© 2006 by Symbol Technologies, Inc. All rights reserved.

No part of this publication may be reproduced or used in any form, or by any electrical or mechanical means, without permission in writing from Symbol. This includes electronic or mechanical means, such as photocopying, recording, or information storage and retrieval systems. The material in this manual is subject to change without notice.

The software is provided strictly on an “as is” basis. All software, including firmware, furnished to the user is on a licensed basis. Symbol grants to the user a non-transferable and non-exclusive license to use each software or firmware program delivered hereunder (licensed program). Except as noted below, such license may not be assigned, sublicensed, or otherwise transferred by the user without prior written consent of Symbol. No right to copy a licensed program in whole or in part is granted, except as permitted under copyright law. The user shall not modify, merge, or incorporate any form or portion of a licensed program with other program material, create a derivative work from a licensed program, or use a licensed program in a network without written permission from Symbol. The user agrees to maintain Symbol’s copyright notice on the licensed programs delivered hereunder, and to include the same on any authorized copies it makes, in whole or in part. The user agrees not to decompile, disassemble, decode, or reverse engineer any licensed program delivered to the user or any portion thereof.

Symbol reserves the right to make changes to any software or product to improve reliability, function, or design.

Symbol does not assume any product liability arising out of, or in connection with, the application or use of any product, circuit, or application described herein.

No license is granted, either expressly or by implication, estoppel, or otherwise under any Symbol Technologies, Inc., intellectual property rights. An implied license only exists for equipment, circuits, and subsystems contained in Symbol products.

Symbol, Spectrum One, and Spectrum24 are registered trademarks of Symbol Technologies, Inc. Bluetooth is a registered trademark of Bluetooth SIG. Microsoft, Windows and ActiveSync are either registered trademarks or trademarks of Microsoft Corporation. Other product names mentioned in this manual may be trademarks or registered trademarks of their respective companies and are hereby acknowledged.

Symbol Technologies, Inc.
One Symbol Plaza
Holtsville, New York 11742-1300
http://www.symbol.com

Patents

This product is covered by one or more of the patents listed on the webbiest: www.symbol.com/patents
# Revision History

Changes to the original manual are listed below:

<table>
<thead>
<tr>
<th>Change</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-01 Rev A</td>
<td>June 2005</td>
<td>Initial Release</td>
</tr>
<tr>
<td>-02 Rev A</td>
<td>July 2006</td>
<td>Added Gen 2 update</td>
</tr>
<tr>
<td>Contents</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Revision History ........................................ iii</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>About This Guide</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Introduction ........................................... xi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter Descriptions ................................... xi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Notational Conventions ................................ xi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Related Documents and Software ....................... xii</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Service Information .................................... xii</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Chapter 1. Getting Started</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Introduction ........................................... 1-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Chapter 2. Structures and Definitions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Introduction ........................................... 2-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Definitions ............................................. 2-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Chapter 3. Initialization</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Introduction ........................................... 3-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RFID_OPEN .............................................. 3-5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RFID_CLOSE ............................................. 3-8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chapter 4. Device Capabilities Discovery</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Introduction ................................</td>
<td>4-3</td>
<td></td>
</tr>
<tr>
<td>RFID_GetCapList ................................</td>
<td>4-4</td>
<td></td>
</tr>
<tr>
<td>RFID_GetCapInfo ................................</td>
<td>4-5</td>
<td></td>
</tr>
<tr>
<td>RFID_GetCapCurrValue .......................</td>
<td>4-7</td>
<td></td>
</tr>
<tr>
<td>RFID_GetCapDfltValue ......................</td>
<td>4-10</td>
<td></td>
</tr>
<tr>
<td>RFID_SetCapCurrValue ......................</td>
<td>4-11</td>
<td></td>
</tr>
<tr>
<td>RFID_SetCapDfltValue ......................</td>
<td>4-12</td>
<td></td>
</tr>
<tr>
<td>RFID_SetCapDfltValue ......................</td>
<td>4-13</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 5. Reading and Writing Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction ................................</td>
</tr>
<tr>
<td>RFID_GetTagID ................................</td>
</tr>
<tr>
<td>RFID_GetTagMask ............................</td>
</tr>
<tr>
<td>RFID_SetTagMask ............................</td>
</tr>
<tr>
<td>RFID_KillTag ..................................</td>
</tr>
<tr>
<td>RFID_LockTag ...................................</td>
</tr>
<tr>
<td>RFID_ProgramTags ............................</td>
</tr>
<tr>
<td>RFID_EraseTag ..................................</td>
</tr>
<tr>
<td>RFID_ReadTagInventory .......................</td>
</tr>
<tr>
<td>GPIO Support ...................................</td>
</tr>
<tr>
<td>Extended Version of Data Structures and Functions</td>
</tr>
<tr>
<td>Error in Tag Reading ..........................</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 6. General Helper Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction ................................</td>
</tr>
<tr>
<td>RFID_GetCommandStatusText .............</td>
</tr>
<tr>
<td>RFID_GetStats ................................</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Appendix A. RFID API Capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction ...........................</td>
</tr>
<tr>
<td>RFID_INFCAP_SUPPORTEDCAPS ..........</td>
</tr>
<tr>
<td>RFID_DEVCAP_IP_NAME ..................</td>
</tr>
<tr>
<td>RFID_DEVCAP_IP_PORT ..................</td>
</tr>
<tr>
<td>RFID_DEVCAP_ANTENNA_SEQUENCE .......</td>
</tr>
<tr>
<td>RFID_READCAP_RF_ATTENUATION ..........</td>
</tr>
<tr>
<td>RFID_READCAP_EVENTTAGPTR ............</td>
</tr>
<tr>
<td>RFID_READCAP_EVENTNAME ...............</td>
</tr>
<tr>
<td>RFID_READCAP_EVENT_ALLTAGS ..........</td>
</tr>
<tr>
<td>RFID_READCAP_METHOD ...................</td>
</tr>
<tr>
<td>RFID_READCAP_OUTLOOP ...................</td>
</tr>
<tr>
<td>RFID_READCAP_INLOOP ....................</td>
</tr>
</tbody>
</table>
Appendix B. Return Values

