is RFLD or external RF is detected, perform listen mode of discovery loop
   e. Notify system task if polling/listening is completed and standby is enabled

3. CT task
   a. Enable CT presence interrupt and wait for CT presence interrupt
   b. If boot reason is CT presence or CT presence interrupt is detected, perform CT example
   c. Notify system task if polling/listening is completed and standby is enabled

- Fig 25 and Fig 26 illustrate one instance of phExMain execution for both standby and non-standby scenarios.
- The CLIF and CT tasks are independent and can concurrently operate the CL and CT interfaces. During such concurrent operation, there is a possibility that CT interface may be unstable. It is up to the application design to configure interrupt and task priorities for a stable operation.

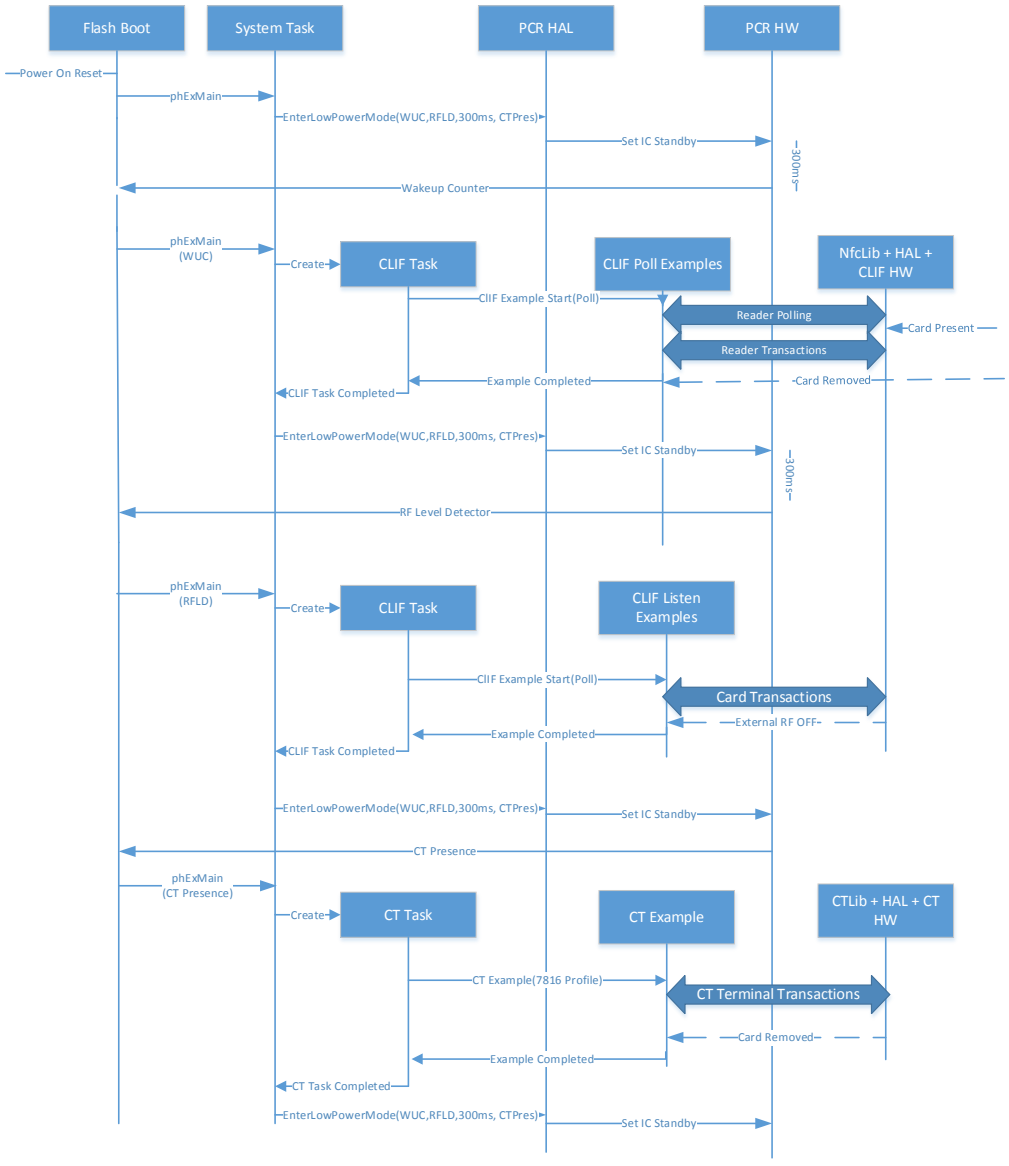

Fig 23. PN7462AU phExMain sequence diagram for standby scenario with RTOS

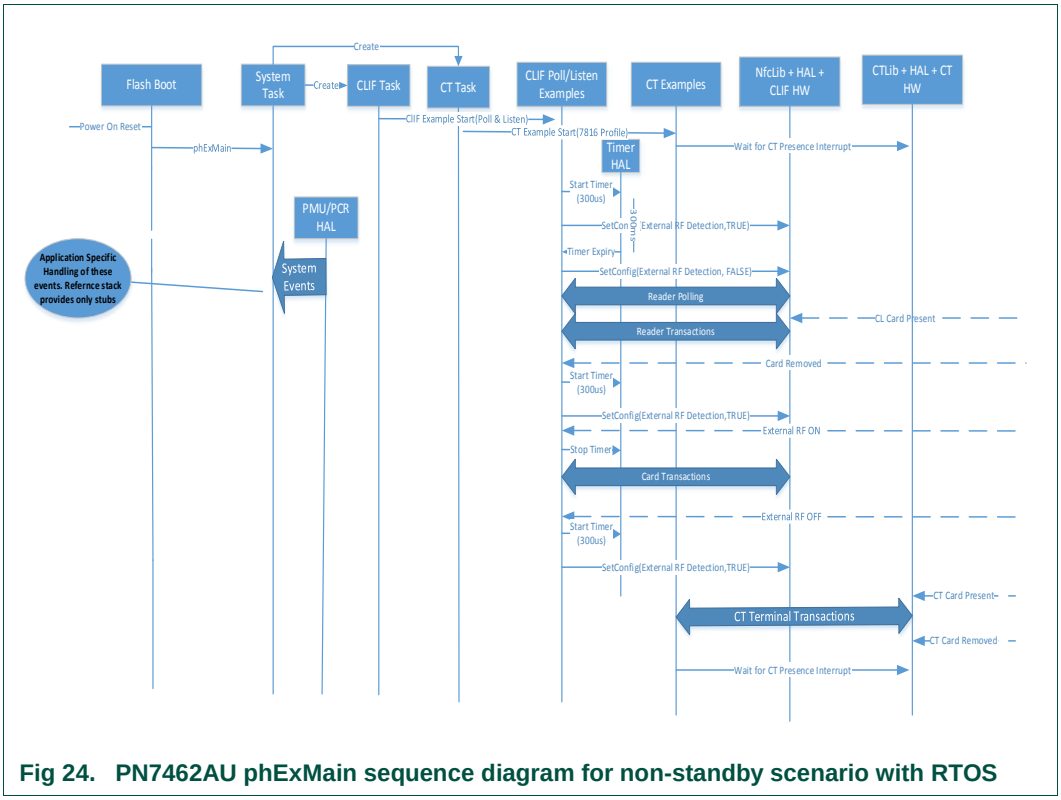

Fig 24. PN7462AU phExMain sequence diagram for non-standby scenario with RTOS

6.1.17 No-RTOS management

In case of No-RTOS, the entry point from flash boot is *phExMain_NoRTOS.* The functionality remains the same except that the CLIF example and the CT examples are called from a single executive while loop. It is based on timer interrupt or external RF detection or CT presence interrupt.

6.2 PN7462AU_ex_phExEMVCo

The phExEMVCo example is provided to demonstrate the following EMV features of PN7462AU FW. This example is a standalone example executing inside PN7462AU without any host interface.

6.2.1 EMVCo discovery loop

This example polls only type A and type B technologies. No other poll or listen technologies are supported. This example also does not support LPCD and standby feature.

6.2.2 ISO14443-4 reader (type A and type B)

In this example, the following commands are transacted between the PN7462AU and Type A EMV Card.

- Select (PPSE)
- SELECT command
- Get Processing options
- Read Record
- Generate AC

6.2.3 Contact EMVCo reader

