

TSSAPI-C

Application programming interface Developer's Guide

Ver. 1.3
09/2015

1

TSS COMPANY s.r.o., Pod Rovnicami 41, 84104 Bratislava, Slovakia
web: www.tsscompany.eu email: info@tsscompany.eu

© Copyright 2015, TSS COMPANY, s.r.o.

No part of this manual may be reproduced in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of TSS COMPANY.

TSS COMPANY reserves the right to modify its products specifications without giving any notice. For up to date information please visit www.tsscompany.eu.

The information contained herein has been carefully checked and is believed to be accurate. However, no responsibility is assumed for inaccuracies.

Contents

1	Introduction.....	4
2	Data Structures.....	5
3	Macro Definitions.....	6
4	Functions	10
5	Function Documentation	12
6	Data Structure Documentation	103
7	Examples	122
8	Importing the libraries	127
9	References.....	128
10	Index.....	129

1 Introduction

This document describes usage of the Application Software Interface (API) offered by TSS Company s.r.o. for the development of software for use, integrate and control TSS Comapany s.r.o. UHF RFID readers.

1.1 Overview

TSSAPI-C.dll is shared library with C language interface written in C++ using Qt Framework. C language interface is used to allow the Software Developers to use the library in other programming languages like C#, JAVA, Visual Basic, etc.

The TSS-APIC.dll in version 1.20 is using Qt libraries version: 5.4 and was built with MSVC 2013 compiler

2 Data Structures

<u>c1g2blockeraseopspecresult t</u>	103
<u>c1g2blockpermalockopspecresult t</u>	104
<u>c1g2blockwriteopspecresult t</u>	105
<u>c1g2customcommandresult t</u>	107
<u>c1g2epc t</u>	108
<u>c1g2getblockpermalockstatusopspecresult t</u>	109
<u>c1g2killopspecresult t</u>	110
<u>c1g2lockopspecresult t</u>	111
<u>c1g2readopspecresult t</u>	112
<u>c1g2v2authenticateandreplyopspecresult t</u>	113
<u>c1g2v2authenticateandstoreopspecresult t</u>	114
<u>c1g2v2readbufferopspecresult t</u>	115
<u>c1g2v2untraceableopspecresult t</u>	116
<u>c1g2writeopspecresult t</u>	117
<u>gpievent t</u>	118
<u>higgs3blockreadlockresult t</u>	119
<u>log t</u>	120
<u>rospecevent t</u>	121

3 Macro Definitions

3.1 Supported LLRP Commands

- #define URMSP_GET_READER_CAPABILITIES 1
- #define URMSP_START_ROSPEC 22
- #define URMSP_STOP_ROSPEC 23
- #define URMSP_GET_ROSPECS 26
- #define URMSP_ROSPECSTARTTRIGGER 179
- #define URMSP_GPIRIGGERVALUE 181
- #define URMSP_AISPECSTOPTRIGGER 184
- #define URMSP_TAGOBSERVATIONTRIGGER 185
- #define URMSP_RFTRANSMITTER 224
- #define URMSP_EVENTNOTIFICATIONSTATE 245
- #define URMSP_C1G2TAGINVENTORYMASK 332
- #define URMSP_C1G2TAGINVENTORYSTATEAWAREFILTERACTION 333
- #define URMSP_C1G2SINGULATIONCONTROL 336
- #define URMSP_C1G2TAGINVENTORYSTATEAWARESINGULATIONACTION 337
- #define URMSP_C1G2READ 341
- #define URMSP_C1G2WRITE 342
- #define URMSP_C1G2KILL 343
- #define URMSP_C1G2LOCK 344
- #define URMSP_C1G2LOCKPAYLOAD 345
- #define URMSP_C1G2BLOCKERASE 346
- #define URMSP_C1G2BLOCKWRITE 347
- #define URMSP_C1G2BLOCKPERMALOCK 358
- #define URMSP_C1G2GETBLOCKPERMALOCKSTATUS 359
- #define URMSP_C1G2V2AUTHENTICATEANDREPLY 8555
- #define URMSP_C1G2V2AUTHENTICATEANDSTORE 8556
- #define URMSP_C1G2V2READBUFFER 8560
- #define URMSP_C1G2V2UNTRACEABLE 8561
- #define URMSP_HIGGS3BLOCKREADLOCK 8961
- #define URMSP_CUSTOMCOMMAND 16382
- #define URMSP_SETPERIPHERALS 16642
- #define URMSP_SETPOWERMODE 16897

3.2 Supported LLRP Command Results

- #define URMSP_EPCDATA 241
- #define URMSP_GPIEVENT 248
- #define URMSP_ROSPECEVENT 249
- #define URMSP_EPC_96 36096
- #define URMSP_C1G2READOPSPECRESULT 349
- #define URMSP_C1G2WRITEOPSPECRESULT 350
- #define URMSP_C1G2KILLOPSPECRESULT 351
- #define URMSP_C1G2LOCKOPSPECRESULT 352
- #define URMSP_C1G2BLOCKERASEOPSPECRESULT 353
- #define URMSP_C1G2BLOCKWRITEOPSPECRESULT 354
- #define URMSP_C1G2BLOCKPERMALOCKOPSPECRESULT 361
- #define URMSP_C1G2GETBLOCKPERMALOCKSTATUSOPSPECRESULT 362
- #define URMSP_C1G2V2AUTHENTICATEANDREPLYRESULT 8566
- #define URMSP_C1G2V2AUTHENTICATEANDSTORERESULT 8567

- #define URMSP_C1G2V2READBUFFERRESULT 8571
- #define URMSP_C1G2V2UNTRACEABLERESULT 8572
- #define URMSP_HIGGS3BLOCKREADLOCKRESULT 8962
- #define URMSP_CUSTOMCOMMANDRESULT 16383

3.3 Device Status

- #define TSS_SUCCESS 000
- #define TSS_PARAMETERERROR 200
- #define TSS_FIELDERROR 201
- #define TSS_UNEXPECTEDPARAMETER 202
- #define TSS_MISSINGPARAMETER 203
- #define TSS_DUPLICATEPARAMETER 204
- #define TSS_OVERFLOWPARAMETER 205
- #define TSS_OVERFLOWFIELD 206
- #define TSS_UNKNOWNPARAMETER 207
- #define TSS_UNKNOWNFIELD 208
- #define TSS_UNSUPPORTEDPARAMETER 209
- #define TSS_INVALID 300
- #define TSS_OUTOFRANGE 301
- #define TSS_DEVICEERROR 401

3.4 GetReaderCapabilities - Request Data

- #define TSS_GENERAL_DEVICE_CAPABILITIES 1

3.5 ROSpecStartTrigger - Trigger Type

7

- #define TSS_ROSPECSTARTTRIGGER_NULL 0
- #define TSS_ROSPECSTARTTRIGGER_IMMEDIATE 1
- #define TSS_ROSPECSTARTTRIGGER_PERIODIC 2
- #define TSS_ROSPECSTARTTRIGGER_GPI 3

3.6 AISpecStopTrigger - Trigger Type

- #define TSS_AISPECSTOPTRIGGER_NULL 0
- #define TSS_AISPECSTOPTRIGGER_DURATION 1
- #define TSS_AISPECSTOPTRIGGER_GPITIMEOUT 2
- #define TSS_AISPECSTOPTRIGGER_TAGOBSERVATION 3

3.7 EventNotification - Event Type

- #define TSS_EVENTNOTIFICATION_GPI 1
- #define TSS_EVENTNOTIFICATION_ROSPEC 2

3.8 EventNotification - Event State

- #define TSS_EVENTNOTIFICATIONSTATE_DISABLED 0
- #define TSS_EVENTNOTIFICATIONSTATE_ENABLED 1

3.9 Tag memory bank

- #define TSS_TAGMEMORYBANK_RESERVED 0
- #define TSS_TAGMEMORYBANK_EPC 1
- #define TSS_TAGMEMORYBANK_TID 2
- #define TSS_TAGMEMORYBANK_USER 3

3.10 C1G2Lock Privilege

- #define TSS_C1G2LOCKPRIVILEGE_READWRITE 0
- #define TSS_C1G2LOCKPRIVILEGE_PERMALOCK 1
- #define TSS_C1G2LOCKPRIVILEGE_PERMAUNLOCK 2
- #define TSS_C1G2LOCKPRIVILEGE_UNLOCK 3

3.11 C1G2Lock Datafield

- #define TSS_C1G2LOCKDATAFIELD_KILLPWD 0
- #define TSS_C1G2LOCKDATAFIELD_ACCESSPWD 1
- #define TSS_C1G2LOCKDATAFIELD_EPCMEMORY 2
- #define TSS_C1G2LOCKDATAFIELD_TIDMEMORY 3
- #define TSS_C1G2LOCKDATAFIELD_USERMEMORY 4

3.12 C1G2CustomCommand Frame Sync or Preamble

- #define TSS_C1G2CUSTOMCOMMAND_FP_NONE 0
- #define TSS_C1G2CUSTOMCOMMAND_FP_NONE 2
- #define TSS_C1G2CUSTOMCOMMAND_FP_PREAMBLE 1

3.13 C1G2CustomCommand - Bit operation

- #define TSS_C1G2CUSTOMCOMMAND_BITOPERATION_NONE 0
- #define TSS_C1G2CUSTOMCOMMAND_BITOPERATION_XORWITHRN16 1

3.14 C1G2CustomCommand - Handle

- #define TSS_C1G2CUSTOMCOMMAND_HANDLE_NONE 0
- #define TSS_C1G2CUSTOMCOMMAND_HANDLE_ADD 1

3.15 C1G2CustomCommand - CRC

- #define TSS_C1G2CUSTOMCOMMAND_CRC_NONE 0
- #define TSS_C1G2CUSTOMCOMMAND_CRC_ADD 1

3.16 C1G2CustomCommand - Tag reply type

- #define TSS_C1G2CUSTOMCOMMAND_REPLYTYPE_NONE 1
- #define TSS_C1G2CUSTOMCOMMAND_REPLYTYPE_IMMEDIATE 2
- #define TSS_C1G2CUSTOMCOMMAND_REPLYTYPE_DELAYED 3
- #define TSS_C1G2CUSTOMCOMMAND_REPLYTYPE_INPROCESS 4

3.17 SetPeripherals

- `#define TSS_SETPERIPHERALS_DISABLED 0`
- `#define TSS_SETPERIPHERALS_ENABLED 1`
- `#define TSS_SETPERIPHERALS_DONOTCHANGE 15`

4 Functions

4.1 Reader Commands

- void [setDebugLevel](#) (int debugLevel)
- void [initDll](#) ()
- int [connectReader](#) (char *comPort)
- int [connectReaderTCP](#) (char *host, unsigned int port)
- int [disconnectReader](#) ()
- int [waitForResultStruct](#) (void *structPointer, unsigned int resultType, int timeout)
- int [urmsp_GetReaderCapabilities](#) (char *returnValue, unsigned char flags, unsigned char reserved, unsigned char requestedData)
- int [urmsp_StartROSpec](#) (unsigned char flags, unsigned char reserved)
- int [urmsp_StopROSpec](#) (unsigned char flags, unsigned char reserved)
- int [urmsp_GetROSpecs](#) (unsigned char *returnValue, unsigned char flags, unsigned char reserved)
- int [urmsp_ROSpecStartTrigger](#) (unsigned char flags, unsigned char reserved, unsigned char rOSpecStartTriggerType)
- int [urmsp_GPITriggerValue](#) (unsigned char flags, unsigned char reserved, unsigned short gPIPortNum, unsigned char gPIEvent, unsigned int timeout)
- int [urmsp_AISpecStopTrigger](#) (unsigned char flags, unsigned char reserved, unsigned char alSpecStopTriggerType, unsigned int durationTriggerValue)
- int [urmsp_TagObservationTrigger](#) (unsigned char flags, unsigned char reserved, unsigned char triggerType, unsigned short numberOfTags, unsigned short numberOfAttempts, unsigned short t, unsigned int timeout)
- int [urmsp_RFTransmitter](#) (unsigned char flags, unsigned char reserved, unsigned short hopTableID, unsigned short channelIndex, unsigned short transmitPower)
- int [urmsp_EventNotificationState](#) (unsigned char flags, unsigned char reserved, unsigned short eventType, unsigned char notificationState)
- int [urmsp_C1G2TagInventoryMask](#) (unsigned char flags, unsigned char reserved, unsigned char mb, unsigned short pointer, unsigned short maskBitCount, const char *tagMask)
- int [urmsp_C1G2TagInventoryStateAwareFilterAction](#) (unsigned char flags, unsigned char reserved, unsigned char target, unsigned char action)
- int [urmsp_C1G2SingulationControl](#) (unsigned char flags, unsigned char reserved, unsigned char s, unsigned short tagPopulation, unsigned int tagTransitTime)
- int [urmsp_C1G2TagInventoryStateAwareSingulationAction](#) (unsigned char flags, unsigned char reserved, bool i, bool s, bool a)
- int [urmsp_C1G2Read](#) (unsigned char flags, unsigned char reserved, unsigned short opSpecID, unsigned int accessPassword, unsigned char MB, unsigned short wordPointer, unsigned short wordCount)
- int [urmsp_C1G2Write](#) (unsigned char flags, unsigned char reserved, unsigned short opSpecID, unsigned int accessPassword, unsigned char MB, unsigned short writeDataWordPointer, unsigned short wordCount, unsigned short writeData)
- int [urmsp_C1G2Kill](#) (unsigned char flags, unsigned char reserved, unsigned short opSpecID, unsigned int killPassword)
- int [urmsp_C1G2Lock](#) (unsigned char flags, unsigned char reserved, unsigned short opSpecID, unsigned int accessPassword)
- int [urmsp_C1G2LockPayload](#) (unsigned char flags, unsigned char reserved, unsigned char privilege, unsigned char dataField)
- int [urmsp_C1G2BlockErase](#) (unsigned char flags, unsigned char reserved, unsigned short opSpecID, unsigned int accessPassword, unsigned char MB, short wordPointer, short wordCount)
- int [urmsp_C1G2BlockWrite](#) (unsigned char flags, unsigned char reserved, unsigned short opSpecID, unsigned int accessPassword, unsigned char MB, short writeDataWordPointer, short wordCount, const char *writeData)
- int [urmsp_C1G2BlockPermalock](#) (unsigned char flags, unsigned char reserved, unsigned short opSpecID, unsigned int accessPassword, unsigned char MB, unsigned short blockPointer, unsigned short blockMaskWordCount, const char *blockMask)

- int [urmsp_C1G2GetBlockPermalockStatus](#) (unsigned char flags, unsigned char reserved, unsigned short opSpecID, unsigned int accessPassword, unsigned char MB, unsigned short blockPointer, unsigned short blockRange)
- int [urmsp_C1G2V2AuthenticateAndReply](#) (unsigned char flags, unsigned char reserved, unsigned short opSpecID, unsigned int accessPassword, unsigned char csi, unsigned short messageLength, const char *message)
- int [urmsp_C1G2V2AuthenticateAndStore](#) (unsigned char flags, unsigned char reserved, unsigned short opSpecID, unsigned int accessPassword, unsigned char csi, unsigned short storedMessageLength, const char *message)
- int [urmsp_C1G2V2Untraceable](#) (unsigned char flags, unsigned char reserved, unsigned short opSpecID, unsigned int accessPassword, bool u, bool e, unsigned char epcLength, unsigned char tid, bool m, unsigned char r)
- int [urmsp_C1G2V2ReadBuffer](#) (unsigned char flags, unsigned char reserved, unsigned short opSpecID, unsigned int accessPassword, unsigned short wordPointer, unsigned short bitCount)
- int [urmsp_Higgs3BlockReadLock](#) (unsigned char flags, unsigned char reserved, unsigned short opSpecID, unsigned int accessPassword, unsigned char readLockBits)
- int [urmsp_C1G2CustomCommand](#) (unsigned char flags, unsigned char reserved, unsigned short opSpecID, unsigned int accessPassword, unsigned char fp, unsigned char rfu, unsigned short bitCount, const char *command, unsigned char rfu2, bool r1, bool r2, bool h, bool c, unsigned char rt, unsigned char rfu3, unsigned short replyBiteCount)
- int [urmsp_SetPeripherals](#) (unsigned char flags, unsigned char reserved, unsigned char powerLED, unsigned char statusLED, unsigned char buzzer, unsigned char pushButton, unsigned char rfu)
- int [urmsp_SetPowerMode](#) (unsigned char flags, unsigned char reserved, unsigned char powerMode)
- void [getErrorString](#) (char *errorSourceString, unsigned int *errorSourceStringLength, char *errorCodeString, unsigned int *errorCodeStringLength, int errorCode)
- void [getResultString](#) (char *resultString, unsigned int *resultStringLength, unsigned int resultType, unsigned char resultCode)

4.2 Callback registration

11

- void [connectCallbackEPC](#) (CallbackFunctionPointer cbf, void *cb)
- void [connectCallbackReaderEvents](#) (CallbackFunctionPointer cbf, void *cb)
- void [connectCallbackC1G2Read](#) (CallbackFunctionPointer cbf, void *cb)
- void [connectCallbackC1G2Write](#) (CallbackFunctionPointer cbf, void *cb)
- void [connectCallbackC1G2Kill](#) (CallbackFunctionPointer cbf, void *cb)
- void [connectCallbackC1G2Lock](#) (CallbackFunctionPointer cbf, void *cb)
- void [connectCallbackC1G2BlockErase](#) (CallbackFunctionPointer cbf, void *cb)
- void [connectCallbackC1G2BlockWrite](#) (CallbackFunctionPointer cbf, void *cb)
- void [connectCallbackC1G2BlockPermalock](#) (CallbackFunctionPointer cbf, void *cb)
- void [connectCallbackC1G2GetBlockPermalockStatus](#) (CallbackFunctionPointer cbf, void *cb)
- void [connectCallbackC1G2V2AuthenticateAndReply](#) (CallbackFunctionPointer cbf, void *cb)
- void [connectCallbackC1G2V2AuthenticateAndStore](#) (CallbackFunctionPointer cbf, void *cb)
- void [connectCallbackC1G2V2Untraceable](#) (CallbackFunctionPointer cbf, void *cb)
- void [connectCallbackC1G2V2ReadBuffer](#) (CallbackFunctionPointer cbf, void *cb)
- void [connectCallbackHiggs3BlockReadLock](#) (CallbackFunctionPointer cbf, void *cb)
- void [connectCallbackC1G2CustomCommand](#) (CallbackFunctionPointer cbf, void *cb)
- void [connectCallbackLog](#) (CallbackFunctionPointer cbf, void *cb)

5 Function Documentation

5.1 void setDebugLevel (int *debugLevel*)

This function set the library debug message level. This function have to be called before you initialize the library.

5.1.1 Parameters:

in	<i>debugLevel</i>	Possible Values: 0x00 - 0x02. 0x00 = No debug messages 0x01 = Console log messages 0x02 = Log to file (TSS-APIC-dd-MM-yyyy.log)
----	-------------------	--

5.2 void initDll ()

This function initialize the library. This function have to be called before you start to use the library.

Example Usage:

[connectReader\(\)](#)

5.3 void connectCallbackEPC (CallbackFunctionPointer *cbf*, void * *cb*)

Connect a callbackfunction of choice. Function will be called each time the reader reads an EPC number. Callback function receive [c1g2epc_t](#) type structure.

5.3.1 Parameters:

In	<i>cbf</i>	Pointer to a Static member function (SomeClass::someCallbackFunction).
In	<i>cb</i>	Object pointer (someclass)

Example Usage:

[urmsp_StartROSpec\(\)](#)

5.4 void connectCallbackReaderEvents (CallbackFunctionPointer *cbf*, void * *cb*)

Connect a callbackfunction of choice. Function will be called each time the reader receive ReaderEvents: GPIEvent Callback function receive [gpievent_t](#) type structure.

5.4.1 Parameters:

In	<i>cbf</i>	Pointer to a Static member function (SomeClass::someCallbackFunction).
In	<i>cb</i>	Object pointer (someclass)

Example Usage:

[urmsp_EventNotificationState\(\)](#)

5.5 void connectCallbackC1G2Read (CallbackFunctionPointer *cbf*, void * *cb*)

Connect a callbackfunction of choice. Function will be called each time the reader receive C1G2ReadResult. Callback function receive [c1g2readopspecresult_t](#) type structure.

5.5.1 Parameters:

In	<i>cbf</i>	Pointer to a Static member function (SomeClass::someCallbackFunction).
In	<i>cb</i>	Object pointer (someclass)

Example Usage:

[urmsp_C1G2Read\(\)](#)

5.6 void connectCallbackC1G2Write (CallbackFunctionPointer *cbf*, void * *cb*)

Connect a callbackfunction of choice. Function will be called each time the reader receive C1G2WriteResult. Callback function receive [c1g2writeopspecresult_t](#) type structure.

5.6.1 Parameters:

In	<i>cbf</i>	Pointer to a Static member function (SomeClass::someCallbackFunction).
In	<i>cb</i>	Object pointer (someclass)

Example Usage:

[urmsp_C1G2Write\(\)](#)

5.7 void connectCallbackC1G2Kill (CallbackFunctionPointer *cbf*, void * *cb*)

Connect a callbackfunction of choice. Function will be called each time the reader receive C1G2KillResult. Callback function receive [c1g2killopspecresult_t](#) type structure.

5.7.1 Parameters:

In	<i>cbf</i>	Pointer to a Static member function (SomeClass::someCallbackFunction).
In	<i>cb</i>	Object pointer (someclass)

Example Usage:

[urmsp_C1G2Kill\(\)](#)

5.8 void connectCallbackC1G2Lock (CallbackFunctionPointer *cbf*, void * *cb*)

Connect a callbackfunction of choice. Function will be called each time the reader receive C1G2LockResult. Callback function receive [c1g2lockopspecresult t](#) type structure.

5.8.1 Parameters:

In	<i>cbf</i>	Pointer to a Static member function (SomeClass::someCallbackFunction).
In	<i>cb</i>	Object pointer (someclass)

Example Usage:

[urmsp_C1G2Lock\(\)](#)

5.9 void connectCallbackC1G2BlockErase (CallbackFunctionPointer *cbf*, void * *cb*)

Connect a callbackfunction of choice. Function will be called each time the reader receive C1G2BlockEraseResult. Callback function receive [c1g2blockeraseopspecresult t](#) type structure.

5.9.1 Parameters:

In	<i>cbf</i>	Pointer to a Static member function (SomeClass::someCallbackFunction).
In	<i>cb</i>	Object pointer (someclass)

Example Usage:

[urmsp_C1G2BlockErase\(\)](#)

5.10 void connectCallbackC1G2BlockWrite (CallbackFunctionPointer *cbf*, void * *cb*)

Connect a callbackfunction of choice. Function will be called each time the reader receive C1G2BlockWriteResult. Callback function receive [c1g2blockwriteopspecresult t](#) type structure.

5.10.1 Parameters:

In	<i>cbf</i>	Pointer to a Static member function (SomeClass::someCallbackFunction).
In	<i>cb</i>	Object pointer (someclass)

Example Usage:

[urmsp_C1G2BlockWrite\(\)](#)

5.11 void connectCallbackC1G2BlockPermalock (CallbackFunctionPointer *cbf*, void * *cb*)

Connect a callbackfunction of choice. Function will be called each time the reader receive C1G2BlockPermalockResult. Callback function receive [c1g2blockpermalockopspecresult t](#) type structure.

5.11.1 Parameters:

In	<i>cbf</i>	Pointer to a Static member function (SomeClass::someCallbackFunction).
In	<i>cb</i>	Object pointer (someclass)

Example Usage:

[urmsp_C1G2BlockPermalock\(\)](#)

5.12 void connectCallbackC1G2GetBlockPermalockStatus (CallbackFunctionPointer *cbf*, void * *cb*)

Connect a callbackfunction of choice. Function will be called each time the reader receive C1G2GetBlockPermalockResult. Callback function receive [c1g2getblockpermalockstatusopspecresult t](#); type structure.