Introduction ................................................................. B-3

Appendix C. Sample Application

Introduction ................................................................. C-3

Index

Tell Us What You Think
About This Guide

Introduction .......................................................... xi
Chapter Descriptions .............................................. xi
Notational Conventions ........................................... xi
Related Documents and Software .............................. xii
Service Information ................................................ xii
Introduction

This guide provides information on using C API to develop applications to read and write tags on an XR Series RFID reader. The C API provides an interface with the XR Series readers using the parameters defined in the XR Series Interface Control Guide, p/n 72E-71803-xx.

Chapter Descriptions

Topics covered in this guide are as follows:

- **Chapter 1, Getting Started** provides an overview of the RFID C API.
- **Chapter 3, Initialization** describes commands associated with device initialization and discovery.
- **Chapter 4, Device Capabilities Discovery** describes how to discover and change the RFID API capabilities.
- **Chapter 5, Reading and Writing Tags** describes tag reading and writing functions.
- **Chapter 6, General Helper Functions** includes commands for retrieving API status codes and module statistics.
- **Appendix A, RFID API Capabilities** provides capabilities information to use for reference when calling functions.
- **Appendix B, Return Values** describes the return values that the C API uses when responding to function calls.
- **Appendix C, Sample Application** provides a sample application for finding and opening a reader, and reading and displaying tags.

Notational Conventions

The following conventions are used in this document:

- **Italics** are used to highlight the following:
  - Chapters and sections in this and related documents
  - Dialog box, window and screen names
  - Drop-down list and list box names
  - Check box and radio button names
  - Icons on a screen.

- **Bold** text is used to highlight the following:
  - Key names on a keypad
  - Button names on a screen.

- **Bullets (•)** indicate:
  - Action items
  - Lists of alternatives
  - Lists of required steps that are not necessarily sequential.
  - Sequential lists (e.g., those that describe step-by-step procedures) appear as numbered lists.
Related Documents and Software

The following documents provide more information about the XR Series RFID readers.

- *XR Series RFID Readers Integrator Guide*, p/n 72E-71773-xx
- *XR Series Interface Control Guide*, p/n 72E-71803-xx
- *TagVis User Guide*, p/n 72E-71804-xx
- *ReaderComm5DLL Developer Guide*, p/n 72E-71805-xx

For the latest version of this guide and all guides, go to: [http://www.symbol.com/manuals](http://www.symbol.com/manuals).

Service Information

For service information, warranty information or technical assistance contact or call the Symbol Support Center. Contact information is provided on the Symbol contact web site go to: [http://www.symbol.com/contactsupport](http://www.symbol.com/contactsupport)

If the problem cannot be solved over the phone, the equipment may need to be returned for servicing. If that is necessary, specific directions will be provided.

Symbol Technologies is not responsible for any damages incurred during shipment if the approved shipping container is not used. Shipping the units improperly can possibly void the warranty.

If the Symbol product was purchased from a Symbol Business Partner, contact that Business Partner for service.
Introduction
The C API enables programmers to write applications which run either directly on an XR Series RFID reader or on a Windows XP SP2 PC host.

This API provides access to the reader features, by utilizing the byte stream protocol as defined in the XR Series Interface Control Guide, p/n 72E-71803-xx. The API exposes these features as a set of predefined capabilities, and carries out operations through function calls.