This example sends SELECT APDU command to the CT card and expects to receive RAPDU command 0x90 0x00. It sends select commands for nine pre-selected types of EMVCO cards.

1. Master card: Credit or Debit (tested)
2. Visa card: Credit or Debit (tested)
3. Master card: Maestro (debit card)
4. Master card: Cirrus (inter-bank network)
5. Master card: Maestro UK
6. Visa card: Electron card
7. Visa card: V PAY card
8. Visa card: VISA Plus card
9. Amex card (tested)

6.2.4 RTOS task management

- For information regarding RTOS task management, refer Section 6.1.16.

6.2.5 No RTOS management

- For information regarding No RTOS management, refer Section 6.1.17.

6.3 PN7462AU_ex_phExHif

The phExHif example is used for:

- Demonstrating the host interface loop back functionality for I^2C, SPI, HSU
- Demonstrating the master interface functionality for I^2CM, SPIM
- Demonstrating secondary downloader functionality to EEPROM and flash memory over SPI Host interface

- Demonstrating CT functionality with SPI Host interface
- Demonstrate usage non-RTOS based integration
- For Host Interface, BUFFER_FORMAT_FREE (struct phhalHif_BuffFormat_t) is used
- One Application@PN7462AU
  - to demonstrate multiple interface features
  - to characterize performance
    - For example: Optimum FIFO threshold of I^2CM is seven words for max performance
    - Context switch latency without RTOS is 11 µs
- Functionality of host interface HALs, master interface HALs, flash/EEPROM memory HALs and forwarding of APDU received from host interface to CT interface

6.3.1 LPC interfacing

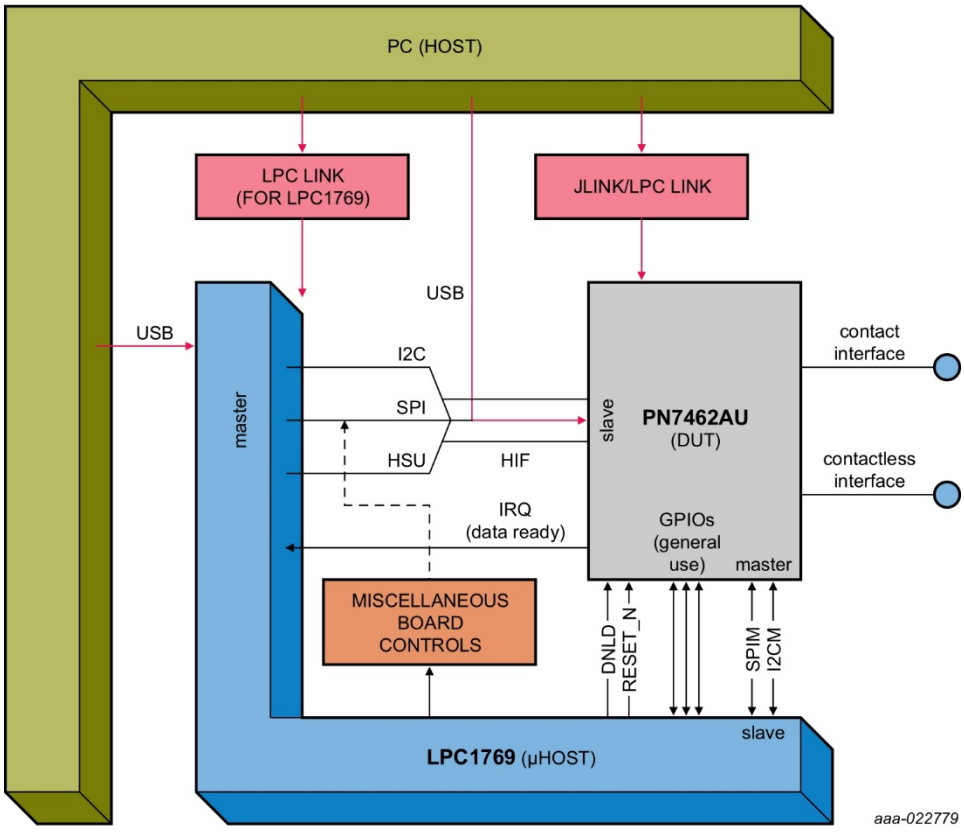

Fig 25. PN7462AU_ex_phExHif example setup

The GPIOs of LPC and PN7462AU are used to determine which functionality of this example has to be executed. It is also used to select the host interface or master interface to be used. For each different functionality, a different LPCXpresso project is required for LPC1769 side while PN7462AU side has only the PN7462AU_ex_phExHif LPCXpresso project.

