

Unified bootloader user guide



Contents

Chapter 1 Introduction.....	3
Chapter 2 Unified bootloader system.....	4
2.1 Scope of unified bootloader.....	4
2.2 Unified bootloader key features.....	4
2.3 UI features.....	5
2.4 Unified bootloader reset/power on process.....	5
2.5 Dividing bootloader, APP flash and RAM storage space.....	6
2.6 Unified bootloader system architecture.....	9
Chapter 3 How to port the stack on new platform.....	11
3.1 Unified bootloader system diagram.....	11
3.2 Unified bootloader stack folder.....	11
3.3 Unified bootloader porting file.....	12
Chapter 4 How to install UI/host.....	33
Chapter 5 How to setup UI/host.....	34
5.1 ISO-CAN-UDS tool.....	34
5.2 ISO-LIN-UDS tool.....	39
5.3 General tools.....	40
Chapter 6 Bootloader code size.....	41
Chapter 7 Test and performance.....	42
Chapter 8 APP needed.....	43
Chapter 9 References.....	46
Chapter 10 Revision history.....	47

Chapter 1

Introduction

Unified bootloader is based on Unified Diagnostic Services (UDS) and Transport Protocol and Network Layer Services Protocol (TP) . The unified bootloader objective:

- Efficiency to implement bootloader over a new part by reusing existing stack.
- Deliver a high-quality new bootloader instance by reusing proven platform.

NOTE

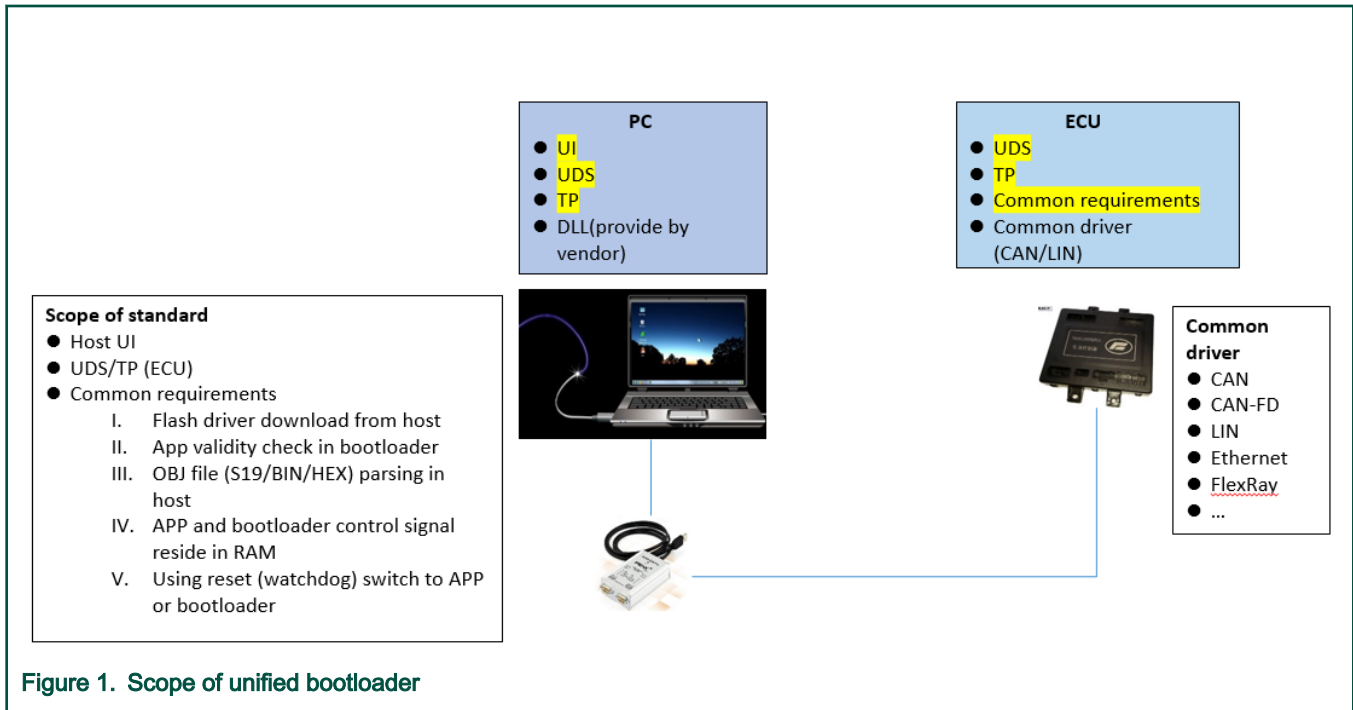
UDS and TP is used by HIS (Herstel Initiative Software, defined the bootloader data and control flow)

Chapter 2

Unified bootloader system

2.1 Scope of unified bootloader

The following figure shows connection of PC and ECU with adapter. The highlighted bullets are common for ECU and PC. For all the platform (MCU) is the same.



2.2 Unified bootloader key features

The following are the key features of unified bootloader.

- UDS (ISO14229)
- TP (ISO1765-2/CAN, ISO17987-2/LIN)
- Common requirements include Flash driver
- Flash driver download from host
- App validity check in bootloader
- APP and bootloader control signal reside in RAM
- Using reset (watchdog) switch to APP or bootloader
- CRC
- AES
- Random
- Hardware driver HAL

Other platform may have more or less features.

2.3 UI features

The following are the key UI features:

- Support JSON control firmware download process
- Support import/export JSON
- Support config JSON
- Support print logging (e.g. all transmitted/received message and timestamp)
- Support parsing OBJ (S19/BIN/HEX)
- Support two chose window: select Flash driver and APP OBJ
- Support config function and physical, respond ID for CAN. LIN can be config NAD ID
- Support CRC and AES. Both can be changed by customer
- Support display progress bar for download firmware
- Status about download firmware
- UDS and TP for communication with ECU over PEAK

2.4 Unified bootloader reset/power on process

The bootloader reset/power on process is explained below. The following checks are performed by bootloader during the power on/reset.

1. Check update APP flag
2. Check APP validity

The following table shows what bootloader needs to do depending on the flag value.

Table 1. Flag value

Update APP flag (TRUE/FALSE)	APP validity (TRUE/FALSE)	Description
TRUE	N/A	Enter bootloader mode for update APP firmware
FALSE	TRUE	Jump to APP
FALSE	FALSE	Enter bootloader mode for waiting update APP firmware

From the above information, bootloader and APP needs to exchange information, the information is stored in retention RAM. The warn reset can hold the information, but the information can be lost because during cold reset (power on/off). The following figure shows bootloader reset/power on process.

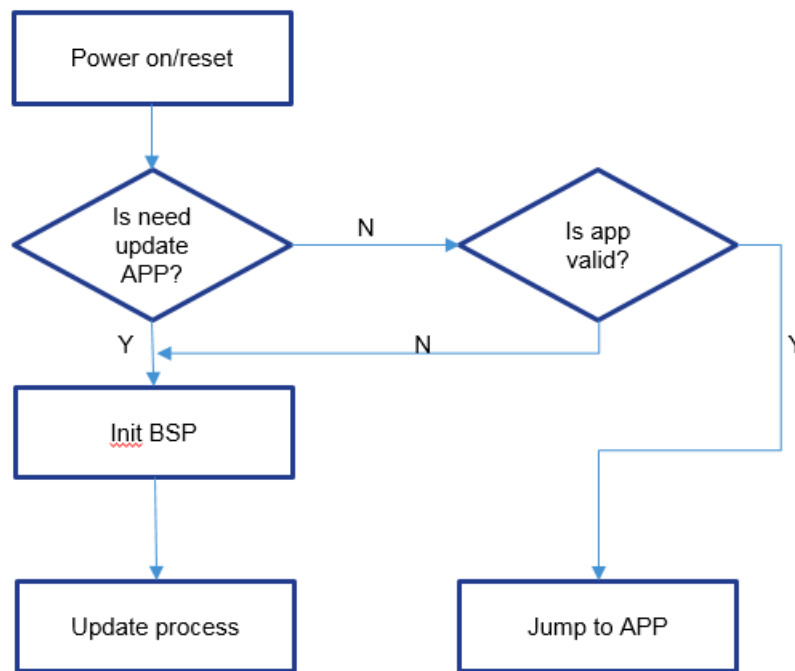


Figure 2. Reset and Power flow diagram

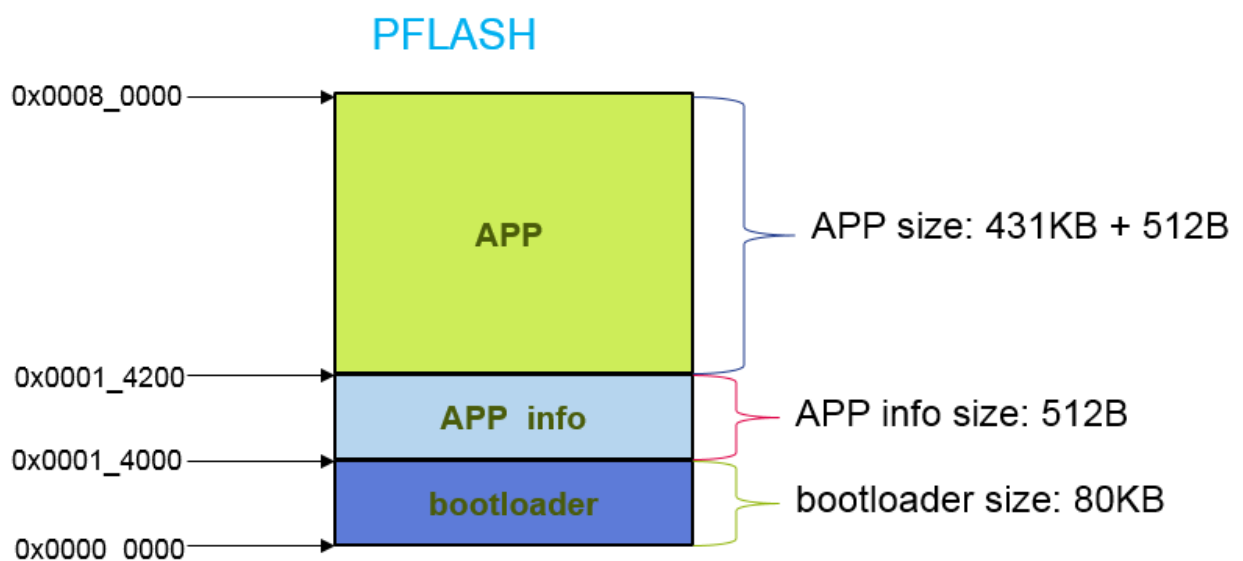
2.5 Dividing bootloader, APP flash and RAM storage space

MCU has two internal memories (FLASH and RAM). The following are examples for dividing FLASH and RAM space for bootloader and APP.

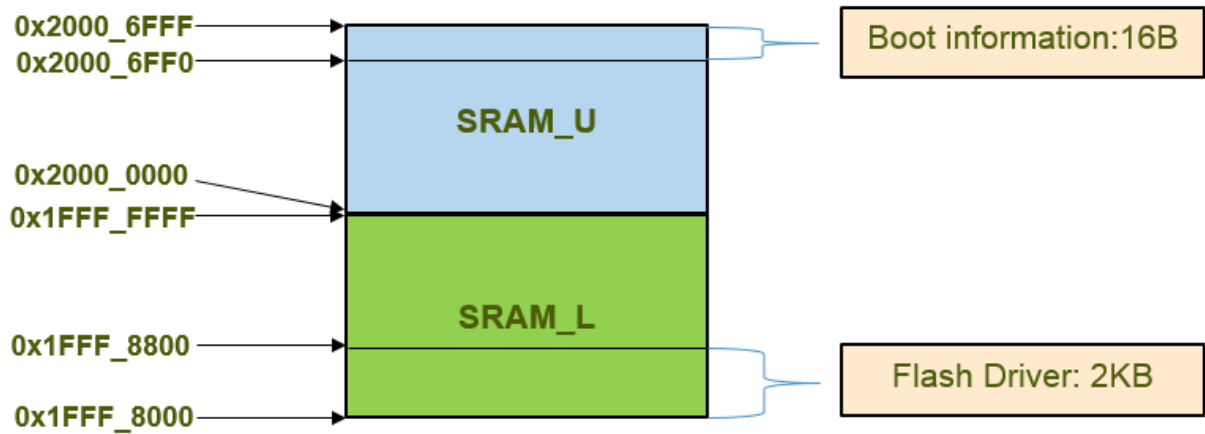
- FLASH: Power on or reset always enter bootloader. Bootloader reset handler stores the default reset handler address.
- RAM: Bootloader and APP should exchange information over retention RAM. This means, bootloader and APP should divide some retention RAM(16 bytes) from link file for exchange of information.

Below is an example for divided flash and RAM space over S32K144 and S12ZVML12.