The API does not save any configuration information. All of the capabilities are initialized to their default values when an instance is created and any changes made by the user(s) is not persisted by the API. The users must manage and adjusting the parameters at runtime. This is true for both the PC and the XR reader platforms.
Structures and Definitions

Introduction ................................................................. 2-3
Definitions ................................................................. 2-3
Introduction
This section describes the code structures and definitions.

Definitions

**Capability Data Type**
// Definitions for capability data types
#define TWTY_INT8 0x0000 /* Means Item is a TW_INT8 */
#define TWTY_INT16 0x0001 /* Means Item is a TW_INT16 */
#define TWTY_INT32 0x0002 /* Means Item is a TW_INT32 */
#define TWTY_UINT8 0x0003 /* Means Item is a TW_UINT8 */
#define TWTY_UINT16 0x0004 /* Means Item is a TW_UINT16 */
#define TWTY_UINT32 0x0005 /* Means Item is a TW_UINT32 */

**Capability Container Type**
// Definitions for capability container types
#define TWON_UNKNOWN 0
#define TWON_ARRAY 3
#define TWON_ENUMERATION 4
#define TWON_ONEVALUE 5
#define TWON_RANGE 6

**Maximum Tag Type Array Size**
#define RFID_TAG_TYPE_MAX20 // Reserve for 20 types of tag in stats block
Tag Type Definition
enum RFID_TAG_TYPE_INDEX {
    RFID_TAG_TYPE_OTHER = 0,  //
    RFID_TAG_TYPE_EPC_CLASS0 = 1,  //
    RFID_TAG_TYPE_EPC_CLASS0PLUS = 2,  //
    RFID_TAG_TYPE_EPC_CLASS1 = 3,  //
    RFID_TAG_TYPE_EPC_CLASSG2 = 4,  //
    RFID_TAG_TYPE_EPC_CLASSOZUMA = 5,  //
    RFID_TAG_TYPE_MAXINDEX = 5,  // Max used
    RFID_TAG_TYPE_ERROR = 0xFFFFFFFF,  // Unassigned type
};

Maximum Tag Type ID Length
#define RFID_MAX_TAG_ID_LENGTH_EX64 // max tag id length

Tag Filter Option
#define RFID_FILTER_OPTION_PASS 0  // pass tags that match the supplied filter
#define RFID_FILTER_OPTION_DROP 1  // drop tags that match the supplied filter
#define RFID_FILTER_OPTION_NOMATCH_PASS 2  // pass tags that do not match the supplied filter
#define RFID_FILTER_OPTION_NOMATCH_DROP 3  // drop tags that do not match the supplied filter

Antenna Structure Size
// RFID_CAPS defines
#define RFID_ANTENNA_SEQUENCE_LENGTH 16
#define MAX_NUMBER_ANTENNA 8

Generic Structure Information
typedef struct tagSTRUCT_INFO {
    DWORD dwAllocated;  // Size of allocated structure in bytes
    DWORD dwUsed;  // Amount of structure actually used
} STRUCT_INFO;
**CAPINFO Structure**

// Capability information structure
typedef struct tagCAPINFO {
    STRUCT_INFO StructInfo; // Information about this structure
    DWORD dwCapId; // The capability ID
    BOOLEAN bSupported; // non zero: supported
    DWORD dwDataTypeId; // data type for this capability
    DWORD dwDataSize; // size (# of bytes) of this data type
    DWORD dwContainerId; // container type for this capability
    DWORD dwContainerSize; // container size for this capability
    DWORD dwNumItems; // Maximum Number of Item in this capability
    TCHAR tszName[MAX_CAP_NAME]; // name of this capability
} CAPINFO, *LPCAPINFO;

**TYPE_TAG Structure**

/* type storing tag data for a Tag List **************/
typedef struct
{
    unsigned char status; //Status of entry - discovered, ... 1
    unsigned char antennaNum; //Last antenna this tag has been read 2
    unsigned char dataLength; //Tag length in bytes 3
    unsigned char tagID[RFID_MAX_TAG_ID_LENGTH]; //ID code of tag 4-20
    SYSTEMTIME lastSeen; //Timestamp of last time tag was seen 21-36
    unsigned long readCount; //Number of reads of this tag 37-40
    DWORD dwType; //Type of Tag 41-44
    DWORD dwFormat; //Tag Data Format 45-48
    DWORD dwOperationStatus; //RFIDAPI status code 49-52
    WORD wStatusDetail; //extended status information 53-54
    WORD wReserved1; // 55-56
    DWORD dwReserved4; // 57-60
    DWORD dwReserved5; // Reserve 5 DWORDS for future 61-64
} TYPE_TAG;
TYPE_TAG_EX structure

typedef struct
{
    int nInit; // This will be initialized by macro
    int nTypeTagStructureFormat;
    int nTypeTagStructureSize;
} TYPE_TAG_STRUCTURE_HEADER;