The PN7462AU_ex_phExHif indicates its readiness to LPC1769 through GPIO1 of PN7462AU connected to GPIO0.0 of LPC1769 (APP ready pin).

6.3.2 HIF selection

Table 26. HIF selection

GPIO5_PN7462AU ← GPIO2.0_LPC	GPIO4_PN7462AU ← GPIO2.1_LPC	Chosen HIF
0	0	invalid

GPIO5_PN7462AU ← GPIO2.0_LPC	GPIO4_PN7462AU ← GPIO2.1_LPC	Chosen HIF
0	1	I^2C
1	0	SPI
1	1	HSU

6.3.3 Operation selection

The tabulated GPIO configuration selects the operation performed by examples shown in Table 27.

Table 27. Operation selection

GPIO8_PN7462AU ← GPIO2.2_LPC	GPIO7_PN7462AU ← GPIO2.3_LPC	GPIO6_PN7462AU ← GPIO2.4_LPC	Operation performed by PN7462AU example
0	0	0	loopback on HIF
0	0	0	forward HIF RX packet to I2CM TX
0	1	0	forward HIF RX packet to SPIM TX
0	1	1	forward HIF RX packet to SPIM & I^2CM TX both
1	0	0	program EEP with HIF RX packet
1	0	1	program flash with HIF RX packet
1	1	0	forward HIF RX packet to CT
1	1	1	RFU

6.3.4 EEPROM configuration dependencies

Values from the following EEPROM structures are used in this example:

- Boot::EEPROM
- Boot::FLASH
- Boot::CT
- Boot::GPIO
- HW::I^2CM
- HW::SPIM
- HW::HIF

6.3.5 LPCXpresso projects provided for LPC1769

- LPCExHif_HSU_LoopBack_App
- LPCExHif_HSU_to_I2CM_SPIM_App
- LPCExHif_I2C_Loopback_App
- LPCExHif_I2C_to_SPIM_App
- LPCExHif_SPI_CT_App

- LPCExHif_SPI_LoopBack_App
- LPCExHif_SPI_to_EEPROM_App
- LPCExHif_SPI_to_FLASH_App
- LPCExHif_SPI_to_I2CM_App
- Supporting libraries
  - PN640_lpc17xx_lib, CMSISv2p00_LPC17xx

6.4 PN7462AU_ex_phExRf

The phExRf example is provided to demonstrate the contactless application development directly using the RF HAL. Since no protocol libraries are used, the feature demonstration is not as extensive as PN7462AU_ex_phExMain. Also, this example demonstrates the non-RTOS integration of application with HALs.

6.4.1 NFC forum + discovery loop

For information regarding NFC forum + discovery loop, refer Section 6.1.1, except that NxpNfcRdLib discovery loop is **not** used and APIs from Section 4.10 are directly used.

6.4.2 MIFARE Classic reader

For information regarding MIFARE classic reader, refer Section 6.1.3.

The implementation is available in *"phExRf_A.c"* file.

6.4.3 MIFARE Ultralight reader

For information regarding NFC forum + discovery loop, refer Section 6.1.4.

The implementation is available in *"phExRf_A.c"* file.

6.4.4 Jewel reader

For information regarding NFC forum + discovery loop, refer Section 6.1.5.

The implementation is available in *"phExRf_A.c"* file.

6.4.5 ISO14443-4 type A reader (MFDF Card)

For information regarding NFC forum + discovery loop, refer Section 6.1.6.

The implementation is available in *"phExRf_A.c"* file.

6.4.6 ISO14443-4 type B reader (EzLink/SLE Card)

For information regarding NFC forum + discovery loop, refer Section 6.1.7.

The implementation is available in *"phExRf_B.c"* file.

6.4.7 FeliCa reader

For information regarding NFC forum + discovery loop, refer Section 6.1.8.

The implementation is available in *"phExRf_F.c"* file.

6.4.8 ISO15693 reader (ICODE SLIX Card)

For information regarding NFC forum + discovery loop, refer Section 6.1.9.

The implementation is available in *"phExRf_15693.c"* file.

6.4.9 EPCV2 (ISO18000p3m3) reader – (ICODE ILT Card)

For information regarding NFC forum + discovery loop, refer Section 6.1.10.

The implementation is available in *"phExRf_18000p3m3.c"* file.

6.4.10 Active mode P2P

This example only performs active mode NFC-DEP activation to demonstrate the usage RF Field control APIs for active mode switching and the SynByte handling for NFC-A active mode.

The implementation is available in *"phExRf_ActInit.c and phExRf_CM.c"* file.

6.4.11 Card mode

This example only performs card mode activation (until RATS-ATS) to demonstrate the usage of AutoColl API.

The implementation is available in *phExRf_CM.c"* file.

6.5 PN7462AU_ex_phExRFPoll

The example implements the polling for contactless cards without NFC Reader Library support and without RTOS (on top of Bare Metal HAL).

6.5.1 Reader Mode

Supports TypeA, TypeB, Felica, ISO15693, ISO18000p3m3 protocols

Device Limit supporting per Technology is 1.

Supports up to read and write for all protocols.

Supports proprietary cards Mifare Classic, Mifare UltraLight till read and write

Supports authentication and reauthentication for Mifare Classic and Inventory read and fast Inventory read command for ICODE SLIX

Note- Since Inventory read is a manufacturer specific proprietary command thus inventory read will work only for NXP Manufactured ICode Cards one of which is ICODE SLIX.

Supports Topaz/Jewel Tags - Command supported RID, READ8.

6.5.2 Peer-to-peer Mode

Supports Active F 212 till ATR REQ only.

6.5.3 Card MODE

Emulates as Type A Card and supports till I-Block exchange.

6.6 PN7462AU_ex_phExCt

This example is provided to demonstrate the application development directly over the CT HAL APIs. PN7462AU_ex_phExCt activates EMVCo card. SELECT master card APDU is sent depending on the protocol supported by the card and expects RAPDU 0x90 0x00. phExCt uses only the CT HAL APIs to demonstrate the CT functionality. The example is also capable of determining the non-EMVCo card or non-Master card. After the transactions, deactivation is performed. Also, this example demonstrates the non-RTOS integration of application with HALs.