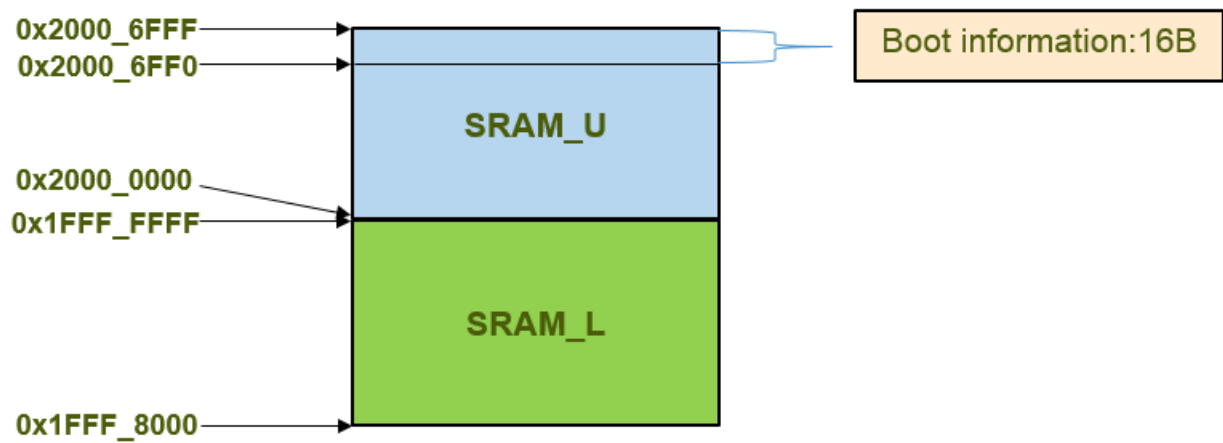
— Divided S32K144 P-flash and RAM space



Bootloader RAM space

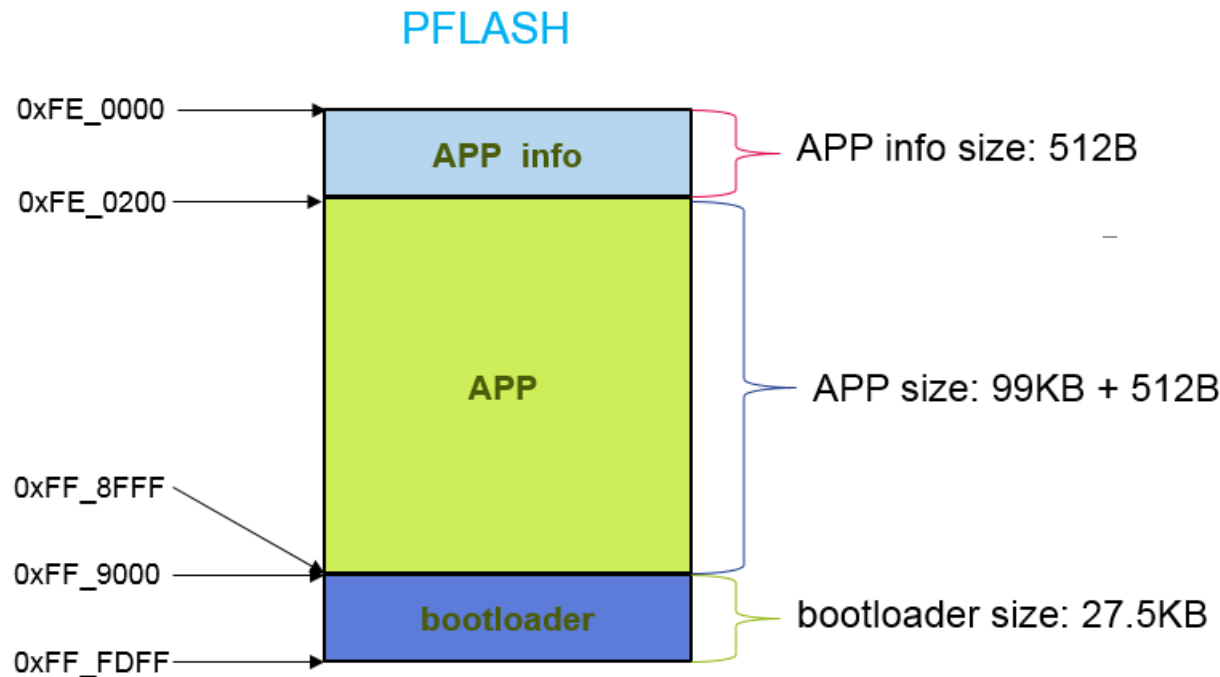


APP RAM space

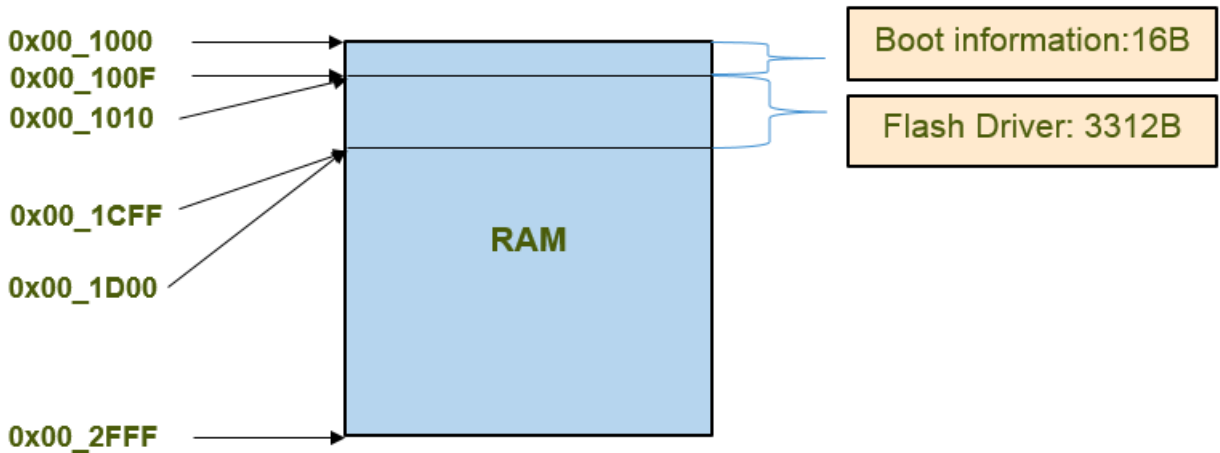


Space for flash driver is not divided by APP.

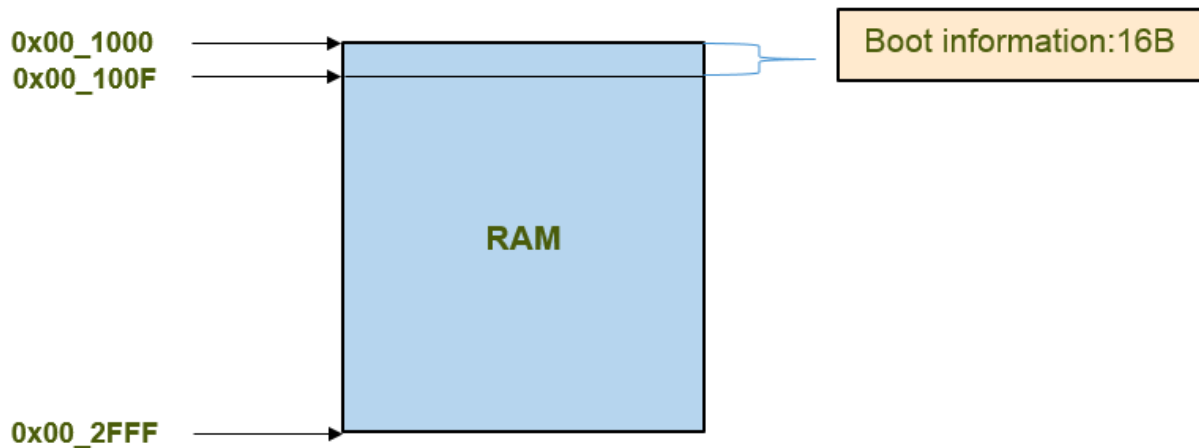
— Divided S12ZVML12 P-flash and RAM space



Divided Bootloader RAM space



Divided APP RAM space

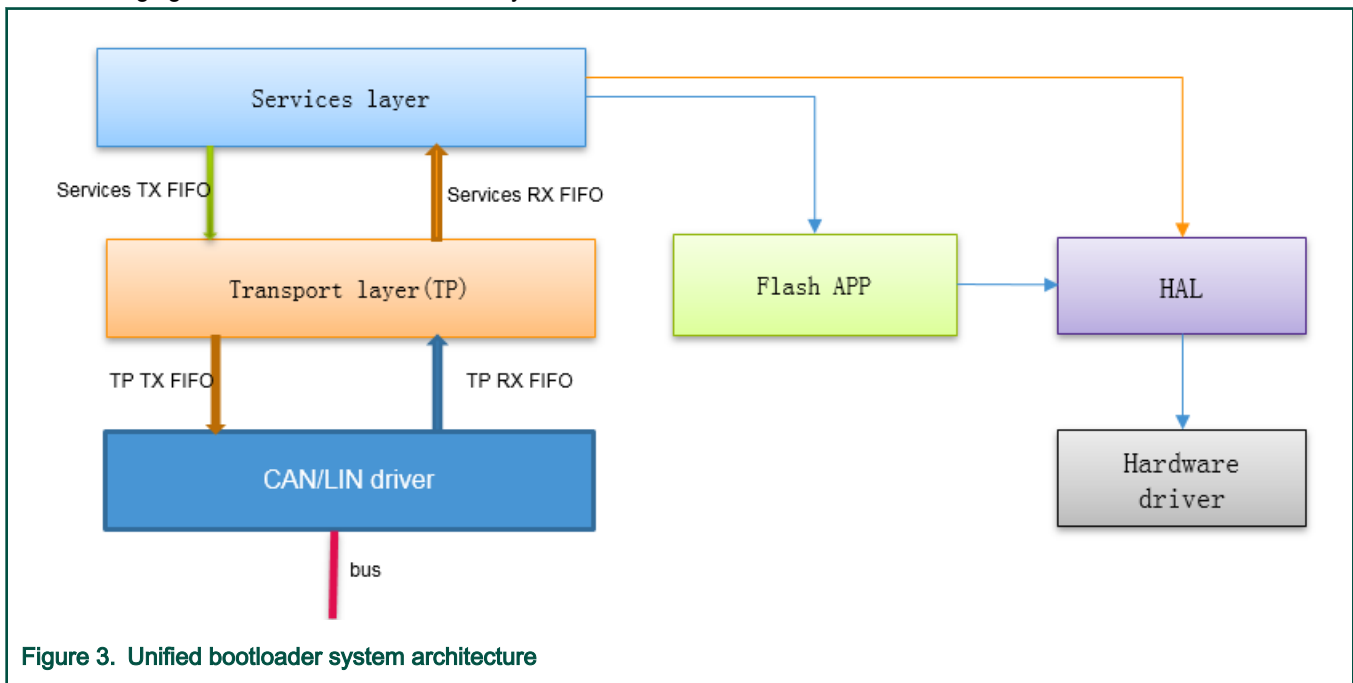


NOTE

Divide bootloader P-flash and flash driver size based on bootloader and flash driver size. For P-flash bootloader used flash size is always N* sectors. Bootloader and APP exchange information including CRC. The CRC algorithm is the same in bootloader and APP.

2.6 Unified bootloader system architecture

The following figure shows Unified bootloader system architecture.



Bootloader has six modules namely, services layer, transport layer, FIFO, hardware communicate driver (CAN/LIN etc.), flash driver and flash operate. The unified bootloader follows the steps to communicate:

1. Services layer

- Receives message from TP layer (services RX FIFO)
- Control bootloader process. Example operate flash init, erase, program etc
- Response message to host (transmit message to services TX FIFO)

2. TP

- Receives message from TP RX FIFO (CAN/LIN etc.)
- Analyze data received from TP RX FIFO. If the message is valid, sends a frame to service RX FIFO
- Transmit message to TP TX FIFO, if message is received from services layer
- TP transmit a frame message to service RX FIFO and deletes the header and checksum
- TP adds header, checksum and also transmits a frame message to TP TX FIFO

3. Hardware driver (CAN/LIN etc.) communicates

- Reads data from bus and transmit data in TP
- RX FIFO Reads data from TP TX FIFO and write to bus

Chapter 3

How to port the stack on new platform

3.1 Unified bootloader system diagram

The following figure shows the system diagram for unified bootloader. Unified bootloader needs some hardware drivers like CAN/LIN, watchdog, timer, flash, debug(I/O, UART) etc, for porting the stack to new platform. You need to prepare all the hardware drivers.

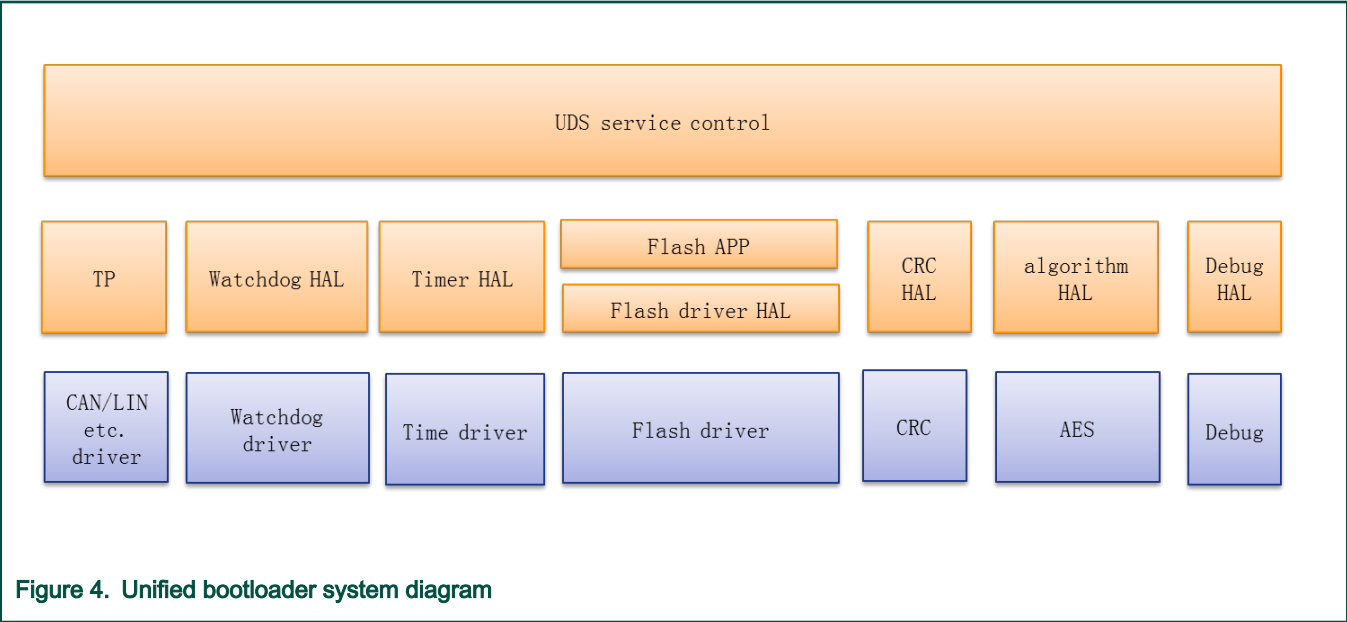


Figure 4. Unified bootloader system diagram

3.2 Unified bootloader stack folder

The following is an unified bootloader stack folder.

Figure 5. Unified bootloader stack folder

auto_lib	2019/1/28 16:17	File folder
boot	2019/1/21 13:46	File folder
Debug	2019/1/23 10:49	File folder
Demo	2019/4/15 16:21	File folder
FIFO	2019/1/18 14:20	File folder
Flah_app	2019/1/22 11:34	File folder
HAL	2019/4/23 11:09	File folder
public_inc	2019/4/26 19:07	File folder
UDS_stack	2019/1/14 11:50	File folder

The following table describes the folder target and whether to modify or not when porting bootloader stack.

No.	Folder name	Description	Porting modify nor not?
1	auto_lib	Public function like memcpy, memset, rand etc.	No

Table continues on the next page...

Table continued from the previous page...

2	boot	Provide some boot services for other module like boot information, jump to APP.	Yes
3	Debug	UART print, debug I/O, timer.	Yes/No
4	Demo	It's a demo for used bootloader stack. User just to add initialize clock and CAN/LIN module in file.	Yes/No ³
5	FIFO	FIFO module.	No ¹
6	Flash_APP	Provide operate FLASH services for UDS.	No ¹
7	HAL	CRC, flash, timer, UDS algorithm, watchdog HAL. It's abstract for hardware. In other modules, used the HAL APIs replace low level APIs.	Yes ²
8	public_inc	Public header file like user configure file and re-typedef data type. All the typedef default as 32-bit MCU. If new platform is not 32-bit MCU, should re-typedef data type.	Yes ²
9	UDS_stack	TP and UDS stack.	Yes ²

1. Do not modify
2. Can be modified
3. Based on user requirement, can be modified

3.3 Unified bootloader porting file

The shaded text in the below sub section denotes that you need to note or modify it when porting bootloader stack.

3.3.1 Public_inc

- includes.h: include common_types.h, toolchain.h, autolibc.h and user_config.h. New platform or compiler may have some conflict for data type like uint32/16/8 etc, to resolve these kind of issues, includes.h should include all the other headers. E.g., cpu.h/stdint.h(typedef uint32/16/8 etc) common_types.h include the macro to prevent re-typedef data type. All the other files should include includes.h.

```

00001: #ifndef __INCLUDES_H__
00002: #define __INCLUDES_H__
00003:
00004: /*common_types.h define uint32/ sint32...*/
00005: #include "stdint.h"
00006: #include "cpu.h"
00007: #include "port.h"
00008:
00009:
00010: #include "common_types.h"
00011: #include "toolchain.h"
00012: #include "autolibc.h"
00013:
00014: /*user_config.h is used define macro for application.*/
00015: #include "user_config.h"
00016:

```

porting include

- user_config.h: is used user enable/disable macros. The following table describes how you can configure macros.