typedef struct
{
    TYPE_TAG_STRUCTURE_HEADER Header;
    unsigned char status; // Status of entry - discovered, ...
    unsigned char antennaNum; // Last antenna this tag has been read
    DWORD dataLength; // Number of bytes in the tagID field. SYSTEMTIME firstSeen;
    // Timestamp of first time tag was seen
    SYSTEMTIME lastSeen; // Timestamp of last time tag was seen
    unsigned long readCount; // Number of reads of this tag
    DWORD dwType; // Type of Tag
    DWORD dwFormat; // Tag Data Format

    int nMemoryPageNumber; // For class 0 tags, memory page 2 is EPC.
    // Memory page 2 information is not stored in this
    // and the following memory page variables
    DWORD dwMemoryPageOffset; // Starting bit offset into tag's memory page where data was read.
    DWORD dwMemoryPageLength; // Length of memory page in bytes.

    DWORD dwOperationStatus; // Status code for the read operation
    WORD wStatusDetail; // Other hardware related detail
    WORD wCRC; // Reserved
    WORD wPC; // Reserved
    WORD wRSSI; // Reserved
    DWORD dwMillisecond; // Reserved
    DWORD dwReserved5; // Reserve DWORDS

    unsigned char tagID[RFID_MAX_TAG_ID_LENGTH_EX]; // ID code of tag
} TYPE_TAG_EX;
**TYPE_MASK Structure**
typedef struct
{
    unsigned char cBitLen;
    unsigned char cBitStartOffset;
    unsigned char cTagMask[RFID_MAX_TAG_ID_LENGTH];
    DWORD dwReserved[20];  // Reserve 20 DWORDS for future
} TAG_MASK;

**TYPE_MASK_EX structure**
typedef struct
{
    int nInit;               // This will be initialized by macro
    int nTagMaskStructureFormat;
    int nTagMaskStructureSize;
} TAG_MASK_STRUCTURE_HEADER;

typedef struct
{
    TAG_MASK_STRUCTURE_HEADER Header;
    unsigned int nBitLen;
    unsigned int nBitStartOffset;
    unsigned char cTagMask[RFID_MAX_TAG_ID_LENGTH_EX];
    DWORD dwReserved[20];  // Reserve 20 DWORDS for future
} TAG_MASK_EX;
**RFID_CAPS structure**

typedef struct 
{
    TCHAR szAPIVersionString[RFID_API_VERSION_STRING_LENGTH];
    TCHAR szFirmwareVersion[RFID_FIRMWARE_VERSION_STRING_LENGTH];
    TCHAR szMfgDateCode[RFID_FIRMWARE_MFGDATECODE_LENGTH];
    TCHAR szSerialInfo[RFID_FIRMWARE_SERIALINFO_LENGTH];
    unsigned char ReaderNumber;
    unsigned char Antenna;
    unsigned char AntennaSequence[RFID_ANTENNA_SEQUENCE_LENGTH];
    BOOL bPowerState;
    BOOL bPortOpen;
    DWORD dwReaderType;
    DWORD dwCurrentPort;
    DWORD dwCurrentBaud;
    DWORD dwRFIDDeviceTable[10];
    DWORD dwRFIDDeviceCount;
    DWORD dwRFAttenuation; // 1
    DWORD dwRFChannel; // 2
    DWORD dwLastReaderCmdSeqNum; // 3
    DWORD dwLastReaderCmd; // 4
    DWORD dwLastReaderRsp; // 5
    DWORD dwTagListSeqNum; // 6
    BYTE bSupportedTagTypes[RFID_TAG_TYPE_MAX]; // 7 - 11
    BYTE bEnabledTagTypes[RFID_TAG_TYPE_MAX]; // 12 - 16
    DWORD dwMonitorStatus; // 17
    BYTE bRFIDModuleType; // 18
    DWORD dwReserved[100-18]; // Reserve 100 DWORDS for future
}RFID_CAPS;
**TAG_LIST structure**
typedef struct
{
    DWORD dwTotalTags; // 1
    DWORD dwTotalReads; // 2
    DWORD dwNewTags; // 3
    DWORD dwReadTimeMS; // 4
    DWORD dwSeqNum; // 5
    DWORD dwMaxTags; // Max tags supported. 0 means 200
    DWORD dwErrorTags; // total of error tag packets appended to end of valid tags.
    DWORD dwReserved[25-7]; // Use a total of 25 DWORDS
    TYPE_TAG Tags[RFID_MAX_TAGS];
} TAG_LIST;