6.6.1 EMVCo activation

The example performs EMVCo activation and EMVCo ATR parsing. It also determines the protocol supported by the card.

6.6.2 SELECT master card

The example sends a SELECT master card APDU and expects a RAPDU 0x90 0x00.

6.7 PN7462AU_ex_phExCT7816

This example is provided to demonstrate the CT interface capability to work in the ISO7816 mode. phExCT7816 activates ISO7816 card and transactions. The example is built to work on a SCOSTA card. phExCT7816 demonstrates the CT Protocol Lib + HAL APIs. After the transactions, deactivation is performed. Also, this example demonstrates the non-RTOS integration of application with HALs.

6.7.1 ISO7816 activation

The example performs an ISO7816 activation and ISO7816 ATR parsing. Also, it determines the protocol supported by the card and applies the protocol supported.

6.7.2 APDU transactions

The following APDU'S are sent after the activation of the card. If the card supports the following APDU'S (e.g. SCOSTA), proper responses come from the card.

- Create MF
- Create EF
- Select EF
- Write binary
- Read binary
- Delete EF

6.8 PN7462AU_ex_phExCTEMVCo

For the contact EMVCo reader, the example activates the card and then sends ONLY SELECT APDU command, and expects RAPDU 0x90 0x00. PN7462AU_ex_phExCTEMVCo demonstrates the CT Protocol Lib + HAL APIs. After the transactions, deactivation is performed. The example is also capable of determining the Non-EMVCo card. Also, it demonstrates the non-RTOS integration of application with HALs.

6.8.1 EMVCo activation

The example performs an EMVCo activation and EMVCo ATR parsing. It also determines the protocol supported by the card and applies the protocol supported.

6.8.2 APDU transactions

The example sends select commands for nine pre-selected types of EMVCO cards after successful activation.

1. Master card: Credit or Debit (tested)
2. Visa card: Credit or Debit (tested)
3. Master card: Maestro (debit card)
4. Master card: Cirrus (inter-bank network)
5. Master card: Maestro UK
6. Visa card: Electron card
7. Visa card: V PAY card
8. Visa card: VISA Plus card
9. Amex card (tested)

6.9 PN7462AU_ex_phExCcid

Example implements USB CCID reader. The PC USB reader example is hosted on the PN7462AU and can be tested with any PC/SC application running on the PC with Windows OS.

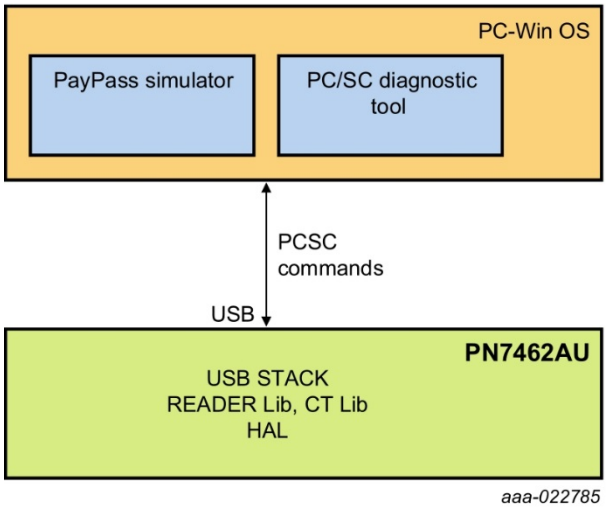

Fig 26. PC USB reader example block diagram

The USB stack and CCID class is implemented in the PN7462AU. The default CCID driver present in PC with Windows OS is used for operation.

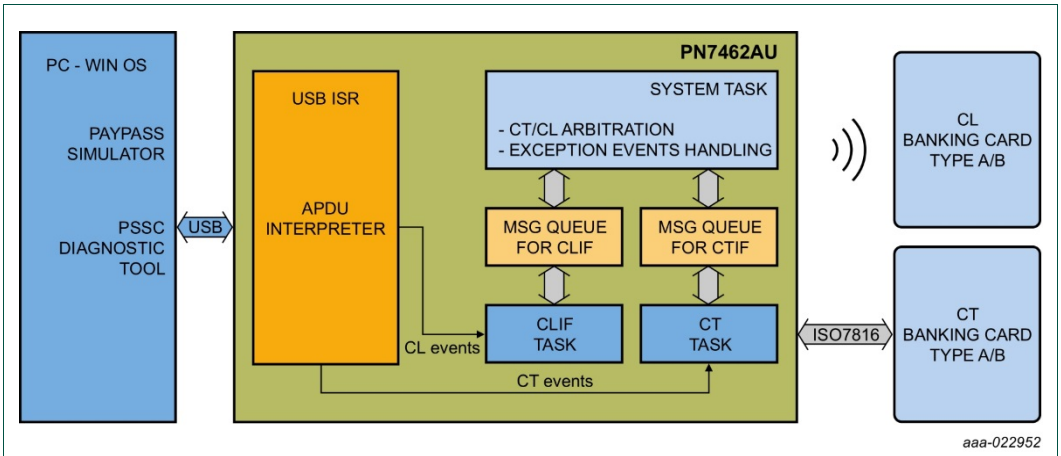

Fig 27. Example architecture

Note: Detailed description and how to use example is described in "PN7462AU PC CCID Reader User Manual".

6.10 PN7462AU_ex_phExDoorAccess

The example application demonstrates card detection, card authentication and HSU interface communication with the PC host. The application is running a NFC polling loop and goes to standby mode after each polling loop in case no CL card is detected by the RF field of the reader. The polling loop is implemented for Type A, B, F, ISO15693 and ISO 1800-P3M3. The application prints out the detected type of the card and UID if available. In case the NFC device is detected, the application sends a NDEF message containing the NXP webpage address.

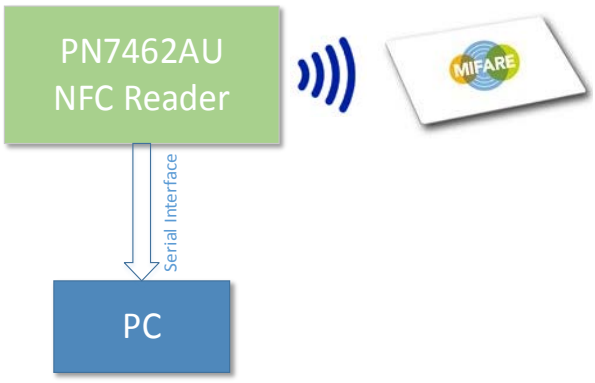

Fig 28. Door access example

After each polling loop the application goes to standby mode and remains for 500ms. A timer is used as a wakeup source from standby. This process continues until a card is detected by the RF field of the reader. If a card is detected, the card type with its UID is sent via HSU and will be printed on the PC console.