Table 2. Configuring macros

No.	Name	Description
1	CORE_NO	Reserved for futures
2	EN_DEBUG_IO	Test time or toggle LED for index bootloader alive
3	EN_DEBUG_TIMER	Reserved
4	EN_ASSERT	Index enable assert or not
5	EN_UDS_DEBUG EN_TP_DEBUG EN_APP_DEBUG EN_DEBUG_FIFO	Index different layer print information. Debug one layer should enable the macro if needed. If enabled one of them initializes the UART for print. It can enable multi debug for print.
6	DebugBootloader_NOTCRC	Enable check flash driver and APP image with CRC or not
7	EN_ALG_SW EN_ALG_HW AES_SEED_LEN	Enable UDS algorithm (AES) with SW or HW. Default enabled EN_ALG_SW.
8	EN_CAN_TP EN_LIN_TP EN_ETHERNET_TP EN_OTHERS_TP	Enable TP type. This macro can be enable only one of them. If enable multi or none will trigger compiler error
9	RX_FUN_ID RX_PHY_ID TX_ID	CAN RX function ID CAN RX physical ID CAN TX message ID If enable CAN TP should config theses ID
10	RX_BOARD_ID RX_FUN_ID RX_PHY_ID TX_ID	LIN RX board NAD LIN RX function NAD LIN RX physical NAD LIN TX message NAD If enable LIN TP should config theses NAD
11	EN_CRC_HARDWARE EN_CRC_SOFTWARE	Enable calculate CRC with SW or HW. Both of them can be enabled EN_CRC_SOFTWARE must be enabled.
12	RX_BUS_FIFO	RX BUS FIFO ID and length

Table continues on the next page...

Table 2. Configuring macros (continued)

	RX_BUS_FIFO_LEN	
13	TX_BUS_FIFO TX_BUS_FIFO_LEN	TX message BUS FIFO ID and length
14	EN_SUPPORT_APP_B EN_NEWEST_APP_INVALID_JUMP_OLD_APP	Reserved for OTA/FOTA
15	DisableAllInterrupts() EnableAllInterrupts()	Disable/enable all interrupts. Bootloader stack, use the API for disable/enable all interrupts. If used interrupts, must config the APIs.
16	MCU_S12Z MCU_S32K14x MCU_TYPE	Set current MCU type. Used erase P-flash sector size and calculate erase P-flash time
17	EN_DELAY_TIME DELAY_MAX_TIME_MS	Enable enter bootloader mode max time. If APP request enter bootloader mode and not send any diagnostic command, bootloader wait max delay time. If APP is valid, then jump to APP.

- common_types.h: This file is typedef basic data type. E.g., uint32/16/8, sint32/16/8. If basic data type is conflict typedef, you should add condition to prevent compiler in this file.

```

00174: #if (!defined TYPEDEFS_H) && (! defined _STDINT) && (! defined SYS_STDINT_H) * ewl_library & S32K SDK - workaround for typedefs collisions */
00175: typedef unsigned char uint8_t;
00176: typedef unsigned short uint16_t;
00177: typedef unsigned int uint32_t;
00178:
00179: typedef volatile unsigned char vuint8_t;
00180: typedef volatile unsigned short vuint16_t; #include "stdint.h" --> define SYS_STDINT_H
00181: typedef volatile unsigned int vuint32_t;
00182: typedef volatile char vint8_t;
00183: typedef volatile short vint16_t;
00184: typedef volatile int vint32_t;
00185: #endif /* _TYPEDEFS_H */
00186:
00187: /* Old type definitions used by FECLD */
00188: typedef uint32_t uint32;
00189: typedef uint16_t uint16;
00190: typedef uint8_t uint8;
00191: #if (!defined TYPEDEFS_H) && (! defined _STDINT) && (! defined SYS_STDINT_H) * eCos & S32K SDK- workaround for typedefs collisions */
00192: typedef sint64_t int64_t;
00193: typedef sint32_t int32_t;
00194: typedef sint16_t int16_t;
00195: typedef sint8_t int8_t;
00196: #endif /* _TYPEDEFS_H */

```

- toolchain.h: redefines some key words. E.g., inline, asm, interrupt etc. Bootloader stack do not use the file key words. The file is reserved, if you have some conflicts, please modify the file.

3.3.2 HAL

- crc_hal: calculates CRC over software and hardware. CRC module based on software must be enabled (EN_CRC_SOFTWARE defined in user_config.h). Bootloader uses CRC module before initiating CRC hardware module. You can modify and calculate CRC method and the CRC table. Bootloader and APP must have the same CRC table and calculation method. If both of them are not equal, information exchanged by APP and bootloader will fail.

Table 3. API description

API/variable name	description
gs_aCrc16Tab	CRC table
boolean CRC_HAL_Init (void)	Init CRC module. If hardware CRC module is used, you must add init in the function. If init CRC hardware module is successful, return TRUE, else return FALSE.
void CRC_HAL_CreatHardwareCrc (const uint8 *i_pucDataBuf, const uint32 i_ulDataLen, uint32*m_pCurCrc)	Calculate CRC value with hardware module. i_pucDataBuf: calculate CRC data i_ulDataLen: message length m_pCurCrc: CRC value.
void CRC_HAL_CreatSoftwareCrc (const uint8_t *i_pucDataBuf, const uint32_t i_ulDataLen, uint32_t*m_pCurCrc)	Calculate CRC value with software module i_pucDataBuf: calculate CRC data i_ulDataLen: message length m_pCurCrc: CRC value.
CRC_HAL_Deinit	reserved

- flash_hal: In flash_hal.c should port

Table 4. API description

API/variable name	description
<pre>#define SECTOR_LEN (xxx) #define MAX_ERASE_SECTOR_FLASH_MS(xx) 00042: /*define a sector = bytes*/ 00043: #if (defined MCU_TYPE) && (MCU_TYPE == MCU_S32K14x) 00044: #define SECTOR_LEN (4096u) 00045: #elif (defined MCU_TYPE) && (MCU_TYPE == MCU_S122) 00046: #define SECTOR_LEN (512u) 00047: #else 00048: #error "Please config the MCU Type" 00049: #endif 00066: define eras flash sector max time 00067: ***** 00068: S32K142/144/146/148 -- max erase time 130ms 00069: MagniV S12ZVL/S32ZVM -- 21ms 00070: */ 00071: #if (defined MCU_TYPE) && (MCU_TYPE == MCU_S32K14x) 00072: #define MAX_ERASE_SECTOR_FLASH_MS (130u) 00073: #endif 00074: 00075: #if (defined MCU_TYPE) && (MCU_TYPE == MCU_S122) 00076: #define MAX_ERASE_SECTOR_FLASH_MS (21u) 00077: #endif</pre>	<p>SECTOR_LEN index is a sector size (Bytes).</p> <p>MAX_ERASE_SECTOR_FLASH_MS index erase a sector in max time (MS).</p>
<pre>#define APP_VECTOR_TABLE_OFFSET</pre>	APP vector table offset from gs_astBlockNumA/B header. The value configs with project.
#define	Reset handle/startup offset from vector table

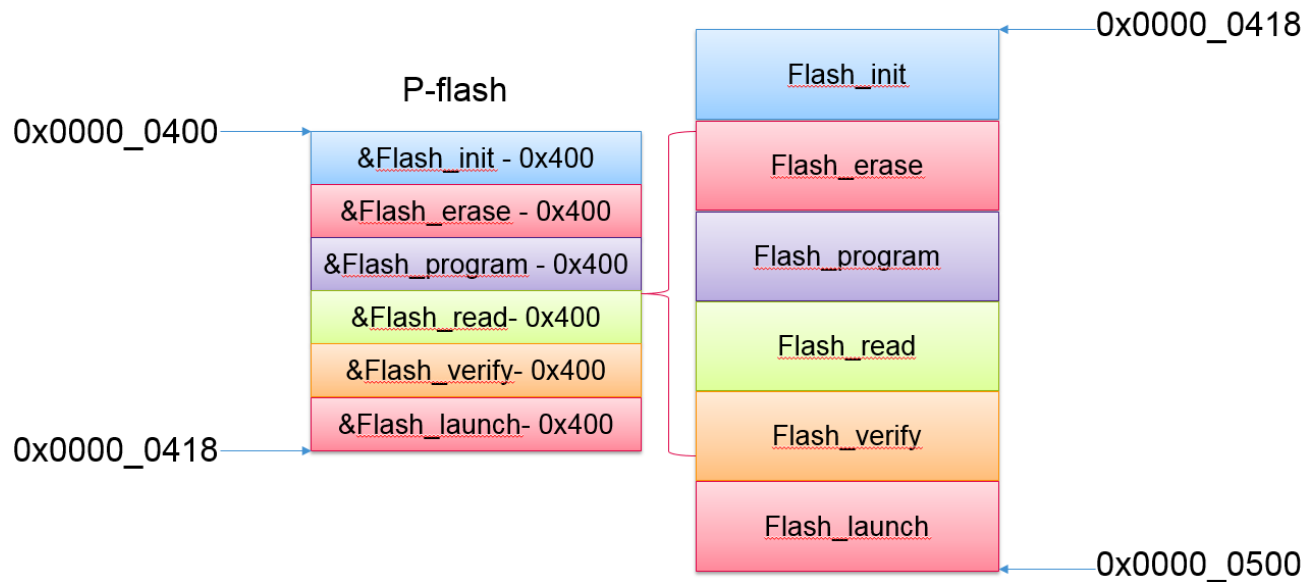
Table continues on the next page...

Table 4. API description (continued)

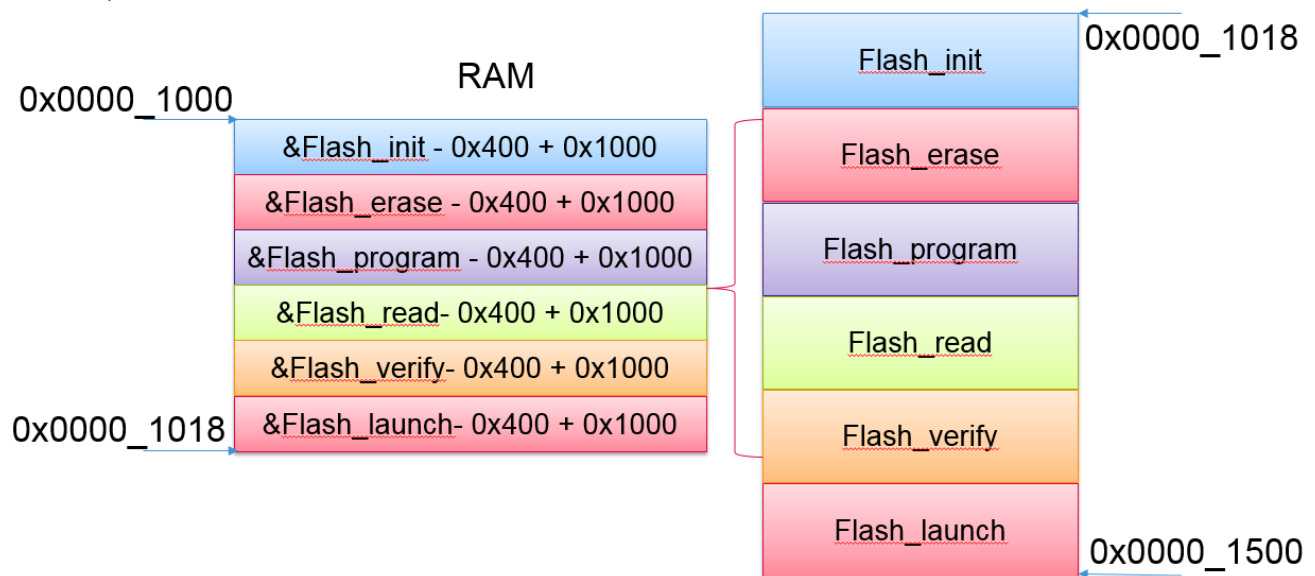
API/variable name	description
RESET_HANDLE_OFFSET	
#define RESET_HANDLER_ADDR_LEN	Reset handle pointer length. It is equal pointer length.
boolean FLASH_HAL_Init(void)	Init flash. In this function, you can calculate and store all the flash driver APIs address in flash driver table. Return initialize flash status. TRUE/FALSE.
boolean FLASH_HAL_EraseSector (const uint32 i_startAddr, const uint32 i_noEraseSectors)	Erase flash sector. Erase flash is N * sector. i_startAddr: erase flash start address i_noEraseSectors: number of erase sectors Return erase flash status (TRUE/FALSE)
boolean FLASH_HAL_WriteData (const uint32 i_startAddr, const uint8 *i_pDataBuf, const uint32 i_dataLen)	Program data in flash. Program data in flash have some conditions. E.g., program always 8 * N bytes data. If program data is not equal 8 * N, should add to 8 * N i_startAddr: program data start address i_pDataBuf: program data buf i_dataLen: program data length NOTE Program data in flash, should confirm the data length. Program data length should always equal 8 * N.
boolean FLASH_HAL_ReadData(const uint32 i_startAddr, const uint32 i_readLen, uint8 *o_pDataBuf)	Reserved
void FLASH_HAL_Deinit(void)	Reserved
boolean FLASH_HAL_RegisterFlashAPI(tFlashOperateAPI *o_pstFlashOperateAPI)	Register flash API. The API register operate flash function. For porting do not modify the API.

Flash driver compile with independent position code. It means that flash driver can copy to any RAM address and run normally, to achieve the target. Flash driver have a table. The table include some APIs like, init/erase/program/read/verify

flash memory. *flash_hal.c* call the APIs to calculate init/erase/program/read/verify flash address in RAM.



If you copy flash driver to 0x1000 (RAM) the flash driver APIs table recalculate the API start address. The APIs start address can be recalculated within Flash_init.



- timer_hal

Table 5. API description

API name	description
void TIMER_HAL_Init (void)	Init timer. Hardware timer init in this function. Timer should be set to tick in 1ms.

Table continues on the next page...

Table 5. API description (continued)

API name	description
void TIMER_HAL_1msPeriod (void)	This function need 1ms called. It may be called by ISR or main function.
boolean TIMER_HAL_Is1msTickTimeout (void)	Check if 1ms tick timeout? If timeout return TRUE, else return FALSE.
boolean TIMER_HAL_Is100msTickTimeout (void)	Check if 100 ms tick timeout? If timeout return TRUE, else return FALSE.
uint32 TIMER_HAL_GetTimerTickCnt (void)	Get timer tick counter. This API is used by random module.
void TIMER_HAL_Deinit (void)	Reserved

NOTE

Hardware timer module should call *TIMER_HAL_1msPeriod* in 1ms period. Timer needs to be configured for 1ms timeout.

- **UDS_algorithm_hal**: UDS algorithm is based on AES. If user do not used the algorithm, user can add it's own algorithm in the file.