**TAG_LIST_EX structure**
typedef struct
{
    int nInit; // This will be initialized by macro
    int nTagListStructureFormat;
    int nTagListStructureSize;
} TAG_LIST_STRUCTURE_HEADER;

typedef struct
{
    TAG_LIST_STRUCTURE_HEADER Header;
    DWORD dwTotalTags; //
    DWORD dwTotalReads; //
    DWORD dwNewTags; //
    DWORD dwReadTimeMS; //
    DWORD dwSeqNum; //
    DWORD dwMaxTags; // Max tags supported. 0 means 200
    DWORD dwErrorTags; // total of error tag packets appended to end of valid tags
    DWORD dwReserved[19]; // Reserve a total of 19 DWORDS
    TYPE_TAG_EX Tags[RFID_MAX_TAGS];
} TAG_LIST_EX;
RFID_STATS structure
typedef struct
{
    DWORD dwVersion; // 1
    DWORD dwTotalTX; // 2
    DWORD dwTotalRX; // 3
    DWORD dwPacketsTX; // 4
    DWORD dwPacketsRX; // 5
    DWORD dwIncompleteTX; // 6
    DWORD dwPacketsFragmented; // 7
    DWORD dwPacketsCRCError; // 8
    DWORD dwSessionIDError; // 9
    DWORD dwTotalReads; // 10
    DWORD dwTotalTagCRCError; // 11
    DWORD dwTotalTagCollisions; // 12
    DWORD dwTimeouts; // 13
    DWORD dwTotal_10MSOnTime; // 14
    DWORD dwTotalMSRFOnTime; // 15
    DWORD dwNoTagErr; // 16
    DWORD dwEraseFailErr; // 17
    DWORD dwProgFailErr; // 18
    DWORD dwTagLockErr; // 19
    DWORD dwKillFailErr; // 20
    DWORD dwHardwareErr; // 21
    DWORD dwDataSizeErr; // 22
    DWORD dwReadTime_10MS; // 23
    DWORD dwReadAttempts; // 24
    DWORD dwReadSuccess; // 25
    DWORD dwProgramAttempts; // 26
    DWORD dwProgramSuccess; // 27
    DWORD dwEraseAttempts; // 28
    DWORD dwEraseSuccess; // 29
    DWORD dwLockAttempts; // 30
    DWORD dwLockSuccess; // 31
    DWORD dwKillAttempts; // 32
    DWORD dwKillSuccess; // 33
    DWORD dwLockFailErr; // 34
    DWORD dwTagTypeReadTotal[RFID_TAG_TYPE_MAX]; // 35-54 (35+(RFID_TAG_TYPE_MAX - 1)
    DWORD dwTagUnderrunErrors; // 55
    DWORD dwDroppedTagEvents; //56
    DWORD dwReserved[256-56];
} RFID_STATS;

// Total of 256 DWORDS for stats(Fixed size total of 1K)
Macro Used to Initialize TYPE_TAG_EX Structure
A TYPE_TAG_EX variable must be initialized by this macro before it is used.

The macro syntax is: TYPE_TAG_EX1_INIT(TypeTag)

Parameter TypeTag is a TYPE_TAG_EX variable.

Macro used to initialize TYPE_MASK_EX structure
A TYPE_MASK_EX variable must be initialized by this macro before it is used.

The macro syntax is: TAG_MASK_EX1_INIT(TagMask)

Parameter TagMask is a TYPE_MASK_EX variable.

Macro used to initialize TAG_LIST_EX structure
A TAG_LIST_EX variable must be initialized by this macro before it is used.

The macro syntax is: TAG_LIST_EX1_INIT(TagList, MaxTags)

Parameter TagList is a TAG_LIST_EX variable.
Parameter MaxTags is either 0 or 200.
Introduction

Clients must initialize the RFID C API and obtain a reader handle before interacting with the devices. All subsequent interactions with the RFID device are based on this handle. For the PC host version of this C-API the client can manage multiple readers by opening a different handle for each reader. Each reader maintains a copy of the capability engine version and tag list. Each of the capability engine version can be operated independently.

The C API library needs to be initialized before any other functions can be called. This is required because the capabilities and functions need to be associated with a reader object instance to be functional. Every reader object instance is uniquely identified by a handle, which the API uses to interact with the corresponding RFID device.

For Windows XP PC version, the library supports multiple concurrent reader instances. The user can interact with all the readers simultaneously by referring to their handles. For the XR device version, only one instance is supported (the local reader).

The reader handle is initialized by RFID_Open(HANDLE *hReader) function. Upon success, the hReader holds a valid handle for the reader object instance. Any further reader operations require the handle as reference. The handle and the reader object instance must be released by calling the RFID_Close(HANDLE *hReader) function.