5.12.1 Parameters:

In	<i>cbf</i>	Pointer to a Static member function (SomeClass::someCallbackFunction).
In	<i>cb</i>	Object pointer (someclass)

Example Usage:

[urmsp_C1G2GetBlockPermalockStatus\(\)](#)

15

5.13 void connectCallbackC1G2V2AuthenticateAndReply (CallbackFunctionPointer *cbf*, void * *cb*)

Connect a callbackfunction of choice. Function will be called each time the reader receive C1G2V2AuthenticateAndReplyResult. Callback function receive [c1g2v2authenticateandreplyopspecresult t](#) type structure.

5.13.1 Parameters:

In	<i>cbf</i>	Pointer to a Static member function (SomeClass::someCallbackFunction).
In	<i>cb</i>	Object pointer (someclass)

Example Usage:

[urmsp_C1G2V2AuthenticateAndReply\(\)](#)

5.14 void connectCallbackC1G2V2AuthenticateAndStore

(CallbackFunctionPointer *cbf*, void * *cb*)

Connect a callbackfunction of choice. Function will be called each time the reader receive C1G2V2AuthenticateAndStoreResult. Callback function receive [c1g2v2authenticateandstoreopspecresult t](#) type structure.

5.14.1 Parameters:

In	<i>cbf</i>	Pointer to a Static member function (SomeClass::someCallbackFunction).
In	<i>cb</i>	Object pointer (someclass)

Example Usage:

[urmsp_C1G2V2AuthenticateAndStore\(\)](#)

5.15 void connectCallbackC1G2V2Untraceable (CallbackFunctionPointer *cbf*, void * *cb*)

Connect a callbackfunction of choice. Function will be called each time the reader receive C1G2V2UntraceableResult. Callback function receive [c1g2v2untraceableopspecresult t](#); type structure.

5.15.1 Parameters:

In	<i>cbf</i>	Pointer to a Static member function (SomeClass::someCallbackFunction).
In	<i>cb</i>	Object pointer (someclass)

Example Usage:

[urmsp_C1G2V2Untraceable\(\)](#)

16

5.16 void connectCallbackC1G2V2ReadBuffer (CallbackFunctionPointer *cbf*, void * *cb*)

Connect a callbackfunction of choice. Function will be called each time the reader receive C1G2V2ReadBufferResult. Callback function receive [c1g2v2readbufferopspecresult t](#) type structure.

5.16.1 Parameters:

In	<i>cbf</i>	Pointer to a Static member function (SomeClass::someCallbackFunction).
In	<i>cb</i>	Object pointer (someclass)

Example Usage:

[urmsp_C1G2V2ReadBuffer\(\)](#)

5.17 void connectCallbackHiggs3BlockReadLock (CallbackFunctionPointer *cbf*, void * *cb*)

Connect a callbackfunction of choice. Function will be called each time the reader receive Higgs3BlockReadLockResult. Callback function receive [higgs3blockreadlockresult_t](#) type structure.

5.17.1 Parameters:

In	<i>cbf</i>	Pointer to a Static member function (SomeClass::someCallbackFunction).
In	<i>cb</i>	Object pointer (someclass)

Example Usage:

[urmsp_Higgs3BlockReadLock\(\)](#)

5.18 void connectCallbackC1G2CustomCommand (CallbackFunctionPointer *cbf*, void * *cb*)

Connect a callbackfunction of choice. Function will be called each time the reader receive CustomCommandResult. Callback function receive [c1g2customcommandresult_t](#) type structure.

5.18.1 Parameters:

In	<i>cbf</i>	Pointer to a Static member function (SomeClass::someCallbackFunction).
In	<i>cb</i>	Object pointer (someclass)

Example Usage:

[urmsp_C1G2CustomCommand\(\)](#)

5.19 void connectCallbackLog (CallbackFunctionPointer *cbf*, void * *cb*)

Connect a callbackfunction of choice. Function will be called for logging purposes(serial port communication, errors, messages). Callback function receive [log_t](#) type structure.

5.19.1 Parameters:

In	<i>cbf</i>	Pointer to a Static member function (SomeClass::someCallbackFunction).
In	<i>cb</i>	Object pointer (someclass)

Example Usage:

[urmsp_StartROSpec\(\)](#)

5.20 int connectReader (char * *comPort*)

This function opens the serial port connection to reader. It must be called before any other call.

5.20.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.20.2 Parameters:

In	<i>comPort</i>	Com port number (COM1, COM2, ...)
----	----------------	-----------------------------------

Example Usage:

```
#ifndef CONNECTDISCONNECTEXP
#define CONNECTDISCONNECTEXP

#include <string>

std::string getError(int errorCode);

#endif // CONNECTDISCONNECTEXP
```

```
#include "connectdisconnectexp.h"
#include "apic.h"

#include <iostream>

int main()
{
    //Initialize the library
    initDll\(\);

    //Set COM port number.
    char prt[] = "com3";

    //Connecting reader
    int errorCode = connectReader(prt);

    if(errorCode == TSS_SUCCESS)
    {
        //Connection successful

        //Do something

        //Disconnecting reader
        errorCode = disconnectReader();

        if(errorCode == TSS_SUCCESS)
        {
            //Disconnection successful
        }
        else
        {
            //Disconnection failed
            std::cout << getError(errorCode);
        }
    }
    else
    {
        //Connection failed
        std::cout << getError(errorCode);
    }
}

//This function translate error code to error strings
std::string getError(int errorCode)
{
```

```
char errorSourceString[1024];
char errorCodeString[1024];

unsigned int errorSourceStringLength;
unsigned int errorCodeStringLength;

getErrorString(errorSourceString,
                &errorSourceStringLength,
                errorCodeString,
                &errorCodeStringLength,
                errorCode);

return std::string(errorCodeString,errorCodeStringLength);
}
```

5.21 int connectReaderTCP (char * *host*, unsigned int *port*)

This function opens TCP/IP connection to reader. It must be called before any other call.

5.21.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.21.2 Parameters:

in	<i>host</i>	host IPv4 address e.g. "192.168.0.110"
in	<i>port</i>	host port number e.g. "5000"

5.22 int disconnectReader ()

This function closes the connection to reader.

5.22.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

Example Usage:

[connectReader\(\)](#)

5.23 int waitForResultStruct (void * *structPointer*, unsigned int *resultType*, unsigned int *timeout*)

DEPRECATED

5.24 int urmsp_GetReaderCapabilities (void * *returnValue*, unsigned char *flags*, unsigned char *reserved*, unsigned char *requestedData*)

This function reads the general capabilities of the device like supported air protocols, version of the reader firmware, device hardware and software information.

5.24.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.24.2 Parameters:

out	<i>returnValue</i>	Returns firmware version and build number of the reader.
in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>requestedData</i>	Possible Value: 0x01. 0x01 = Firmware version and build number.

5.24.3 Parameters:

out	<i>returnValue</i>	Returns getReaderCapabilitiesResponse_t structure.
in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>requestedData</i>	Possible Value: 0x01. 0x01 = Firmware version and build number.

Example Usage:

```
#ifndef GETREADERCAPABILITIESEXP
#define GETREADERCAPABILITIESEXP

#include <string>

std::string getError(int errorCode);

#endif // GETREADERCAPABILITIESEXP
```

```
#include "getreadercapabilitiesexp.h"
#include "apic.h"

#include <iostream>

int main()
{
    //Initialize the library
    initDll\(\);

    //Set COM port number.
    char prt[] = "com3";

    //Connecting reader
    int errorCode = connectReader(prt);

    if(errorCode == TSS_SUCCESS)
    {
```

```
//Connection successful

//Get the reader firmware version.
getReaderCapabilitiesResponse_t returnValue;

errorCode = urmsp\_GetReaderCapabilities(&returnValue,
                                         0x00, //Reserved value
                                         0x00, //Reserved value
                                         TSS_GENERAL_DEVICE_CAPABILITIES);

if(errorCode == TSS_SUCCESS)
{
    //Show firmware version
    std::cout << "\n Reader F/W : "
              << std::string(returnValue.readerFirmwareVersion,
                             returnValue.firmwareVersionByteCount);
    //Show reader model
    std::cout << "\n Reader model : "
              << std::string(returnValue.modelName,4);

    //Show maximum number of antennas supported
    std::cout << "\n Maximum number of antennas supported : "
              << returnValue.maxNumberOfAntennaSupported;    }
else
{
    //Command failed
    std::cout << getError(errorCode);
}

//Disconnecting reader
errorCode = disconnectReader();

if(errorCode == TSS_SUCCESS)
{
    //Disconnection successful
}
else
{
    //Disconnection failed
    std::cout << getError(errorCode);
}
}
else
{
    //Connection failed
    std::cout << getError(errorCode);
}
}

//This function translate error code to error strings
std::string getError(int errorCode)
{
    char errorSourceString[1024];
    char errorCodeString[1024];

    unsigned int errorSourceStringLength;
    unsigned int errorCodeStringLength;

    getErrorString(errorSourceString,
                  &errorSourceStringLength,
                  errorCodeString,
                  &errorCodeStringLength,
                  errorCode);

    return std::string(errorCodeString,errorCodeStringLength);
}
```

5.25 int urmsp_StartROSpec (unsigned char *flags*, unsigned char *reserved*)

This function starts the reader operation.

5.25.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.25.2 Parameters:

in	<i>flags</i>	
in	<i>reserved</i>	

Example Usage:

```
#ifndef STARTROSPECEXP
#define STARTROSPECEXP

#include <string>

static void callBackLog(void *p,unsigned int commandType, void *structPointer);
static void callBackEPC(void *p, unsigned int commandType, void *structPointer);

int waitForResult(bool *flag, int timeout);

std::string getError(int errorCode);

#endif // STARTROSPECEXP
```

```
#include "startrospecexp.h"
#include "apic.h"

#include <iostream>
#include <mutex>

std::mutex mtx;

c1g2epc t m_epcStruct;
bool m_epcReceived = false;

int main()
{
    //pointer to class object
    void *cb = NULL;

    //Initialize the library
    initDll\(\);

    //Connect callback for log messages
    connectCallbackLog(&callBackLog,cb);

    //Connect callback for EPC messages
    connectCallbackEPC(&callBackEPC,cb);

    //Set COM port number.
    char prt[] = "com3";

    //Connecting reader
    int errorCode = connectReader(prt);

    if(errorCode == TSS_SUCCESS)
    {
        //Connection successful

        //Start reader operation
        errorCode = urmstp\_StartROSpec(0x00, //Reserved value
            0x00); //Reserved value
```

```

if(errorCode == TSS_SUCCESS)
{
    //Command successful

    //This function wait for EPC result for 1500ms
    errorCode = waitForResult(&m_epcReceived,1500);

    if(errorCode == TSS_SUCCESS)
    {
        std::cout << "\nEPC received";

        std::string epcString(m_epcStruct.epc,m_epcStruct.epcLength*2);
        std::cout << "\nEPC Length: " << m_epcStruct.epcLength;
        std::cout << "\nEPC: " << epcString;
    }
}
else
{
    //Command failed
    std::cout << getError(errorCode);
}

//Disconnecting reader
errorCode = disconnectReader();

if(errorCode == TSS_SUCCESS)
{
    //Disconnection successful
}
else
{
    //Disconnection failed
    std::cout << getError(errorCode);
}
}
else
{
    //Connection failed
    std::cout << getError(errorCode);
}
}

//This callback function receives log messages(serial port
//communication, events, error messages, etc.)
static void callBackLog(void *p,unsigned int commandType, void *structPointer)
{
    log\_t logStruct = *(log\_t*)structPointer;

    std::string logString(logStruct.log,logStruct.logLength);
    std::cout << "\nLog: " << logString;
}

int waitForResult(bool *flag, int timeout)
{
    int x;

    for(x = 0; x < timeout; x += 100)    //Waiting x millisecond
    {
        //Lock the mutex to avoid access to to flag from different thread
        mtx.lock();

        //Check if structure was written
        if(*flag)
        {
            //Reset the flag
            *flag = false;

            mtx.unlock();
            return 0;
        }
    }
}

```

```

    mtx.unlock();

    Sleep(100);
}

if(x == timeout)
{
    //Timeout
    return -1;
}
}

//This callback function is called each time the reader receives EPC number.
static void callBackEPC(void *p, unsigned int commandType, void *structPointer)
{
    //Lock the mutex to avoid access to m_epcStruct from different thread
    std::lock_guard<std::mutex> lock(mtx);

    //EPC96 or EPCDATA command type value.
    if(commandType == URMSP_EPC_96 || URMSP_EPCDATA)
    {
        m_epcStruct = *(c1g2epc_t*)structPointer;

        m_epcReceived = true;
    }
}

//This function translate error code to error strings
std::string getError(int errorCode)
{
    char errorSourceString[1024];
    char errorCodeString[1024];

    unsigned int errorSourceStringLength;
    unsigned int errorCodeStringLength;

    getErrorString(errorSourceString,
        &errorSourceStringLength,
        errorCodeString,
        &errorCodeStringLength,
        errorCode);

    return std::string(errorCodeString,errorCodeStringLength);
}

```

5.26 int urmsp_StopROSpec (unsigned char *flags*, unsigned char *reserved*)

This function stops the execution of the reader operation. Urmstp_StopROSpec overrides all other priorities and stops the execution. This basically moves the reader operation state to inactive.

5.26.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.26.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00

Example Usage:

```
#ifndef STOPROSPECEXP
#define STOPROSPECEXP

#include <string>

static void callBackLog(void *p,unsigned int commandType, void *structPointer);
static void callBackEPC(void *p, unsigned int commandType, void *structPointer);

std::string getError(int errorCode);

#endif // STOPROSPECEXP
```

```
#include "stoprospecexp.h"
#include "apic.h"

#include <iostream>

int main()
{
    //pointer to class object
    void *cb = NULL;

    //Initialize the library
    initDll\(\);

    //Connect callback for log messages
    connectCallbackLog(&callBackLog,cb);

    //Connect callback for EPC messages
    connectCallbackEPC(&callBackEPC,cb);

    //Set COM port number.
    char prt[] = "com3";

    //Connecting reader
    int errorCode = connectReader(prt);

    if(errorCode == TSS_SUCCESS)
    {
        //Connection successful

        //Set the stop trigger to null.
        //Reader operation will be terminated by urmsp_StopROSpec command
        errorCode = urmstp\_AISpecStopTrigger(0x00, //Reserved value
            0x00, //Reserved value
            TSS_AISPECSTOPTRIGGER_NULL,
            0x00);

        if(errorCode == TSS_SUCCESS)
        {
            //Command successful
```

```

//Start reader operation.
errorCode = urmsp\_StartROSpec(0x00, //Reserved value
                                0x00); //Reserved value

if(errorCode == TSS_SUCCESS)
{
    //Command successful

    //Wait while the reader read tags.
    Sleep(500);

    //Stop the reader operations.
    errorCode = urmsp\_StopROSpec(0x00, //Reserved value
                                0x00); //Reserved value

    if(errorCode == TSS_SUCCESS)
    {
        //Command successful
    }
    else
    {
        //Command failed
        std::cout << getError(errorCode);
    }
}
else
{
    //Command failed
    std::cout << getError(errorCode);
}
}
else
{
    //Command failed
    std::cout << getError(errorCode);
}
}

//Disconnecting reader
errorCode = disconnectReader();

if(errorCode == TSS_SUCCESS)
{
    //Disconnection successful
}
else
{
    //Disconnection failed
    std::cout << getError(errorCode);
}
}
else
{
    //Connection failed
    std::cout << getError(errorCode);
}
}

//This callback function receives log messages(serial port
//communication, events, error messages, etc.)
static void callBackLog(void *p,unsigned int commandType, void *structPointer)
{
    log\_t logStruct = *(log\_t*)structPointer;

    std::string logString(logStruct.log,logStruct.logLength);
    std::cout << "\nLog: " << logString;
}

//This callback function is called each time the reader receives EPC number.
static void callBackEPC(void *p, unsigned int commandType, void *structPointer)
{

```

```
std::cout << "\nCallback EPC";

c1g2epc_t m_epcStruct = *(c1g2epc_t*)structPointer;

std::string epcString(m_epcStruct.epc,m_epcStruct.epcLength*2);

std::cout << "\nEPC Length: " << m_epcStruct.epcLength;
std::cout << "\nEPC: " << epcString;
}

//This function translate error code to error strings
std::string getError(int errorCode)
{
    char errorSourceString[1024];
    char errorCodeString[1024];

    unsigned int errorSourceStringLength;
    unsigned int errorCodeStringLength;

    getErrorString(errorSourceString,
        &errorSourceStringLength,
        errorCodeString,
        &errorCodeStringLength,
        errorCode);

    return std::string(errorCodeString,errorCodeStringLength);
}
```

5.27 int urmsp_GetROSpecs (unsigned char * *returnValue*, unsigned char *flags*, unsigned char *reserved*)

This function retrieves all the reader operation that have been configured.

5.27.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.27.2 Parameters:

out	<i>returnValue</i>	0x01 Inactive 0x02 Active
in	<i>flags</i>	
in	<i>reserved</i>	Example Usage:

```
#ifndef GETROSPECEXP
#define GETROSPECEXP

#include <string>

static void callBackLog(void *p,unsigned int commandType, void *structPointer);

std::string getError(int errorCode);

#endif // GETROSPECEXP
```

```
#include "getrospecexp.h"
#include "apic.h"

#include <iostream>

int main()
{
    //pointer to class object
    void *cb = NULL;

    //Initialize the library
    initDll\(\);

    //Connect callback for log messages
    connectCallbackLog(&callBackLog,cb);

    //Set COM port number.
    char prt[] = "com3";

    //Connecting reader
    int errorCode = connectReader(prt);

    if(errorCode == TSS_SUCCESS)
    {
        //Connection successful

        //This function retrieves the reader operation status
        unsigned char retVal[1];

        errorCode = urmsp\_GetROSpecs(retVal,
                                     0x00, //Reserved value
                                     0x00); //Reserved value

        if(errorCode == TSS_SUCCESS)
        {
            if(*retVal == 1)
```

```

    {
        std::cout << "\nReader operation: Inactive";
    }
    else if (*retVal == 2)
    {
        std::cout << "\nReader operation: Active";
    }

    Sleep(100);

    //Disconnecting reader
    errorCode = disconnectReader\(\);

    if(errorCode == TSS_SUCCESS)
    {
        //Disconnection successful
    }
    else
    {
        //Disconnection failed
        std::cout << getError\(errorCode\);
    }
}
else
{
    //Command failed
    std::cout << getError\(errorCode\);
}
}
else
{
    //Connection failed
    std::cout << getError\(errorCode\);
}
}

//This callback function receives log messages(serial port
//communication, events, error messages, etc.)
static void callBackLog(void *p,unsigned int commandType, void *structPointer)
{
    log\_t logStruct = *(log\_t*)structPointer;

    std::string logString(logStruct.log,logStruct.logLength);
    std::cout << "\nLog: " << logString;
}

//This function translate error code to error strings
std::string getError(int errorCode)
{
    char errorSourceString[1024];
    char errorCodeString[1024];

    unsigned int errorSourceStringLength;
    unsigned int errorCodeStringLength;

    getErrorString(errorSourceString,
        &errorSourceStringLength,
        errorCodeString,
        &errorCodeStringLength,
        errorCode);

    return std::string(errorCodeString,errorCodeStringLength);
}

```

5.28 int urmsp_ROSpecStartTrigger (unsigned char *flags*, unsigned char *reserved*, unsigned char *rOSpecStartTriggerType*)

FUNCTION IS RESERVED FOR FUTURE USE.

This function sets the desired reader operation trigger type.

5.28.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.28.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>rOSpecStartTriggerType</i>	Possible Values: 0x00 - 0x03. 0x00 = Null – No start trigger. The only way to start the ROSpec is with a START_ROSPEC from the Client. 0x01 = Immediate 0x02 = Periodic 0x03 = GPI

5.29 int urmsp_GPITriggerValue (unsigned char *flags*, unsigned char *reserved*, unsigned short *gPIPortNum*, unsigned char *gPIEvent*, unsigned int *timeout*)

FUNCTION IS RESERVED FOR FUTURE USE.

This function sets the parameters of GPI trigger. This trigger is tied to an event on the General Purpose Input (GPI) of the Reader. The event is represented as a boolean type.

5.29.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.29.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>gPIPortNum</i>	Possible Values: 0x00. General Purpose Input Port Number.
in	<i>gPIEvent</i>	Possible Value: 0x00 - 0x01. The Boolean value that causes a GPI event to trigger.
in	<i>timeout</i>	Possible Value: 0x00. Trigger timeout in milliseconds. If set to zero, it indicates there is no timeout.

5.30 int urmsp_AISpecStopTrigger (unsigned char *flags*, unsigned char *reserved*, unsigned char *aiSpecStopTriggerType*, unsigned int *durationTriggerValue*)

This function defines the stop (i.e., terminating boundary) of an antenna inventory operation.

5.30.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.30.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>aiSpecStopTriggerType</i>	Possible Values: 0x00 - 0x01 0x00 = Null – Stop when ROSpec is done. 0x01 = Duration - NOT SUPPORTED 0x02 = GPI with a timeout value - NOT SUPPORTED 0x03 = Tag observation - NOT SUPPORTED
in	<i>durationTriggerValue</i>	Duration of AISpec in milliseconds. NOT SUPPORTED

Example Usage:

[urmsp_StopROSpec\(\)](#)

5.31 int urmsp_TagObservationTrigger (unsigned char *flags*, unsigned char *reserved*, unsigned char *triggerType*, unsigned short *numberOfTags*, unsigned short *numberOfAttempts*, unsigned short *t*, unsigned int *timeout*)

FUNCTION IS RESERVED FOR FUTURE USE.

This function sets the antenna inventory stop trigger related to tag observation.

5.31.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.31.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>triggerType</i>	Possible Values: 0x00 - 0x04 0x00 = Upon seeing N tag observations, or timeout. The definition of an "observation" is vendor specific. 0x01 = Upon seeing no more new tag observations for T ms, or timeout. The definition of an "observation" is vendor specific. 0x02 = N attempts to see all tags in the FOV, or timeout. 0x03 = Upon seeing N unique tag observations, or timeout. 0x04 = Upon seeing no more new unique tag observations for T ms, or timeout.
in	<i>numberOfTags</i>	Number of Tags.
in	<i>numberOfAttempts</i>	Number of Attempts.
in	<i>t</i>	Idle time between tag responses in milliseconds.
in	<i>timeout</i>	Trigger timeout value in milliseconds. If set to zero, it indicates that there is no timeout.

5.32 int urmsp_RFTransmitter (unsigned char *flags*, unsigned char *reserved*, unsigned short *hopTableID*, unsigned short *channelIndex*, unsigned short *transmitPower*)

This function defines the RF transmitter information.

hopTableID: The HopTableID is the index of the frequency hop table to be used by the Reader and is used when operating in frequency-hopping regulatory regions. This field is ignored in non-frequency-hopping regulatory regions.

channelIndex: The ChannelIndex is the one-based channel index in the FixedFrequencyTable to use during transmission and is used when operating in non-frequency-hopping regulatory regions. This field is ignored in frequency-hopping regulatory regions.

transmitPower: The Transmit Power defines the transmit power for the antenna expressed as an index into the TransmitPowerTable.