In case a MIFARE Classic card is detected, application tries to authenticate the card using the default MIFARE key. If the authentication is successful a block of data is read from the card. The type of the card, UID and data are sent via HSU and printed on the PC console.

P2P functionality is integrated. When a NFC enabled phone is detected from the RF field as an active or passive target the LLCP SNEP will be activated and NDEF message will be sent to the mobile device.

If LPCD is enabled, the reader checks during the wakeup time the presence of a card and enters the card detected mode when a card is present & repeats the cycle. If no card is present it goes back to standby mode. LPCD is enabled by default.

Note:
Detailed description and how to use example is described in “Door Access User Manual”.

6.11 PN7462AU_ex_phExDoorAccessEC

This example is related to the 6.10 but in this version the application is using a MIFARE DESFire EV1 card for the authentication, data exchange is done over contactless interface and software key store or SAM (Secure Access Module) key store is used for storing the authentication key. By default, the software key store is enabled. The user can use the SAM key store by enabling corresponding. SAM is a key storage element and it should be inserted in the CT main.

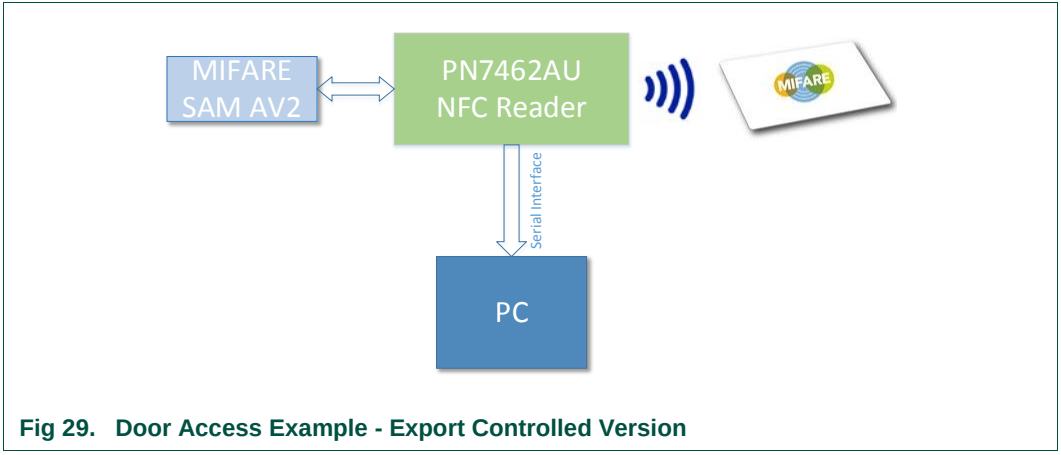

Fig 29. Door Access Example - Export Controlled Version

On power-up, NFC Reader starts polling for (PICC) cards (Type A, B or F, ISO15693, ISO18000-3M3) and if no card is present, the reader goes to standby mode. Timer is used as a wakeup source from standby - timer periodically every 500 ms. This process continues until a card is detected by the RF field of the reader. If a card is detected, the card type with its UID is sent via HSU & this is printed on the PC console

If a MIFARE DESFire EV1 card is detected, the reader tries to select a pre-written custom application on the card. It tries to authenticate the card using the key stored in SAM. If SAM is not present, then software keys can be used for authentication. If authentication is successful a block of data will be read from the card.

Note:
Detailed description and how to use example is described in “Door Access User Manual”. This example is available only with PSP package from NXP DocStore.

6.12 PN7462AU_ex_phExPos

POS use-case demo application shows how to use PN7462AU in combination with second application hosted on the MCU. In this example LPC1769 is used and connection is established through SPI host interface. POS use-case demonstrate the Pay pass transaction on the contact and contactless frontend.

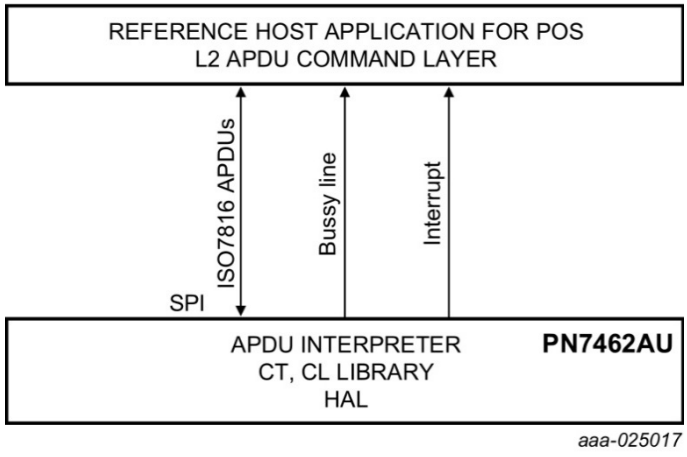

Fig 30. POS use case demo architecture

The POS demo architecture is split into application layer (L2) and low level EMVCo compliant layer L1 which is hosted on the PN7462AU. The application layer L2 commands are simulated in reference microcontroller board (LPC1769) and L1 layer components are placed in PN7462AU.

The application APDU commands (L2) are communicated to PN7462AU through SPI host interface. PN7462AU GPIO pin is used to synchronize command / response between LPC1769 and PN7462AU.

IRQ pin is used to notify valid ISO 14443-4 card to LPC1769.

Note:
Detailed description and how to use example is described in “POS Use Case Demo Setup Manual”.