The standard procedure for using a reader is as following:

1. Initialize a handle
2. Configure the IP address and TCP port for the reader
3. Open connection to the reader
4. Carry out other operations
5. Close the reader connection
6. Release the handle by closing the reader
The following code segment demonstrates this procedure:

```c
/// Opens and establishes connection to a reader
/// @param[in] tszNewIPAddress The IP address of the reader
/// @param[in] wPort The TCP port of the reader
HANDLE OpenReader(TCHAR tszNewIPAddress[32], WORD wPort)
{
    HANDLE hReader = 0;
    // open the API object for the reader
    if (RFID_Open(&hReader) == RFID_SUCCESS)
    {
        // configure TCP/IP parameter for the reader
        if (ConfigureTCPIP(hReader, tszNewIPAddress, wPort))
        {
            // connect to the reader
            if (RFID_OpenReader(hReader, 0) == RFID_SUCCESS)
            {
                printf("Found reader\n");
                // configure other parameters of the reader
                ConfigureReader(hReader);
            }
        }
        RFID_CAPS Caps;
        if (RFID_GetCaps(hReader, &Caps) == RFID_SUCCESS)
        {
            printf("RFIDAPI Version %S Firmware Version %S\n",
                   Caps.szAPIVersionString, Caps.szFirmwareVersion);
            printf("   Serial # Info: %S\n", Caps.szSerialInfo);
        }
    }
    else
    {
        printf("Failed to open RFIDAPI\n");
    }
    return hReader;
}
```
**RFID_Open**

**Description**
Call this function first to initialize the RFID API and obtain the reader handle. It returns RFID_SUCCESS on successful opening.

This function opens the RFID communications port.

To open an RFID device over a TCP socket, first configure the capabilities `RFID_DEVCAP_IP_PORT` and `RFID_DEVCAP_IP_NAME`. See `RFID_DEVCAP_IP_PORT` on page A-6 and `RFID_DEVCAP_IP_NAME` on page A-5.

**Function Prototype**

```c
DWORD RFID_Open(*phReader);
DWORD RFID_OpenReader(HANDLE hReader, int port);
```

**Parameters**

- `*phReader` Pointer to the handle that receives the RFID device handle on RFID_SUCCESS.
- `hReader` Handle to open the RFID device.
- `Port` Desired port. Any valid port number, if the port number is 0, the value for `RFID_DEVCAP_IP_PORT` will be used.

**Return Values**

If successful:

- RFID_SUCCESS

Upon failure, it returns one of the following values:

- RFID_INVALID_HANDLE
- RFID_CANNOT_ALLOC_MEM
- RFID_ENGINE_BUSY
- RFID_PORT_OPEN_ERROR

**Example 1**

```c
HANDLE hRFIDReader;
RFID_Open(&hRFIDReader);
```
Example 2

ConfigureTCP/IP (HANDLE hReader, TCHAR *pszIPAddress, DWORD dwPort)

BOOL FindAndOpenReader(void)
DWORD dwItems;
DWORD dwStatus;

dwItems = 1;

dwStatus = RFID_SetCapCurrValue (hReader, RFID_DEVCAP_IP_PORT, &dwItems, sizeof (dwPort), &dwPort);
if (dwStatus == RFID_SUCCESS)
{
    bSuccess = TRUE;
}
else
{
    printf ("RFID_DEVCAP_IP_NAME Set Cap Error %S\n", RFID_GetCommandStatusText (hReader, dwStatus));
}
else
{
    printf ("RFID_DEVCAP_IP_NAME Set Cap Error %S\n", RFID_GetCommandStatusText (hReader, dwStatus));
}
return (bSuccess);
BOOL FindAndOpenReader(void)
{
    BOOL bSuccess = FALSE;

    // Open, and Initialize API
    if(RFID_Open(&hReader)== RFID_SUCCESS)
    {
        // Now try to find the first RFID reader.
        // It will look for a device on the local com ports...
        if(ConfigureTCPIP (hReader, TEXTC "$192.168.0.101" ), 3000))
            // A reader was found, now try and open using the port number returned.
            if(RFID_OpenReader(hReader, RFIDFindInfo.nPortNumber) == RFID_SUCCESS)
                {
                    bSuccess = TRUE;
                }

    RFID_FindClose(hReader, hFind);
    
    return(bSuccess);
    
}
RFID_Close

Description
The RFID_Close function releases handle. It returns RFID_SUCCESS on successful closing.

This function closes the communication port.

Function Prototype

DWORD RFID_Close(HANDLE *phReader);
DWORD WINAPI RFID_CloseReader(HANDLE hReader);

Parameters

*phReader Pointer to the handle that receives the RFID device handle on RFID_SUCCESS.
hReader Handle for the open RFID device.