5.32.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.32.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>hopTableID</i>	Possible Value: 0x00 - NOT SUPPORTED
in	<i>channelIndex</i>	Possible Value: 0x00 - NOT SUPPORTED
in	<i>transmitPower</i>	Possible Values: 0x00 - 0x0E

Example Usage:

```
#ifndef RFTRANSMITTER
#define RFTRANSMITTER

#include <string>

static void callBackLog(void *p,unsigned int commandType, void *structPointer);

std::string getError(int errorCode);

#endif // RFTRANSMITTER
```

```
#include "rftransmitter.h"
#include "apic.h"

#include <iostream>

int main()
{
    //pointer to class object
    void *cb = NULL;

    //Initialize the library
    initDll\(\);

    //Connect callback for log messages
    connectCallbackLog(&callBackLog,cb);

    //Set COM port number.
    char prt[] = "com3";
```

```
//Connecting reader
int errorCode = connectReader(prt);

if(errorCode == TSS_SUCCESS)
{
    //Connection successful

    if(errorCode == TSS_SUCCESS)
    {
        //Command successful

        //Set RF trnasmitter power to MINIMUM
        errorCode = urmsp\_RFTransmitter(0x00, //Reserved value
            0x00, //Reserved value
            0x00, //Reserved value
            0x00, //Reserved value
            0x00); //Lowest RF Trnasmitter power (0).

        if(errorCode == TSS_SUCCESS)
        {
            //Command successful

            std::cout << "\nRF Transmitter power on minimum level.";
        }
        else
        {
            //Command failed
            std::cout << getError(errorCode);
        }
    }
    else
    {
        //Command failed
        std::cout << getError(errorCode);
    }
}

//Disconnecting reader
errorCode = disconnectReader();

if(errorCode == TSS_SUCCESS)
{
    //Disconnection successful
}
else
{
    //Disconnection failed
    std::cout << getError(errorCode);
}
}
else
{
    //Connection failed
    std::cout << getError(errorCode);
}
}

//This callback function receives log messages(serial port
//communication, events, error messages, etc.)
static void callBackLog(void *p,unsigned int commandType, void *structPointer)
{
    log\_t logStruct = *(log\_t*)structPointer;

    std::string logString(logStruct.log,logStruct.logLength);
    std::cout << "\nLog: " << logString;
}

//This function translate error code to error strings
std::string getError(int errorCode)
{
    char errorSourceString[1024];
```

```
char errorCodeString[1024];

unsigned int errorSourceStringLength;
unsigned int errorCodeStringLength;

getErrorString(errorSourceString,
               &errorSourceStringLength,
               errorCodeString,
               &errorCodeStringLength,
               errorCode);

return std::string(errorCodeString,errorCodeStringLength);
}
```

5.33 int urmsp_EventNotificationState (unsigned char *flags*, unsigned char *reserved*, unsigned short *eventType*, unsigned char *notificationState*)

This function is used to enable or disable notification of a single Reader event type.

5.33.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.33.2 Parameters:

In	<i>flags</i>	Possible Value: 0x00
In	<i>reserved</i>	Possible Value: 0x00
In	<i>eventType</i>	Possible Values: 0x00 - 0x09. 0x00 = Upon hopping to next channel - NOT SUPPORTED 0x01 = GPI event 0x02 = ROSpec event (start/end/preempt) 0x03 = Report buffer fill warning - NOT SUPPORTED 0x04 = Reader exception event - NOT SUPPORTED 0x05 = RFSurvey event (start/end) - NOT SUPPORTED 0x06 = AISpec event (end) - NOT SUPPORTED 0x07 = AISpec event (end) with singulation details - NOT SUPPORTED 0x08 = Antenna event (disconnect/connect) - NOT SUPPORTED 0x09 = SpecLoop event - NOT SUPPORTED
In	<i>notificationState</i>	Possible Values: 0x00 - 0x01. 0x00 = Disabled 0x01 = Enabled

Example Usage:

```
#ifndef EVENTNOTIFICATIONSTATEEXP
#define EVENTNOTIFICATIONSTATEEXP

#include <string>

static void callBackLog(void *p,unsigned int commandType, void *structPointer);
static void callBackEPC(void *p, unsigned int commandType, void *structPointer);
static void callBackEventNotification(void *p, unsigned int commandType, void *structPointer);

void gPIOEventFunction(void *s);
void rOSpecEventFunction(void *s);

std::string getError(int errorCode);

#endif // EVENTNOTIFICATIONSTATEEXP
```

```
#include "eventnotificationstateexp.h"
#include "apic.h"

#include <iostream>

int main()
{
    //pointer to class object
    void *cb = NULL;
```

```
//Initialize the library
initDll();

//Connect callback for log messages
connectCallbackLog(&callBackLog,cb);

//Connect callback for EPC messages
connectCallbackEPC(&callBackEPC,cb);

//Connect callback for Reader events
connectCallbackReaderEvents(&callBackEventNotification, cb);

//Set COM port number.
char prt[] = "com3";

//Connecting reader
int errorCode = connectReader(prt);

if(errorCode == TSS_SUCCESS)
{
    //Connection successful

    //EventNotificationState Command
    errorCode = urmsp_EventNotificationState(0x00, //Reserved value
                                             0x00, //Reserved value
                                             TSS_EVENTNOTIFICATION_ROSPEC, //Notification type
                                             TSS_EVENTNOTIFICATIONSTATE_ENABLED); //Notification state

    if(errorCode == TSS_SUCCESS)
    {
        //Command successful

        //Start Reader operation
        errorCode = urmsp_StartROSpec(0x00, //Reserved value
                                      0x00); //Reserved value

        if(errorCode == TSS_SUCCESS)
        {
            //Command successful
        }
        else
        {
            //Command failed
            std::cout << getError(errorCode);
        }
    }

    Sleep(2000);

    //Disconnecting reader
    errorCode = disconnectReader();

    if(errorCode == TSS_SUCCESS)
    {
        //Disconnection successful
    }
    else
    {
        //Disconnection failed
        std::cout << getError(errorCode);
    }
}
else
{
    //Command failed
    std::cout << getError(errorCode);
}
}
else
{
    //Connection failed
}
```

```

        std::cout << getError(errorCode);
    }
}

//This callback function receives log messages(serial port
//communication, events, error messages, etc.)
static void callBackLog(void *p,unsigned int commandType, void *structPointer)
{
    log_t logStruct = *(log_t*)structPointer;

    std::string logString(logStruct.log,logStruct.logLength);
    std::cout << "\nLog: " << logString;
}

//This callback function is called each time the reader receives EPC number.
static void callBackEPC(void *p, unsigned int commandType, void *structPointer)
{
    std::cout << "\nCallback EPC";

    c1g2epc_t m_epcStruct = *(c1g2epc_t*)structPointer;

    std::string epcString(m_epcStruct.epc,m_epcStruct.epcLength*2);

    std::cout << "\nEPC Length: " << m_epcStruct.epcLength;
    std::cout << "\nEPC: " << epcString;
}

//This callback function is called each time the reader receives EPC number.
static void callBackEventNotification(void *p, unsigned int commandType, void *structPointer)
{
    std::cout << "\nCallback Event Notification";

    switch(commandType)
    {
        case URMSP_GPIEVENT:
            gPIOEventFunction(structPointer);
            break;
        case URMSP_ROSPECEVENT:
            rOSpecEventFunction(structPointer);
            break;
    }
}

void gPIOEventFunction(void *s)
{
    gpievent_t *event = (gpievent_t*)s;

    if(event->e)
    {
        std::cout << "GPIO Port Number: " << event->gPIPortNumber;
        std::cout << "\nGPIOEvent: GPIO8 True";
    }
    else
    {
        std::cout << "GPIO Port Number: " << event->gPIPortNumber;
        std::cout << "\nGPIOEvent: GPIO8 False";
    }
}

void rOSpecEventFunction(void *s)
{
    rospecevent_t *event = (rospecevent_t*)s;

    switch(event->eventType)
    {
        case 0:
            std::cout << "\nROSpec Event: Start of Reader opeartion";
            break;
        case 1:
            std::cout << "\nROSpec Event: End of Reader operation";
            break;
    }
}

```

```
case 2:
    std::cout << "\nROSpec Event: Preemption of ROSpec";
    break;
}
}

//This function translate error code to error strings
std::string getError(int errorCode)
{
    char errorSourceString[1024];
    char errorCodeString[1024];

    unsigned int errorSourceStringLength;
    unsigned int errorCodeStringLength;

    getErrorString(errorSourceString,
        &errorSourceStringLength,
        errorCodeString,
        &errorCodeStringLength,
        errorCode);

    return std::string(errorCodeString,errorCodeStringLength);
}
```

5.34 int urmsp_C1G2TagInventoryMask (unsigned char *flags*, unsigned char *reserved*, unsigned char *mb*, unsigned short *pointer*, unsigned short *maskBitCount*, const char * *tagMask*)

This function set the information for the tag memory data pattern used for the select operation.

5.34.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.34.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>mb</i>	Memory bank to use. Possible Values: 0x00 - 0x03. 0x00 = RESERVED Memory 0x01 = EPC Memory 0x02 = TID Memory 0x03 = USER Memory
in	<i>pointer</i>	The first (msb) bit location of the specified memory bank against which to compare the TagMask.
in	<i>maskBitCount</i>	Number of bits from tagMask used for matching.
in	<i>tagMask</i>	Bit array. The pattern against which to compare. If this array is empty (0 * length), the tag mask is considered a "null filter" and will match all tags.

Example Usage:

```
#ifndef SELECTTAGEXP
#define SELECTTAGEXP

#include <string>

static void callBackLog(void *p,unsigned int commandType, void *structPointer);
static void callBackEPC(void *p, unsigned int commandType, void *structPointer);

int waitForResult(bool *flag, int timeout);

std::string getError(int errorCode);
std::string getResult(unsigned int resultType, unsigned char resultCode);

#endif // SELECTTAGEXP
```

```
#include "selecttagexp.h"
#include "apic.h"

#include <iostream>
#include <mutex>

std::mutex mtx;

c1g2epc t m_epcStruct;
bool m_epcReceived = false;

int main()
{
    //pointer to class object
    void *cb = NULL;

    //Initialize the library
```

```

initDll();

//Connect callback for log messages
connectCallbackLog(&callBackLog,cb);

//Connect callback for EPC message
connectCallbackEPC(&callBackEPC,cb);

//Set COM port number.
char prt[] = "com3";

//Connecting reader
int errorCode = connectReader(prt);

if(errorCode == TSS_SUCCESS)
{
    //Connection successful

    errorCode = urmsp_C1G2TagInventoryStateAwareSingulationAction(0x00, //Reserved value
                                                                    0x00, //Reserved value
                                                                    0x00, //Inventoried state
                                                                    0x00, //Select flag
                                                                    0x00); //A flag

    if(errorCode == TSS_SUCCESS)
    {
        Sleep(30);
        errorCode = urmsp_C1G2TagInventoryStateAwareFilterAction(0x00, //Reserved value
                                                                    0x00, //Reserved value
                                                                    0x00, //Target
                                                                    0x00); //Action
    }
    else
    {
        std::cout << getError(errorCode);
    }

    if(errorCode == TSS_SUCCESS)
    {
        Sleep(30);
        //This is the EPC number of tag we want to select
        const char tagMask[] = {0XE2, 0X00, 0X10, 0X71, 0X29, 0X12,
                                0X00, 0X91, 0X02, 0X40, 0XF1, 0X00};

        errorCode = urmsp_C1G2TagInventoryMask(0x00, //Reserved value
                                                0x00, //Reserved value
                                                TSS_TAGMEMORYBANK_EPC, // Memory bank
                                                0x20, //The first (msb) bit location of
                                                //the specified memory bank
                                                96, //Number of bits to use from tagMask
                                                tagMask); //Tag Mask
    }
    else
    {
        std::cout << getError(errorCode);
    }

    if(errorCode == TSS_SUCCESS)
    {
        Sleep(30);

        //Only the tag with matching EPC will respond
        errorCode = urmsp_StartROSpec(0x00,
                                       0x00);
    }
    else
    {
        std::cout << getError(errorCode);
    }

    if(errorCode == TSS_SUCCESS)

```

```

{
    //Command successful
    //This function wait for EPC result for 1500ms
    errorCode = waitForResult(&m_epcReceived,1500);
    if(errorCode == TSS_SUCCESS)
    {
        std::cout << "\nEPC received";
        std::string epcString(m_epcStruct.epc,m_epcStruct.epcLength*2);
        std::cout << "\nEPC Length: " << m_epcStruct.epcLength;
        std::cout << "\nEPC: " << epcString;
    }
}
else
{
    //Command failed
    std::cout << getError(errorCode);
}

if(errorCode == TSS_SUCCESS)
{
    //Command successful

    //Disconnecting reader
    errorCode = disconnectReader();

    if(errorCode == TSS_SUCCESS)
    {
        //Disconnection successful
    }
    else
    {
        //Disconnection failed
        std::cout << getError(errorCode);
    }
}
else
{
    //Command failed
    std::cout << getError(errorCode);
}
}
else
{
    //Connection failed
    std::cout << getError(errorCode);
}
}

//This callback function receives log messages(serial port
//communication, events, error messages, etc.)
static void callBackLog(void *p,unsigned int commandType, void *structPointer)
{
    log_t logStruct = *(log_t*)structPointer;

    std::string logString(logStruct.log,logStruct.logLength);
    std::cout << "\nLog: " << logString;
}

//This callback function is called each time the reader receives EPC number.
static void callBackEPC(void *p, unsigned int commandType, void *structPointer)
{
    //Lock the mutex to avoid access to m_epcStruct from different thread
    std::lock_guard<std::mutex> lock(mtx);

    //EPC96 or EPCDATA command type value.
    if(commandType == URMSP_EPC_96 || URMSP_EPCDATA)
    {
        m_epcStruct = *(c1g2epc_t*)structPointer;

        m_epcReceived = true;
    }
}

```

```

}

int waitForResult(bool *flag, int timeout)
{
    int x;
    //Waiting x millisecond
    for(x = 0; x < timeout; x += 100)
    {
        //Lock the mutex to avoid access to to flag from different thread
        mtx.lock();
        //Check if structure was written
        if(*flag)
        {
            //Reset the flag
            *flag = false;

            mtx.unlock();
            return 0;
        }
        mtx.unlock();
        Sleep(100);
    }
    if(x == timeout)
    {
        //Timeout
        return -1;
    }
}

//This function translate result code to result strings
std::string getResult(unsigned int resultType, unsigned char resultCode)
{
    char resultString[1024];

    unsigned int resultStringLength;

    getResultString(resultString,&resultStringLength,resultType,resultCode);

    return std::string(resultString,resultStringLength);
}

//This function translate error code to error strings
std::string getError(int errorCode)
{
    char errorSourceString[1024];
    char errorCodeString[1024];

    unsigned int errorSourceStringLength;
    unsigned int errorCodeStringLength;

    getErrorString(errorSourceString,
        &errorSourceStringLength,
        errorCodeString,
        &errorCodeStringLength,
        errorCode);

    return std::string(errorCodeString,errorCodeStringLength);
}

```

5.35 int urmsp_C1G2TagInventoryStateAwareFilterAction (unsigned char *flags*, unsigned char *reserved*, unsigned char *target*, unsigned char *action*)

This function is used to manage the tag states during an inventory operation.

target: This value indicates which flag in the tag to modify – whether the SL flag or its inventoried flag for a particular session.

action: Describes the action for matching and non-matching tags. The actions are specific about the tag-inventory states - e.g., do nothing, assert or deassert* SL, assign inventoried S0/S1/S2/S3 to A or B.

5.35.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.35.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00.
in	<i>reserved</i>	Possible Value: 0x00.
in	<i>target</i>	Possible Values: 0x00 - 0x04. 0x00 = SL 0x01 = Inventoried state for session S0 0x02 = Inventoried state for session S1 0x03 = Inventoried state for session S2 0x04 = Inventoried state for session S3
in	<i>action</i>	Possible Values: 0x00 - 0x07 0x00 = Matching tags: assert SL or inventoried state => A. Non-matching tags: deassert SL or inventoried state =>B. 0x01 = Matching tags: assert SL or inventoried state => A. Non-matching tags: do nothing 0x02 = Matching tags: do nothing Non-matching tags: deassert SL or inventoried state =>B 0x03 = Matching tags: negate SL or (A=>B, B=>A) Non-matching tags: do nothing 0x04 = Matching tags: deassert SL or inventoried state => B Non-matching tags: assert SL or inventoried state =>A 0x05 = Matching tags: deassert SL or inventoried state => B Non-matching tags: do nothing 0x06 = Matching tags: do nothing Non-matching tags: assert SL or inventoried state => A 0x07 = Matching tags: do nothing Non-matching tags: negate SL or (A=>B, B=>A)

Example Usage:

[urmsp_C1G2TagInventoryMask\(\)](#)

5.36 int urmsp_C1G2SingulationControl (unsigned char *flags*, unsigned char *reserved*, unsigned char *s*, unsigned short *tagPopulation*, unsigned int *tagTransitTime*)

This function provides controls particular to the singulation process in the C1G2 air protocol. The singulation process is started using a Query command in the C1G2 protocol. The Query command describes the session number, tag state, the start Q value to use, and the RF link parameters. This Singulation Parameter specifies the session, tag state and description of the target singulation environment. The following attributes are specified to provide guidance to the Reader for the singulation algorithm:

5.36.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.36.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>s</i>	Possible Values: 0x00 - 0x03. Session number to use for the inventory operation. 0x00 = Session S0 0x01 = Session S1 0x02 = Session S2 0x03 = Session S3
in	<i>tagPopulation</i>	Possible Value: 0x00(Not working yet). Tag population: This is the expected tag population in the field of view of the antenna. NOT SUPPORTED
in	<i>tagTransitTime</i>	Possible Value: 0x00(Not working yet). Tag transit time: This is the measure of expected tag mobility in the field of view of the antenna where this inventory operation is getting executed. NOT SUPPORTED

5.37 int urmsp_C1G2TagInventoryStateAwareSingulationAction (unsigned char *flags*, unsigned char *reserved*, bool *i*, bool *s*, bool *a*)

i: This is the inventoried state of the target tag population in the selected session and it corresponds to the Target field of the Class1 Gen2 Query command. Only tags that match the session state participate in the inventory round. If the TagInventoryStateAware flag is false, then the Reader ignores this field, and its up to the Reader implementation to determine the value of I used in the inventory round.

s: This is the state of the SL flag in the tag and it corresponds to the Sel field of the Class1 Gen2 Query command. Only tags that match that tag state participate in the inventory round. If the TagInventoryStateAware flag is false, then the Reader ignores this field, and its up to the Reader implementation to determine the value of S used in the inventory round.

a: If set to zero, reference the S field. If set to one, the S field is ignored.

5.37.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.37.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>i</i>	Possible Values: 0x00 - 0x01 0x00 = State A 0x01 = State B
in	<i>s</i>	Possible Values: 0x00 - 0x01 0x00 = SL 0x01 = ~SL
in	<i>a</i>	Possible Values: 0x00 - 0x01 0x00 = State A 0x01 = State B

Example Usage:

[urmsp_C1G2TagInventoryMask\(\)](#)

5.38 int urmsp_C1G2Read (unsigned char *flags*, unsigned char *reserved*, unsigned short *opSpecID*, unsigned int *accessPassword*, unsigned char *MB*, unsigned short *wordPointer*, unsigned short *wordCount*)

This function can be used to read a portion of the user memory in a tag.

5.38.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.38.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>opSpecID</i>	Possible Value: 0x00
in	<i>accessPassword</i>	Access Password is the password used by the Reader to transition the tag to the secure state so that it can read from protected tag memory regions. Password length is 4 Byte(2 Word).
in	<i>MB</i>	Memory bank to use. Possible Values: 0x00 - 0x03. 0x00 = RESERVED Memory 0x01 = EPC Memory 0x02 = TID Memory 0x03 = USER Memory
in	<i>wordPointer</i>	Starting word address.
in	<i>wordCount</i>	Number of 16-bit words to be read.

Example Usage:

47

```
#ifndef C1G2READEXP
#define C1G2READEXP

#include <string>

static void callBackLog(void *p,unsigned int commandType, void *structPointer);
static void callBackRead(void *p, unsigned int commandType, void *structPointer);

int waitForResult(bool *flag, int timeout);

std::string getResult(unsigned int resultType, unsigned char resultCode);
std::string getError(int errorCode);

#endif // C1G2READEXP
```

```
#include "c1g2readexp.h"
#include "apic.h"

#include <iostream>
#include <mutex>

std::mutex mtx;

c1g2readopspecresult t m_readResult;
bool m_readResultReceived = false;

int main()
{
    //pointer to class object
    void *cb = NULL;
```

```
//Initialize the library
initDll();

//Connect callback for log messages
connectCallbackLog(&callBackLog,cb);

//Connect callback for C1G2Read
connectCallbackC1G2Read(&callBackRead,cb);

//Set COM port number.
char prt[] = "com3";

//Connecting reader
int errorCode = connectReader(prt);

if(errorCode == TSS_SUCCESS)
{
    //Connection successful
    errorCode = urmsp_C1G2Read(0x00, //Reserved value
        0x00, //Reserved value
        0x00, //Reserved Value
        0x00000000, //Access password
        TSS_TAGMEMORYBANK_USER, //Memory bank
        0x01, //Read Address
        0x04); //Number of words to read

    if(errorCode == TSS_SUCCESS)
    {
        //This function wait for C1G2Read result for 1500ms
        errorCode = waitForResult(&m_readResultReceived,1500);

        if(errorCode == TSS_SUCCESS)
        {
            std::cout << "\nC1G2Read Result received";

            if(m_readResult.result == TSS_SUCCESS)
            {
                std::cout << "\nC1G2Read result code: "
                    << (int)m_readResult.result;

                std::cout << "\nC1G2Read result string: "
                    << getResult(URMSP_C1G2READOPSPECRESULT,
                        m_readResult.result);

                std::cout << "\nC1G2Read read data: "
                    << std::string(m_readResult.data);

                std::cout << "\nC1G2Read wordCount: "
                    << m_readResult.wordCount;
            }
            else
            {
                std::cout << getResult(URMSP_C1G2READOPSPECRESULT,
                    m_readResult.result);
            }
        }
        else
        {
            std::cout << "\nwaitForResult timeout";
            std::cout << getError(errorCode);
        }
    }

    //Disconnecting reader
    errorCode = disconnectReader();

    if(errorCode == TSS_SUCCESS)
    {
        //Disconnection successful
    }
    else

```

```

    {
        //Disconnection failed
        std::cout << getError(errorCode);
    }
}
else
{
    //Command failed
    std::cout << getError(errorCode);
}
}
else
{
    //Connection failed
    std::cout << getError(errorCode);
}
}

//This callback function receives log messages(serial port
//communication, events, error messages, etc.)
static void callBackLog(void *p,unsigned int commandType, void *structPointer)
{
    log_t logStruct = *(log_t*)structPointer;

    std::string logString(logStruct.log,logStruct.logLength);
    std::cout << "\nLog: " << logString;
}

//This callback function is called each time the reader receives
//C1G2ReadOpSpecResult the result of the C1G2Read command.
static void callBackRead(void *p, unsigned int commandType, void *structPointer)
{
    std::lock_guard<std::mutex> lock(mtx);

    //C1G2Read command type value.
    if(commandType == URMSP_C1G2READ)
    {
        m_readResult = *(c1g2readopspecresult_t*)structPointer;

        m_readResultReceived = true;
    }
}

//This function wait x milliscenond until the the EPC is written in m_epcStruct
int waitForResult(bool *flag, int timeout)
{
    int x;

    //Waiting x millisecond
    for(x = 0; x < timeout; x += 100)
    {
        //Lock the mutex to avoid access to to flag from different thread
        mtx.lock();

        //Check if structure was written
        if(*flag)
        {
            //Reset the flag
            *flag = false;

            mtx.unlock();
            return 0;
        }
        mtx.unlock();

        Sleep(100);
    }

    if(x == timeout)
    {
        //Timeout
    }
}

```

```
        return -1;
    }
}

//This function translate result code to result strings
std::string getResult(unsigned int resultType, unsigned char resultCode)
{
    char resultString[1024];

    unsigned int resultStringLength;

    getResultString(resultString,&resultStringLength,resultType,resultCode);

    return std::string(resultString,resultStringLength);
}

//This function translate error code to error strings
std::string getError(int errorCode)
{
    char errorSourceString[1024];
    char errorCodeString[1024];

    unsigned int errorSourceStringLength;
    unsigned int errorCodeStringLength;

    getErrorString(errorSourceString,
        &errorSourceStringLength,
        errorCodeString,
        &errorCodeStringLength,
        errorCode);

    return std::string(errorCodeString,errorCodeStringLength);
}
```

5.39 int urmsp_C1G2Write (unsigned char *flags*, unsigned char *reserved*, unsigned short *opSpecID*, unsigned int *accessPassword*, unsigned char *MB*, unsigned short *writeDataWordPointer*, unsigned short *wordCount*, unsigned short *writeData*)

Function used to write 4 Byte(2 Word) data to chosen memory bank.