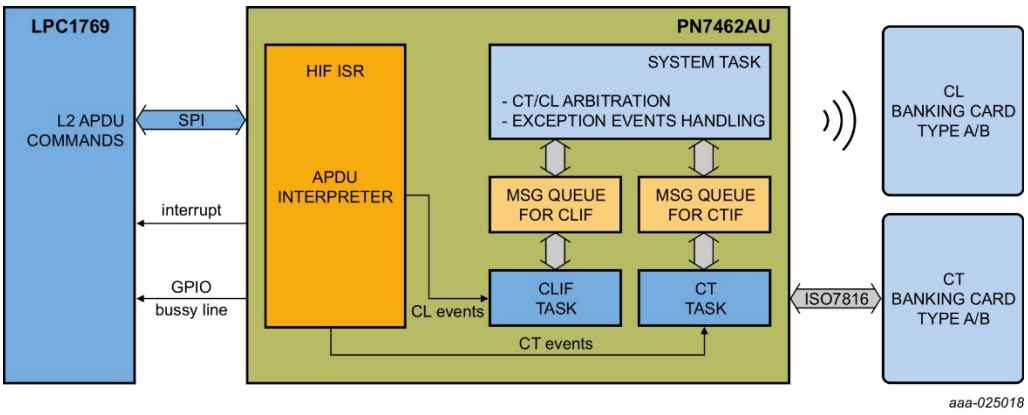

Fig 31. POS demo architecture

6.13 PN7462AU_ex_phExNFCCcid

The NFC CCID is versatile demo application that features:

- Card detection for TypeA, TypeB, Felica, ISO15693, ISO18000p3m3 technologies
- Support for proprietary commands for MIFARE Classic, MIFARE Ultralight and MIFARE Ultralight C for read and write.
- CCID USB device protocol implementation. Supports the Suspend Resume and Wakeup Feature.
- Communication of the CLIF information with the PC using a PCSC application.
- P2P passive initiator mode. Supports LLCP Initiator Mode for sending the NDEF message to the mobile.

6.14 PN7462AU_ex_phExMfCryptoEC

The PN7462AU_ex_phExMfCrypto demo application includes both CL and CT library components. Example is based on the Discovery Loop alongside Crypto layers and it is intended to evaluate MIFARE DESFire card functionalities. Application performs following operations:

- Application and Data and Value File creation inside the card.
- AES authentication of the application.
- Changing of the key.
- Enciphered Read of Value File and Plain Read of Data File.
- Enciphered Write of Value File and Plain write of Data File.
- Supports Different Keys for Read and write.
- CT part does not include any crypto example for CT but gives scope to include the CT example later

Note:
This example is available only with PSP package from NXP DocStore.

6.15 PN7462AU_ex_phExRfPCDA

This example demonstrates simple low level API usage to perform detection, anti-collision, activation, authentication and R/W operation on the Type A cards according to ISO14443 and MIFARE standard.

Application is using low level RF interface HAL implementation in flash. There is limitation to only one card at the time. Supported TypeA cards are Type1 TOPAZ, MIFARE Ultralight, MIFARE Classic, MIFARE DESFire cards will pass through activation and anti-collision and. In the case of MIFARE Classic card also authentication with default key is demonstrated. In the case of MIFARE DESFire card L4 activation is demonstrated.

6.16 PN7462AU_ex_phExVCom

This example application features TypeA card detection, RF filed control and communication with the PC host over USB CDC interface (VCOM). The example shows CDC USB device class implementation.

6.17 PN7462AU_ex_phSystemServices

This example application demonstrates system services invocation. The PN7462AU provides ROM services, also described in `/PN7462AU/phROMIntf/phhalSysSer/inc/phhalSysSer.h` and with more detailed description in API documentation.

This application requires user interface for performing the operations so it is needed to use debug mode. Some of the featured system service commands could be irreversible or reversible depending on the application mode configured by `ENABLE_IR_REVERSIBLE_COMMANDS macro.`

PN7462AU_ex_phSystemServices features

Feature	description
SECROW Lock	The HW SecRow contains the SWD access bits, code write-protection bits and RSTN pin behavior bits. For blocking any further writes to SecRow, the phhalSysSer_OTP_SetSecrowLock() is used. It prevents further usage of phhalSysSer_OTP_SecrowConfig() function.
Code write protection	It is required to lock flash memory from write at HW level. It is locked possibly at a stage when secure secondary upgrade is not planned for the remaining lifecycle of the product. For such use cases, phhalSysSer_OTP_SecrowConfig() is used to lock flash memory from any further write. Any flash programming after locking the flash results in hard fault. Once SECROW functionality is locked, this feature cannot be used anymore.
Block SWD debugging	This command disables PN7462AU SWD debug interface. When the PN7462AU IC is delivered from production to user, the default SWD

Feature	description
	access level enables the user to view and debug user flash memory, user EEPROM memory, user RAM memory, and peripheral registers. The access level can be irreversibly changed to prevent view/debug access to any memory region or peripheral registers, before deploying the IC to the field. phhalSysSer_OTP_SecrowConfig() can be used to lock the SWD against any further access. Once SECROW functionality is locked, this feature cannot be used anymore.
Disable primary download	Command is used to irreversibly disable the ROM primary download feature. On subsequent boots, the ROM boot never enters ROM primary download mode, even if DWL_REQ pin and USB_VBUS pin is high. This feature is typically used after development and flashing of secondary downloader in the flash memory, for subsequent code/data upgrades.
Update Product ID	USB Product ID PID update
Update Vendor ID	USB vendor ID update
Perform In Application Programming	Application asks for FLASH page number. Page is 128bytes long, for 158kb of the flash memory, the page number is in range 0-1263. The selected flash page is updated from user programmable values.
Set internal PVDD	PVDD is pad voltage reference and supply of the host interface (HSU, USB, I2C, and SPI) and the GPIOs. This command sets PVDD configuration to internal.
Get ROM version	Commands returns current ROM firmware version

7. PN7462AU RTOS abstraction layer

The NxpNfcReaderLibrary can be executed either in RTOS environment or non-RTOS environment. The HAL code does not change for both the environments.

phRtos is provided to RTOS abstraction*,* which wraps either an RTOS or a stub RTOS (No RTOS).

In case of no RTOS, an event wait is modelled as “WFI” and event variable is a global variable. Also, tasks are **not** abstracted in no-RTOS. It means, the examples that execute in non-RTOS abstraction run on ARM thread mode only.

The build time macro *PHFL_HALAPI_WITH_RTOS in APP_NxpBuild.h* is used to control whether RTOS or non-RTOS is used.

8. PN7462AU common utilities layer

The common utilities layer provides an abstraction to standard library functions such as memcpy, memset, memcmp etc. It also provides a delay function that counts, based on the CPU clock of 20 MHz.