Return Values
If successful:
RFID_SUCCESS

Upon failure:
RFID_ENGINE_BUSY
RFID_INVALID_HANDLE
RFID_PORT_NOT_OPEN

Example
HANDLE hRFIDReader;
If (RFID_Open(&HRFIDReader) == RFID_SUCCESS)
{
    RFID_Close(&hRFIDReader);
}

Device Capabilities Discovery

Introduction ................................................................. 4-3
RFID_GetCapList ............................................................. 4-4
RFID_GetCapInfo ............................................................ 4-5
RFID_GetCapCurrValue ...................................................... 4-7
RFID_GetCapDfltValue ....................................................... 4-10
RFID_SetCapCurrValue ...................................................... 4-11
RFID_SetCapDflts ............................................................ 4-12
RFID_SetCapDfltValue ....................................................... 4-13
Introduction

This chapter describes how to discover and change the API capabilities. These capabilities allow configuring the API and retrieving the current API settings.

The API supports a limited set of capabilities, including attenuation settings, supported tag types, and reading modes. To get the set a current or default values, query the API for a list of supported capabilities.

The following functions allow performing operations on the API capabilities.
RFID_GetCapList

Description
This function returns a list of capability IDs. Each ID represents a unique capability provided by the API.

Function Prototype
   DWORD RFID_GetCapList(HANDLE hReader, DWORD *pdwCapId, DWORD *pNumCaps);

Parameters
   pdwCapId       Pointer to an array of DWORDs for placing the CapIDs.
   pNumCaps       Pointer to DWORD that contains the number of CapIDs in the list.

Note
If pdwCapId or *pNumCaps is 0, the function sets *pNumCaps to the number of DWORDs required to hold the entire list. The user can then allocate sufficient storage for retrieving the capability list on the next call to RFID_GetCapList.

Return Values
If successful:
   RFID_SUCCESS

Upon failure, it returns one of the following values:
   RFID_PARAM_ERROR
   RFID_INVALID_HANDLE
   RFID_BUFFER_TOO_SMALL
RFID_GetCapInfo

Description
This function returns information regarding the capability with ID \( dwCapId \). Information includes unique name, data type, and contained type.

Function Prototype
\[
\text{DWORD RFID_GetCapInfo(HANDLEhReader, DWORD dwCapId, LPCAPINFO lpCapInfo);}\]

Parameters
- \( dwCapId \): Cap Id on which to return information.
- \( lpCapInfo \): Pointer to CapInfo structure to be filled with information.

See CAPINFO structure.

Return Values
If successful:
- RFID_SUCCESS

Upon failure, it returns one of the following values:
- RFID_INVALID_HANDLE
- RFID_PARAM_ERROR
- RFID_CAPNOTSUPPORTED
Example

Get the List of Capabilities and Display the Human Readable Names of Each

```c
void DisplayCapabilities(void)
{
    DWORD *pCaps;
    DWORD dwNumCaps;
    CAPINFO CapInfo;

    // ask for the number of capabilities supported
    if(RFID_GetCapList(hReader, 0, &dwNumCaps) == RFID_SUCCESS)
    {
        // allocate a buffer for them
        pCaps = malloc(dwNumCaps * sizeof(DWORD));
        if(pCaps)
        {
            // now we know how many there are, and we have a valid buffer to hold them
            // Ask for the cap list
            if(RFID_GetCapList(hReader, pCaps, &dwNumCaps) == RFID_SUCCESS)
            {
                DWORD dwCount;

                // now go through the list, and print the human readable name
                for(dwCount = 0; dwCount < dwNumCaps; dwCount++)
                {
                    if(RFID_GetCapInfo(hReader, *(pCaps + dwCount), &CapInfo) == RFID_SUCCESS)
                    {
                        printf("Cap Name: %S", CapInfo.tszName);
                    }
                }
                free(pCaps);
            }
        }
    }
}
```
**RFID_GetCapCurrValue**

*Description*
This function returns the current value of the desired capability.

*Function Prototype*

```c
DWORD RFID_GetCapCurrValue(HANDLE hReader, DWORD dwCapId, DWORD *pdwNumItems, DWORD dwValueBufSize, LPVOID pvValueBuf);
```

*Parameters*

- `dwCapId` ID of capability to get.
- `pdwNumItems` Pointer to DWORD indicating the maximum number of items to get.
- `dwValueBufSize` Size in bytes pointed to by `pvValueBuf`.
- `pvValueBuf` Block of memory receiving the capability value(s).

*Notes*
A capability can be a single entry, or multiple entries long. Each entry must be 8, 16, or 32 bits (1, 2, or 4 bytes) long. The parameters imply the size of each entry, which the API calculates as follows:

```
EntrySize = dwValueBufSize / *pdwNumItems
```

The API converts smaller entries into larger entries, but not vice-versa. For example, the tag lock code is an array of bytes. The caller can provide this function with an array of DWORDs. The API places each lock code byte into a single DWORD.