5.39.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.39.2 Parameters:

In	<i>flags</i>	Possible Value: 0x00
In	<i>reserved</i>	Possible Value: 0x00
In	<i>opSpecID</i>	Possible Value: 0x00
In	<i>accessPassword</i>	Access Password is the password used by the Reader to transition the tag to the secure state so that it can write to protected tag memory regions. Password length is 4 Byte(2 Word).
In	<i>MB</i>	Memory bank to use. Possible Values: 0x00 - 0x03. 0x00 = RESERVED Memory 0x01 = EPC Memory 0x02 = TID Memory 0x03 = USER Memory
in	<i>writeDataWordPointer</i>	The word addresss of the first word to be written to the chosen memory bank.
in	<i>wordCount</i>	Word Count is the number of 16-bit words to be written.
in	<i>writeData</i>	Write The data to write to the chosen memory bank.

Example Usage:

```
#ifndef C1G2WRITEEXP
#define C1G2WRITEEXP

#include <string>

static void callBackLog(void *p,unsigned int commandType, void *structPointer);
static void callBackWrite(void *p, unsigned int commandType, void *structPointer);

int waitForResult(bool *flag, int timeout);

std::string getResult(unsigned int resultType, unsigned char resultCode);
std::string getError(int errorCode);

#endif // C1G2WRITEEXP
```

```
#include "c1g2writeexp.h"
#include "apic.h"

#include <iostream>
#include <mutex>

std::mutex mtx;

c1g2writeopspecresult t m_writeResult;
```

```

bool m_writeResultReceived = false;

int main()
{
    //pointer to class object
    void *cb = NULL;

    //Initialize the library
    initDll\(\);

    //Connect callback for log messages
    connectCallbackLog(&callBackLog,cb);

    //Connect callback for C1G2Write
    connectCallbackC1G2Write(&callBackWrite,cb);

    //Set COM port number.
    char prt[] = "com3";

    //Connecting reader
    int errorCode = connectReader(prt);

    if(errorCode == TSS_SUCCESS)
    {
        //Connection successful

        //C1G2Write Command
        errorCode = urmsp\_C1G2Write(0x00, //Reserved value
                                0x00, //Reserved value
                                0x00, //Reserved Value
                                0x00000000, //Access password
                                TSS_TAGMEMORYBANK_USER, //Memory bank
                                0x01, //Write Address
                                0x01, //Word count(This function support only 1 Word)
                                0x1234); //Write data

        if(errorCode == TSS_SUCCESS)
        {
            //Command successful

            //This function wait for C1G2Write result for 1500ms
            errorCode = waitForResult(&m_writeResultReceived,1500);

            if(errorCode == TSS_SUCCESS)
            {
                std::cout << "\nC1G2Write Result received";

                if(m_writeResult.result == TSS_SUCCESS)
                {
                    std::cout << "\nC1G2Write result code: "
                                << (int)m_writeResult.result;

                    std::cout << "\nC1G2Write result string: "
                                << getResult(URMSP_C1G2WRITEOPSPECRESULT,
                                    m_writeResult.result);

                    std::cout << "\nC1G2Write wordCount: "
                                << m_writeResult.wordCount;
                }
                else
                {
                    std::cout << getResult(URMSP_C1G2WRITEOPSPECRESULT,
                                    m_writeResult.result);
                }
            }
            else
            {
                std::cout << "\nwaitForResult timeout";
                std::cout << getError(errorCode);
            }
        }
    }
}

```

```

//Disconnecting reader
errorCode = disconnectReader();

if(errorCode == TSS_SUCCESS)
{
    //Disconnection successful
}
else
{
    //Disconnection failed
    std::cout << getError(errorCode);
}
}
else
{
    //Command failed
    std::cout << getError(errorCode);
}
}
else
{
    //Connection failed
    std::cout << getError(errorCode);
}
}

//This callback function receives log messages(serial port
//communication, events, error messages, etc.)
static void callBackLog(void *p,unsigned int commandType, void *structPointer)
{
    log\_t logStruct = *(log\_t*)structPointer;

    std::string logString(logStruct.log,logStruct.logLength);
    std::cout << "\nLog: " << logString;
}

//This callback function is called each time the reader receives
//C1G2WriteOpSpecResult the result of the C1G2Write command.
static void callBackWrite(void *p, unsigned int commandType, void *structPointer)
{
    std::lock_guard<std::mutex> lock(mtx);

    //C1G2Write command type value.
    if(commandType == URMSP_C1G2WRITE)
    {
        m_writeResult = *(c1g2writeopspecresult\_t*)structPointer;

        m_writeResultReceived = true;
    }
}

int waitForResult(bool *flag, int timeout)
{
    int x;

    for(x = 0; x < timeout; x += 100)    //Waiting x millisecond
    {
        //Lock the mutex to avoid access to to flag from different thread
        mtx.lock();

        //Check if structure was written
        if(*flag)
        {
            //Reset the flag
            *flag = false;

            mtx.unlock();
            return 0;
        }
        mtx.unlock();
    }
}

```

```
Sleep(100);
}

if(x == timeout)
{
    //Timeout
    return -1;
}
}

//This function translate result code to result strings
std::string getResult(unsigned int resultType, unsigned char resultCode)
{
    char resultString[1024];

    unsigned int resultStringLength;

    getResultString(resultString,&resultStringLength,resultType,resultCode);

    return std::string(resultString,resultStringLength);
}

//This function translate error code to error strings
std::string getError(int errorCode)
{
    char errorSourceString[1024];
    char errorCodeString[1024];

    unsigned int errorSourceStringLength;
    unsigned int errorCodeStringLength;

    getErrorString(errorSourceString,
        &errorSourceStringLength,
        errorCodeString,
        &errorCodeStringLength,
        errorCode);

    return std::string(errorCodeString,errorCodeStringLength);
}
```

5.40 int urmsp_C1G2Kill (unsigned char *flags*, unsigned char *reserved*, unsigned short *opSpecID*, unsigned int *killPassword*)

This function can be used to kill a EPC Class 1 Gen 2 tag.

5.40.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.40.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>opSpecID</i>	Possible Value: 0x00
in	<i>killPassword</i>	Password required to execute C1G2Kill command. Password length 4 Byte(2 Word).

Example Usage:

```
#ifndef C1G2KILLEXP
#define C1G2KILLEXP

#include <string>

static void callBackLog(void *p,unsigned int commandType, void *structPointer);
static void callBackKill(void *p, unsigned int commandType, void *structPointer);

int waitForResult(bool *flag, int timeout);

std::string getResult(unsigned int resultType, unsigned char resultCode);
std::string getError(int errorCode);

#endif // C1G2KILLEXP
```

```
#include "c1g2killexp.h"
#include "apic.h"

#include <iostream>
#include <mutex>

std::mutex mtx;

c1g2writeopspecresult_t m_writeResult;
bool m_writeResultReceived = false;

c1g2killopspecresult_t m_killResult;
bool m_killResultReceived = false;

int main()
{
    //pointer to class object
    void *cb = NULL;

    //Initialize the library
    initDll\(\);

    //Connect callback for log messages
    connectCallbackLog(&callBackLog,cb);

    //Connect callback for C1G2kill
    connectCallbackC1G2Kill(&callBackKill,cb);

    //Set COM port number.
    char prt[] = "com3";
```

```
//Connecting reader
int errorCode = connectReader(prt);

if(errorCode == TSS_SUCCESS)
{
    //Connection successful

    //C1G2Kill Command
    errorCode = urmsp\_C1G2Kill(0x00, //Reserved value
        0x00, //Reserved value
        0x00, //Reserved value
        0x12345678); //Kill password

    if(errorCode == TSS_SUCCESS)
    {
        //Command successful

        //This function wait for C1G2Write result for 1500ms
        errorCode = waitForResult(&m_killResultReceived,1500);

        if(errorCode == TSS_SUCCESS)
        {
            std::cout << "\nC1G2Kill Result received";

            if(m_killResult.result == TSS_SUCCESS)
            {
                std::cout << "\nC1G2Kill result code: "
                    << (int)m_killResult.result;

                std::cout << "\nC1G2Kill result string: "
                    << getResult(URMSP_C1G2KILLOPSPECRESULT,
                        m_killResult.result);
            }
            else
            {
                std::cout << "\n" << getResult(URMSP_C1G2KILLOPSPECRESULT,
                    m_killResult.result);
            }
        }
        else
        {
            std::cout << "\nwaitForResult timeout";
            std::cout << getError(errorCode);
        }
    }

    //Disconnecting reader
    errorCode = disconnectReader();

    if(errorCode == TSS_SUCCESS)
    {
        //Disconnection successful
    }
    else
    {
        //Disconnection failed
        std::cout << getError(errorCode);
    }
}
else
{
    //Command failed
    std::cout << getError(errorCode);
}
}
else
{
    //Connection failed
    std::cout << getError(errorCode);
}
}
```

```
//This callback function receives log messages(serial port
//communication, events, error messages, etc.)
static void callBackLog(void *p,unsigned int commandType, void *structPointer)
{
    log_t logStruct = *(log_t*)structPointer;

    std::string logString(logStruct.log,logStruct.logLength);
    std::cout << "\nLog: " << logString;
}

//This callback function is called each time the reader receives
//C1G2KillOpSpecResult the result of the C1G2Kill command.
static void callBackKill(void *p, unsigned int commandType, void *structPointer)
{
    std::lock_guard<std::mutex> lock(mtx);

    //C1G2Kill command type value.
    if(commandType == URMSP_C1G2KILL)
    {
        m_killResult = *(c1g2killopspecresult_t*)structPointer;

        m_killResultReceived = true;
    }
}

int waitForResult(bool *flag, int timeout)
{
    int x;

    //Waiting x millisecond
    for(x = 0; x < timeout; x += 100)
    {
        //Lock the mutex to avoid access to to flag from different thread
        mtx.lock();

        //Check if structure was written
        if(*flag)
        {
            //Reset the flag
            *flag = false;

            mtx.unlock();
            return 0;
        }
        mtx.unlock();

        Sleep(100);
    }

    if(x == timeout)
    {
        //Timeout
        return -1;
    }
}

//This function translate result code to result strings
std::string getResult(unsigned int resultType, unsigned char resultCode)
{
    char resultString[1024];

    unsigned int resultStringLength;

    getResultString(resultString,&resultStringLength,resultType,resultCode);

    return std::string(resultString,resultStringLength);
}

//This function translate error code to error strings
std::string getError(int errorCode)
{

```

```
char errorSourceString[1024];  
char errorCodeString[1024];  
  
unsigned int errorSourceStringLength;  
unsigned int errorCodeStringLength;  
  
getErrorString(errorSourceString,  
                &errorSourceStringLength,  
                errorCodeString,  
                &errorCodeStringLength,  
                errorCode);  
  
return std::string(errorCodeString,errorCodeStringLength);  
}
```

5.41 int urmsp_C1G2Lock (unsigned char *flags*, unsigned char *reserved*, unsigned short *opSpecID*, unsigned int *accessPassword*)

This function execute the Lock command. The five data fields for which we can define access control using the lock command are: Kill Password, Access Password, EPC memory, TID memory and User memory. The access privilege updates are expressed as a list of C1G2LockPayload Parameters, one for each memory location. The Access Password provides the password to enter the secured state. A Reader can perform a lock operation on a tag only if the tag is in the secured state. The tag enters the secured state only using the Access Password (if a non-zero value).

5.41.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.41.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>opSpecID</i>	Possible Value: 0x00
in	<i>accessPassword</i>	Access Password is the password used by the Reader to transition the tag to the secure state so that it can write to protected tag memory regions. Password length is 4 Byte(2 Word).

Example Usage:

```
#ifndef C1G2LOCKEXP
#define C1G2LOCKEXP

#include <string>

static void callBackLog(void *p,unsigned int commandType, void *structPointer);
static void callBackLock(void *p, unsigned int commandType, void *structPointer);

int waitForResult(bool *flag, int timeout);

std::string getResult(unsigned int resultType, unsigned char resultCode);
std::string getError(int errorCode);

#endif // C1G2LOCKEXP
```

```
#include "c1g2lockexp.h"
#include "apic.h"

#include <iostream>
#include <mutex>

std::mutex mtx;

c1g2lockopspecresult t m_lockResult;
bool m_lockResultReceived = false;

int main()
{
    //pointer to class object
    void *cb = NULL;

    //Initialize the library
    initDll\(\);

    //Connect callback for log messages
    connectCallbackLog(&callBackLog,cb);
```

```
//Connect callback for C1G2Read
connectCallbackC1G2Lock(&callBackLock,cb);

//Set COM port number.
char prt[] = "com3";

//Connecting reader
int errorCode = connectReader(prt);

if(errorCode == TSS_SUCCESS)
{
    //Connection successful

    //Set Lock Action
    errorCode = urmsp_C1G2LockPayload(0x00, //Reserved value
                                     0x00, //Reserved value
                                     TSS_C1G2LOCKPRIVILEGE_READWRITE,
                                     TSS_C1G2LOCKDATAFIELD_ACCESSPWD);

    if(errorCode == TSS_SUCCESS)
    {
        //Apply the lock actions
        //To perform C1G2Lock Access password have to be a non zero value
        errorCode = urmsp_C1G2Lock(0x00, //Reserved value
                                   0x00, //Reserved value
                                   0x00, //Reserved value
                                   0x12345678); //Access Password

        if(errorCode == TSS_SUCCESS)
        {
            //This function wait for C1G2Read result for 1500ms
            errorCode = waitForResult(&m_lockResultReceived,1500);

            if(errorCode == TSS_SUCCESS)
            {
                std::cout << "\nC1G2Lock Result received";

                if(m_lockResult.result == TSS_SUCCESS)
                {
                    std::cout << "\nC1G2Lock result code: "
                                << (int)m_lockResult.result;

                    std::cout << "\nC1G2Lock result string: "
                                << getResult(URMSP_C1G2LOCKOPSPECRESULT,
                                              m_lockResult.result);
                }
                else
                {
                    std::cout << getResult(URMSP_C1G2LOCKOPSPECRESULT,
                                              m_lockResult.result);
                }
            }
            else
            {
                std::cout << "\nwaitForResult timeout";
                std::cout << getError(errorCode);
            }
        }
    }

    //Disconnecting reader
    errorCode = disconnectReader();

    if(errorCode == TSS_SUCCESS)
    {
        //Disconnection successful
    }
    else
    {
        //Disconnection failed
    }
}
```

```

        std::cout << getError(errorCode);
    }
}
else
{
    //Command failed
    std::cout << getError(errorCode);
}
}
else
{
    //Connection failed
    std::cout << getError(errorCode);
}
}

//This callback function receives log messages(serial port
//communication, events, error messages, etc.)
static void callBackLog(void *p,unsigned int commandType, void *structPointer)
{
    log_t logStruct = *(log_t*)structPointer;

    std::string logString(logStruct.log,logStruct.logLength);
    std::cout << "\nLog: " << logString;
}

//This callback function is called each time the reader receives
//C1G2LockOpSpecResult the result of the C1G2Lock command.
static void callBackLock(void *p, unsigned int commandType, void *structPointer)
{
    std::lock_guard<std::mutex> lock(mtx);

    //C1G2Lock command type value.
    if(commandType == URMSP_C1G2LOCK)
    {
        m_lockResult = *(c1g2lockopspecresult_t*)structPointer;

        m_lockResultReceived = true;
    }
}

//This function wait x milliscenond until the the EPC is written in m_epcStruct
int waitForResult(bool *flag, int timeout)
{
    int x;

    //Waiting x millisecond
    for(x = 0; x < timeout; x += 100)
    {
        //Lock the mutex to avoid access to to flag from different thread
        mtx.lock();

        //Check if structure was written
        if(*flag)
        {
            //Reset the flag
            *flag = false;

            mtx.unlock();
            return 0;
        }
        mtx.unlock();

        Sleep(100);
    }

    if(x == timeout)
    {
        //Timeout
        return -1;
    }
}

```

```
}

//This function translate result code to result strings
std::string getResult(unsigned int resultType, unsigned char resultCode)
{
    char resultString[1024];

    unsigned int resultStringLength;

    getResultString(resultString,&resultStringLength,resultType,resultCode);

    return std::string(resultString,resultStringLength);
}

//This function translate error code to error strings
std::string getError(int errorCode)
{
    char errorSourceString[1024];
    char errorCodeString[1024];

    unsigned int errorSourceStringLength;
    unsigned int errorCodeStringLength;

    getErrorString(errorSourceString,
        &errorSourceStringLength,
        errorCodeString,
        &errorCodeStringLength,
        errorCode);

    return std::string(errorCodeString,errorCodeStringLength);
}
```

5.42 int urmsp_C1G2LockPayload (unsigned char *flags*, unsigned char *reserved*, unsigned char *privilege*, unsigned char *dataField*)

This function sets the access privileges (read/write/permalock). The five data fields for which we can define access control using the lock command are: Kill Password, Access Password, EPC memory, TID memory and User memory.

5.42.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.42.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>privilege</i>	Value indicates the access privilege to be applied. Possible Values: 0x00 - 0x03 0x00 = Read/Write 0x01 = Permalock 0x02 = Permaunlock 0x03 = Unlock
in	<i>dataField</i>	Value indicates to which data field the access privilege will be applied. Possible Values: 0x00 - 0x04 0x00 = Kill Password 0x01 = Access Password 0x02 = EPC Memory 0x03 = TID Memory 0x04 = User Memory

Example Usage:

[urmsp_C1G2Lock\(\)](#)

5.43 int urmsp_C1G2BlockErase (unsigned char *flags*, unsigned char *reserved*, unsigned short *opSpecID*, unsigned int *accessPassword*, unsigned char *MB*, unsigned short *wordPointer*, unsigned short *wordCount*)

BlockErase allows to erase multiple words in a Tag's Reserved memory, EPC memory, TID memory, or the currently open file in User memory.

5.43.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.43.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>opSpecID</i>	Possible Value: 0x00
in	<i>accessPassword</i>	Access Password is the password used by the Reader to transition the tag to the secure state so that it can write to protected tag memory regions. Password length is 4 Byte(2 Word).
in	<i>MB</i>	Memory bank to use. Possible Values: 0x00 - 0x03. 0x00 = RESERVED Memory 0x01 = EPC Memory 0x02 = TID Memory 0x03 = USER Memory
in	<i>wordPointer</i>	specifies the starting word address for the memory erase, where words are 16 bits in length.
in	<i>wordCount</i>	specifies the number of 16-bit words to be erased.

Example Usage:

```
#ifndef C1G2BLOCKERASEEXP
#define C1G2BLOCKERASEEXP

#include <string>

static void callBackLog(void *p,unsigned int commandType, void *structPointer);
static void callBackBlockErase(void *p, unsigned int commandType, void *structPointer);

int waitForResult(bool *flag, int timeout);

std::string getResult(unsigned int resultType, unsigned char resultCode);
std::string getError(int errorCode);

#endif // C1G2BLOCKERASEEXP
```

```
#include "c1g2blockeraseexp.h"
#include "apic.h"

#include <iostream>
#include <mutex>

std::mutex mtx;

c1g2blockeraseopspecresult_t m_blockEraseResult;
bool m_blockEraseResultReceived = false;
```

```

int main()
{
    //pointer to class object
    void *cb = NULL;

    //Initialize the library
    initDll\(\);

    //Connect callback for log messages
    connectCallbackLog(&callBackLog,cb);

    //Connect callback for C1G2Write
    connectCallbackC1G2BlockErase(&callBackBlockErase,cb);

    //Set COM port number.
    char prt[] = "com3";

    //Connecting reader
    int errorCode = connectReader(prt);

    if(errorCode == TSS_SUCCESS)
    {
        //Connection successful

        //C1G2BlockErase Command
        errorCode = urmsp\_C1G2BlockErase(0x00, //Reserved value
            0x00, //Reserved value
            0x00, //Reserved value
            0x00000000, //Access password
            TSS_TAGMEMORYBANK_USER, //Memory bank
            0x00, //Erase address
            0x02); //Number of word to erase

        if(errorCode == TSS_SUCCESS)
        {
            //Command successful

            //This function wait for C1G2BlockErase result for 1500ms
            errorCode = waitForResult(&m_blockEraseResultReceived,1500);

            if(errorCode == TSS_SUCCESS)
            {
                std::cout << "\nC1G2BlockErase Result received";

                if(m_blockEraseResult.result == TSS_SUCCESS)
                {
                    std::cout << "\nC1G2BlockErase result code: "
                        << (int)m_blockEraseResult.result;

                    std::cout << "\nC1G2BlockErase result string: "
                        << getResult(URMSP_C1G2BLOCKERASEOPSPECRESULT,
                            m_blockEraseResult.result);

                    std::cout << "\nC1G2BlockErase wordCount: "
                        << m_blockEraseResult.wordCount;
                }
                else
                {
                    std::cout << getResult(URMSP_C1G2BLOCKERASEOPSPECRESULT,
                        m_blockEraseResult.result);
                }
            }
            else
            {
                std::cout << "\nwaitForResult timeout";
                std::cout << getError(errorCode);
            }

            //Disconnecting reader
            errorCode = disconnectReader();
        }
    }
}

```

```

        if(errorCode == TSS_SUCCESS)
        {
            //Disconnection successful
        }
        else
        {
            //Disconnection failed
            std::cout << getError(errorCode);
        }
    }
    else
    {
        //Command failed
        std::cout << getError(errorCode);
    }
}
else
{
    //Connection failed
    std::cout << getError(errorCode);
}
}

//This callback function receives log messages(serial port
//communication, events, error messages, etc.)
static void callBackLog(void *p,unsigned int commandType, void *structPointer)
{
    log t logStruct = *(log t*)structPointer;

    std::string logString(logStruct.log,logStruct.logLength);
    std::cout << "\nLog: " << logString;
}

//This callback function is called each time the reader receives
//C1G2BlockEraseOpSpecResult the result of the C1G2BlockErase command.
static void callBackBlockErase(void *p, unsigned int commandType, void *structPointer)
{
    std::lock_guard<std::mutex> lock(mtx);

    //C1G2BlockErase command type value.
    if(commandType == URMSP_C1G2BLOCKERASE)
    {
        m_blockEraseResult = *(c1g2blockerasespecresult t*)structPointer;

        m_blockEraseResultReceived = true;
    }
}

int waitForResult(bool *flag, int timeout)
{
    int x;

    for(x = 0; x < timeout; x += 100)    //Waiting x millisecond
    {
        //Lock the mutex to avoid access to to flag from different thread
        mtx.lock();

        //Check if structure was written
        if(*flag)
        {
            //Reset the flag
            *flag = false;

            mtx.unlock();
            return 0;
        }
        mtx.unlock();

        Sleep(100);
    }
}

```

```
if(x == timeout)
{
    //Timeout
    return -1;
}

//This function translate result code to result strings
std::string getResult(unsigned int resultType, unsigned char resultCode)
{
    char resultString[1024];

    unsigned int resultStringLength;

    getResultString(resultString,&resultStringLength,resultType,resultCode);

    return std::string(resultString,resultStringLength);
}

//This function translate error code to error strings
std::string getError(int errorCode)
{
    char errorSourceString[1024];
    char errorCodeString[1024];

    unsigned int errorSourceStringLength;
    unsigned int errorCodeStringLength;

    getErrorString(errorSourceString,
        &errorSourceStringLength,
        errorCodeString,
        &errorCodeStringLength,
        errorCode);

    return std::string(errorCodeString,errorCodeStringLength);
}
```

5.44 int urmsp_C1G2BlockWrite (unsigned char *flags*, unsigned char *reserved*, unsigned short *opSpecID*, unsigned int *accessPassword*, unsigned char *MB*, unsigned short *writeDataWordPointer*, unsigned short *wordCount*, const char * *writeData*)

BlockWrite allows to write multiple words in a Tag's Reserved memory, EPC memory, TID memory, or the currently open file in User memory.