Table 6. API description

API name/variable name	description
gs_aKey	AES key
gs_UDS_SWTimerTickCnt	Storage software timer tick counter. It is used for AES. The variable is not init when defined. Because, you get random data.
void UDS_ALG_HAL_Init (void)	Reserved for hardware AES module init.
boolean UDS_ALG_HAL_EncryptData (const uint8 *i_pPlainText, const uint32 i_dataLen, uint8 *o_pCipherText)	Encrypt data. Return encrypt status (TRUE/FALSE). i_pPlainText: input plaintext message buf i_dataLen: plaintext message length o_pCipherText: output ciphertext <div style="text-align: center;">NOTE the length of AES input and output message is equal</div>
boolean UDS_ALG_HAL_DecryptData	Decrypt data. Return decrypt status (TRUE/FALSE). i_pCipherText: input ciphertext data buff

Table continues on the next page...

Table 6. API description (continued)

API name/variable name	description
(const uint8 *i_pCipherText, const uint32 i_dataLen, uint8 *o_pPlainText)	i_dataLen: ciphertext data length o_pPlainText: output plaintext data buff <div style="text-align: center;"> NOTE Ciphertext data buff length is equal to the output ciphertext data length </div>
boolean UDS_ALG_HAL_GetRandom (const uint32 i_needRandomDataLen, uint8 *o_pRandomDataBuf)	Get random. Return get status (TRUE/FALSE). i_needRandomDataLen: need to get random data length o_pRandomDataBuf: random data buffer
void UDS_ALG_HAL_AddSWTimerTickCnt(void)	When this API is called it triggers gs_UDS_SWTimerTickCnt + 1
void UDS_ALG_HAL_Deinit(void)	Reserved

- watchdog_hal: In unified bootloader, watchdog module is used to check if the system is working or not and trigger bootloader reset.

Table 7. API description

API name	description
void WATCHDOG_HAL_Init (void)	Init watchdog module
void WATCHDOG_HAL_Fed (void)	Fed watchdog
void WATCHDOG_HAL_SystemRest (void)	Trigger system reset
void WATCHDOG_HAL_Deinit(void)	Reserved

NOTE

In BOOTLOADER_MAIN_Demo function, after every 100 ms the erase/program flash memory needs to check watchdog. You should set watchdog timeout to over 300 ms.

3.3.3 UDS_stack

UDS_stack include TP and UDS layer. For porting bootloader stack, you just need to modify xx_cfg.c.

- TP: For current version, TP include CAN and LIN. CAN TP is based on ISO15765-2 and LIN TP is based on ISO17987-2.

- TP_cfg.c: Provides basic services for TP and UDS layer. If you use CAN or LIN TP, there is no need to modify this file. To add another ISO (E.g., Ethernet/I2C...) standard in bootloader, you need to add some services in the file. The following table shows service description for TP_cfg.c.

Table 8. TP_cfg.c API description

API name	description
uint32 TP_GetConfigTxMsgID(void)	Get config TX message ID
uint32 TP_GetConfigRxMsgFUNID(void)	Get config RX function message ID
uint32 TP_GetConfigRxMsgPHYID(void)	Get config RX physical message ID
uint32 TP_GetConfigRxMsgBoardcastID(void)	Get config RX broadcast NAD for LIN
void TP_RegisterTransmittedAFrmMsgCallBack(const tpfUDSTxMsgCallBack i_pfTxMsgCallBack)	Register transmits a frame and wait for callback
void TP_DoTransmittedAFrameMsgCallBack(const uint8 i_result)	Transmits a frame callback
boolean TP_DriverWriteDataInTP (const uint32 i_RxID, const uint32 i_RxDataLen, const uint8 *i_pRxDataBuf)	When driver receives message from BUS, driver calls this API to send the message to TP layer. i_RxID: RX message ID (E.g., CAN ID: 0x7DE, LIN: NAD) i_RxDataLen: RX message length i_pRxDataBuf: RX message buf
boolean TP_DriverReadDataFromTP (const uint32 i_readDataLen, uint8 *o_pReadDataBuf, uint32 *o_pTxMsgID, uint32 *o_pTxMsgLength)	Called the API to get TP to send driver message. i_readDataLen: TX message length. For LIN it is 8 bytes. o_pReadDataBuf: message buff o_pTxMsgID: TX message ID, for CAN is CAN ID, for LIN is NAD o_pTxMsgLength: TX message length
void TP_RegisterAbortTxMsg(const void (*i_pfAbortTxMsg)(void))	When TP sends message timeout, TP calls the API to abort transmit. i_pfAbortTxMsg: abort TX message pointer
void TP_DoTxMsgSuccessfulCallback (void)	When driver transmits message, it calls the API. The API must be called. As, xx_TP_cfg.c transmit API have a parameter which shows TX message successful, if you have used the pointer, this API should not call again.

TP layer provides three APIs for driver. When driver receives a message, it calls *TP_DriverWriteDataInTP* to send the message to TP. When the message is transmitted it needs to call *TP_DoTxMsgSuccessfulCallback* for notification to TP layer. Driver needs to check TP layer, if the message needs to be transmitted again *TP_DriverReadDataFromTP*. The following figures shows the *TP_DriverWriteDataInTP*, *TP_DriverReadDataFromTP* and *TP_DoTxMsgSuccessfulCallback* example. When LIN driver receives a message from LIN BUS, it should transmit the message to TP layer.

```

00136:         case LIN_RX_COMPLETED:
00137:             TP_DriverWriteDataInTP(0x3Cu, 8u, g_aLINRxBuf);
00138:             break;

```

TP_DriverReadDataFromTP should call period. Below is the example to check message needs to be transmitted or not?

```

00259:         if(TRUE == TP_DriverReadDataFromTP(8u, g_aLINTxBuf, &msgId, &msgLength))
00260:         {
00261:             g_bIsNeedTxMsg = TRUE;
00262:
00263:             APPDebugPrintf("Read data from TP\n");
00264:         }

```

When message is transmitted to BUS, it should call *TP_DoTxMsgSuccessfulCallback* to notify the upper layer.

```

00130:         case LIN_TX_COMPLETED:
00131:             g_bIsNeedTxMsg = FALSE;
00132:             TP_DoTxMsgSuccessfulCallback();|

```

- CAN_TP_cfg.c : If you enable CAN_TP, this file configures information and modify some APIs for CAN TP layer.

CAN TP has information which needs to be configured e.g., STmin, N_As, N_Bs etc. For most of the time, you do not need to modify these parameters. The following images show the parameters.

```

00020: /*uds network layer cfg info */
00021: const tUdsCANNetLayerCfg g_stCANUdsNetLayerCfgInfo =
00022: {
00023:     1u, /*called can tp period*/
00024:     RX_FUN_ID, /*can tp rx function ID*/
00025:     RX_PHY_ID, /*can tp rx phy ID*/
00026:     TX_ID, /*can tp tx ID*/
00027:     0u, /*BS = block size*/
00028:     1u, /*STmin*/
00029:     25u, /*N_As*/
00030:     25u, /*N_Ar*/
00031:     75u, /*N_Bs*/
00032:     0u, /*N_Br*/
00033:     100u, /*N_Cs < 0.9 N_Cr*/
00034:     150u, /*N_Cr*/
00035:     0u, /*max blocking time 0ms, > 0u mean waiting send successful. equal 0 is not waiting.*/
00036:     CANTP_TxMsg, /*can tp tx*/
00037:     CANTP_RxMsg, /*can tp rx*/
00038:     CANTP_AbortTxMsg,|
00039: };

```

```

00031: typedef struct
00032: {
00033:     uint8 ucCalledPeriod; /*called CAN tp main function period*/
00034:     tUdsId xRxFunId; /*rx function ID*/
00035:     tUdsId xRxPhyId; /*Rx phy ID*/
00036:     tUdsId xTxId; /*Tx ID*/
00037:     tBlockSize xBlockSize; /*BS*/
00038:     tNetTime xSTmin; /*STmin*/
00039:     tNetTime xNAs; /*N_As*/
00040:     tNetTime xNAr; /*N_Ar*/
00041:     tNetTime xNBs; /*N_Bs*/
00042:     tNetTime xNBr; /*N_Br*/
00043:     tNetTime xNCs; /*N_Cs*/
00044:     tNetTime xNCr; /*N_Cr*/
00045:     uint32 txBlockingMaxTimeMs; /*TX message blocking max time (MS)*/
00046:     tNetTxMsg pfNetTxMsg; /*net tx message with non blocking*/
00047:     tNetRx pfNetRx; /*net rx*/
00048:     tpfAbortTxMsg pfAbortTXMsg; /*abort tx message*/
00049: } tUdsCANNetLayerCfg;

```

The following table describes the APIs.

Table 9. CAN_TP_cfg.c API description

API name	description
uint8 CANTP_TxMsg(const tUdsId i_xTxId, const uint16 i_DataLen, const uint8* i_pDataBuf, const tpfNetTxCallBack i_pfNetTxCallBack, const uint32 txBlockingMaxtime)	CAN TP TX message. This function transmits message to hardware API. i_xTxId: TX message ID i_DataLen: TX message length i_pDataBuf: message buff i_pfNetTxCallBack: TX message successful callback txBlockingMaxtime: reserved
uint8 CANTP_RxMsg(tUdsId * o_pxRxId, uint8 * o_pRxDataLen, uint8 *o_pRxBuf)	CAN TP RX message from FIFO. Usually the API does not need to be modified. CAN driver receives message from BUS and checks ID validity and writes the message in FIFO. o_pxRxId: RX message ID o_pRxDataLen: RX message data length o_pRxBuf: RX message buff
tUdsId CANTP_GetConfigTxMsgID(void)	Get CAN TP config TX message ID
tUdsId CANTP_GetConfigRxMsgFUNID(void)	Get CAN TP config RX function ID
boolean CANTP_IsReceivedMsgIDValid(const uint32 i_receiveMsgID)	Check if the received message ID is valid? i_receiveMsgID: RX message ID
tUdsId CANTP_GetConfigRxMsgPHYID(void)	Get CAN TP config RX message physical ID
tNetTxMsg CANTP_GetConfigTxHandle(void)	Reserved
tNetRx CANTP_GetConfigRxHandle(void)	Reserved
void CANTP_AbortTxMsg(void)	When transmit message timeout, TP layer aborts hardware transmit message. This function is used to abort the transmitted message over hardware.
void CANTP_RegisterAbortTxMsg(const tpfAbortTxMsg i_pfAbortTxMsg)	Register abort TX message call back.
boolean CANTP_DriverWriteDataInCANTP(const uint32 i_RxID, const uint32 i_dataLen, const uint8 *i_pDataBuf)	This API is provided for TP_cfg.c. It is used by driver to receive message and transmit the message to TP layer.
boolean CANTP_DriverReadDataFromCANTP(const uint32 i_readDataLen, uint8 *o_pReadDataBuf, tPTxMsgHeader *o_pstTxMsgHeader)	This API is provided for TP_cfg.c. It is used by driver to read message from TP and where it needs to be sent.
void CANTP_DoTxMsgSuccessfulCallBack(void)	This API is provided for TP_cfg.c. It is used by driver to send message successfully and call callback the API.
static boolean CANTP_ClearTXBUSFIFO(void)	This API is used for clear TX BUS FIFO.

- LIN_TP_cfg.c: If you enable LIN_TP, you should configure and modify some functions for LIN TP layer. LIN TP has some information related to time that should be configured, e.g., STmin, N_As, N_Bs etc. For most of the time, you do not need to modify these parameters. The following images shows the parameters.

```

00022: /*uds network layer cfg info */
00023: const tUdsLINNetLayerCfg g_stUdsLINNetLayerCfgInfo =
00024: {
00025:     1u,          /*called LIN tp period*/
00026:     RX_BOARD_ID, /*LIN TP rx boardcast IDs*/
00027:     RX_FUN_ID,   /*LIN tp rx function ID*/
00028:     RX_PHY_ID,   /*LIN tp rx phy ID*/
00029:     TX_ID,       /*LIN tp tx ID*/
00030:     0u,          /*BS = block size*/
00031:     0u,          /*STmin*/
00032:     300u,        /*N_As*/
00033:     300u,        /*N_Ar*/
00034:     300u,        /*N_Bs*/
00035:     0u,          /*N_Br*/
00036:     300u,        /*N_Cs*/
00037:     500u,        /*N_Cr*/
00038:     0u,          /*max blocking time 0ms, > 0u mean waiting send successful. equal 0 is not waiting.*/
00039:     LINTP_TxMsg, /*LIN tp tx*/
00040:     LINTP_RxMsg, /*LIN tp rx*/
00041:     LINTP_AbortTxMsg, /*abort tx message*/
00042: };

-----
00031: typedef struct
00032: {
00033:     unsigned char ucCalledPeriod; /*called LIN tp main function period*/
00034:     tUdsId xRxBoardcastId;        /*rx boardcast ID*/
00035:     tUdsId xRxFunId;              /*rx function ID*/
00036:     tUdsId xRxPhyId;              /*Rx phy ID*/
00037:     tUdsId xTxId;                 /*Tx ID*/
00038:     tBlockSize xBlockSize;        /*BS*/
00039:     tNetTime xSTmin;              /*STmin*/
00040:     tNetTime xNAs;                /*N_As*/
00041:     tNetTime xNAr;                /*N_Ar*/
00042:     tNetTime xNBs;                /*N_Bs*/
00043:     tNetTime xNBr;                /*N_Br*/
00044:     tNetTime xNCs;                /*N_Cs*/
00045:     tNetTime xNCr;                /*N_Cr*/
00046:     uint32 txBlockingMaxTimeMs; /*TX message blocking max time (MS)*/
00047:     tNetTxMsg pfNetTxMsg; /*net tx message with non blocking*/
00048:     tNetRx pfNetRx;         /*net rx*/
00049:     tpfAbortTxMsg pfAbortTxMsg; /*abort tx message*/
00050: } ? end {anontUdsLINNetLayerCfg} ? tUdsLINNetLayerCfg;

```

The following table describes the APIs

Table 10. LIN_TP_cfg.c API description

API name	Description
uint8 LINTP_TxMsg(const tUdsId i_xTxId, const uint16 i_DataLen, const uint8* i_pDataBuf, const tpfNetTxCallBack i_pfNetTxCallBack, const uint32 txBlockingMaxtime)	<p>LIN TP TX message over hardware. For LIN TP this function is used to transmit message to FIFO. LIN driver read the message from FIFO and transmit message over hardware. Usually this function is not modified.</p> <p>i_xTxId: TX message NAD</p> <p>i_DataLen: TX message length</p> <p>i_pDataBuf: TX message buff</p> <p>i_pfNetTxCallBack: TX message successful over hardware call this callback</p>

Table continues on the next page...