9. PN7462AU critical sections in HAL

The implementation of HAL is supporting applications with several threads which can be synchronously activated by software. Additionally, interrupt service routines (ISRs) asynchronously interrupt the CPU at any time when signaled by the hardware. Using these two mechanisms together can lead to difficulties, which arise when same resources are used in different threads. To avoid this critical sections are implemented.

Critical sections are used to synchronize the thread mode and handler mode in HAL. To disable IRQ “cpsie” instruction is used and to enable IRQ “cpsid” instruction is used. Table 28 contains description of the places in the code where critical sections are implemented, times for how long interrupts are disabled and the reason why critical section is required.

During the critical section, all interrupts are disabled and any interrupt, which should have triggered in this time period, will be delayed or missing.

In the Fig 32 it is presented, as an example, how critical section impact on the interrupts during its execution.

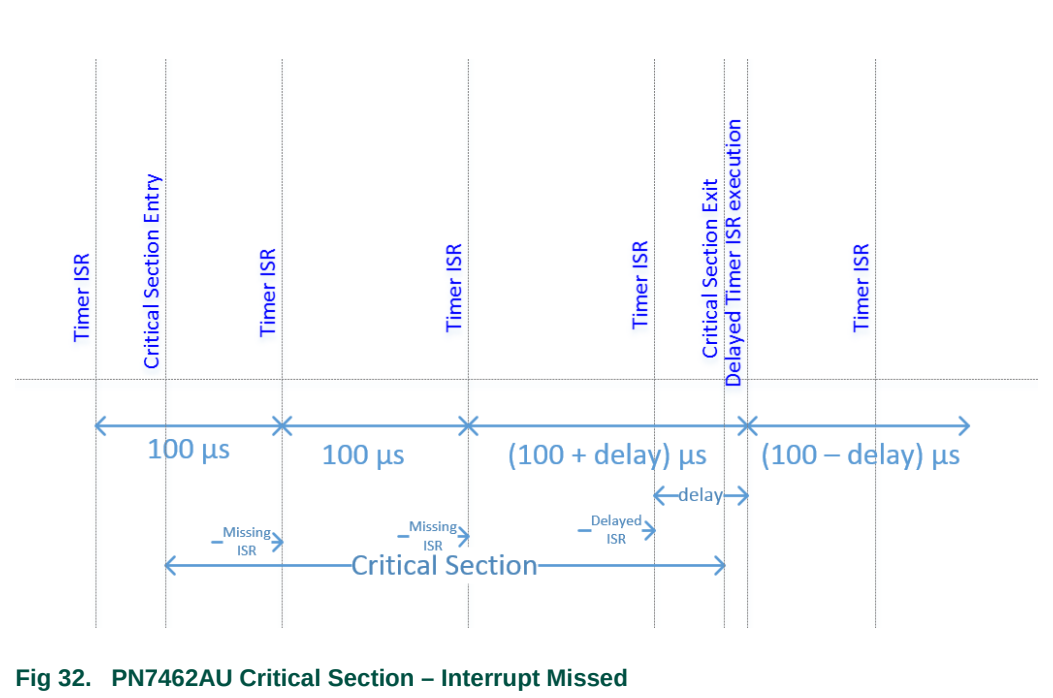

Fig 32. PN7462AU Critical Section – Interrupt Missed

As presented in the Fig 32 there is a possibility of missing interrupts when the interrupt is configured to occur every 100 µs and if the interrupt arrives when the “*phhalRf_Receive()*” API is executing the critical section in passive target mode, the interrupt gets missed couple of times.

Another example when interrupt is delayed:
A timer interrupt is configured to occur at every 500 µs and if the interrupt arrives when the "*phhalRf_Receive()*" API is executing the critical section in passive target mode, the interrupt gets delayed by maximum by 437µs (see Table 28).