5.44.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.44.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>opSpecID</i>	Possible Value: 0x00
in	<i>accessPassword</i>	Access Password is the password used by the Reader to transition the tag to the secure state so that it can write to protected tag memory regions. Password length is 4 Byte(2 Word).
in	<i>MB</i>	Memory bank to use. Possible Values: 0x00 - 0x03. 0x00 = RESERVED Memory 0x01 = EPC Memory 0x02 = TID Memory 0x03 = USER Memory
in	<i>writeDataWordPointer</i>	The word addresss of the first word to be written to the chosen memory bank.
in	<i>wordCount</i>	Word Count is the number of 16-bit words to be written.
in	<i>writeData</i>	Write The data to write to the chosen memory bank.

Example Usage:

```
#ifndef C1G2BLOCKWRITEEXP
#define C1G2BLOCKWRITEEXP

#include <string>

static void callBackLog(void *p,unsigned int commandType, void *structPointer);
static void callBackBlockWrite(void *p, unsigned int commandType, void *structPointer);

int waitForResult(bool *flag, int timeout);

std::string getResult(unsigned int resultType, unsigned char resultCode);
std::string getError(int errorCode);

#endif // C1G2BLOCKWRITEEXP
```

```
#include "c1g2blockwriteexp.h"
#include "apic.h"

#include <iostream>
#include <mutex>

std::mutex mtx;
```

```

c1g2blockwriteopspecresult t m_blockWriteResult;
bool m_blockWriteResultReceived = false;

int main()
{
    //pointer to class object
    void *cb = NULL;

    //Initialize the library
    initDll\(\);

    //Connect callback for log messages
    connectCallbackLog(&callBackLog,cb);

    //Connect callback for C1G2BlockWrite
    connectCallbackC1G2BlockWrite(&callBackBlockWrite,cb);

    //Set COM port number.
    char prt[] = "com3";

    //Connecting reader
    int errorCode = connectReader(prt);

    if(errorCode == TSS_SUCCESS)
    {
        //Connection successful

        //Write data 4 Byte = 2 Word
        const char writeData[4] = {0x12,0x34,0x56,0x78};

        //C1G2BlockWrite Command
        errorCode = urmsp\_C1G2BlockWrite(0x00, //Reserved value
            0x00, //Reserved value
            0x00, //Reserved Value
            0x00000000, //Access password
            TSS_TAGMEMORYBANK_USER, //Memory bank
            0x00, //Write Address
            0x02, //Word count(Tag specific)
            writeData); //Write data

        //NOTE: Maximum word count is Tag specific value

        if(errorCode == TSS_SUCCESS)
        {
            //Command successful

            //This function wait for C1G2BlockWrite result for 1500ms
            errorCode = waitForResult(&m_blockWriteResultReceived,1500);

            if(errorCode == TSS_SUCCESS)
            {
                std::cout << "\nC1G2BlockWrite Result received";

                if(m_blockWriteResult.result == TSS_SUCCESS)
                {
                    std::cout << "\nC1G2BlockWrite result code: "
                        << (int)m_blockWriteResult.result;

                    std::cout << "\nC1G2BlockWrite result string: "
                        << getResult(URMSP_C1G2BLOCKWRITEOPSPECRESULT,
                            m_blockWriteResult.result);

                    std::cout << "\nC1G2BlockWrite wordCount: "
                        << m_blockWriteResult.wordCount;
                }
            }
            else
            {
                std::cout << getResult(URMSP_C1G2BLOCKWRITEOPSPECRESULT,
                    m_blockWriteResult.result);
            }
        }
    }
}

```

```

else
{
    std::cout << "\nwaitForResult timeout";
    std::cout << getError(errorCode);
}

//Disconnecting reader
errorCode = disconnectReader();

if(errorCode == TSS_SUCCESS)
{
    //Disconnection successful
}
else
{
    //Disconnection failed
    std::cout << getError(errorCode);
}
}
else
{
    //Command failed
    std::cout << getError(errorCode);
}
}
else
{
    //Connection failed
    std::cout << getError(errorCode);
}
}

//This callback function receives log messages(serial port
//communication, events, error messages, etc.)
static void callBackLog(void *p,unsigned int commandType, void *structPointer)
{
    log\_t logStruct = *(log\_t*)structPointer;

    std::string logString(logStruct.log,logStruct.logLength);
    std::cout << "\nLog: " << logString;
}

//This callback function is called each time the reader receives
//C1G2BlockWriteOpSpecResult the result of the C1G2BlockWrite command.
static void callBackBlockWrite(void *p, unsigned int commandType, void *structPointer)
{
    std::lock_guard<std::mutex> lock(mtx);

    //C1G2BlockWrite command type value.
    if(commandType == URMSP_C1G2BLOCKWRITE)
    {
        m_blockWriteResult = *(c1g2blockwriteopspecresult\_t*)structPointer;

        m_blockWriteResultReceived = true;
    }
}

int waitForResult(bool *flag, int timeout)
{
    int x;

    for(x = 0; x < timeout; x += 100)    //Waiting x millisecond
    {
        //Lock the mutex to avoid access to to flag from different thread
        mtx.lock();

        //Check if structure was written
        if(*flag)
        {
            //Reset the flag
            *flag = false;

```

```
        mtx.unlock();
        return 0;
    }
    mtx.unlock();

    Sleep(100);
}

if(x == timeout)
{
    //Timeout
    return -1;
}
}

//This function translate result code to result strings
std::string getResult(unsigned int resultType, unsigned char resultCode)
{
    char resultString[1024];

    unsigned int resultStringLength;

    getResultString(resultString,&resultStringLength,resultType,resultCode);

    return std::string(resultString,resultStringLength);
}

//This function translate error code to error strings
std::string getError(int errorCode)
{
    char errorSourceString[1024];
    char errorCodeString[1024];

    unsigned int errorSourceStringLength;
    unsigned int errorCodeStringLength;

    getErrorString(errorSourceString,
        &errorSourceStringLength,
        errorCodeString,
        &errorCodeStringLength,
        errorCode);

    return std::string(errorCodeString,errorCodeStringLength);
}
```

5.45 int urmsp_C1G2BlockPermalock (unsigned char *flags*, unsigned char *reserved*, unsigned short *opSpecID*, unsigned int *accessPassword*, unsigned char *MB*, unsigned short *blockPointer*, unsigned short *blockMaskWordCount*, const char * *blockMask*)

BlockPermalock allows to Permalock one or more memory blocks in the User memory

5.45.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.45.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>opSpecID</i>	Possible Value: 0x00
in	<i>accessPassword</i>	Access Password is the password used by the Reader to transition the tag to the secure state so that it can write to protected tag memory regions. Password length is 4 Byte(2 Word).
in	<i>MB</i>	Memory bank to use. Possible Values: 0x00 - 0x03. 0x00 = RESERVED Memory 0x01 = EPC Memory 0x02 = TID Memory 0x03 = USER Memory
in	<i>blockPointer</i>	specifies the starting address for Mask, in units of 16 blocks.
in	<i>blockMaskWordCount</i>	specifies the range of Mask, starting at blockPointer and ending (16×BlockRange)–1 blocks later.
in	<i>blockMask</i>	specifies which memory blocks a Tag permalocks. Mask bit = 0: Retain the current permalock setting for the corresponding memory block. Mask bit = 1: Permalock the corresponding memory block. If a block is already permalocked then a Tag shall retain the current permalock setting. A memory block, once permalocked, cannot be un-permalocked.

Example Usage:

```
#ifndef C1G2BLOCKPERMALOCKEXP
#define C1G2BLOCKPERMALOCKEXP

#include <string>

static void callBackLog(void *p,unsigned int commandType, void *structPointer);
static void callBackBlockPermalock(void *p, unsigned int commandType, void *structPointer);

int waitForResult(bool *flag, int timeout);

std::string getResult(unsigned int resultType, unsigned char resultCode);
std::string getError(int errorCode);

#endif // C1G2BLOCKPERMALOCKEXP
```

```
#include "c1g2blockpermalockexp.h"
#include "apic.h"

#include <iostream>
#include <mutex>

std::mutex mtx;

c1g2blockpermalockopspecresult t m_blockPermalockResult;
bool m_blockPermalockResultReceived = false;

int main()
{
    //pointer to class object
    void *cb = NULL;

    //Initialize the library
    initDll\(\);

    //Connect callback for log messages
    connectCallbackLog(&callBackLog,cb);

    //Connect callback for C1G2BlockPermalock
    connectCallbackC1G2BlockPermalock(&callBackBlockPermalock,cb);

    //Set COM port number.
    char prt[] = "com3";

    //Connecting reader
    int errorCode = connectReader(prt);

    if(errorCode == TSS_SUCCESS)
    {
        //Connection successful

        //Block Mask to lock first block
        //| 1000 0000|0000 0000|
        const char blockMask[2] = {0x80,0x00};

        //C1G2BlockPermalock Command
        errorCode = urmsp\_C1G2BlockPermalock(0x00, //Reserved value
            0x00, //Reserved value
            0x00, //Reserved value
            0x00000000, //Access Password
            TSS_TAGMEMORYBANK_USER, //Memory bank
            0x00, //Block pointer
            0x01, //Block Mask Word Count
            blockMask); //BlockMask

        if(errorCode == TSS_SUCCESS)
        {
            //Command successful

            //This function wait for C1G2BlockPermalock result for 1500ms
            errorCode = waitForResult(&m_blockPermalockResultReceived,1500);

            if(errorCode == TSS_SUCCESS)
            {
                std::cout << "\nC1G2BlockPermalock Result received";

                if(m_blockPermalockResult.result == TSS_SUCCESS)
                {
                    std::cout << "\nC1G2BlockPermalock result code: "
                        << (int)m_blockPermalockResult.result;

                    std::cout << "\nC1G2BlockPermalock result string: "
                        << getResult(URMSP_C1G2BLOCKPERMALOCKOPSPECRESULT,
                            m_blockPermalockResult.result);
                }
            }
            else
            {

```

```

        std::cout << getResult(URMSP_C1G2BLOCKPERMALOCKOPSPECRESULT,
                               m_blockPermalockResult.result);
    }
}
else
{
    std::cout << "\nwaitForResult timeout";
    std::cout << getError(errorCode);
}

//Disconnecting reader
errorCode = disconnectReader();

if(errorCode == TSS_SUCCESS)
{
    //Disconnection successful
}
else
{
    //Disconnection failed
    std::cout << getError(errorCode);
}
}
else
{
    //Command failed
    std::cout << getError(errorCode);
}
}
else
{
    //Connection failed
    std::cout << getError(errorCode);
}
}

//This callback function receives log messages(serial port
//communication, events, error messages, etc.)
static void callBackLog(void *p,unsigned int commandType, void *structPointer)
{
    log_t logStruct = *(log_t*)structPointer;

    std::string logString(logStruct.log,logStruct.logLength);
    std::cout << "\nLog: " << logString;
}

//This callback function is called each time the reader receives
//C1G2BlockPermalockOpSpecResult the result of the C1G2BlockPermalock command.
static void callBackBlockPermalock(void *p, unsigned int commandType, void *structPointer)
{
    std::lock_guard<std::mutex> lock(mtx);

    //C1G2BlockPermalock command type value.
    if(commandType == URMSP_C1G2BLOCKPERMALOCK)
    {
        m_blockPermalockResult = *(c1g2blockpermalockopspecresult_t*)structPointer;

        m_blockPermalockResultReceived = true;
    }
}

int waitForResult(bool *flag, int timeout)
{
    int x;

    for(x = 0; x < timeout; x += 100)    //Waiting x millisecond
    {
        //Lock the mutex to avoid access to to flag from different thread
        mtx.lock();

        //Check if structure was written
    }
}

```

```
if(*flag)
{
    //Reset the flag
    *flag = false;

    mtx.unlock();
    return 0;
}
mtx.unlock();

Sleep(100);
}

if(x == timeout)
{
    //Timeout
    return -1;
}
}

//This function translate result code to result strings
std::string getResult(unsigned int resultType, unsigned char resultCode)
{
    char resultString[1024];

    unsigned int resultStringLength;

    getResultString(resultString,&resultStringLength,resultType,resultCode);

    return std::string(resultString,resultStringLength);
}

//This function translate error code to error strings
std::string getError(int errorCode)
{
    char errorSourceString[1024];
    char errorCodeString[1024];

    unsigned int errorSourceStringLength;
    unsigned int errorCodeStringLength;

    getErrorString(errorSourceString,
        &errorSourceStringLength,
        errorCodeString,
        &errorCodeStringLength,
        errorCode);

    return std::string(errorCodeString,errorCodeStringLength);
}
```

5.46 int urmsp_C1G2GetBlockPermalockStatus (unsigned char *flags*, unsigned char *reserved*, unsigned short *opSpecID*, unsigned int *accessPassword*, unsigned char *MB*, unsigned short *blockPointer*, unsigned short *blockRange*)

GetBlockPermalockStatus command retrieves the BlockPermalock status from a tag.

5.46.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.46.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>opSpecID</i>	Possible Value: 0x00
in	<i>accessPassword</i>	Access Password is the password used by the Reader to transition the tag to the secure state so that it can write to protected tag memory regions. Password length is 4 Byte(2 Word).
in	<i>MB</i>	Memory bank to use. Possible Values: 0x00 - 0x03. 0x00 = RESERVED Memory 0x01 = EPC Memory 0x02 = TID Memory 0x03 = USER Memory
in	<i>blockPointer</i>	Specifies the starting address to retrieve in units of 16 blocks.
in	<i>blockRange</i>	The range of blocks to retrieve, starting at blockPointer and ending ((16*blockRange) – 1) blocks later.

Example Usage:

```
#ifndef C1G2GETBLOCKPERMALOCKSTATUSEXP
#define C1G2GETBLOCKPERMALOCKSTATUSEXP

#include <string>

static void callBackLog(void *p,unsigned int commandType, void *structPointer);
static void callBackGetBlockPermalockStatus(void *p, unsigned int commandType, void *structPointer);

int waitForResult(bool *flag, int timeout);

std::string getResult(unsigned int resultType, unsigned char resultCode);
std::string getError(int errorCode);

#endif // C1G2GETBLOCKPERMALOCKSTATUSEXP
```

```
#include "c1g2getblockpermalockstatusexp.h"
#include "apic.h"

#include <iostream>
#include <mutex>

std::mutex mtx;

c1g2getblockpermalockstatusopspecresult_t m_getBlockPermalockStatusResult;
bool m_getBlockPermalockStatusResultReceived = false;
```

```

int main()
{
    //pointer to class object
    void *cb = NULL;

    //Initialize the library
    initDll\(\);

    //Connect callback for log messages
    connectCallbackLog(&callBackLog,cb);

    //Connect callback for C1G2GetBlockPermalockStatus
    connectCallbackC1G2GetBlockPermalockStatus(&callBackGetBlockPermalockStatus,cb);

    //Set COM port number.
    char prt[] = "com3";

    //Connecting reader
    int errorCode = connectReader(prt);

    if(errorCode == TSS_SUCCESS)
    {
        //Connection successful

        //C1G2GetBlockPermalockStatus Command
        errorCode = urmsp\_C1G2GetBlockPermalockStatus(0x00, //Reserved value
                                                    0x00, //Reserved value
                                                    0x00, //Reserved value
                                                    0x00000000, //Access Password
                                                    TSS_TAGMEMORYBANK_USER, //Memory bank
                                                    0x00, //Block pointer
                                                    0x01); //Block Mask Word Count

        if(errorCode == TSS_SUCCESS)
        {
            //Command successful

            //This function wait for C1G2GetBlockPermalockStatus result for 1500ms
            errorCode = waitForResult(&m_getBlockPermalockStatusResultReceived,1500);

            if(errorCode == TSS_SUCCESS)
            {
                std::cout << "\nC1G2GetBlockPermalockStatus Result received";

                if(m_getBlockPermalockStatusResult.result == TSS_SUCCESS)
                {
                    std::cout << "\nC1G2GetBlockPermalockStatus result code: "
                                << (int)m_getBlockPermalockStatusResult.result;

                    std::cout << "\nC1G2GetBlockPermalockStatus result string: "
                                << getResult(URMSP_C1G2GETBLOCKPERMALOCKSTATUSOPSPECRESULT,
                                                m_getBlockPermalockStatusResult.result);

                    std::cout << "\nGetBlockPermalockStatus statusWordCount: "
                                << m_getBlockPermalockStatusResult.statusWordCount;

                    std::cout << "\nGetBlockPermalockStatus permalockStatus "
                                << std::string(m_getBlockPermalockStatusResult.permalockStatus);
                }
                else
                {
                    std::cout << getResult(URMSP_C1G2GETBLOCKPERMALOCKSTATUSOPSPECRESULT,
                                                m_getBlockPermalockStatusResult.result);
                }
            }
            else
            {
                std::cout << "\nwaitForResult timeout";
                std::cout << getError(errorCode);
            }
        }
    }
}

```

```
//Disconnecting reader
errorCode = disconnectReader\(\);

if(errorCode == TSS_SUCCESS)
{
    //Disconnection successful
}
else
{
    //Disconnection failed
    std::cout << getError\(errorCode\);
}
}
else
{
    //Command failed
    std::cout << getError\(errorCode\);
}
}
else
{
    //Connection failed
    std::cout << getError\(errorCode\);
}
}

//This callback function receives log messages(serial port
//communication, events, error messages, etc.)
static void callBackLog(void *p,unsigned int commandType, void *structPointer)
{
    log\_t logStruct = *(log\_t*)structPointer;

    std::string logString(logStruct.log,logStruct.logLength);
    std::cout << "\nLog: " << logString;
}

//This callback function is called each time the reader receives
//C1G2GetBlockPermalockStatusOpSpecResult the result of the C1G2GetBlockPermalockStatus command.
static void callBackGetBlockPermalockStatus(void *p, unsigned int commandType, void *structPointer)
{
    std::lock_guard<std::mutex> lock(mtx);

    //C1G2GetBlockPermalockStatus command type value.
    if(commandType == URMSP_C1G2GETBLOCKPERMALOCKSTATUS)
    {
        m_getBlockPermalockStatusResult = *(c1g2getblockpermalockstatusopspecresult\_t*)structPointer;

        m_getBlockPermalockStatusResultReceived = true;
    }
}

int waitForResult(bool *flag, int timeout)
{
    int x;

    for(x = 0; x < timeout; x += 100)    //Waiting x millisecond
    {
        //Lock the mutex to avoid access to to flag from different thread
        mtx.lock();

        //Check if structure was written
        if(*flag)
        {
            //Reset the flag
            *flag = false;

            mtx.unlock();
            return 0;
        }
        mtx.unlock();
    }
}
```

```
Sleep(100);
}

if(x == timeout)
{
    //Timeout
    return -1;
}
}

//This function translate result code to result strings
std::string getResult(unsigned int resultType, unsigned char resultCode)
{
    char resultString[1024];

    int resultStringLength;

    getResultString(resultString,&resultStringLength,resultType,resultCode);

    return std::string(resultString,resultStringLength);
}

//This function translate error code to error strings
std::string getError(int errorCode)
{
    char errorSourceString[1024];
    char errorCodeString[1024];

    int errorSourceStringLength;
    int errorCodeStringLength;

    getErrorString(errorSourceString,
        &errorSourceStringLength,
        errorCodeString,
        &errorCodeStringLength,
        errorCode);

    return std::string(errorCodeString,errorCodeStringLength);
}
```

5.47 int urmsp_C1G2V2AuthenticateAndReply (unsigned char *flags*, unsigned char *reserved*, unsigned short *opSpecID*, unsigned int *accessPassword*, unsigned char *csi*, unsigned short *messageLength*, const char * *message*)

AuthenticateAndReply command is used to pass data to the cryptographic processor on the Tag. The tag backscatters its response immediately after this command.

5.47.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.47.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>opSpecID</i>	Possible Value: 0x00
in	<i>accessPassword</i>	Access Password is the password used by the Reader to transition the tag to the secure state so that it can write to protected tag memory regions. Password length is 4 Byte(2 Word).
in	<i>csi</i>	selects the cryptographic suite that Tag use for the authentication as well as for all subsequent communications.
in	<i>messageLength</i>	is the message length in bits.
in	<i>message</i>	includes parameters for the authentication.