Table 10. LIN_TP_cfg.c API description (continued)

API name	Description
	txBlockingMaxtime: reserved for blocking send
uint8 LINTP_RxMsg(tUdsId *o_pRxId, uint8 *o_pRxDataLen, uint8 *o_pRxBuf)	RX message from FIFO. LIN TP read message from FIFO. LIN driver receives message from BUS and the ID and NAD is valid, write the message in FIFO. Usually the function is not modified. o_pRxId: RX message NAD o_pRxDataLen: RX message length o_pRxBuf: RX message buff
tUdsId LINTP_GetConfigTxMsgID(void)	Get LIN TP config TX message NAD
tUdsId LINTP_GetConfigRxMsgFUNID(void)	Get LIN TP config RX message function NAD
tUdsId LINTP_GetConfigRxMsgPHYID(void)	Get LIN TP config RX physical NAD
tUdsId LINTP_GetConfigRxMsgBroadcastID(void)	Get LIN TP config RX message broadcast NAD
tNetTxMsg LINTP_GetConfigTxHandle(void)	Get LIN TP config TX handle
tNetRx LINTP_GetConfigRxHandle(void)	Get LIN TP config RX handle
boolean LINTP_IsReceivedMsgIDValid(const uint32 i_receiveMsgID)	Check LIN TP RX message NAD validity?
void LINTP_AbortTxMsg(void)	When transmit message timeout, TP layer aborts hardware transmit message. This function is used abort transmit message over hardware. Usually LIN driver transmit message is based on ISR. LIN driver should store and send data in global variable, when it receives diagnostic ID, LIN driver transmit the message to LIN BUS. If global variable stores the previous data, the UI/host will receive error message. When TP transmit message, timeout will call the API to abort TX message (E.g., clear global variable).
boolean LINTP_DriverWriteDataInLINTP(const uint32 i_RxNAD, const uint32 i_dataLen, const uint8 *i_pDataBuf)	This API is provided for TP_cfg.c. When driver receives message from BUS, it calls the API to transmit message to TP layer.
boolean LINTP_DriverWriteDataInLINTP(const uint32 i_RxNAD, const uint32 i_dataLen, const uint8 *i_pDataBuf)	This API is provided for TP_cfg.c. This API provides read message from TX FIFO.
void LINTP_RegisterAbortTxMsg(const tpfAbortTxMsg i_pfAbortTxMsg)	This API is provided for TP_cfg.c. Register abort TX message.
void LINTP_DoTxMsgSuccessfulCallBack(void)	This API is provided for TP_cfg.c. When transmit message is successful, it calls the API.
static boolean LINTP_ClearTXBUSFIFO(void)	This API is used to clear TX BUS FIFO.

The below image is a demo for LIN TP abort TX message API, in the API set the global data status to invalid.


```

00222: void LINBUSAbortTxMsg(void)
00223: {
00224:     g_bIsNeedTxMsg = FALSE;
00225:     g_aLINTxBuf[0u] = 0xFFu;
00226:
00227:     APPDebugPrintf("%s\n", __func__);
00228: }

```

- UDS: UDS layer is based on ISO14229. For porting you need to modify *uds_app_cfg.c*. In this file, all the UDS diagnostic services needs to be configured and defined. All the diagnostic services will not be called by the user. In this file, you should modify the table over download firmware process. The table element are shown in the following figure.

```

00020: typedef struct UDSServiceInfo
00021: {
00022:     uint8 SerNum; /*service num. eg 0x3e/0x87...*/
00023:     uint8 SessionMode; /*default session / program session / extend session*/
00024:     uint8 SupReqMode; /*support physical / function addr*/
00025:     uint8 ReqLevel; /*request level.Lock/unlock*/
00026:     void (*pfSerNameFun) (struct UDSServiceInfo*, tUdsAppMsgInfo *);
00027: } tUDSService;

```

Every item (diagnostic service) includes information as structure.

For downloading firmware process, NXP provide a basic process. If you need to move, add or delete any service, you can modify the table. Only diagnostic service in table can be run by UDS manager layer.

```

00280: /*dig service config table*/
00281: const static tUDSService gs_astUDSService[] =
00282: {
00283:     /*diagnose mode control*/
00284:     {
00285:         0x10u,
00286:         DEFALUT_SESSION | PROGRAM_SESSION | EXTEND_SESSION,
00287:         SUPPORT_PHYSICAL_ADDR | SUPPORT_FUNCTION_ADDR,
00288:         NONE_SECURITY,
00289:         DigSession
00290:     },
00291:
00292:     /*communication control*/
00293:     {
00294:         0x28u,
00295:         DEFALUT_SESSION | PROGRAM_SESSION | EXTEND_SESSION,
00296:         SUPPORT_PHYSICAL_ADDR | SUPPORT_FUNCTION_ADDR,
00297:         NONE_SECURITY,
00298:         CommunicationControl
00299:     },
00300:
00301:     /*control DTC setting*/
00302:     {
00303:         0x85u,
00304:         DEFALUT_SESSION | PROGRAM_SESSION | EXTEND_SESSION,
00305:         SUPPORT_PHYSICAL_ADDR | SUPPORT_FUNCTION_ADDR,
00306:         NONE_SECURITY,
00307:         ControlDTCSetting
00308:     },
00309:
00310:     /*security access*/
00311:     {
00312:         0x27u,
00313:         PROGRAM_SESSION,
00314:         SUPPORT_PHYSICAL_ADDR,
00315:         NONE_SECURITY,
00316:         SecurityAccess
00317:     },
00318:
00319:     /*write data by identifier*/
00320:     {
00321:         0x2Eu,
00322:         PROGRAM_SESSION,
00323:         SUPPORT_PHYSICAL_ADDR,
00324:         SECURITY_LEVEL_1,
00325:         WriteDataByIdentifier
00326:     },

```

All the diagnostic services have the same input parameters.

```

00025: void (*pfSerNameFun) (struct UDSServiceInfo*, tUdsAppMsgInfo *);
00026:

```

Example.

```

00283:     /*diagnose mode control*/
00284:     {
00285:         0x10u,
00286:         DEFALUT_SESSION | PROGRAM_SESSION | EXTEND_SESSION,
00287:         SUPPORT_PHYSICAL_ADDR | SUPPORT_FUNCTION_ADDR,
00288:         NONE_SECURITY,
00289:         DigSession
00290:     },

```

0x10 is the service number. The number is from UDS standard (ISO14229). DEFALUT_SESSION | PROGRAM_SESSION | EXTEND_SESSION configures which session can request the service. SUPPORT_PHYSICAL_ADDR | SUPPORT_FUNCTION_ADDR configures which ID can request the service (only support function and physical ID). NONE_SECURITY configures which security level can request. DigSession points the service address. The following figure is the diagnostic session, if you want more information please check the ISO14229.

```

00405: static void DigSession(struct UDSServiceInfo* i_pstUDSServiceInfo, tUdsAppMsgInfo *m_pstPDUMsg)
00406: {
00407:     uint8 ucRequestSubfunction = 0u;
00408:     ASSERT(NULL_PTR == m_pstPDUMsg);
00409:     ASSERT(NULL_PTR == i_pstUDSServiceInfo);
00410:
00411:     ucRequestSubfunction = m_pstPDUMsg->aDataBuf[1u];
00412:
00413:     /*set send postive message*/
00414:     m_pstPDUMsg->aDataBuf[0u] = i_pstUDSServiceInfo->SerNum + 0x40u;
00415:     m_pstPDUMsg->aDataBuf[1u] = ucRequestSubfunction;
00416:     m_pstPDUMsg->xDataLen = 2u;
00417:
00418:     /*sub function*/
00419:     switch(ucRequestSubfunction)
00420:     {
00421:     case 0x01u : /*default mode*/
00422:     case 0x81u :
00423:         SetCurrentSession(DEFALUT_SESSION);
00424:         if(0x81u == ucRequestSubfunction)
00425:         {
00426:             m_pstPDUMsg->xDataLen = 0u;
00427:         }
00428:         break;
00429:
00430:     case 0x02u : /*program mode*/
00431:     case 0x82u :
00432:         SetCurrentSession(PROGRAM_SESSION);
00433:         if(0x82u == ucRequestSubfunction)
00434:         {
00435:             m_pstPDUMsg->xDataLen = 0u;
00436:         }
00437:
00438:         /*set slave mcu in bootloader*/
00439:         //SetSlaveMcuInBootloader();
00440:
00441:         /*restart s3server time*/
00442:         RestartS3Server();
00443:         break;
00444:
00445:     case 0x03u : /*extend mode*/
00446:     case 0x83u :
00447:         SetCurrentSession(EXTEND_SESSION);
00448:         if(0x83u == ucRequestSubfunction)
00449:         {
00450:             m_pstPDUMsg->xDataLen = 0u;
00451:         }
00452:
00453:         /*restart s3server time*/
00454:         RestartS3Server();
00455:         break;
00456:     }
00457: }
00458:
00459:
00460:
00461:
00462:
00463:
00464:
00465:

```

service information

RX and TX message information

sub function

set current session based on ISO14229

TX message length = 0/not TX message data

RX diagnostic message, so restart S3 server time

3.3.4 Boot

Boot module includes *boot.c* and *boot_cfg.c*. Boot module checks APP validity, jumps to APP (or not) and manage exchange of information. Usually user can not modify *boot.c*. In *boot_cfg.c* there are two important features for porting, first one is jump to APP reset handler address and the second is to configure, exchange information address and check valid value.

- *Boot_cfg.c*
- Two scenarios for APP and bootloader to exchange information

Scenario #1 APP receives updated firmware command. APP follows these steps:

1. Send request, enter bootloader mode
2. Update exchange information CRC

3. Send request over UDS layer
4. Trigger MCU reset over watchdog

Enter bootloader, check request enter bootloader was set and CRC is valid

1. Clear request enter bootloader mode
2. Update exchange information CRC
3. Send enter bootloader mode command to UI/host over UDS

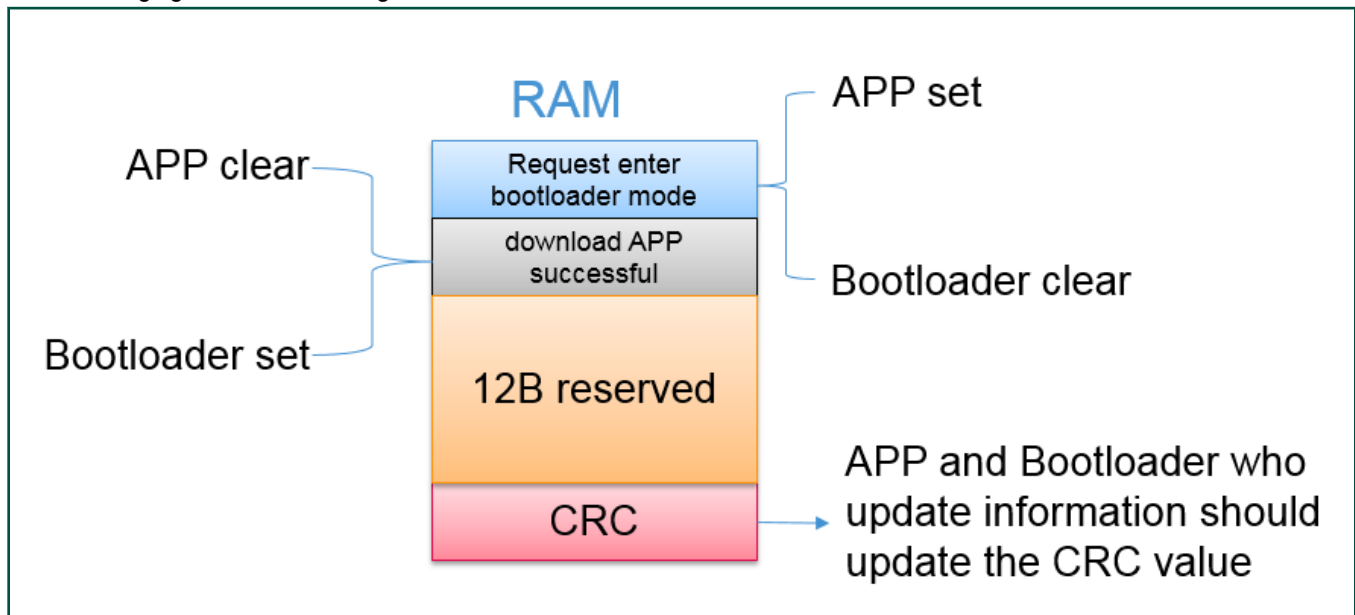
Scenarios #2 successful bootloader firmware update. Bootloader follows these steps:

1. Set APP update (success)
2. Update exchange information CRC
3. Send request over UDS protocol to UI/host
4. Watchdog reset

Enter APP check the following:

1. Check APP update is successful and exchange information CRC is valid
2. Clear APP update is successful, flag and update exchange information CRC
3. Send positive message over UDS layer to UI/host

The following figure shows exchange of information:



Boot_cfg.c provide services for boot, like APP and bootloader exchange information address config and value, check request enter bootloader mode.

Table 11. Boot_cfg.c API description

API/variable name	Description
<pre> 00043: typedef struct 00044: { 00045: uint8 infoDataLen; 00046: uint8 requestEnterBootloader; 00047: uint8 downloadAPPSuccessful; 00048: uint32 infoStartAddr; 00049: uint32 requestEnterBootloaderAddr; 00050: uint32 downloadAppSuccessfulAddr; 00051: }tBootInfo; 00052: 00053: const static tBootInfo gs_stBootInfo = { 00054: 16u, 00055: 0x5Au, 00056: 0xA5u, 00057: 0x20006FF0u, 00058: 0x20006FF1u, 00059: 0x20006FF0u, 00060: }; </pre> <p>Config the address for different platforms (infoStartAddr, requestEnterBootloaderAddr and downloadAppSuccessfulAddr)</p>	<p>infoDataLen // information length</p> <p>requestEnterBootloader// request enter bootloader valid value</p> <p>downloadAPPSuccessful//download APP successful valid value</p> <p>infoStartAddr//information start address</p> <p>requestEnterBootloaderAddr//request enter bootloader address</p> <p>downloadAppSuccessfulAddr//download APP successful address</p> <p style="text-align: center;">NOTE</p> <p>The information address, value and CRC must have the same APP.</p>
void SetDownloadAppSuccessful(void)	Set download APP successful
boolean IsRequestEnterBootloader(void)	Does request enter bootloader mode? Return TRUE/FALSE
void ClearRequestEnterBootloaderFlag(void)	Clear request enter bootloader mode
boolean Boot_IsPowerOnTriggerReset(void)	The function is used to check power on trigger reset or not? Return TRUE/FALSE.
void Boot_PowerONClearAllFlag(void)	When Powered on, clears all exchange information for ECC.
void Boot_RemapApplication(void)	Reserved
void Boot_JumpToApp (const uint32 i_AppAddr)	Jump to APP. This function is used to jump to APP. i_AppAddr: jump to APP address.
boolean Boot_IsInfoValid(void)	Check information validity? Return TRUE/FALSE
uint16 Boot_CalculateInfoCRC(void)	Calculate information CRC. Return CRC value.

3.3.5 Demo

Demo folder include *bootloader_main.c* and *bootloader_main.h*. These files provides information on how to call other modules.