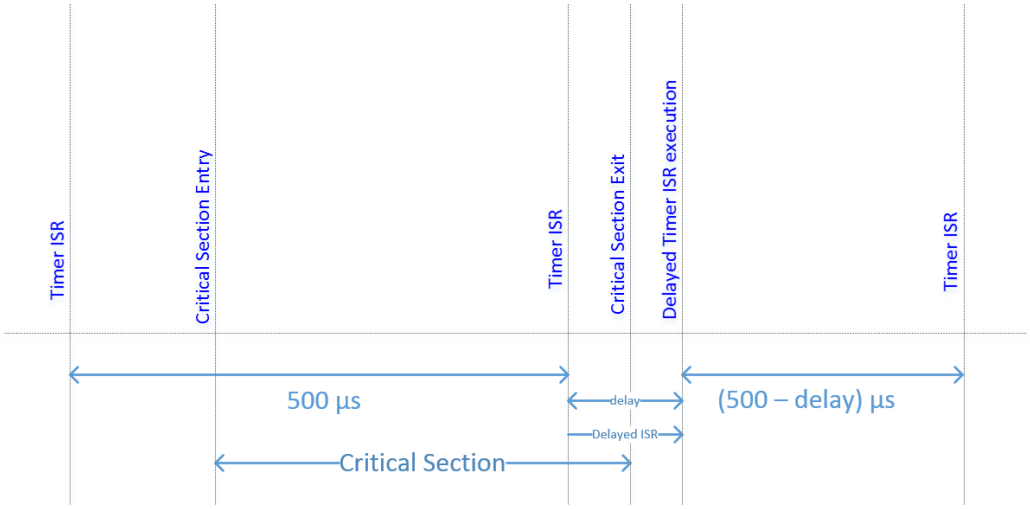

Fig 33. PN7462AU Critical Section – Interrupt Delayed

Table 28. API table with critical section

File Name	API Name	Time	Reason
phFlashBoot.c	phFlashBoot_PreCheck()	12.2 us	Setting up the initial interrupt and setting priorities for the interrupts. Assuming the PVDD is already started
phhalCt_Activate.c	phhalCt_CardDeactivate()	3 us	A variable updated which is shared in thread mode and ISR mode, this is required to synchronize the CT in thread mode and ISR mode for variable update
phhalCt_Transceive.c	phhalCt_Receive()	10.2 us	There is one variable updated which is shared in thread mode and ISR mode, this is required to synchronize the CT in thread mode and ISR mode for variable update
phhalPcr.c	phhalPcr_EnterLowPowerMode()	1.01 ms	When entering into the low power mode, no interrupts are allowed. For e.g. If the chip is entering due to increase in temperature, there should not be any interrupt which stops the chip entering into the standby
phhalPmu.c	phhalPmu_TxLdoStandby()	973.4 us	This time is due to the TVDD cap to settle down.
	phhalPmu_TxLdoStart()	10.4 us in FULL_POWER 9.2 us in LOW POWER	While starting the LDO (Analog Block), it is better not to serve any interrupt so that the analog block starts correctly.
phhalI2CM.c	phhalI2CM_AsmFill_OR_CopyFifo	3 us	Filling the I2CM Fifo has to be atomic for efficiency
phhalRf	phhalRf_SetIdleState()	4.2 us	No interrupt is allowed while entering Idle State. For e.g. While Hal Shut down no clif irqs are acceptable.
	phhalRf_PCD_Exchg()	28.8 us	This is to avoid pre-emption due Async Shut Down/SetIdle or other external IRQs
	phhalRf_PCD_Exchg()	20 us	A system service Api is being called to set up TxRx buffer and during this time the interrupts get disabled.
	phhalRf_Transmit()	46 uS	This is to avoid pre-emption due Async Shut Down/SetIdle or other external IRQs
	phhalRf_Receive()	437 us	Passive Target: This is to avoid pre-emption due Async Shut Down/SetIdle or other external IRQs
	phhalRf_Receive()	33 us	Active Mode: This is to avoid pre-emption due Async Shut Down/SetIdle or other external IRQs
	phhalRf_AutoColl()	26 us	A system service Api is being called during the autocoll, where the interrupts get disabled during this time.
	phhalRf_LoadProtocol_Initiator()	187.8 us	Tx Ocp Irq is not allowed during updating of the registers as this will over-write them.
	phhalRf_PCD_ExchgMFC_Auth()	240 us	A system service Api is being called for the Mifare Authentication, where

	interrupts get disabled during this time.

Note 1
Values in Table 28 are taken at 20MHz HFO clock. These values can vary due to the HFO clock tolerance.

Note 2
The critical sections used in the HAL implementation, can be disabled using the "__phUser_EnterCriticalSection()" macro the in "ph_User.h" file.

"#define __phUser_EnterCriticalSection()".

Note 3
When critical sections are disabled the API's may not work as expected.
There could be some risks like, expected IRQ processing may not happen, the expected behavior of the system may be unstable, the expected functions would not be called or the expected data update may not happen.

10. Abbreviations

Table 29. Abbreviations

Acronym	Description
ALM	Active Load Modulation
API	Application Programming Interface
CLIF	ContactLess InterFace
CRC	Cyclic Redundancy Code
CT	ConTact Interface
DMA	Direct Memory Access
EoF	End of File
EEPROM	Electrically Erasable Programmable Read Only Memory
FW	FirmWare
GPIO	General-Purpose Input Output
HAL	Hardware Abstraction Layer
HPD	Hard Power-Down
HSU	High Speed UART
HW	HardWare
LDO	Low Drop Out
NFC	Near Field Communication
NMI	Non-Maskable Interrupt
P2P	Peer to Peer
PAL	Protocol Abstraction Layer
PLM	Passive Load Modulation
PMU	Power Management Unit
POR	Power-On Reset
PSP	Product Support Package
RF	Radio Frequency
ROM	Read Only Memory
RTOS	Real Time Operating System
SDA	Serial Data Signal
SPI	Serial Peripheral Interface
SPIM	SPI Master interface
SRAM	Static Random Access Memory
SW	SoftWare
SWD	Serial Wire Debug
TXLDO	Transmitter Low Drop Out
USB	Universal Serial Bus

11. References

[1] LPCXpresso webpage: http://www.nxp.com/products/software-and-tools/software-development-tools/software-tools/lpc-microcontroller-utilities/lpcxpresso-ide-v8.2.2:LPCXPRESSO

[2] EEPROM description: “PN7462AU\phHal\phCfg\xml_output_xml_sizes.html”

12. Legal information

12.1 Definitions

Draft — The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

12.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors accepts no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Evaluation products — This product is provided on an "as is" and "with all faults" basis for evaluation purposes only. NXP Semiconductors, its affiliates and their suppliers expressly disclaim all warranties, whether express, implied or statutory, including but not limited to the implied warranties of non-infringement, merchantability and fitness for a particular purpose. The entire risk as to the quality, or arising out of the use or performance, of this product remains with customer.

In no event shall NXP Semiconductors, its affiliates or their suppliers be liable to customer for any special, indirect, consequential, punitive or incidental damages (including without limitation damages for loss of business, business interruption, loss of use, loss of data or information, and the like) arising out the use of or inability to use the product, whether or not based on tort (including negligence), strict liability, breach of contract, breach of warranty or any other theory, even if advised of the possibility of such damages.

Notwithstanding any damages that customer might incur for any reason whatsoever (including without limitation, all damages referenced above and all direct or general damages), the entire liability of NXP Semiconductors, its affiliates and their suppliers and customer's exclusive remedy for all of the foregoing shall be limited to actual damages incurred by customer based on reasonable reliance up to the greater of the amount actually paid by customer for the product or five dollars (US$5.00). The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails of its essential purpose.

12.3 Licenses

Purchase of NXP ICs with NFC technology

Purchase of an NXP Semiconductors IC that complies with one of the Near Field Communication (NFC) standards ISO/IEC 18092 and ISO/IEC 21481 does not convey an implied license under any patent right infringed by implementation of any of those standards. Purchase of NXP Semiconductors IC does not include a license to any NXP patent (or other IP right) covering combinations of those products with other products, whether hardware or software.

Purchase of NXP ICs with ISO 14443 type B functionality

This NXP Semiconductors IC is ISO/IEC 14443 Type B software enabled and is licensed under Innovatron's Contactless Card patents license for ISO/IEC 14443 B.

The license includes the right to use the IC in systems and/or end-user equipment.

RATP/Innovatron Technology

12.4 Trademarks

Notice: All referenced brands, product names, service names and trademarks are property of their respective owners.

MIFARE — is a trademark of NXP B.V.

MIFARE Ultralight — is a trademark of NXP B.V.

MIFARE DESFire — is a trademark of NXP B.V.

I²C-bus logo — is a trademark of NXP B.V.V

ICODE and I•CODE — are trademarks of NXP B.V.

13. List of figures

14. List of tables

15. Contents

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 20 March 2017
336613
Document identifier: UM10913