Example Usage:

80

```
#ifndef C1G2V2AUTHENTICATEANDREPLYEXP
#define C1G2V2AUTHENTICATEANDREPLYEXP

#include <string>

static void callBackLog(void *p,unsigned int commandType, void *structPointer);
static void callBackAuthenticateAndReply(void *p, unsigned int commandType, void *structPointer);

int waitForResult(bool *flag, int timeout);

std::string getResult(unsigned int resultType, unsigned char resultCode);
std::string getError(int errorCode);

#endif // C1G2V2AUTHENTICATEANDREPLYEXP
```

```
#include "c1g2v2authenticateandreplyexp.h"
#include "apic.h"

#include <iostream>
#include <mutex>

std::mutex mtx;

c1g2v2authenticateandreplyopspecresult t m_authenticateAndReplyResult;
bool m_authenticateAndReplyResultReceived = false;

int main()
{
    //pointer to class object
    void *cb = NULL;
```

```
//Initialize the library
initDll();

//Connect callback for log messages
connectCallbackLog(&callBackLog,cb);

//Connect callback for C1G2V2AuthenticateAndReply
connectCallbackC1G2V2AuthenticateAndReply(&callBackAuthenticateAndReply,cb);

//Set COM port number.
char prt[] = "com3";

//Connecting reader
int errorCode = connectReader(prt);

if(errorCode == TSS_SUCCESS)
{
    //Connection successful

    //Authenticate message
    //The authenticate message is Tag specific(this example is tested on UCODE DNA Tag)
    const char message[12] = {0x00, 0x00, 0xD6, 0xEB, 0x2C, 0xA9, 0x03, 0x21, 0xBB, 0xEF, 0x5F, 0x5F};

    //C1G2V2AuthenticateAndReply Command
    errorCode = urmsp_C1G2V2AuthenticateAndReply(0x00, //Reserved value
        0x00, //Reserved value
        0x00, //Reserved value
        0x00000000, //Access Password
        0x00, //Cryptographic suite
        0x60, //Message length
        message); //Message

    if(errorCode == TSS_SUCCESS)
    {
        //Command successful

        //This function wait for C1G2V2AuthenticateAndReply result for 1500ms
        errorCode = waitForResult(&m_authenticateAndReplyResultReceived,1500);

        if(errorCode == TSS_SUCCESS)
        {
            std::cout << "\nC1G2V2AuthenticateAndReply Result received";

            if(m_authenticateAndReplyResult.result == TSS_SUCCESS)
            {
                std::cout << "\nC1G2V2AuthenticateAndReply result code: "
                    << (int)m_authenticateAndReplyResult.result;

                std::cout << "\nC1G2V2AuthenticateAndReply result string: "
                    << getResult(URMSP_C1G2V2AUTHENTICATEANDREPLYRESULT,
                        m_authenticateAndReplyResult.result);

                std::cout << "\nC1G2V2AuthenticateAndReply messageLength: "
                    << m_authenticateAndReplyResult.messageLength;

                std::cout << "\nC1G2V2AuthenticateAndReply message: "
                    << std::string(m_authenticateAndReplyResult.message);
            }
            else
            {
                std::cout << getResult(URMSP_C1G2V2AUTHENTICATEANDREPLYRESULT,
                    m_authenticateAndReplyResult.result);
            }
        }
        else
        {
            std::cout << "\nwaitForResult timeout";
            std::cout << getError(errorCode);
        }
    }
}
```

```

//Disconnecting reader
errorCode = disconnectReader\(\);

if(errorCode == TSS_SUCCESS)
{
    //Disconnection successful
}
else
{
    //Disconnection failed
    std::cout << getError(errorCode);
}
}
else
{
    //Command failed
    std::cout << getError(errorCode);
}
}
else
{
    //Connection failed
    std::cout << getError(errorCode);
}
}

//This callback function receives log messages(serial port
//communication, events, error messages, etc.)
static void callBackLog(void *p,unsigned int commandType, void *structPointer)
{
    log\_t logStruct = *(log\_t*)structPointer;

    std::string logString(logStruct.log,logStruct.logLength);
    std::cout << "\nLog: " << logString;
}

//This callback function is called each time the reader receives
//C1G2V2AuthenticateAndReplyOpSpecResult the result of the C1G2V2AuthenticateAndReply command.
static void callBackAuthenticateAndReply(void *p, unsigned int commandType, void *structPointer)
{
    std::lock_guard<std::mutex> lock(mtx);

    //C1G2V2AuthenticateAndReply command type value.
    if(commandType == URMSP_C1G2V2AUTHENTICATEANDREPLY)
    {
        m_authenticateAndReplyResult = *(c1g2v2authenticateandreplyopspecresult\_t*)structPointer;

        m_authenticateAndReplyResultReceived = true;
    }
}

int waitForResult(bool *flag, int timeout)
{
    int x;

    for(x = 0; x < timeout; x += 100)    //Waiting x millisecond
    {
        //Lock the mutex to avoid access to to flag from different thread
        mtx.lock();

        //Check if structure was written
        if(*flag)
        {
            //Reset the flag
            *flag = false;

            mtx.unlock();
            return 0;
        }
        mtx.unlock();
    }
}

```

```
Sleep(100);
}

if(x == timeout)
{
    //Timeout
    return -1;
}
}

//This function translate result code to result strings
std::string getResult(unsigned int resultType, unsigned char resultCode)
{
    char resultString[1024];

    unsigned int resultStringLength;

    getResultString(resultString,&resultStringLength,resultType,resultCode);

    return std::string(resultString,resultStringLength);
}

//This function translate error code to error strings
std::string getError(int errorCode)
{
    char errorSourceString[1024];
    char errorCodeString[1024];

    unsigned int errorSourceStringLength;
    unsigned int errorCodeStringLength;

    getErrorString(errorSourceString,
        &errorSourceStringLength,
        errorCodeString,
        &errorCodeStringLength,
        errorCode);

    return std::string(errorCodeString,errorCodeStringLength);
}
```

5.48 int urmsp_C1G2V2AuthenticateAndStore (unsigned char *flags*, unsigned char *reserved*, unsigned short *opSpecID*, unsigned int *accessPassword*, unsigned char *csi*, unsigned short *storedMessageLength*, const char * *message*)

FUNCTION IS RESERVED FOR FUTURE USE.

AuthenticateAndStore command is used to pass on data to the cryptographic processor on the Tag. The tag response to this command is stored in responseBuffer and can be read by urmsp_C1G2V2ReadBuffer command.

5.48.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.48.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>opSpecID</i>	Possible Value: 0x00
in	<i>accessPassword</i>	Access Password is the password used by the Reader to transition the tag to the secure state so that it can write to protected tag memory regions. Password length is 4 Byte(2 Word).
in	<i>csi</i>	selects the cryptographic suite that Tag use for the authentication as well as for all subsequent communications.
in	<i>storedMessageLength</i>	is the message length in bits.
in	<i>message</i>	includes parameters for the authentication.

5.49 int urmsp_C1G2V2Untraceable (unsigned char *flags*, unsigned char *reserved*, unsigned short *opSpecID*, unsigned int *accessPassword*, bool *u*, bool *e*, unsigned char *epcLength*, unsigned char *tid*, bool *m*, unsigned char *r*)

Untraceable allows to instruct a Tag to Hide/Unhide parts of the EPC,TID and User Memory and also reduce Tag's operation range.

A Tag only executes Untraceable when it is in the secured state.

5.49.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.49.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>opSpecID</i>	Possible Value: 0x00
in	<i>accessPassword</i>	Access Password is the password used by the Reader to transition the tag to the secure state so that it can execute the Untraceable command. Password length is 4 Byte(2 Word).
in	<i>u</i>	Assert and Deassert U in XPC_W1. Possible value: 0x00 - 0x01. 0x00 = Deassert U 0x01 = Assert U
in	<i>e</i>	Specifies whether a Tag untraceably hides part of EPC memory. Possible Values: 0x00 - 0x01. 0x00 = Tag exposes EPC memory. 0x01 = Tag untraceably hides EPC memory above that set by <i>epcLength</i> .
in	<i>epcLength</i>	specifies a new EPC length.
in	<i>tid</i>	specifies the TID memory that a Tag untraceably hides. Possible Values: 0x00 - 0x03. 0x00 = Tag exposes TID memory. 0x01 = If the Tag's allocation class identifier is E0h then the Tag untraceably hides TID memory above 10h, inclusive. If the Tag's allocation class identifier is E2h then the Tag untraceably hides TID memory above 20h, inclusive. 0x10 = Tag untraceably hides all of TID memory. 0x11 = Reserved.
in	<i>m</i>	specifies whether a Tag untraceably hides User memory. Possible value: 0x00 - 0x01 0x00 = Tag exposes User memory. 0x01 = Tag untraceably hides User memory.
in	<i>r</i>	specifies a Tag's operating range. Possible value: 0x00 - 0x03.

		0x00 = Tag persistently enables normal operating range. 0x01 = Tag persistently enables reduced operating range. 0x02 = Tag temporarily toggles its operating range (if normal then to reduced; if reduced then to normal) but reverts to its prior persistent operating range when the Tag loses power. 0x03 = Reserved
--	--	--

Example Usage:

```
#ifndef C1G2V2UNTRACEABLEEXP
#define C1G2V2UNTRACEABLEEXP

#include <string>

static void callBackLog(void *p,unsigned int commandType, void *structPointer);
static void callBackUntraceable(void *p, unsigned int commandType, void *structPointer);

int waitForResult(bool *flag, int timeout);

std::string getResult(unsigned int resultType, unsigned char resultCode);
std::string getError(int errorCode);

#endif // C1G2V2UNTRACEABLEEXP
```

```
#include "c1g2v2untraceableexp.h"
#include "apic.h"

#include <iostream>
#include <mutex>

std::mutex mtx;

c1g2v2untraceableopspecresult_t m_untraceableResult;
bool m_untraceableResultReceived = false;

int main()
{
    //pointer to class object
    void *cb = NULL;

    //Initialize the library
    initDll\(\);

    //Connect callback for log messages
    connectCallbackLog(&callBackLog,cb);

    //Connect callback for C1G2V2Untraceable
    connectCallbackC1G2V2Untraceable(&callBackUntraceable,cb);

    //Set COM port number.
    char prt[] = "com3";

    //Connecting reader
    int errorCode = connectReader(prt);

    if(errorCode == TSS_SUCCESS)
    {
        //Connection successful

        //C1G2V2Untraceable Command
```

```
//Only works with C1G2V2 Tags
errorCode = urmsp\_C1G2V2Untraceable(0x00, //Reserved value
    0x00, //Reserved value
    0x00, //Reserved value
    0x12345678, //Access password
    0x00, //Reserved value
    0x01, //E Flag - Hide Epc
    0x01, //Epc lenght - Set to 1 Word
    0x00, //Hide TID - No
    0x00, //Hide User memory - No
    0x00); //Tag operation range - Normal

if(errorCode == TSS_SUCCESS)
{
    //Command successful

    //This function wait for C1G2V2Untraceable result for 1500ms
    errorCode = waitForResult(&m_untraceableResultReceived,1500);

    if(errorCode == TSS_SUCCESS)
    {
        std::cout << "\nC1G2V2Untraceable Result received";

        if(m_untraceableResult.result == TSS_SUCCESS)
        {
            std::cout << "\nC1G2V2Untraceable result code: "
                << (int)m_untraceableResult.result;

            std::cout << "\nC1G2V2Untraceable result string: "
                << getResult(URMSP_C1G2V2UNTRACEABLERESULT,
                    m_untraceableResult.result);
        }
        else
        {
            std::cout << getResult(URMSP_C1G2V2UNTRACEABLERESULT,
                m_untraceableResult.result);
        }
    }
    else
    {
        std::cout << "\nwaitForResult timeout";
        std::cout << getError(errorCode);
    }
}

//Disconnecting reader
errorCode = disconnectReader();

if(errorCode == TSS_SUCCESS)
{
    //Disconnection successful
}
else
{
    //Disconnection failed
    std::cout << getError(errorCode);
}
else
{
    //Command failed
    std::cout << getError(errorCode);
}
else
{
    //Connection failed
    std::cout << getError(errorCode);
}
}

//This callback function receives log messages(serial port
```

```

//communication, events, error messages, etc.)
static void callBackLog(void *p,unsigned int commandType, void *structPointer)
{
    log t logStruct = *(log t*)structPointer;

    std::string logString(logStruct.log,logStruct.logLength);
    std::cout << "\nLog: " << logString;
}

//This callback function is called each time the reader receives
//C1G2V2UntraceableOpSpecResult the result of the C1G2V2Untraceable command.
static void callBackUntraceable(void *p, unsigned int commandType, void *structPointer)
{
    std::lock_guard<std::mutex> lock(mtx);

    //C1G2V2Untraceable command type value.
    if(commandType == URMSP_C1G2V2UNTRACEABLE)
    {
        m_untraceableResult = *(c1g2v2untraceableopspecresult t*)structPointer;

        m_untraceableResultReceived = true;
    }
}

int waitForResult(bool *flag, int timeout)
{
    int x;

    for(x = 0; x < timeout; x += 100)    //Waiting x millisecond
    {
        //Lock the mutex to avoid access to to flag from different thread
        mtx.lock();

        //Check if structure was written
        if(*flag)
        {
            //Reset the flag
            *flag = false;

            mtx.unlock();
            return 0;
        }
        mtx.unlock();

        Sleep(100);
    }

    if(x == timeout)
    {
        //Timeout
        return -1;
    }
}

//This function translate result code to result strings
std::string getResult(unsigned int resultType, unsigned char resultCode)
{
    char resultString[1024];

    unsigned int resultStringLength;

    getResultString(resultString,&resultStringLength,resultType,resultCode);

    return std::string(resultString,resultStringLength);
}

//This function translate error code to error strings
std::string getError(int errorCode)
{
    char errorSourceString[1024];
    char errorCodeString[1024];
}

```

```
unsigned int errorSourceStringLength;  
unsigned int errorCodeStringLength;  
  
getErrorString(errorSourceString,  
                &errorSourceStringLength,  
                errorCodeString,  
                &errorCodeStringLength,  
                errorCode);  
  
return std::string(errorCodeString,errorCodeStringLength);  
}
```

5.50 int urmsp_C1G2V2ReadBuffer (unsigned char *flags*, unsigned char *reserved*, unsigned short *opSpecID*, unsigned int *accessPassword*, unsigned short *wordPointer*, unsigned short *bitCount*)

FUNCTION IS RESERVED FOR FUTURE USE.

ReadBuffer allows to read data stored in a Tag's ResponseBuffer.

5.50.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.50.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>opSpecID</i>	Possible Value: 0x00
in	<i>accessPassword</i>	Access Password is the password used by the Reader to transition the tag to the secure state so that it can write to protected tag memory regions. Password length is 4 Byte(2 Word).
in	<i>wordPointer</i>	specifies the starting word address for the read in units of 16 blocks.
in	<i>bitCount</i>	specifies the number of bits to read.

5.51 int urmsp_Higgs3BlockReadLock (unsigned char *flags*, unsigned char *reserved*, unsigned short *opSpecID*, unsigned int *accessPassword*, unsigned char *readLockBits*)

Higgs3BlockReadLock command is used to prevent reading from selected 4 Word blocks. This command work only with ALIEN Higgs 3 Tags.

5.51.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.51.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>opSpecID</i>	Possible Value: 0x00
in	<i>accessPassword</i>	Access Password is the password used by the Reader to transition the tag to the secure state so that it can write to protected tag memory regions. Password length is 4 Byte(2 Word).
in	<i>readLockBits</i>	each bit specifies the 4 Word blocks to be locked.

Example Usage:

```
#ifndef HIGGS3BLOCKREADLOCKEXP
#define HIGGS3BLOCKREADLOCKEXP

#include <string>

static void callBackLog(void *p,unsigned int commandType, void *structPointer);
static void callBackHiggs3BlockReadLock(void *p, unsigned int commandType, void *structPointer);

int waitForResult(bool *flag, int timeout);

std::string getResult(unsigned int resultType, unsigned char resultCode);
std::string getError(int errorCode);

#endif // HIGGS3BLOCKREADLOCKEXP
```

```
#include "higgs3blockreadlockexp.h"
#include "apic.h"

#include <iostream>
#include <mutex>

std::mutex mtx;

higgs3blockreadlockresult t m_higgs3BlockReadLockResult;
bool m_higgs3BlockReadLockResultReceived = false;

int main()
{
    //pointer to class object
    void *cb = NULL;

    //Initialize the library
    initDll\(\);

    //Connect callback for log messages
    connectCallbackLog(&callBackLog,cb);
```

```
//Connect callback for Higgs3BlockReadLock
connectCallbackHiggs3BlockReadLock(&callBackHiggs3BlockReadLock,cb);

//Set COM port number.
char prt[] = "com7";

//Connecting reader
int errorCode = connectReader(prt);

if(errorCode == TSS_SUCCESS)
{
    //Connection successful

    //Higgs3BlockReadLock Command
    errorCode = urmsp_Higgs3BlockReadLock(0x00, //Reserved value
        0x00, //Reserved value
        0x00, //Reserved value
        0x12345678, //Access password
        0x80); //Read lock bits - Lock the 1st block

    if(errorCode == TSS_SUCCESS)
    {
        //Command successful

        //This function wait for Higgs3BlockReadLock result for 1500ms
        errorCode = waitForResult(&m_higgs3BlockReadLockResultReceived,1500);

        if(errorCode == TSS_SUCCESS)
        {
            std::cout << "\nHiggs3BlockReadLock Result received";

            if(m_higgs3BlockReadLockResult.result == TSS_SUCCESS)
            {
                std::cout << "\nHiggs3BlockReadLock result code: "
                    << (int)m_higgs3BlockReadLockResult.result;

                std::cout << "\nHiggs3BlockReadLock result string: "
                    << getResult(URMSP_C1G2WRITEOPSPECRESULT,
                        m_higgs3BlockReadLockResult.result);
            }
            else
            {
                std::cout << getResult(URMSP_HIGGS3BLOCKREADLOCKRESULT,
                    m_higgs3BlockReadLockResult.result);
            }
        }
        else
        {
            std::cout << "\nwaitForResult timeout";
            std::cout << getError(errorCode);
        }
    }

    //Disconnecting reader
    errorCode = disconnectReader();

    if(errorCode == TSS_SUCCESS)
    {
        //Disconnection successful
    }
    else
    {
        //Disconnection failed
        std::cout << getError(errorCode);
    }
}
else
{
    //Command failed
    std::cout << getError(errorCode);
}
}
```

```

else
{
    //Connection failed
    std::cout << getError(errorCode);
}
}

//This callback function receives log messages(serial port
//communication, events, error messages, etc.)
static void callBackLog(void *p,unsigned int commandType, void *structPointer)
{
    log_t logStruct = *(log_t*)structPointer;

    std::string logString(logStruct.log,logStruct.logLength);
    std::cout << "\nLog: " << logString;
}

//This callback function is called each time the reader receives
//BackHiggs3BlockReadLockResult the result of the BackHiggs3BlockReadLock command.
static void callBackHiggs3BlockReadLock(void *p, unsigned int commandType, void *structPointer)
{
    std::lock_guard<std::mutex> lock(mtx);

    //C1G2Write command type value.
    if(commandType == URMSP_HIGGS3BLOCKREADLOCK)
    {
        m_higgs3BlockReadLockResult = *(higgs3blockreadlockresult_t*)structPointer;

        m_higgs3BlockReadLockResultReceived = true;
    }
}

int waitForResult(bool *flag, int timeout)
{
    int x;

    for(x = 0; x < timeout; x += 100)    //Waiting x millisecond
    {
        //Lock the mutex to avoid access to to flag from different thread
        mtx.lock();

        //Check if structure was written
        if(*flag)
        {
            //Reset the flag
            *flag = false;

            mtx.unlock();
            return 0;
        }
        mtx.unlock();

        Sleep(100);
    }

    if(x == timeout)
    {
        //Timeout
        return -1;
    }
}

//This function translate result code to result strings
std::string getResult(unsigned int resultType, unsigned char resultCode)
{
    char resultString[1024];

    unsigned int resultStringLength;

    getResultString(resultString,&resultStringLength,resultType,resultCode);

```

```
    return std::string(resultString,resultStringLength);
}

//This function translate error code to error strings
std::string getError(int errorCode)
{
    char errorSourceString[1024];
    char errorCodeString[1024];

    unsigned int errorSourceStringLength;
    unsigned int errorCodeStringLength;

    getErrorString(errorSourceString,
        &errorSourceStringLength,
        errorCodeString,
        &errorCodeStringLength,
        errorCode);

    return std::string(errorCodeString,errorCodeStringLength);
}
```

5.52 int urmsp_C1G2CustomCommand (unsigned char *flags*, unsigned char *reserved*, unsigned short *opSpecID*, unsigned int *accessPassword*, unsigned char *fp*, unsigned char *rfu*, unsigned short *bitCount*, const char * *command*, unsigned char *rfu2*, bool *r1*, bool *r2*, bool *h*, bool *c*, unsigned char *rt*, unsigned char *rfu3*, unsigned short *replyBiteCount*)

C1G2CustomCommand enables the user to implement air protocol custom commands specific for particular Tag type. The content of the "command" will be sent to the Tag and the reply will be received via C1G2CustomCommand Result.

5.52.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.52.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>opSpecID</i>	Possible Value: 0x00
in	<i>accessPassword</i>	Access Password is the password used by the Reader to transition the tag to the secure state so that it can write to protected tag memory regions. Password length is 4 Byte(2 Word).
in	<i>fp</i>	Possible value: 0x00 = Send the command withouth a preamble or a frame-sync 0x01 = Send a preamble at the beining of the command 0x02 = Send a frame-sync at the beining of the command 0x03 = RFU
in	<i>rfu</i>	RFU Possible Value: 0x00
in	<i>bitCount</i>	Bit count of command
in	<i>command</i>	Command to be sent
in	<i>rfu2</i>	RFU Possible Value: 0x00
in	<i>r1</i>	Possible value: 0x00 = No Operation 0x01 = XOR bits [command end - 31, command end - 16] with RN16
in	<i>r2</i>	Possible value: 0x00 = No Operation 0x01 = XOR bits [command end - 15, command end] with RN16
in	<i>h</i>	Handle Possible value: 0x00 = No Operation 0x01 = Add Handle at the end of command
in	<i>c</i>	CRC

		Possible Value: 0x00 = No operation 0x01 = Add CRC-16 at the end of command
in	<i>rt</i>	Tag reply type. Possible value: 0x00 = RFU(None) 0x01 = Immediate 0x02 = Delayed 0x03 = RFU(In-Process)
in	<i>rfu3</i>	RFU Possible Value: 0x00
in	<i>replyBiteCount</i>	Bit count of expected tag reply

5.53 int urmsp_SetPeripherals (unsigned char *flags*, unsigned char *reserved*, unsigned char *powerLED*, unsigned char *statusLED*, unsigned char *buzzer*, unsigned char *pushButton*, unsigned char *rfu*)

SetPeripherals command is used to configure reader peripherals like LEDs, buzzers and push buttons.

5.53.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.53.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>powerLED</i>	Possible Value: 0x0 = Power LED off 0x1 = Power LED active 0x2 - 0x0E = RFU 0xF = Do not change the settings of Power LED
in	<i>statusLED</i>	Possible Value: 0x0 = Status LED off 0x1 = Status LED active 0x2 - 0x0E = RFU 0xF = Do not change the settings of Status LED
in	<i>buzzer</i>	Possible Value: 0x0 = Buzzer off 0x1 = Buzzer active 0x2 - 0x0E = RFU 0xF = Do not change the settings of Buzzer
in	<i>pushButton</i>	Possible Value: 0x0 = PushButton off 0x1 = PushButton active 0x2 - 0x0E = RFU 0xF = Do not change the settings of PushButton
in	<i>rfu</i>	Possible Value: 0x00

Example Usage:

```
#ifndef SETPERIPHERALSEXP
#define SETPERIPHERALSEXP

#include <string>

static void callBackLog(void *p,unsigned int commandType, void *structPointer);

std::string getError(int errorCode);

#endif // SETPERIPHERALSEXP
```

```
#include "setperipheralsexp.h"
#include "apic.h"

#include <iostream>

int main()
{
    //pointer to class object
    void *cb = NULL;

    //Initialize the library
    initDll\(\);

    //Connect callback for log messages
    connectCallbackLog(&callBackLog,cb);

    //Set COM port number.
    char prt[] = "com4";

    //Connecting reader
    int errorCode = connectReader(prt);

    if(errorCode == TSS_SUCCESS)
    {
        //Connection successful

        if(errorCode == TSS_SUCCESS)
        {
            //Command successful

            //Set Peripherals
            errorCode = urmsp\_SetPeripherals(0x00, //Reserved value
                0x00, //Reserved value
                TSS_SETPERIPHERALS_DISABLED, //PowerLED Disabled
                TSS_SETPERIPHERALS_ENABLED, //StatusLED Enabled
                TSS_SETPERIPHERALS_DONOTCHANGE, //Buzzer - No change
                TSS_SETPERIPHERALS_DONOTCHANGE, //PushButton - No change
                0x00); //Reserved value

            if(errorCode == TSS_SUCCESS)
            {
                //Command successful
            }
            else
            {
                //Command failed
                std::cout << getError(errorCode);
            }
        }
        else
        {
            //Command failed
            std::cout << getError(errorCode);
        }

        //Disconnecting reader
        errorCode = disconnectReader();

        if(errorCode == TSS_SUCCESS)
        {
            //Disconnection successful
        }
        else
        {
            //Disconnection failed
            std::cout << getError(errorCode);
        }
    }
}
```

```
else
{
    //Connection failed
    std::cout << getError(errorCode);
}
}

//This callback function receives log messages(serial port
//communication, events, error messages, etc.)
static void callBackLog(void *p,unsigned int commandType, void *structPointer)
{
    log_t logStruct = *(log_t*)structPointer;

    std::string logString(logStruct.log,logStruct.logLength);
    std::cout << "\nLog: " << logString;
}

//This function translate error code to error strings
std::string getError(int errorCode)
{
    char errorSourceString[1024];
    char errorCodeString[1024];

    unsigned int errorSourceStringLength;
    unsigned int errorCodeStringLength;

    getErrorString(errorSourceString,
        &errorSourceStringLength,
        errorCodeString,
        &errorCodeStringLength,
        errorCode);

    return std::string(errorCodeString,errorCodeStringLength);
}
```

5.54 int urmsp_SetPowerMode (unsigned char *flags*, unsigned char *reserved*, unsigned char *powerMode*)

SetPowerMode command is used to change the reader power mode. The reader will wake up from power down mode by receiving any urmsp command.