Table 12. API description

API name	Description
void BOOTLOADER_MAIN_Init (void (*pfBSP_Init)(void), void (*pfAbortTxMsg)(void))	Init all the bootloader module. For this function, you should add clock, CAN/LIN init etc. pfBSP_Init: pointer function. It includes BSP init. BSP should include: clock/BUS(CAN/LIN)/flash init. Flash initialize is not important, because for flash driver initialize there is a method for multi-initialization. pfAbortTxMsg: pointer function is used to abort TX message and clear some global variables. If you have not used, please set to NULL_PTR;
void BOOTLOADER_MAIN_Demo (void)	Bootloader main function. It called by user in main. E.g., Void main(void) { for(;;){BOOTLOADER_MAIN_Demo();} }

The following figure shows BSP Init and TX message abort demo.

```

00181: void BSP_init(void)
00182: {
00183:     CLOCK_SYS_Init(g_clockManConfigsArr, CLOCK_MANAGER_CONFIG_CNT, g_clockManCallbacksArr, CLOCK_MANAGER_CALLBACK_CNT);
00184:     CLOCK_SYS_UpdateConfiguration(OU, CLOCK_MANAGER_POLICY_AGREEMENT);
00185:
00186:     PINS_DRV_Init(NUM_OF_CONFIGURED_PINS, g_pin_mux_InitConfigArr);
00187:     /* Set LED pin direction as output */
00188:     PINS_DRV_SetPinsDirection(LED_GPIO_PORT, (1 << LED0_PIN_INDEX));
00189:     /* Turn off LED0 */
00190:     PINS_DRV_SetPins(LED_GPIO_PORT, (1 << LED0_PIN_INDEX));
00191:
00192:     LPIT_DRV_Init(INST_LPIT1, &lpit1_InitConfig);
00193:     /* Initialize LPIT channel 0 and configure it as a periodic counter
00194:      * which is used to generate an interrupt every second.
00195:      */
00196:     LPIT_DRV_InitChannel(INST_LPIT1, LPIT_CHANNEL, &lpit1_ChnConfig0);
00197:
00198:     /* Install LPIT_ISR as LPIT interrupt handler */
00199:     INT_SYS_InstallHandler(LPIT_Channel_IRQn, &LPIT_ISR, (isr_t *)0);
00200:
00201:     /* Start LPIT0 channel 0 counter */
00202:     LPIT_DRV_StartTimerChannels(INST_LPIT1, (1 << LPIT_CHANNEL));
00203:
00204:     LPTMR_DRV_Init(INST_LPTMR1, &lpTmr1_config0, false);
00205:     INT_SYS_InstallHandler(LPTMR0_IRQn, LPTMR_ISR, (isr_t *)NULL);
00206:     INT_SYS_EnableIRQ(LPTMR0_IRQn);
00207:     LPTMR_DRV_StartCounter(INST_LPTMR1);
00208:
00209:     //Enable TIJ1027
00210:     PINS_DRV_WritePin(PTE, 9, 1);
00211:     LIN_DRV_Init(INST_LIN1, &lin1_InitConfig0, &lin1_State);
00212:     LIN_DRV_InstallCallback(INST_LIN1, LIN_Callback);
00213:     LIN_DRV_EnableIRQ(INST_LIN1);
00214:     INT_SYS_EnableIRQ(LPUART2_RxTx_IRQn);
00215:     INT_SYS_EnableIRQGlobal();
00216:     //PINS_DRV_WritePin(LED_GPIO_PORT, 0, 1u);
00217:
00218:     InitFlash();
00219:
00220: } ? end BSP_init ?

```

```

00222: void LINBUSAbortTxMsg(void)
00223: {
00224:     g_bIsNeedTxMsg = FALSE;
00225:     g_aLINTxBuf[0u] = 0xFFu;
00226:
00227:     APPDebugPrintf("%s\n", __func__);
00228: }

```

3.3.6 Debug

Debug include bootloader_debug.c, debug_IO.c and debug_timer.c. debug_timer.c is reserved.

Bootloader_debug.c includes debug IO, debug timer and debug print.

Table 13. Bootloader_debug.c API description

API name	Description
void BOOTLOADER_DEBUG_Init (void)	Init debug IO, timer and print. Usually do not modify for porting.
void Bootloader_DebugPrintInit (void)	UART hardware init.
void Bootloader_DebugPrint (const char *fmt, ...)	Print message over UART. Usually porting need to replace hardware transmit message function.

Debug_IO.c

Table 14. Debug_IO.c API description

API name	Description
void DEBUG_IO_Init (void)	If enable debug_IO, should add debug IO init in this function.
void DEBUG_IO_Deinit(void)	Reserved
void DEBUG_IO_SetDebugIOLow (void)	Set output debug IO low level
void DEBUG_IO_SetDebugIOHigh (void)	Set output debug IO high level

Table continues on the next page...

Table 14. Debug_IO.c API description (continued)

API name	Description
void DEBUG_IO_ToggleDebugIO (void)	Toggle debug IO

3.3.7 FIFO

FIFO is used by different layers for information exchange. FIFO module can not be modified. You just only need to config FIFO length or support application of max FIFO number. It mean's that FIFO support max storage data. If you apply length over total data length in FIFO, the length apply will fail.

The following figure shows the config information.

```

00042: #ifdef EN_LIN_TP
00043: #define FIFO_NUM (4u) /*Fifo num*/
00044: #define TOTAL_FIFO_BYTES (450u) /*config total bytes*/
00045: #else
00046: #define FIFO_NUM (3u) /*Fifo num*/
00047: #define TOTAL_FIFO_BYTES (600u + 99u) /*config total bytes*/
00048: #endif

```

3.3.8 FLS_APP

FLS_APP module uses CRC, UDS and watchdog APIs. FLS_APP module can not be modified. Default program FLASH memory is 128 bytes. If you need to change the program bytes or config finger, print length in FLS_APP.h can modify it.

```

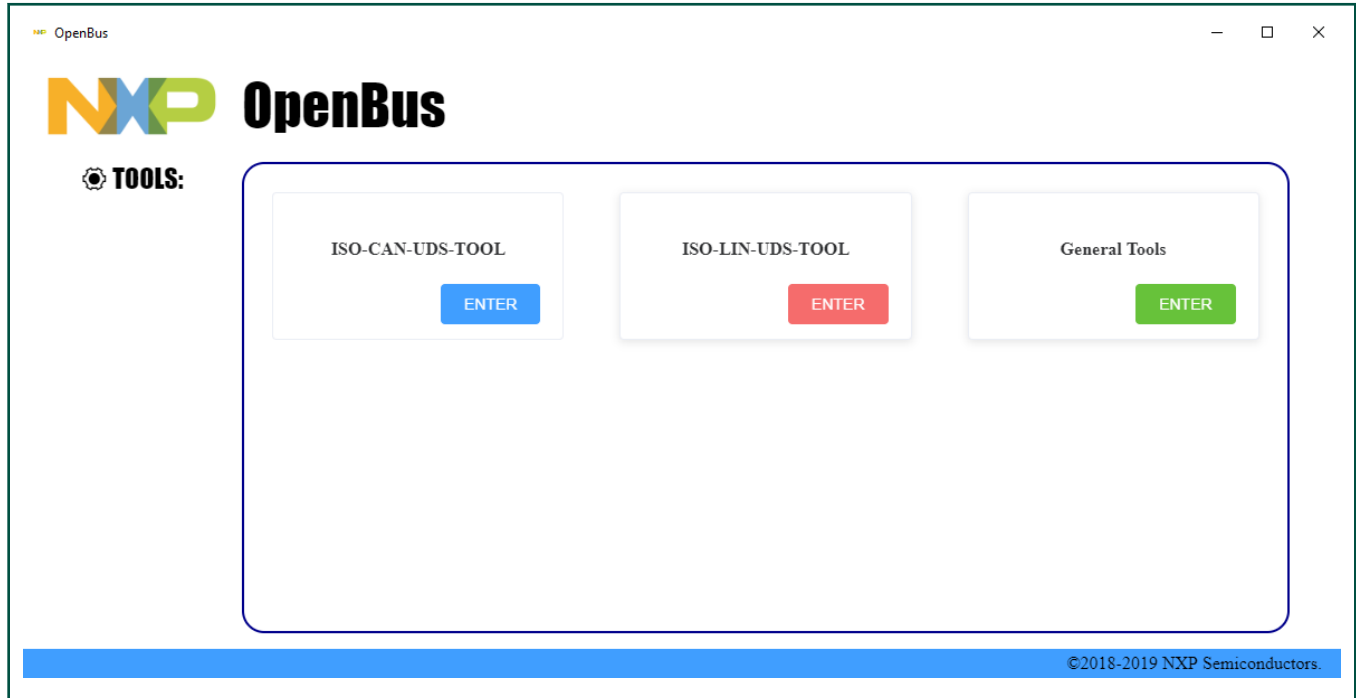
00007: /*every program flash size*/
00008: #define PROGRAM_SIZE (128u)
00009:
00010: /*Flash finger print length*/
00011: #define FL_FINGER_PRINT_LENGTH (17u)|

```


Chapter 4

How to install UI/host

The PC tool is useful to use, without installing. You just need to enter the folder and open NxpOpenBus.exe program.



NOTE

The tool is tested on Window versions - Window10 64bit, Window7 64bit.

Chapter 5

How to setup UI/host

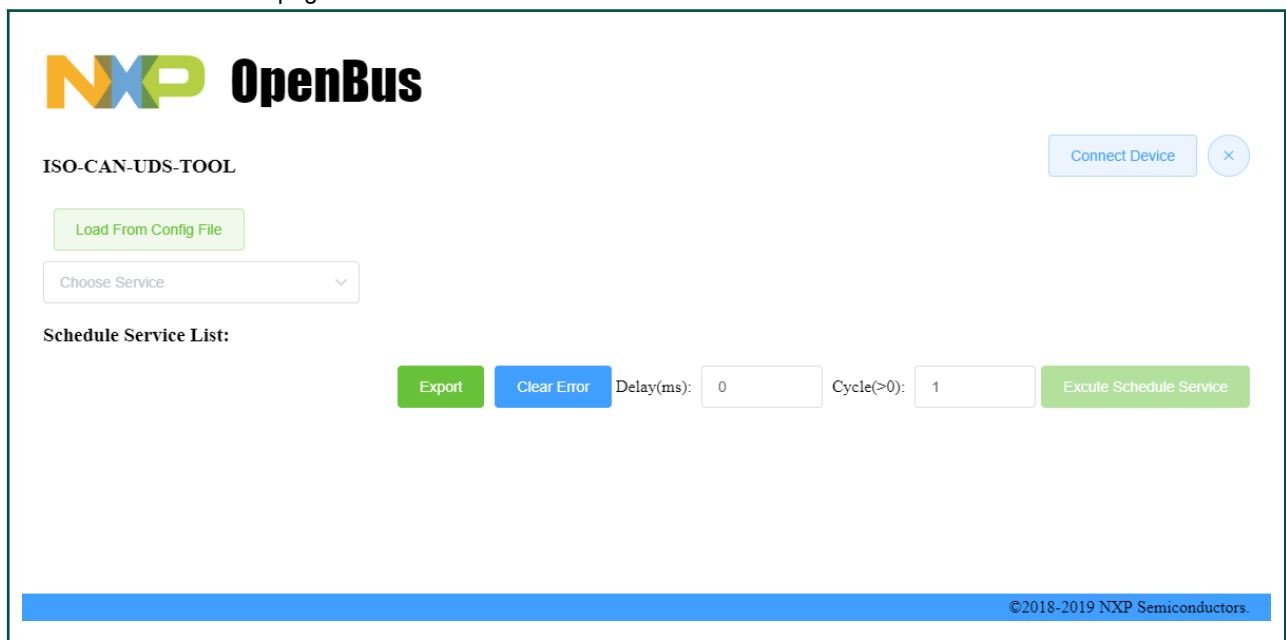
There are three feature in 0.6 Version

- ISO-CAN-UDS-TOOL
- ISO-LIN-UDS-TOOL
- General Tools.

5.1 ISO-CAN-UDS tool

This feature upgrades ECU firmware in CAN, CAN-FD is currently not supported. The following steps needs to be followed:

1. Click to enter the feature page.



The screenshot shows the NXP OpenBus ISO-CAN-UDS-TOOL interface. At the top left is the NXP OpenBus logo. Below it, the text "ISO-CAN-UDS-TOOL" is displayed. On the right side, there is a "Connect Device" button and a close button (X). Below the title, there is a green button labeled "Load From Config File". Underneath that is a dropdown menu labeled "Choose Service". Below the dropdown menu, the text "Schedule Service List:" is shown. To the right of this text are several controls: a green "Export" button, a blue "Clear Error" button, a text input field for "Delay(ms):" with the value "0", a text input field for "Cycle(>0):" with the value "1", and a green "Excute Schedule Service" button. At the bottom right of the interface, there is a blue bar containing the copyright text "©2018-2019 NXP Semiconductors."

2. Connect: You need to connect PEAK-CAN, then click on Connect Device.

Choose the device, speed, enter the physical(must) address and enter the functional(option) address.

The image displays two screenshots of a 'Connect' dialog box, illustrating the configuration steps for connecting a device.

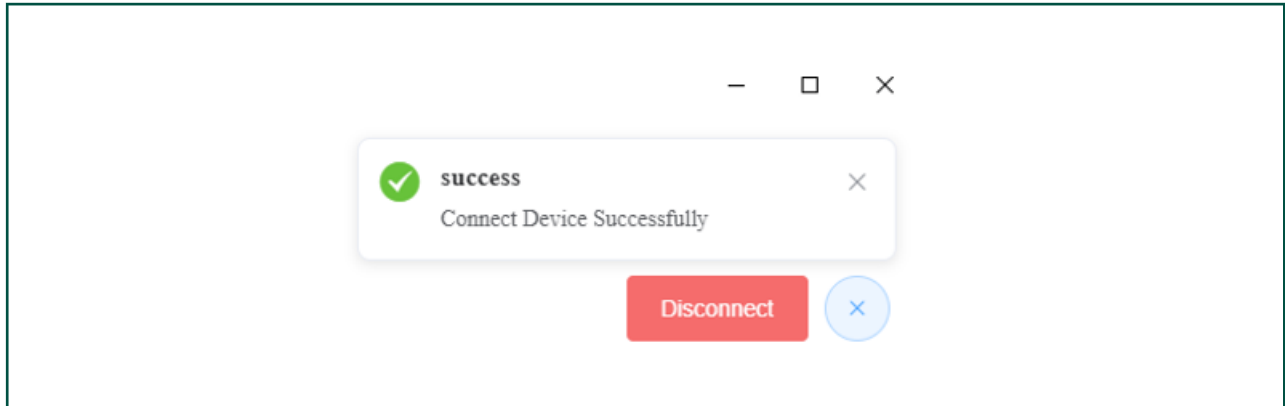
Top Screenshot (Initial State):

- Device:** * Device: Choose Device (dropdown menu)
- Speed:** * Speed: Choose Speed (dropdown menu)
- Physical Address:** TA, SA, RA (buttons)
- Functional Address:** TA, SA, RA (buttons)
- Address Protocol:** ISO_15765_2_29B (dropdown menu)
- CONNECT** (blue button)

Bottom Screenshot (Configured State):