5.54.1 Returns:

int error code. To translate error code to error message use [getErrorString\(\)](#) function.

5.54.2 Parameters:

in	<i>flags</i>	Possible Value: 0x00
in	<i>reserved</i>	Possible Value: 0x00
in	<i>powerMode</i>	Possible Value: 0x00 = Power down mode

Example Usage:

```
#ifndef SETPOWERMODEEXP
#define SETPOWERMODEEXP

#include <string>

static void callBackLog(void *p,unsigned int commandType, void *structPointer);

std::string getError(int errorCode);

#endif // SETPOWERMODEEXP
```

```
#include "setpowermodeexp.h"
#include "apic.h"

#include <iostream>

int main()
{
    //pointer to class object
    void *cb = NULL;

    //Initialize the library
    initDll\(\);

    //Connect callback for log messages
    connectCallbackLog(&callBackLog,cb);

    //Set COM port number.
    char prt[] = "com4";

    //Connecting reader
    int errorCode = connectReader(prt);

    if(errorCode == TSS_SUCCESS)
    {
        //Connection successful

        if(errorCode == TSS_SUCCESS)
        {
            //Command successful

            //Set power mode
            errorCode = urmSP\_SetPowerMode(0x00, //Reserved value
                                           0x00, //Reserved value
```

```

0x00); //Power Down

if(errorCode == TSS_SUCCESS)
{
    //Command successful

    std::cout << "\nPower down";
}
else
{
    //Command failed
    std::cout << getError(errorCode);
}
}
else
{
    //Command failed
    std::cout << getError(errorCode);
}

//Disconnecting reader
errorCode = disconnectReader\(\);

if(errorCode == TSS_SUCCESS)
{
    //Disconnection successful
}
else
{
    //Disconnection failed
    std::cout << getError(errorCode);
}
}
else
{
    //Connection failed
    std::cout << getError(errorCode);
}
}

//This callback function receives log messages(serial port
//communication, events, error messages, etc.)
static void callBackLog(void *p,unsigned int commandType, void *structPointer)
{
    log\_t logStruct = *(log\_t*)structPointer;

    std::string logString(logStruct.log,logStruct.logLength);
    std::cout << "\nLog: " << logString;
}

//This function translate error code to error strings
std::string getError(int errorCode)
{
    char errorSourceString[1024];
    char errorCodeString[1024];

    unsigned int errorSourceStringLength;
    unsigned int errorCodeStringLength;

    getErrorString(errorSourceString,
        &errorSourceStringLength,
        errorCodeString,
        &errorCodeStringLength,
        errorCode);

    return std::string(errorCodeString,errorCodeStringLength);
}

```

5.55 void **getErrorString** (char * *errorSourceString*, unsigned int * *errorSourceStringLength*, char * *errorCodeString*, unsigned int * *errorCodeStringLength*, int *errorCode*)

This function translate error codes to strings.

5.55.1 Parameters:

out	<i>errorSourceString</i>	
out	<i>errorSourceStringLength</i>	
out	<i>errorCodeString</i>	
out	<i>errorCodeStringLength</i>	
in	<i>errorCode</i>	

5.56 void **getResultString** (char * *resultString*, unsigned int * *resultStringLength*, unsigned int *resultType*, unsigned char *resultCode*)

This function translate result codes to strings.

5.56.1 Parameters:

out	<i>resultString</i>	
out	<i>resultStringLength</i>	
in	<i>resultType</i>	Possible Values:

[Supported LLRP Command Results](#)

5.56.2 Parameters:

in	<i>resultCode</i>	
----	-------------------	--

6 Data Structure Documentation

6.1 c1g2blockeraseopspecresult_t Struct Reference

6.1.1 Data Fields

- unsigned char [result](#)
- unsigned short [reserved](#)
- unsigned short [wordCount](#)

6.1.2 Detailed Description

C1G2BlockEraseOpSpecResult Structure Contains the results of the C1G2BlockErase reader operation.

Result	value
C1G2BLOCKERASE_SUCCESS	0
C1G2BLOCKERASE_TAG_MEMORY_OVERRUN_ERROR	1
C1G2BLOCKERASE_TAG_MEMORY_LOCKED_ERROR	2
C1G2BLOCKERASE_INSUFFICIENT_POWER_TO_PERFORM_MEMORY_WRITE_OPERATION	3
C1G2BLOCKERASE_NON_SPECIFIC_TAG_ERROR	4
C1G2BLOCKERASE_NO_RESPONSE_FROM_TAG	5
C1G2BLOCKERASE_NON_SPECIFIC_READER_ERROR	6
C1G2BLOCKERASE_INCORRECT_PASSWORD_ERROR	7

6.1.3 Field Documentation

6.1.3.1 *unsigned char result*

Result values

6.1.3.2 *unsigned short reserved*

Reserved field

6.1.3.3 *unsigned short wordCount*

Word Count is the number of 16-bit words received.

6.1.3.4

6.2 c1g2blockpermalockopspecresult_t Struct Reference

6.2.1 Data Fields

- unsigned char [result](#)
- unsigned short [reserved](#)

6.2.2 Detailed Description

C1G2BlockPermalockOpSpecResult Structure Contains the results of the C1G2BlockPermalock reader operation.

Result	value
C1G2BLOCKPERMALOCK_SUCCESS	0
C1G2BLOCKPERMALOCK_INSUFFICIENT_POWER_TO_PERFORM_LOCK_OPERATION	1
C1G2BLOCKPERMALOCK_NON_SPECIFIC_TAG_ERROR	2
C1G2BLOCKPERMALOCK_NO_RESPONSE_FROM_TAG	3
C1G2BLOCKPERMALOCK_NON_SPECIFIC_READER_ERROR	4
C1G2BLOCKPERMALOCK_INCORRECT_PASSWORD_ERROR	5
C1G2BLOCKPERMALOCK_TAG_MEMORY_OVERRUN_ERROR	6

6.2.3 Field Documentation

6.2.3.1 *unsigned char result*

Result values

6.2.3.2 *unsigned short reserved*

Reserved field

6.2.3.3

6.3 c1g2blockwriteopspecresult_t Struct Reference

6.3.1 Data Fields

- unsigned char [result](#)
- unsigned short [reserved](#)
- unsigned short [wordCount](#)

6.3.2 Detailed Description

C1G2BlockWriteOpSpecResult Structure Contains the results of the C1G2BlockWrite reader operation.

Result	value
C1G2BLOCKWRITE_SUCCESS	0
C1G2BLOCKWRITE_TAG_MEMORY_OVERRUN_ERROR	1
C1G2BLOCKWRITE_TAG_MEMORY_LOCKED_ERROR	2
C1G2BLOCKWRITE_INSUFFICIENT_POWER_TO_PERFORM_MEMORY_WRITE_OPERATION	3
C1G2BLOCKWRITE_NON_SPECIFIC_TAG_ERROR	4
C1G2BLOCKWRITE_NO_RESPONSE_FROM_TAG	5
C1G2BLOCKWRITE_NON_SPECIFIC_READER_ERROR	6
C1G2BLOCKWRITE_INCORRECT_PASSWORD_ERROR	7

6.3.3 Field Documentation

6.3.3.1 *unsigned char result*

Result values

6.3.3.2 *unsigned short reserved*

Reserved field

6.3.3.3 *unsigned short wordCount*

Word Count is the number of 16-bit words received.

6.4 generalDeviceCapabilities_t Struct Reference

6.4.1 Data Fields

- unsigned short [maxNumberOfAntennaSupported](#)
- unsigned short [reserved](#)
- unsigned int [deviceManufacturerName](#)
- char [modelName](#) [4]
- unsigned short [firmwareVersionByteCount](#)
- char [readerFirmwareVersion](#) [255]

6.4.2 Detailed Description

GeneralDeviceCapabilities Structure Contains the informations about the reader.

6.4.3 Field Documentation

6.4.3.1 *unsigned short maxNumberOfAntennaSupported*

Maximum number of antenna supported

6.4.3.2 *unsigned short reserved*

Reserved value

6.4.3.3 *unsigned int deviceManufacturerName*

Device manufacturer identification number

6.4.3.4 *char modelName[4]*

Reader model

6.4.3.5 *unsigned short firmwareVersionByteCount*

Length of firmware version string

6.4.3.6 *char readerFirmwareVersion[255]*

Firmware version string

6.5 c1g2customcommandresult_t Struct Reference

6.5.1 Data Fields

- unsigned char [result](#)
- unsigned short [reserved](#)
- unsigned short [bitCount](#)
- char [replyData](#) [64]

6.5.2 Detailed Description

C1G2CustomCommandResult Structure Contains the results of the CustomCommand Sreader operation.

Result	value
CUSTOMCOMMAND_SUCCESS	0
CUSTOMCOMMAND_TAG_ERROR	1
CUSTOMCOMMAND_NO_RESPONSE_FROM_TAG	2
CUSTOMCOMMAND_NON_SPECIFIC_READER_ERROR	3

6.5.3 Field Documentation

6.5.3.1 *unsigned char result*

Result values

6.5.3.2 *unsigned short reserved*

Reserved field

6.5.3.3 *unsigned short bitCount*

Amount of reply data in bits.

6.5.3.4 *char replyData[64]*

Reply Data.

6.6 c1g2epc_t Struct Reference

6.6.1 Data Fields

- unsigned short [epcLength](#)
 - char [epc](#) [255]
-

6.6.2 Detailed Description

c1g2EPC Structure Contains the results of the StartROSpec reader operation.

6.6.3 Field Documentation

6.6.3.1 *unsigned short epcLength*

EPC number length

6.6.3.2 *char epc[255]*

EPC number

6.7 c1g2getblockpermalockstatusopspecresult_t Struct Reference

6.7.1 Data Fields

- unsigned char [result](#)
- unsigned short [reserved](#)
- unsigned short [statusWordCount](#)
- char [permalockStatus](#) [255]

6.7.2 Detailed Description

C1G2GetBlockPermalockOpSpecResult Structure Contains the results of the C1G2GetBlockPermalockStatus reader operation.

Result	value
C1G2GETBLOCKPERMALOCKSTATUS_SUCCESS	0
C1G2GETBLOCKPERMALOCKSTATUS_NON_SPECIFIC_TAG_ERROR	1
C1G2GETBLOCKPERMALOCKSTATUS_NO_RESPONSE_FROM_TAG	2
C1G2GETBLOCKPERMALOCKSTATUS_NON_SPECIFIC_READER_ERROR	3
C1G2GETBLOCKPERMALOCKSTATUS_INCORRECT_PASSWORD_ERROR	4
C1G2GETBLOCKPERMALOCKSTATUS_TAG_MEMORY_OVERRUN_ERROR	5

6.7.3 Field Documentation

6.7.3.1 *unsigned char result*

Result values

6.7.3.2 *unsigned short reserved*

Reserved field

6.7.3.3 *unsigned short statusWordCount*

Word Count is the number of 16-bit words received.

6.7.3.4 *char permalockStatus[255]*

Specifies the Permalock status of each block requested.

6.8 c1g2killopspecresult_t Struct Reference

6.8.1 Data Fields

- unsigned char [result](#)
- unsigned short [reserved](#)

6.8.2 Detailed Description

C1G2KillOpSpecResult Structure Contains the results of the C1G2Kill reader operation.

Result	value
C1G2KILL_SUCCESS	0
C1G2KILL_ZERO_KILL_PASSWORD_ERROR	1
C1G2KILL_INSUFFICIENT_POWER_TO_PERFORM_KILL_OPERATION	2
C1G2KILL_NON_SPECIFIC_TAG_ERROR	3
C1G2KILL_NO_RESPONSE_FROM_TAG	4
C1G2KILL_NON_SPECIFIC_READER_ERROR	5
C1G2KILL_INCORRECT_PASSWORD_ERROR	6

6.8.3 Field Documentation

6.8.3.1 *unsigned char result*

Result values

6.8.3.2 *unsigned short reserved*

Reserved field

6.9 c1g2lockopspecresult_t Struct Reference

6.9.1 Data Fields

- unsigned char [result](#)
- unsigned short [reserved](#)

6.9.2 Detailed Description

C1G2LockOpSpecResult Structure Contains the results of the C1G2Lock reader operation.

Result	value
C1G2LOCK_SUCCESS	0
C1G2LOCK_INSUFFICIENT_POWER_TO_PERFORM_LOCK_OPERATION	1
C1G2LOCK_NON_SPECIFIC_TAG_ERROR	2
C1G2LOCK_NO_RESPONSE_FROM_TAG	3
C1G2LOCK_NON_SPECIFIC_READER_ERROR	4
C1G2LOCK_INCORRECT_PASSWORD_ERROR	5
C1G2LOCK_TAG_MEMORY_OVERRUN_ERROR	6
C1G2LOCK_TAG_MEMORY_LOCKED_ERROR	7

6.9.3 Field Documentation

6.9.3.1 *unsigned char result*

Result values

6.9.3.2 *unsigned short reserved*

Reserved field

6.10 c1g2readopspecresult_t Struct Reference

6.10.1 Data Fields

- unsigned char [result](#)
- unsigned short [reserved](#)
- unsigned short [wordCount](#)
- char [data](#) [255]

6.10.2 Detailed Description

C1G2ReadOpSpecResult Structure Contains the results of the C1G2Read reader operation.

Result	value
C1G2READ_SUCCESS	0
C1G2READ_NON_SPECIFIC_TAG_ERROR	1
C1G2READ_NO_RESPONSE_FROM_TAG	2
C1G2READ_NON_SPECIFIC_READER_ERROR	3
C1G2READ_MEMORY_OVERRUN_ERROR	4
C1G2READ_MEMORY_LOCKED_ERROR	5
C1G2READ_INCORRECT_PASSWORD_ERROR	6

6.10.3 Field Documentation

6.10.3.1 unsigned char result

Result values

6.10.3.2 unsigned short reserved

Reserved field

6.10.3.3 unsigned short wordCount

Word Count is the number of 16-bit words received.

6.10.3.4 char data[255]

Read data

6.11 c1g2v2authenticateandreplyopspecresult_t Struct Reference

6.11.1 Data Fields

- unsigned char [result](#)
- unsigned short [reserved](#)
- unsigned short [messageLength](#)
- char [message](#) [64]

6.11.2 Detailed Description

C1G2V2AuthenticateAndReplyOpSpecResult Structure Contains the results of the C1G2V2AuthenticateAndReply reader operation.

Result	value
C1G2V2AUTHENTICATEANDREPLY_SUCCESS	0
C1G2V2AUTHENTICATEANDREPLY_TAG_MEMORY_OVERRUN_ERROR	1
C1G2V2AUTHENTICATEANDREPLY_TAG_MEMORY_LOCKED_ERROR	2
C1G2V2AUTHENTICATEANDREPLY_INSUFFICIENT_POWER_TO_PERFORM_AUTHENTICATE_OPERATION	3
C1G2V2AUTHENTICATEANDREPLY_NON_SPECIFIC_TAG_ERROR	4
C1G2V2AUTHENTICATEANDREPLY_NOT_SUPPORTED_PARAMETER_ERROR	5
C1G2V2AUTHENTICATEANDREPLY_INSUFFICIENT_PRIVILEGES_ERROR	6
C1G2V2AUTHENTICATEANDREPLY_CRYPTO_SUITE_ERROR	7
C1G2V2AUTHENTICATEANDREPLY_COMMAND_NOT_ENCAPSULATED_ERROR	8
C1G2V2AUTHENTICATEANDREPLY_RESPONSEBUFFER_OVERFLOW_ERROR	9
C1G2V2AUTHENTICATEANDREPLY_SECURITY_TIMEOUT_ERROR	10
C1G2V2AUTHENTICATEANDREPLY_NO_RESPONSE_FROM_TAG	11
C1G2V2AUTHENTICATEANDREPLY_NON_SPECIFIC_READER_ERROR	12
C1G2V2AUTHENTICATEANDREPLY_INCORRECT_PASSWORD_ERROR	14
C1G2V2AUTHENTICATEANDREPLY_TAG_WORKING	15
C1G2V2AUTHENTICATEANDREPLY_CRYPTO_SUITE_NOT_SUPPORTED_ERROR	65
C1G2V2AUTHENTICATEANDREPLY_CRYPTO_SUITE_MEMORY_OVERRUN_ERROR	67
C1G2V2AUTHENTICATEANDREPLY_CRYPTO_SUITE_INSUFFICIENT_POWER_ERROR	75

113

6.11.3 Field Documentation

6.11.3.1 unsigned char result

Result values

6.11.3.2 unsigned short reserved

Reserved field

6.11.3.3 unsigned short messageLength

Message length in bits.

6.11.3.4 char message[64]

Response message to urmsp_C1G2V2AuthenticateAndReply command.

6.12 c1g2v2authenticateandstoreopspecresult_t Struct Reference

6.12.1 Data Fields

- unsigned char [result](#)
- unsigned short [reserved](#)
- unsigned short [storedMessageLength](#)

6.12.2 Detailed Description

C1G2V2AuthenticateAndStoreOpSpecResult Structure Contains the results of the C1G2V2AuthenticateAndStore reader operation.

Result	value
C1G2V2AUTHENTICATEANDSTORE_SUCCESS	0
C1G2V2AUTHENTICATEANDSTORE_TAG_MEMORY_OVERRUN_ERROR	1
C1G2V2AUTHENTICATEANDSTORE_TAG_MEMORY_LOCKED_ERROR	2
C1G2V2AUTHENTICATEANDSTORE_INSUFFICIENT_POWER_TO_PERFORM_AUTHENTICATE_OPERATION	3
C1G2V2AUTHENTICATEANDSTORE_NON_SPECIFIC_TAG_ERROR	4
C1G2V2AUTHENTICATEANDSTORE_NOT_SUPPORTED_PARAMETER_ERROR	5
C1G2V2AUTHENTICATEANDSTORE_INSUFFICIENT_PRIVILEGES_ERROR	6
C1G2V2AUTHENTICATEANDSTORE_CRYPTO_SUITE_ERROR	7
C1G2V2AUTHENTICATEANDSTORE_COMMAND_NOT_ENCAPSULATED_ERROR	8
C1G2V2AUTHENTICATEANDSTORE_RESPONSEBUFFER_OVERFLOW_ERROR	9
C1G2V2AUTHENTICATEANDSTORE_SECURITY_TIMEOUT_ERROR	10
C1G2V2AUTHENTICATEANDSTORE_NO_RESPONSE_FROM_TAG	11
C1G2V2AUTHENTICATEANDSTORE_NON_SPECIFIC_READER_ERROR	12
C1G2V2AUTHENTICATEANDSTORE_INCORRECT_PASSWORD_ERROR	14
C1G2V2AUTHENTICATEANDSTORE_TAG_WORKING	15
C1G2V2AUTHENTICATEANDSTORE_CRYPTO_SUITE_NOT_SUPPORTED_ERROR	65
C1G2V2AUTHENTICATEANDSTORE_CRYPTO_SUITE_MEMORY_OVERRUN_ERROR	67
C1G2V2AUTHENTICATEANDSTORE_CRYPTO_SUITE_INSUFFICIENT_POWER_ERROR	75

114

6.12.3 Field Documentation

6.12.3.1 unsigned char result

Result values

6.12.3.2 unsigned short reserved

Reserved field

6.12.3.3 unsigned short storedMessageLength

Length of the stored message to urmsp_C1G2V2AuthenticateAndStore command.

6.13 c1g2v2readbufferopspecresult_t Struct Reference

6.13.1 Data Fields

- unsigned char [result](#)
- unsigned short [reserved](#)
- unsigned short [bitCount](#)
- char [data](#) [64]

6.13.2 Detailed Description

C1G2V2ReadBufferResult Structure Contains the results of the C1G2V2ReadBuffer reader operation.

Result	value
C1G2V2READBUFFER_SUCCESS	0
C1G2V2READBUFFER_TAG_MEMORY_OVERRUN_ERROR	1
C1G2V2READBUFFER_TAG_MEMORY_LOCKED_ERROR	2
C1G2V2READBUFFER_INSUFFICIENT_POWER_TO_PERFORM_READBUFFER_OPERATION	3
C1G2V2READBUFFER_NON_SPECIFIC_TAG_ERROR	4
C1G2V2READBUFFER_NOT_SUPPORTED_PARAMETER_ERROR	5
C1G2V2READBUFFER_INSUFFICIENT_PRIVILEGES_ERROR	6
C1G2V2READBUFFER_CRYPTO_SUITE_ERROR	7
C1G2V2READBUFFER_COMMAND_NOT_ENCAPSULATED_ERROR	8
C1G2V2READBUFFER_RESPONSEBUFFER_OVERFLOW_ERROR	9
C1G2V2READBUFFER_SECURITY_TIMEOUT_ERROR	10
C1G2V2READBUFFER_NO_RESPONSE_FROM_TAG	11
C1G2V2READBUFFER_NON_SPECIFIC_READER_ERROR	12
C1G2V2READBUFFER_INCORRECT_PASSWORD_ERROR	14

115

6.13.3 Field Documentation

6.13.3.1 unsigned char result

Result values

6.13.3.2 unsigned short reserved

Reserved field

6.13.3.3 unsigned short bitCount

Amount of returned data from read buffer in bits.

6.13.3.4 char data[64]

Data requested from read buffer.

6.14 c1g2v2untraceableopspecresult_t Struct Reference

6.14.1 Data Fields

- unsigned char [result](#)
- unsigned short [reserved](#)

6.14.2 Detailed Description

C1G2V2UntraceableResult Structure Contains the results of the C1G2V2Untraceable reader operation.

Result	value
C1G2V2UNTRACEABLE_SUCCESS	0
C1G2V2UNTRACEABLE_TAG_MEMORY_OVERRUN_ERROR	1
C1G2V2UNTRACEABLE_TAG_MEMORY_LOCKED_ERROR	2
C1G2V2UNTRACEABLE_INSUFFICIENT_POWER_TO_PERFORM_UNTRACEABLE_OPERATION	3
C1G2V2UNTRACEABLE_NON_SPECIFIC_TAG_ERROR	4
C1G2V2UNTRACEABLE_NOT_SUPPORTED_PARAMETER_ERROR	5
C1G2V2UNTRACEABLE_INSUFFICIENT_PRIVILEGES_ERROR	6
C1G2V2UNTRACEABLE_CRYPTO_SUITE_ERROR	7
C1G2V2UNTRACEABLE_COMMAND_NOT_ENCAPSULATED_ERROR	8
C1G2V2UNTRACEABLE_RESPONSEBUFFER_OVERFLOW_ERROR	9
C1G2V2UNTRACEABLE_SECURITY_TIMEOUT_ERROR	10
C1G2V2UNTRACEABLE_NO_RESPONSE_FROM_TAG	11
C1G2V2UNTRACEABLE_NON_SPECIFIC_READER_ERROR	12
C1G2V2UNTRACEABLE_INCORRECT_PASSWORD_ERROR	14

116

6.14.3 Field Documentation

6.14.3.1 unsigned char result

Result values

6.14.3.2 unsigned short reserved

Reserved field

6.15 c1g2writeopspecresult_t Struct Reference

6.15.1 Data Fields

- unsigned char [result](#)
- unsigned short [reserved](#)
- unsigned short [wordCount](#)

6.15.2 Detailed Description

C1G2WriteOpSpecResult Structure Contains the results of the C1G2Write reader operation.