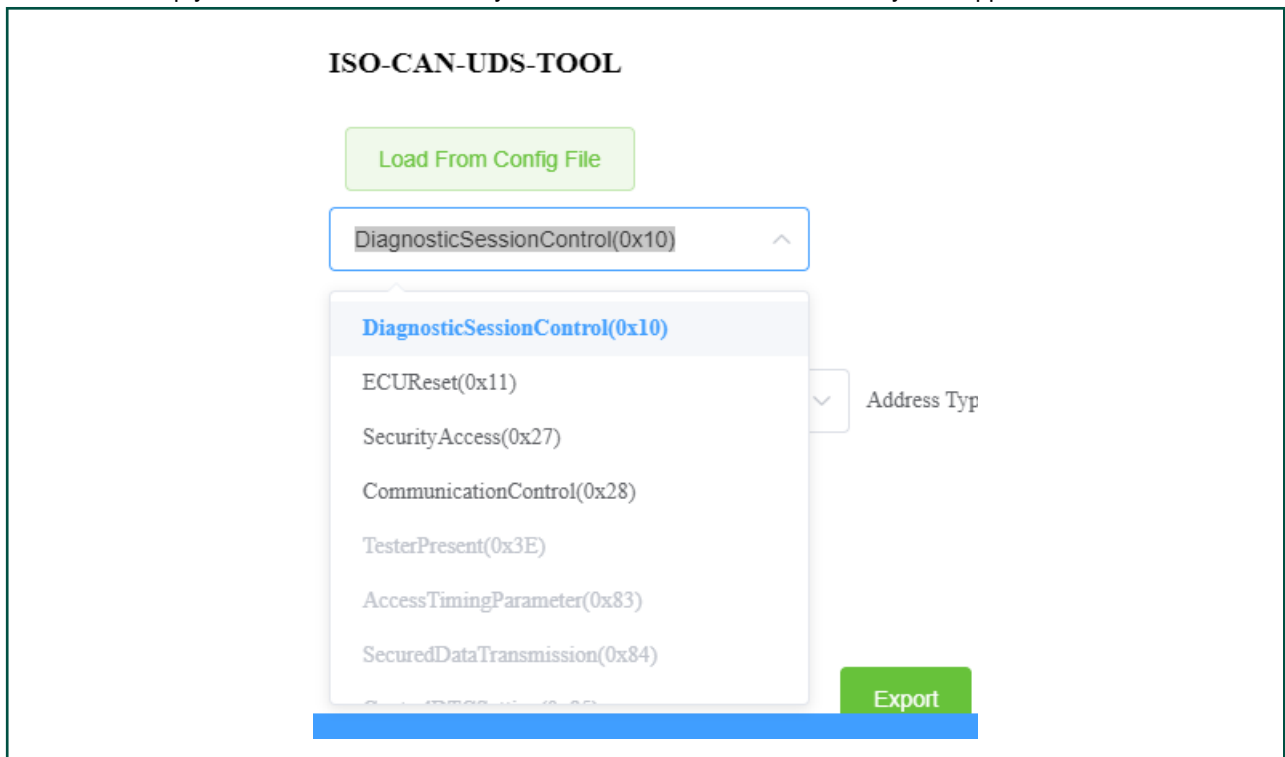
- Device:** * Device: USBBUS1 (dropdown menu)
- Speed:** * Speed: 500 kBit/sec (dropdown menu)
- Physical Address:** 55, 35, RA (buttons; 55 and 35 are highlighted with green borders)
- Functional Address:** TA, SA, RA (buttons)
- Address Protocol:** ISO_15765_2_29B (dropdown menu)
- CONNECT** (blue button)

If successfully connected, you will see the following icon.

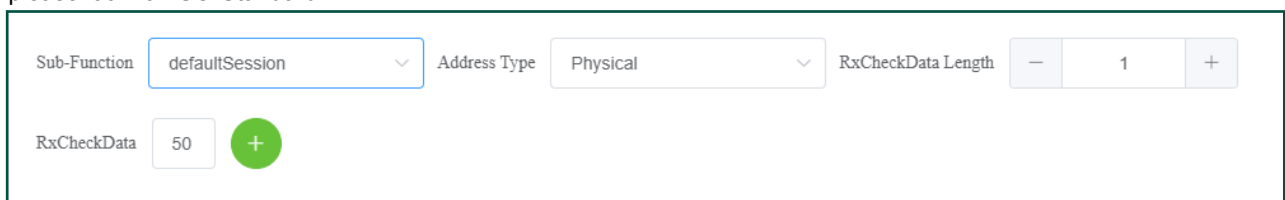


3. Add Schedule: Every step-in upgrades depends upon you. You can add, change and export your configuration. The next time when you log in, you can import your old configuration.

Click on the empty bar and chose the service you need. Some services are currently not supported.



In the following image, the DiagnosticSessionControl(0x10) is used as an example. If you want to know more about service, please look for ISO-Standard.



You should choose sub-function, address type and RX data check length, if the RX check length is zero, then the service is always successful.

If you want to add the service to schedule, just click on the green (+) bar.

Schedule Service List:

Export Clear Error Delay(ms): 0 Cycle(>0): 1 Execute Schedule Service

0:	Service ID: 10	Service Name: DiagnosticSessionControl	Address Type: Physical	Sub-Function ID: 1	Sub-Function Name: defaultSession
rxData: ["50"]					

⬆ ⬇ 🗑 ✖ ➡

The schedule has a service and id is 0. Every service has a functional bar as shown in the following figure.

⬆ ⬇ 🗑 ✖ ➡

The icons displayed in the above figure are used for the following purpose:

- Move up the service.
- Move down the service.
- Delete the service from schedule.
- Clear the service error information.
- Execute this step.

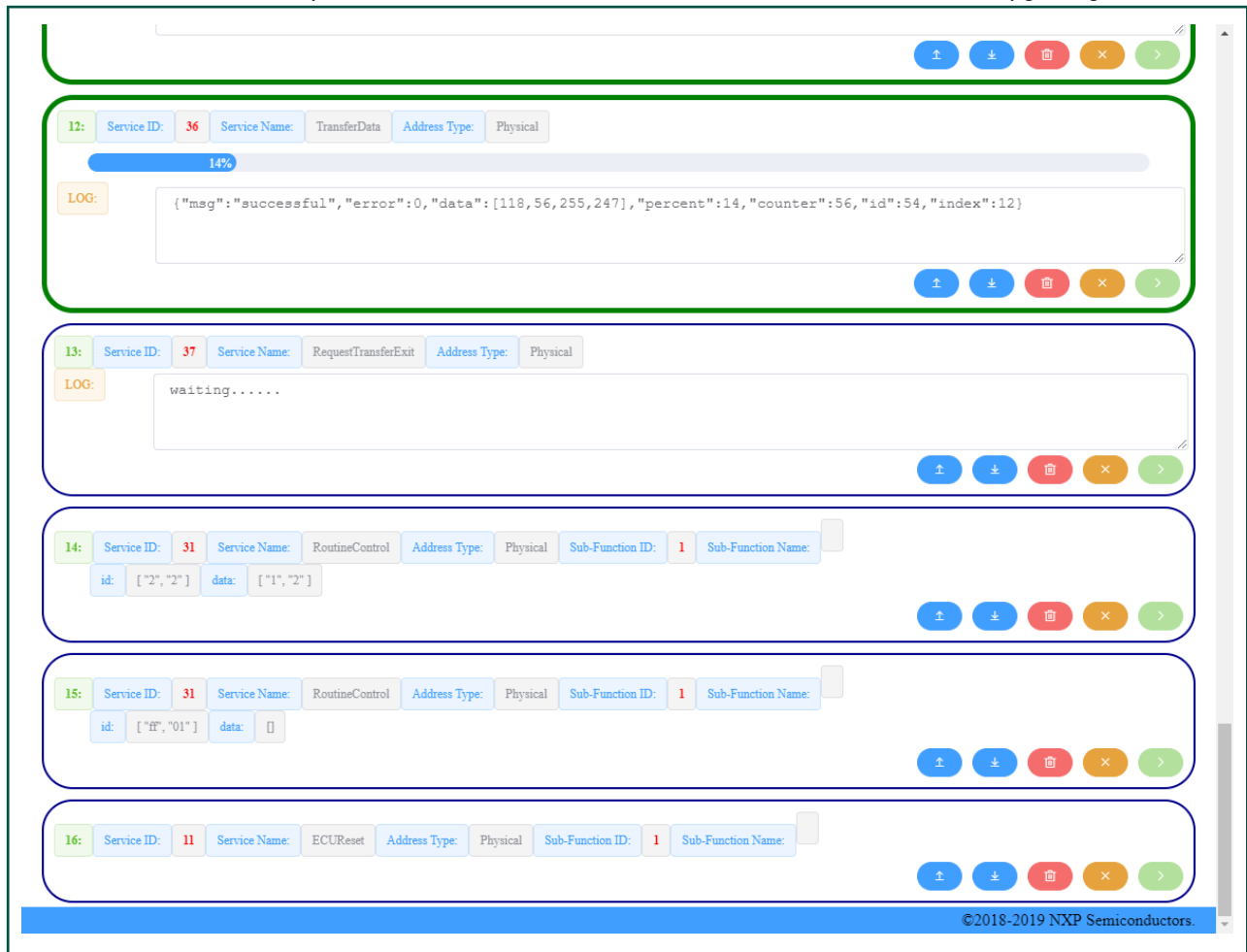
4. Check and Execute Schedule: Add service into the schedule.

Export Clear Error Delay(ms): 0 Cycle(>0): 1 Execute Schedule Service

0:	Service ID: 10	Service Name: DiagnosticSessionControl	Address Type: Physical	Sub-Function ID: 3	Sub-Function Name: extendedDiagnosticSession
⬆ ⬇ 🗑 ✖ ➡					
1:	Service ID: 28	Service Name: CommunicationControl	Address Type: Physical	Sub-Function ID: 3	Sub-Function Name: disableRxAndTx
commType: 3 ⬆ ⬇ 🗑 ✖ ➡					
2:	Service ID: 10	Service Name: DiagnosticSessionControl	Address Type: Physical	Sub-Function ID: 2	Sub-Function Name: programmingSession
⬆ ⬇ 🗑 ✖ ➡					
3:	Service ID: 27	Service Name: SecurityAccess	Address Type: Physical	Sub-Function ID: 1	Sub-Function Name: requestSeed
cryptoMode: key: ["00","00","00","00","00","00","00","00","00","00","00","00","00","00"] iv: ["00","00","00","00","00","00","00","00","00","00","00","00","00","00"] ⬆ ⬇ 🗑 ✖ ➡					
4:	Service ID: 27	Service Name: SecurityAccess	Address Type: Physical	Sub-Function ID: 2	Sub-Function Name: sendKey
cryptoMode: ECB key: ["0","1","2","3","4","5","6","7","8","9","A","B","C","D","E","F"] iv: ["00","00","00","00","00","00","00","00","00","00","00","00","00","00"] ⬆ ⬇ 🗑 ✖ ➡					
5:	Service ID: 2E	Service Name: WriteDataByIdentifier	Address Type: Physical		
id: ["F1","5A"] data: ["55","55"] ⬆ ⬇ 🗑 ✖ ➡					

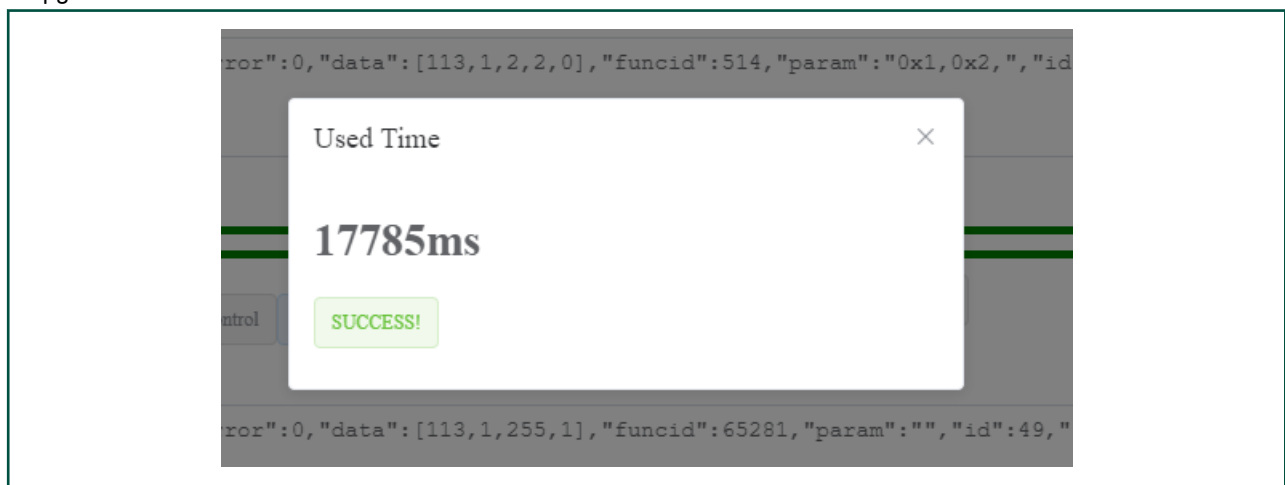
©2018-2019 NXP Semiconductors.

Check and make sure all the parameters are ok. Click Execute Schedule Service button to start upgrading.

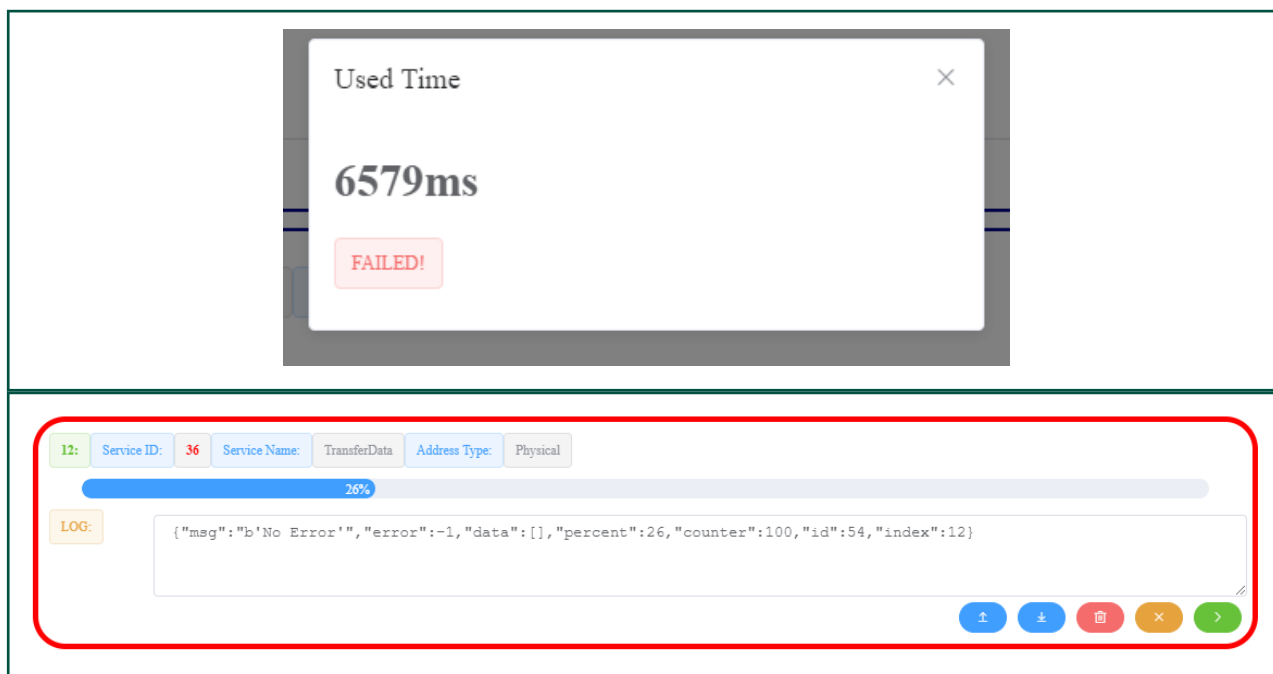


Every service shows the information needed, such as receive data, upload percent and so on.

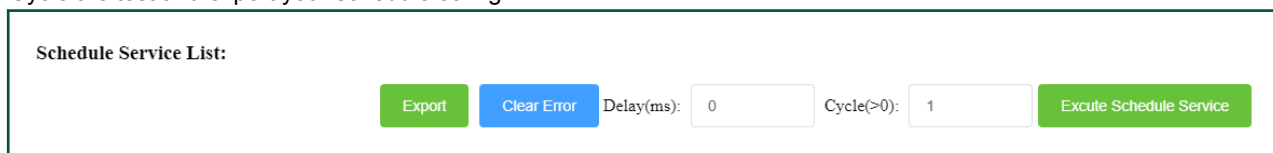
If upgrade is successful. The tool shows used time.



If it fails, check the error information, failed service shows red color, successful service shows green color.



5. Cycle the test and export your schedule config.

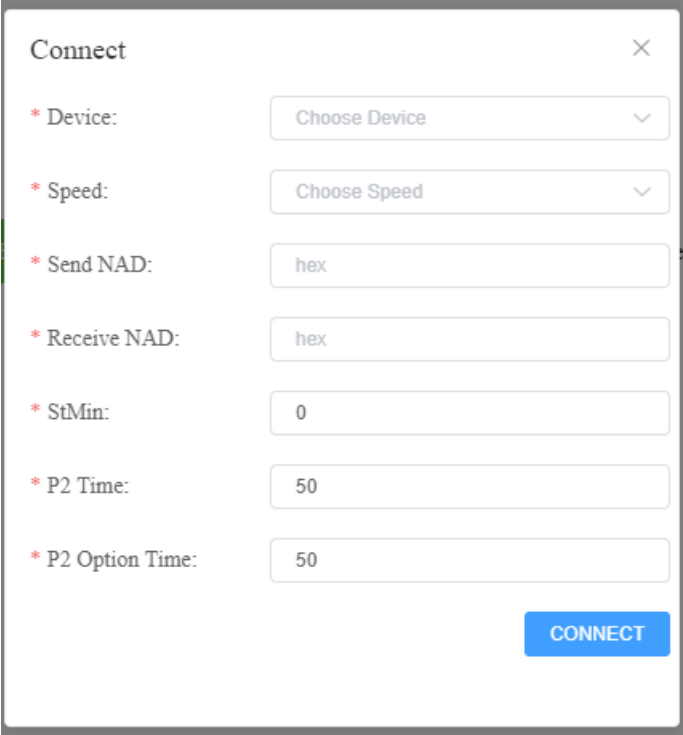


The tool supports time to time upgrades. You just need to config cycle times and delay time in every cycle. If you think the schedule is fine, just click export button, the config information is saved in **json** format. You can import your config file when you run the tool next time.

5.2 ISO-LIN-UDS tool

This feature can upgrade ECU firmware in LIN, click enter the feature page. Make sure to make the tool as LIN master node in a LIN network.

All the features are similar to CAN except connect part.



The image shows a 'Connect' dialog box with a close button (X) in the top right corner. It contains seven fields, each with a red asterisk indicating it is required:

- * Device: A dropdown menu with 'Choose Device' and a downward arrow.
- * Speed: A dropdown menu with 'Choose Speed' and a downward arrow.
- * Send NAD: A text input field containing 'hex'.
- * Receive NAD: A text input field containing 'hex'.
- * StMin: A text input field containing '0'.
- * P2 Time: A text input field containing '50'.
- * P2 Option Time: A text input field containing '50'.

A blue 'CONNECT' button is located at the bottom right of the dialog box.

All the parameters are important. If you have issues with the know how of these parameters, check the ISO standard document.

Send NAD as target ECU NAD.

Receive NAD is the tool NAD.

5.3 General tools

This tool only supports the calculation of CRC16 or CRC32 value currently. If you want to check firmware integrity, you can use this feature.

Chapter 6

Bootloader code size

It test's the bootloader stack code size in different platform and different compiler. The following figure shows detail information of the code size.

stack size					
No.	Compiler	Optimization level	description	code flash size	RAM size
1	GCC	-O0	exclude print information over UART	54KB	
		-O0 & -O1	include print information over UART SDK -O1 stack - O0	41KB	
		-O1	include print information over UART SDK -O1 stack - O1	33KB	
		-O0	only stack	24KB	4.4KB
		-O1	only stack	15KB	4.4KB
2	IAR	-O0	only stack	18KB	3.5KB
		-O1	only stack	17KB	3.5KB
3	Codewarrior	off	stack + driver + print debug info	27KB	
		off	stack + driver	25KB	
		off	only stack	21KB	
		1	only stack	19.4KB	

Chapter 7

Test and performance

The following figure shows the performance for different platform over CAN and LIN BUS based on CAN and LIN TP.

Platform	System clock	Flash driver size	APP size	Baud rate	STMin	Total test times	Every times used time
S32K144	48MHz	2KB	48KB	500K(CAN)	1	5000	14s
					0	100	7s
					2	100	20s
S12ZVML	50MHz	2KB	20KB	19200(LIN)	0	5000	58s

The following figure shows the stress test based on S32K144 over CAN BUS.

No.	Test case	Description	Test time	Result
1	Normally test	About 5000times (S32K144/CAN) About 4500times (S12ZVML/LIN)	5000 * 14s=19.4h 4500 * 1min=75h	pass
2	Reset test	Period trigger reset and reupdate firmware	About 6h(1000 times)	pass
3	Power on/off test	Period trigger power off/on and reupdate firmware	About 7h(1400 times)	pass

Chapter 8

APP needed

The APP and bootloader have some similar features like, CRC, exchange information defined and valid value. If APP support FOTA, additional modify APP information.

Table 15.

CRC - DNP 16bit	<pre> 00227: /***** 00228: ** Description : using software lookup table to create crc. Input data in @ i_pucDataBuf 00229: ** and data len in @ i_ulDataLen. When create crc successful 00230: ** return CRC. 00231: *****/ 00232: static void CreatSoftwareCrc16(const uint8 *i_pDataBuf, const uint32 i_dataLen, uint32 *m_pCurCrc) 00233: { 00234: uint16 crc = 0u; 00235: uint32 index = 0u; 00236: 00237: #if (defined FLASH_ADDRESS_CONTINUE) && (FLASH_ADDRESS_CONTINUE == TRUE) 00238: crc = *m_pCurCrc; 00239: #endif 00240: for(index = 0u; index < i_dataLen; index++) 00241: { 00242: crc = (crc >> 8) ^ g_dnpCrcTable[(crc ^ i_pDataBuf[index]) & 0x00ff] ; 00243: } 00244: 00245: *m_pCurCrc = (uint32)((~crc) & 0xFFFFu); 00246: } </pre> <pre> 00087: /* 00088: *crc table dnp crc table 00089: *Poly: 0x3D65 00090: *Init: 0x0000 00091: *RefIn:true 00092: *RefOut:true 00093: *XorOut:0xFFFF 00094: */ 00095: static const uint16 g_dnpCrcTable[256u] = { 00096: 0x0000, 0x365e, 0x6cbc, 0x5ae2, 0xd978, 0xef26, 0xb5c4, 0x839a, 00097: 0xff89, 0xc9d7, 0x9335, 0xa56b, 0x26f1, 0x10af, 0x4a4d, 0x7c13, 00098: 0xb26b, 0x8435, 0xded7, 0xe889, 0x6b13, 0x5d4d, 0x07af, 0x31f1, 00099: 0x4de2, 0x7bbc, 0x215e, 0x1700, 0x949a, 0xa2c4, 0xf826, 0xce78, 00100: 0x29af, 0x1ff1, 0x4513, 0x734d, 0xf0d7, 0xc689, 0x9c6b, 0xaa35, 00101: 0xd626, 0xe078, 0xba9a, 0x8cc4, 0x0f5e, 0x3900, 0x63e2, 0x55bc, 00102: 0x9bc4, 0xad9a, 0xf778, 0xc126, 0x42bc, 0x74e2, 0x2e00, 0x185e, 00103: 0x644d, 0x5213, 0x08f1, 0x3eaf, 0xbd35, 0x8b6b, 0xd189, 0xe7d7, 00104: 0x535e, 0x6500, 0x3fe2, 0x09bc, 0x8a26, 0xbc78, 0xe69a, 0xd0c4, 00105: 0xacd7, 0x9a89, 0xc06b, 0xf635, 0x75af, 0x43f1, 0x1913, 0x2f4d, 00106: 0xe135, 0xd76b, 0x8d89, 0xbbd7, 0x384d, 0x0e13, 0x54f1, 0x62af, 00107: 0x1ebc, 0x28e2, 0x7200, 0x445e, 0xc7c4, 0xf19a, 0xab78, 0x9d26, 00108: 0x7af1, 0x4caf, 0x164d, 0x2013, 0xa389, 0x95d7, 0xcf35, 0xf96b, 00109: 0x8578, 0xb326, 0xe9c4, 0xdf9a, 0x5c00, 0x6a5e, 0x30bc, 0x06e2, 00110: 0xc89a, 0xfec4, 0xa426, 0x9278, 0x11e2, 0x27bc, 0x7d5e, 0x4b00, 00111: 0x3713, 0x014d, 0x5baf, 0x6df1, 0xee6b, 0xd835, 0x82d7, 0xb489, 00112: 0xa6bc, 0x90e2, 0xca00, 0xfc5e, 0x7fc4, 0x499a, 0x1378, 0x2526, 00113: 0x5935, 0x6fb6, 0x3589, 0x03d7, 0x804d, 0xb613, 0xecf1, 0xdaaf, 00114: 0x14d7, 0x2289, 0x786b, 0x4e35, 0xcda7, 0xfbf1, 0xa113, 0x974d, 00115: 0xeb5e, 0xdd00, 0x87e2, 0xb1bc, 0x3226, 0x0478, 0x5e9a, 0x68c4, 00116: 0x8f13, 0xb94d, 0xe3af, 0xd5f1, 0x566b, 0x6035, 0x3ad7, 0x0c89, 00117: 0x709a, 0x46c4, 0x1c26, 0x2a78, 0xa9e2, 0x9fbc, 0xc55e, 0xf300, 00118: 0x3d78, 0x0b26, 0x51c4, 0x679a, 0xe400, 0xd25e, 0x88bc, 0xbbee, 00119: 0xc2f1, 0xf4af, 0xae4d, 0x9813, 0x1b89, 0x2dd7, 0x7735, 0x416b, 00120: 0xf5e2, 0xc3bc, 0x995e, 0xaf00, 0x2c9a, 0x1ac4, 0x4026, 0x7678, 00121: 0x0a6b, 0x3c35, 0x66d7, 0x5089, 0xd313, 0xe54d, 0xbfaf, 0x89f1, 00122: 0x4789, 0x71d7, 0x2b35, 0x1db6, 0x9ef1, 0xa8af, 0xf24d, 0xc413, 00123: 0xb800, 0x8e5e, 0xd4bc, 0xe2e2, 0x6178, 0x5726, 0x0dc4, 0x3b9a, 00124: 0xdc4d, 0xea13, 0xb0f1, 0x86af, 0x0535, 0x336b, 0x6989, 0x5fd7, 00125: 0x23c4, 0x159a, 0x4f78, 0x7926, 0xfabc, 0xcc2, 0x9600, 0xa05e, 00126: 0x6e26, 0x5878, 0x029a, 0x34c4, 0xb75e, 0x8100, 0xdb2, 0xedbc, 00127: 0x91af, 0xa7f1, 0xfd13, 0xcb4d, 0x48d7, 0x7e89, 0x246b, 0x1235, 00128: }; </pre>
-----------------	---

Table continues on the next page...

Table 15. (continued)

	Use software to calculate CRC, set *m_pCurCrc = 0u before calling the function.
Exchange information	<pre> 00043: typedef struct 00044: { 00045: uint8 infoDataLen; 00046: uint8 requestEnterBootloader; 00047: uint8 downloadAPPSuccessful; 00048: uint32 infoStartAddr; 00049: uint32 requestEnterBootloaderAddr; 00050: uint32 downloadAppSuccessfulAddr; 00051: }tBootInfo; 00052: 00053: const static tBootInfo gs_stBootInfo = { 00054: 16u, 00055: 0x5Au, 00056: 0xA5u, 00057: 0x20006FF0u, 00058: 0x20006FF1u, 00059: 0x20006FF0u, 00060: }; </pre> <p>typedef struct</p> <pre> { uint8 infoDataLen; uint8 requestEnterBootloader; uint8 downloadAPPSuccessful; uint32 infoStartAddr; uint32 requestEnterBootloaderAddr; uint32 downloadAppSuccessfulAddr; }tBootInfo; const static tBootInfo gs_stBootInfo = { 16u, 0x5Au, 0xA5u, 0x20006FF0u, 0x20006FF1u, 0x20006FF0u, }; </pre> <p>CRC is stored in the last 2 bytes.</p>
APP information	<pre> typedef struct { /*flash programming successfull? If programming successfull, the value set TRUE, else set FALSE*/ uint8 isFlashProgramSuccessfull; /*Is erase flash successfull? If erased flash successfull, set TRUE, else set FALSE.*/ </pre>

Table continues on the next page...

Table 15. (continued)

	<pre> uint8 isFlashErasedSuccessful; /*Is Flash struct data valid? If written set value is TRUE, else set valid FALSE*/ uint8 isFlashStructValid; /*indicate app Counter. Before download. */ uint8 appCnt; /* flag if fingerprint buffer */ uint8 aFingerPrint[FL_FINGER_PRINT_LENGTH]; /*reset handler length*/ uint32 appStartAddrLen; /*app Start address -- reset handler*/ uint32 appStartAddr; /*count CRC*/ uint32 crc; }tAppFlashStatus; </pre>
	<pre> 00023: typedef struct 00024: { 00025: /*flash programming successful? If programming successful, the value set TRUE, else set FALSE*/ 00026: uint8 isFlashProgramSuccessful; 00027: 00028: /*Is erase flash successful? If erased flash successful, set the TRUE, else set the FALSE.*/ 00029: uint8 isFlashErasedSuccessful; 00030: 00031: /*Is Flash struct data valid? If written set the value is TRUE, else set the valid FALSE*/ 00032: uint8 isFlashStructValid; 00033: 00034: /*indicate app Counter. Before download. */ 00035: uint8 appCnt; 00036: 00037: /* flag if fingerprint buffer */ 00038: uint8 aFingerPrint[FL_FINGER_PRINT_LENGTH]; 00039: 00040: /*reset handler length*/ 00041: uint32 appStartAddrLen; 00042: 00043: /*app Start address -- reset handler*/ 00044: uint32 appStartAddr; 00045: 00046: /*count CRC*/ 00047: uint32 crc; 00048: } ? end {anonAppFlashStatus} ? tAppFlashStatus; </pre> <p>APP update</p>

Chapter 9

References

ISO14229

ISO17987-2

ISO15765-2

Chapter 10

Revision history

Rev. No.	Date	Substantive Change(s)
0	February 2020	Initial version.
1	March 2020	1. Removed the bullet "Professional/production level bootloader solution deliver to customer" from Introduction .

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer's applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX, EMBRACE, GREENCHIP, HITAG, I2C BUS, ICODE, JCOP, LIFE VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, Altivec, C-5, CodeTEST, CodeWarrior, ColdFire, ColdFire+, C-Ware, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, Ready Play, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, SMARTMOS, Tower, TurboLink, UMEMS, EdgeScale, EdgeLock, eIQ, and Immersive3D are trademarks of NXP B.V. All other product or service names are the property of their respective owners. AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamiQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, µVision, Versatile are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© NXP B.V. 2020.

All rights reserved.

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

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: March, 2020

Document identifier: UBLUG