Result	value
C1G2WRITE_SUCCESS	0
C1G2WRITE_TAG_MEMORY_OVERRUN_ERROR	1
C1G2WRITE_TAG_MEMORY_LOCKED_ERROR	2
C1G2WRITE_INSUFFICIENT_POWER_TO_PERFORM_MEMORY_WRITE_OPERATION	3
C1G2WRITE_NON_SPECIFIC_TAG_ERROR	4
C1G2WRITE_NO_RESPONSE_FROM_TAG	5
C1G2WRITE_NON_SPECIFIC_READER_ERROR	6
C1G2WRITE_INCORRECT_PASSWORD_ERROR	7

6.15.3 Field Documentation

6.15.3.1 *unsigned char result*

Result values

6.15.3.2 *unsigned short reserved*

Reserved field

6.15.3.3 *unsigned short wordCount*

Word Count is the number of 16-bit words written.

6.16 gpievent_t Struct Reference

6.16.1 Data Fields

- unsigned short [gPIPortNumber](#)
- bool [e](#)
- unsigned char [reserved](#)

6.16.2 Detailed Description

GPIEvent Structure Contains the results of reader event.

6.16.3 Field Documentation

6.16.3.1 *unsigned short gPIPortNumber*

General Purpose Input Port Number.

6.16.3.2 *bool e*

Transition on an input pin of the Reader

0x01 = 0=>1

0x00 = 1=>0

6.16.3.3 *unsigned char reserved*

Reserved field

6.17 higgs3blockreadlockresult_t Struct Reference

6.17.1 Data Fields

- unsigned char [result](#)
- unsigned short [reserved](#)

6.17.2 Detailed Description

Higgs3BlockReadLockResult Structure Contains the results of the Higgs3BlockReadLock reader operation.

Result	value
HIGGS3BLOCKREADLOCK_SUCCESS	0
HIGGS3BLOCKREADLOCK_INVALID_START_STATE_FIELD_VALUES	1
HIGGS3BLOCKREADLOCK_TAG_MEMORY_OVERRUN_ERROR	2
HIGGS3BLOCKREADLOCK_TAG_MEMORY_LOCKED_ERROR	3
HIGGS3BLOCKREADLOCK_INSUFFICIENT_POWER_TO_PERFORM_OPERATION	4
HIGGS3BLOCKREADLOCK_NON_SPECIFIC_TAG_ERROR	5
HIGGS3BLOCKREADLOCK_NO_RESPONSE_FROM_TAG	6
HIGGS3BLOCKREADLOCK_NON_SPECIFIC_READER_ERROR	7

6.17.3 Field Documentation

6.17.3.1 *unsigned char result*

Result values

6.17.3.2 *unsigned short reserved*

Reserved field

6.18 log_t Struct Reference

6.18.1 Data Fields

- unsigned short [logLength](#)
- char [log](#) [255]

6.18.2 Detailed Description

Log Structure Contains log message.

6.18.3 Field Documentation

6.18.3.1 *unsigned short logLength*

Log message length

6.18.3.2 *char log[255]*

Log message

6.19 rospevent_t Struct Reference

6.19.1 Data Fields

- unsigned char [eventType](#)
- unsigned int [rOSpecID](#)
- unsigned int [preemptingROSpecID](#)

6.19.2 Detailed Description

ROSpecEvent Structure Contains the results of reader event.

Event type	value
Start of ROSpec	0
End of ROSpec	1
Preemption of ROSpec	2

6.19.3 Field Documentation

6.19.3.1 *unsigned char eventType*

Event type

6.19.3.2 *unsigned int rOSpecID*

This is the ID of the ROSpec that started, ended or got preempted.

6.19.3.3 *unsigned int preemptingROSpecID*

NOT SUPPORTED This field carries the ID of the preempting ROSpec. This field is ignored when EventType != 2.

7 Examples

7.1 Access Password Example

```
#ifndef ACCESSPASSWORDEXAMPLE
#define ACCESSPASSWORDEXAMPLE

#include <string>

static void callBackLog(void *p,unsigned int commandType, void *structPointer);
static void callBackWrite(void *p, unsigned int commandType, void *structPointer);

int waitForResult(bool *flag, int timeout);

std::string getResult(unsigned int resultType, unsigned char resultCode);
std::string getError(int errorCode);

#endif // ACCESSPASSWORDEXAMPLE
```

```
#include "accesspasswordexample.h"
#include "apic.h"

#include <iostream>
#include <mutex>

std::mutex mtx;

c1g2writeopspecresult t m_writeResult;
bool m_writeResultReceived = false;

int main()
{
    //pointer to class object
    void *cb = NULL;

    //This variable must be set based on access password lock status.
    bool accessPasswordLocked = true;

    int accessPassword;
    int oldAccessPassword = 0x87654321;
    int newAccessPassword = 0x12345678;

    //Initialize the library
    initDll();

    //Connect callback for log messages
    connectCallbackLog(&callBackLog,cb);

    //Connect callback for C1G2Write
    connectCallbackC1G2Write(&callBackWrite,cb);

    //Set COM port number.
    char prt[] = "com4";

    //Connecting reader
    int errorCode = connectReader(prt);

    if(errorCode == TSS_SUCCESS)
    {
        //Connection successful

        if(accessPasswordLocked)
        {
            //If the access password is locked
```

```

//we have to use the old access password(0x87654321) to rewrite it.
accessPassword = oldAccessPassword;
}
else
{
    accessPassword = 0x00000000;
    //If the access password is not locked
    //we dont use access password so we set it to zero.
}

//Write first 2 bytes of newAccessPassword(0x1234) to Reserved memory bank.
errorCode = urmsp\_C1G2Write(0x00,//flags
    0x00,//reserved
    0x00,//OpSpecID
    accessPassword,//accessPassword
    0x00,//memory bank (0x00 = Reserved Memory)
    0x02,//wordPointer
    0x01,//wordCount (0x01 = 1 word)
    newAccessPassword >> 16);//writeData

if(errorCode == TSS_SUCCESS)
{
    //Command successful

    //This function wait for C1G2Write result for 1500ms
    errorCode = waitForResult(&m_writeResultReceived,1500);

    if(errorCode == TSS_SUCCESS)
    {
        std::cout << "\nC1G2Write Result received";

        if(m_writeResult.result == TSS_SUCCESS)
        {
            std::cout << "\nC1G2Write result code: "
                << (int)m_writeResult.result;

            std::cout << "\nC1G2Write result string: "
                << getResult(URMSP_C1G2WRITEOPSPECRESULT,
                    m_writeResult.result);

            std::cout << "\nC1G2Write wordCount: "
                << m_writeResult.wordCount;
        }
        else
        {
            std::cout << getResult(URMSP_C1G2WRITEOPSPECRESULT,
                m_writeResult.result);
        }
    }
    else
    {
        std::cout << "\nwaitForResult timeout";
        std::cout << getError(errorCode);
    }
}

if(errorCode == TSS_SUCCESS)
{
    if(accessPasswordLocked)
    {
        //Because we changed the first 2 bytes of the access password
        //we have to use the new one(0x12344321)
        accessPassword = ((newAccessPassword & 0xFFFF0000) | (oldAccessPassword & 0x0000FFFF));
    }
    else
    {
        accessPassword = 0x00000000;
        //If the access password is not locked
        //we dont use access password so we set it to zero.
    }
}

```

```

//Write last 2 bytes of newAccessPassword(0x5678) to Reserved memory bank.
errorCode = urmsp\_C1G2Write(0x00,//flags
    0x00,//reserved
    0x00,//OpSpecID
    accessPassword,//accessPassword
    0x00,//memory bank (0x00 = Reserved Memory)
    0x03,//wordPointer
    0x01,//wordCount (0x01 = 1 word)
    newAccessPassword);//writeData
}

if(errorCode == TSS_SUCCESS)
{
    //Command successful

    //This function wait for C1G2Write result for 1500ms
    errorCode = waitForResult(&m_writeResultReceived,1500);

    if(errorCode == TSS_SUCCESS)
    {
        std::cout << "\nC1G2Write Result received";

        if(m_writeResult.result == TSS_SUCCESS)
        {
            std::cout << "\nC1G2Write result code: "
                << (int)m_writeResult.result;

            std::cout << "\nC1G2Write result string: "
                << getResult(URMSP_C1G2WRITEOPSPECRESULT,
                    m_writeResult.result);

            std::cout << "\nC1G2Write wordCount: "
                << m_writeResult.wordCount;
        }
        else
        {
            std::cout << getResult(URMSP_C1G2WRITEOPSPECRESULT,
                m_writeResult.result);
        }
    }
    else
    {
        std::cout << "\nwaitForResult timeout";
        std::cout << getError(errorCode);
    }
}

//Disconnecting reader
errorCode = disconnectReader();

if(errorCode == TSS_SUCCESS)
{
    //Disconnection successful
}
else
{
    //Disconnection failed
    std::cout << getError(errorCode);
}
else
{
    //Connection failed
    std::cout << getError(errorCode);
}
}

//This callback function receives log messages(serial port
//communication, events, error messages, etc.)
static void callBackLog(void *p,unsigned int commandType, void *structPointer)

```

```

{
    log_t logStruct = *(log_t*)structPointer;

    std::string logString(logStruct.log,logStruct.logLength);
    std::cout << "\nLog: " << logString;
}

//This callback function is called each time the reader receives
//C1G2WriteOpSpecResult the result of the C1G2Write command.
static void callBackWrite(void *p, unsigned int commandType, void *structPointer)
{
    std::lock_guard<std::mutex> lock(mtx);

    //C1G2Write command type value.
    if(commandType == URMSP_C1G2WRITE)
    {
        m_writeResult = *(c1g2writeopspecresult_t*)structPointer;

        m_writeResultReceived = true;
    }
}

int waitForResult(bool *flag, int timeout)
{
    int x;

    for(x = 0; x < timeout; x += 100)    //Waiting x millisecond
    {
        //Lock the mutex to avoid access to to flag from different thread
        mtx.lock();

        //Check if structure was written
        if(*flag)
        {
            //Reset the flag
            *flag = false;

            mtx.unlock();
            return 0;
        }
        mtx.unlock();

        Sleep(100);
    }

    if(x == timeout)
    {
        //Timeout
        return -1;
    }
}

//This function translate result code to result strings
std::string getResult(unsigned int resultType, unsigned char resultCode)
{
    char resultString[1024];

    unsigned int resultStringLength;

    getResultString(resultString,&resultStringLength,resultType,resultCode);

    return std::string(resultString,resultStringLength);
}

//This function translate error code to error strings
std::string getError(int errorCode)
{
    char errorSourceString[1024];
    char errorCodeString[1024];

    unsigned int errorSourceStringLength;

```

```
unsigned int errorCodeStringLength;  
  
getErrorString(errorSourceString,  
                &errorSourceStringLength,  
                errorCodeString,  
                &errorCodeStringLength,  
                errorCode);  
  
return std::string(errorCodeString,errorCodeStringLength);  
}
```

8 Importing the libraries

8.1 Import libraries to Microsoft Visual Studio

- Go to Project -> Properties -> Configuration Properties -> Linker -> Additional Dependencies and add path to your TSS-APIC.lib as a next line.
- Include apic.h header file in project.
- Make sure that the TSS-APIC.dll and all third party .dll files is either in the directory contained by the PATH% environment variable or that its copy is in Output Directory (by default, this is Debug \ Release under your project's folder).

8.2 Import libraries to Qt Creator

- Make sure Qt libraries version are same as the TSS-APIC.dll was built with.
- Edit your application project file (.pro):
 - INCLUDEPATH += "C:\Path\To\The\TSS-APIC"
 - LIBS += "C:\Path\To\The\TSS-APIC\TSS-APIC.lib"
- Include the header file apic.h
- Make sure that the TSS-APIC.dll and all third party .dll files is either in the directory contained by the PATH% environment variable or that its copy is in Output Directory (by default, this is Debug \ Release under your project's folder).

9 References

1. URMSP, Serial Communication Protocol, Reference Manual, Ver. 1.1, 03/2015
<https://tsscompany.eu/customerzone/>
2. Low Level Reader Protocol
http://www.gs1.org/sites/default/files/docs/epc/llrp_1_1-standard-20101013.pdf

10Index

- bitCount
 - c1g2customcommandresult_t, 107
 - c1g2v2readbufferopspecresult_t, 115
- c1g2blockeraseopspecresult_t, 103
 - reserved, 103
 - result, 103
 - wordCount, 103
- c1g2blockpermalockopspecresult_t, 104
 - reserved, 104
 - result, 104
- c1g2blockwriteopspecresult_t, 105
 - reserved, 105
 - result, 105
 - wordCount, 105
- c1g2customcommandresult_t, 107
 - bitCount, 107
 - replyData, 107
 - reserved, 107
 - result, 107
- c1g2epc_t, 108
 - epc, 108
 - epcLength, 108
- c1g2getblockpermalockstatusopspecresult_t, 109
 - permalockStatus, 109
 - reserved, 109
 - result, 109
 - statusWordCount, 109
- c1g2killopspecresult_t, 110
 - reserved, 110
 - result, 110
- c1g2lockopspecresult_t, 111
 - reserved, 111
 - result, 111
- c1g2readopspecresult_t, 112
 - data, 112
 - reserved, 112
 - result, 112
 - wordCount, 112
- c1g2v2authenticateandreplyopspecresult_t, 113
 - message, 113
 - messageLength, 113
 - reserved, 113
 - result, 113
- c1g2v2authenticateandstoreopspecresult_t, 114
 - reserved, 114
 - result, 114
 - storedMessageLength, 114
- c1g2v2readbufferopspecresult_t, 115
 - bitCount, 115
 - data, 115
 - reserved, 115
 - result, 115
- c1g2v2untraceableopspecresult_t, 116
 - reserved, 116
 - result, 116
- c1g2writeopspecresult_t, 117
 - reserved, 117
 - result, 117
 - wordCount, 117
- connectCallbackC1G2BlockErase
 - TSS API-C, 14
- connectCallbackC1G2BlockPermalock
 - TSS API-C, 15
- connectCallbackC1G2BlockWrite
 - TSS API-C, 14
- connectCallbackC1G2CustomCommand
 - TSS API-C, 17
- connectCallbackC1G2GetBlockPermalockStatus
 - TSS API-C, 15
- connectCallbackC1G2Kill
 - TSS API-C, 13
- connectCallbackC1G2Lock
 - TSS API-C, 14
- connectCallbackC1G2Read
 - TSS API-C, 13
- connectCallbackC1G2V2AuthenticateAndReply
 - TSS API-C, 15
- connectCallbackC1G2V2AuthenticateAndStore
 - TSS API-C, 16
- connectCallbackC1G2V2ReadBuffer
 - TSS API-C, 16
- connectCallbackC1G2V2Untraceable
 - TSS API-C, 16
- connectCallbackC1G2Write
 - TSS API-C, 13
- connectCallbackEPC
 - TSS API-C, 12
- connectCallbackHiggs3BlockReadLock
 - TSS API-C, 17
- connectCallbackLog
 - TSS API-C, 17
- connectCallbackReaderEvents
 - TSS API-C, 12
- connectReader
 - TSS API-C, 18
- connectReaderTCP
 - TSS API-C, 19
- data
 - c1g2readopspecresult_t, 112
 - c1g2v2readbufferopspecresult_t, 115
- deviceManufacturerName
 - generalDeviceCapabilities_t, 106
- disconnectReader
 - TSS API-C, 19
- e
 - gpievent_t, 118
- epc
 - c1g2epc_t, 108
- epcLength
 - c1g2epc_t, 108
- eventType
 - rospecevent_t, 121
- Examples, 122
- firmwareVersionByteCount

generalDeviceCapabilities_t, 106
 generalDeviceCapabilities_t, 106
 deviceManufacturerName, 106
 firmwareVersionByteCount, 106
 maxNumberOfAntennaSupported, 106
 modelName, 106
 readerFirmwareVersion, 106
 reserved, 106
 getErrorString
 TSS API-C, 102
 getResultString
 TSS API-C, 102
 gpievent_t, 118
 e, 118
 gPIPortNumber, 118
 reserved, 118
 gPIPortNumber
 gpievent_t, 118
 higgs3blockreadlockresult_t, 119
 reserved, 119
 result, 119
 Importing the libraries, 127
 initDll
 TSS API-C, 12
 log
 log_t, 120
 log_t, 120
 log, 120
 logLength, 120
 logLength
 log_t, 120
 Macro Definitions, 6
 maxNumberOfAntennaSupported
 generalDeviceCapabilities_t, 106
 message
 c1g2v2authenticateandreplyopspecresult_t, 113
 messageLength
 c1g2v2authenticateandreplyopspecresult_t, 113
 modelName
 generalDeviceCapabilities_t, 106
 permalockStatus
 c1g2getblockpermalockstatusopspecresult_t, 109
 preemtingROSpecID
 rospecevent_t, 121
 readerFirmwareVersion
 generalDeviceCapabilities_t, 106
 References, 128
 replyData
 c1g2customcommandresult_t, 107
 reserved
 c1g2blockeraseopspecresult_t, 103
 c1g2blockpermalockopspecresult_t, 104
 c1g2blockwriteopspecresult_t, 105
 c1g2customcommandresult_t, 107
 c1g2getblockpermalockstatusopspecresult_t, 109
 c1g2killopspecresult_t, 110
 c1g2lockopspecresult_t, 111
 c1g2readopspecresult_t, 112
 c1g2v2authenticateandreplyopspecresult_t, 113
 c1g2v2authenticateandstoreopspecresult_t, 114
 c1g2v2readbufferopspecresult_t, 115
 c1g2v2untraceableopspecresult_t, 116
 c1g2writeopspecresult_t, 117
 generalDeviceCapabilities_t, 106
 gpievent_t, 118
 higgs3blockreadlockresult_t, 119
 result
 c1g2blockeraseopspecresult_t, 103
 c1g2blockpermalockopspecresult_t, 104
 c1g2blockwriteopspecresult_t, 105
 c1g2customcommandresult_t, 107
 c1g2getblockpermalockstatusopspecresult_t, 109
 c1g2killopspecresult_t, 110
 c1g2lockopspecresult_t, 111
 c1g2readopspecresult_t, 112
 c1g2v2authenticateandreplyopspecresult_t, 113
 c1g2v2authenticateandstoreopspecresult_t, 114
 c1g2v2readbufferopspecresult_t, 115
 c1g2v2untraceableopspecresult_t, 116
 c1g2writeopspecresult_t, 117
 higgs3blockreadlockresult_t, 119
 rospecevent_t, 121
 eventType, 121
 preemtingROSpecID, 121
 rOSpecID, 121
 rOSpecID
 rospecevent_t, 121
 setDebugLevel
 TSS API-C, 12
 statusWordCount
 c1g2getblockpermalockstatusopspecresult_t, 109
 storedMessageLength
 c1g2v2authenticateandstoreopspecresult_t, 114
 TSS API-C
 connectCallbackC1G2BlockErase, 14
 connectCallbackC1G2BlockPermalock, 15
 connectCallbackC1G2BlockWrite, 14
 connectCallbackC1G2CustomCommand, 17
 connectCallbackC1G2GetBlockPermalockStatus, 15
 connectCallbackC1G2Kill, 13
 connectCallbackC1G2Lock, 14
 connectCallbackC1G2Read, 13
 connectCallbackC1G2V2AuthenticateAndReply, 15
 connectCallbackC1G2V2AuthenticateAndStore, 16
 connectCallbackC1G2V2ReadBuffer, 16
 connectCallbackC1G2V2Untraceable, 16
 connectCallbackC1G2Write, 13
 connectCallbackEPC, 12
 connectCallbackHiggs3BlockReadLock, 17
 connectCallbackLog, 17
 connectCallbackReaderEvents, 12
 connectReader, 18
 connectReaderTCP, 19
 disconnectReader, 19
 getErrorString, 102
 getResultString, 102
 initDll, 12

setDebugLevel, 12
 urmsp_AISpecStopTrigger, 31
 urmsp_C1G2BlockErase, 64
 urmsp_C1G2BlockPermalock, 72
 urmsp_C1G2BlockWrite, 68
 urmsp_C1G2CustomCommand, 95
 urmsp_C1G2GetBlockPermalockStatus, 76
 urmsp_C1G2Kill, 55
 urmsp_C1G2Lock, 59
 urmsp_C1G2LockPayload, 63
 urmsp_C1G2Read, 47
 urmsp_C1G2SingulationControl, 45
 urmsp_C1G2TagInventoryMask, 40
 urmsp_C1G2TagInventoryStateAwareFilterAction, 44
 urmsp_C1G2TagInventoryStateAwareSingulationAction, 46
 urmsp_C1G2V2AuthenticateAndReply, 80
 urmsp_C1G2V2AuthenticateAndStore, 84
 urmsp_C1G2V2ReadBuffer, 90
 urmsp_C1G2V2Untraceable, 85
 urmsp_C1G2Write, 51
 urmsp_EventNotificationState, 36
 urmsp_GetReaderCapabilities, 20
 urmsp_GetROSpecs, 28
 urmsp_GPITriggerValue, 30
 urmsp_Higgs3BlockReadLock, 91
 urmsp_RFTransmitter, 33
 urmsp_ROSpecStartTrigger, 30
 urmsp_SetPeripherals, 97
 urmsp_SetPowerMode, 100
 urmsp_StartROSpec, 22
 urmsp_StopROSpec, 25
 urmsp_TagObservationTrigger, 32
 waitForResultStruct, 20
 urmsp_AISpecStopTrigger
 TSS API-C, 31
 urmsp_C1G2BlockErase
 TSS API-C, 64
 urmsp_C1G2BlockPermalock
 TSS API-C, 72
 urmsp_C1G2BlockWrite
 TSS API-C, 68
 urmsp_C1G2CustomCommand
 TSS API-C, 95
 urmsp_C1G2GetBlockPermalockStatus
 TSS API-C, 76
 urmsp_C1G2Kill
 TSS API-C, 55
 urmsp_C1G2Lock
 TSS API-C, 59
 urmsp_C1G2LockPayload
 TSS API-C, 63
 urmsp_C1G2Read
 TSS API-C, 47
 urmsp_C1G2SingulationControl
 TSS API-C, 45
 urmsp_C1G2TagInventoryMask
 TSS API-C, 40
 urmsp_C1G2TagInventoryStateAwareFilterAction
 TSS API-C, 44
 urmsp_C1G2TagInventoryStateAwareSingulationAction
 TSS API-C, 46
 urmsp_C1G2V2AuthenticateAndReply
 TSS API-C, 80
 urmsp_C1G2V2AuthenticateAndStore
 TSS API-C, 84
 urmsp_C1G2V2ReadBuffer
 TSS API-C, 90
 urmsp_C1G2V2Untraceable
 TSS API-C, 85
 urmsp_C1G2Write
 TSS API-C, 51
 urmsp_EventNotificationState
 TSS API-C, 36
 urmsp_GetReaderCapabilities
 TSS API-C, 20
 urmsp_GetROSpecs
 TSS API-C, 28
 urmsp_GPITriggerValue
 TSS API-C, 30
 urmsp_Higgs3BlockReadLock
 TSS API-C, 91
 urmsp_RFTransmitter
 TSS API-C, 33
 urmsp_ROSpecStartTrigger
 TSS API-C, 30
 urmsp_SetPeripherals
 TSS API-C, 97
 urmsp_SetPowerMode
 TSS API-C, 100
 urmsp_StartROSpec
 TSS API-C, 22
 urmsp_StopROSpec
 TSS API-C, 25
 urmsp_TagObservationTrigger
 TSS API-C, 32
 waitForResultStruct
 TSS API-C, 20
 wordCount
 c1g2blockeraseopspecresult_t, 103
 c1g2blockwriteopspecresult_t, 105
 c1g2readopspecresult_t, 112
 c1g2writeopspecresult_t, 117