*Return Values*
If successful:

- `RFID_SUCCESS`

Upon failure, it returns one of the following values:

- `RFID_INVALID_HANDLE`
- `RFID_PARAM_ERROR`
- `RFID_CAPNOTSUPPORTED`
- `RFID_BUFFER_TO_SMALL`
Example

Get the List of Supported Tag Types, Enable One, and Disable the Rest
This example uses `RFID_GetCapCurrValue()` and `RFID_SetCapCurrValue()`.

```c
BOOL ConfigureTagTypes(void)
{
    BOOL bSuccess = FALSE;
    BYTE bSupportedTypes[5];
    BYTE bEnabledTypes[5];
    DWORD dwNumItems;
    DWORD dwSize;

    // init enabled type list to 0's. Indicates no types are yet enabled.
    ZeroMemory(bEnabledTypes, sizeof(bEnabledTypes));

    // Set the number of items in our buffer for API to use.
    // The number of items in the array is the same as the size of the array
    // if each item is a byte. This calculation ensures that dwNumItems is
    // correct regardless of the size of each item
    dwNumItems = sizeof(bSupportedTypes) / sizeof(bSupportedTypes[0]);

    // Buffer size is in bytes
    dwSize = sizeof(bSupportedTypes);

    // Now ask for the list of supported tag types
    if(RFID_GetCapCurrValue(hReader, RFID_TAGCAP_SUPPORTED_TYPES, &dwNumItems, dwSize, bSupportedTypes) == RFID_SUCCESS)
    {
        // The API updates dwNumItems to reflect number of items returned.

        // One byte is returned for each possible Tag type
        // The defines for tag type are indexes into the tag type list
        // If a tag type’s byte is 0, the type is not supported

        // Now we can see what tag types are supported, and turn some off, and some on

        // If class 0 is supported
        if(bSupportedTypes[RFID_TAG_TYPE_EPC_CLASS0])
        {
        }
    }
}
```
bEnabledTypes[RFID_TAG_TYPE_EPC_CLASS0] = 1; // Enable Class 0
}

// If class 1 is supported
if(bSupportedTypes[RFID_TAG_TYPE_EPC_CLASS1])
{
    bEnabledTypes[RFID_TAG_TYPE_EPC_CLASS1] = 0; // Disable Class 1
}

// If class 1 Gen 2 is supported
if(bSupportedTypes[RFID_TAG_TYPE_EPC_CLASSG2])
{
    bEnabledTypes[RFID_TAG_TYPE_EPC_CLASSG2] = 0; // Disable Class 1 Gen 2
}

// dwNumItems has been updated by GetCapCurrValue to be equal to the number of tag
// type entries. Use the same number for this call.

// Make sure the buffer size is based on the number of items returned when
// getting the supported tag type list.
dwSize = dwNumItems * sizeof(bEnabledTypes[0]);

// Now set the enabled types
if(RFID_SetCapCurrValue(hReader, RFID_TAGCAP_ENABLED_TYPES, &dwNumItems, dwSize, bEnabledTypes) ==
    RFID_SUCCESS)
{
    bSuccess = TRUE;
}

return(bSuccess);
RFID_GetCapDfltValue

**Description**
This function returns the default value for the specified capability ID.

**Function Prototypes**
```
DWORD RFID_GetCapDfltValue(HANDLE hReader, DWORD dwCapId, DWORD *pdwNumItems, DWORD dwValueBufSize, LPVOID pvValueBuf);
```

**Notes**
This function is similar to `RFID_GetCapCurrValue` except it applies to the default values. See `RFID_GetCapCurrValue` on page 4-7 for a function description.

**Parameters**
- **dwCapId**: ID of capability to get.
- **pdwNumItems**: Pointer to DWORD indicating the maximum number of items to get.
- **dwValueBufSize**: Size in bytes pointed to by `pvValueBuf`.
- **pvValueBuf**: Block of memory receiving the capability value(s).

**Return Values**
If successful:
```
RFID_SUCCESS
```
Upon failure, it returns one of the following values:
```
RFID_INVALID_HANDLE
RFID_PARAM_ERROR
RFID_CAPNOTSUPPORTED
RFID_BUFFER_TO_SMALL
```
**RFID_SetCapCurrValue**

**Description**
This function sets the current value of the specified capability ID.

**Function Prototypes**

```
DWORD RFID_SetCapCurrValue(HANDLE hReader, DWORD dwCapId, DWORD *pdwNumItems, DWORD dwValueBufSize, LPVOID pvValueBuf);
```

**Notes**
This function is similar to `RFID_SetCapCurrValue` except it sets caps. See `RFID_GetCapCurrValue` on page 4-7 for a function description.

**Parameters**

- `dwCapId` ID of the capability to get.
- `pdwNumItems` Pointer to DWORD indicating the maximum number of items to get.
- `dwValueBufSize` Size in bytes pointed to by `pvValueBuf`.
- `pvValueBuf` Block of memory receiving the capability value(s).

**Return Values**
If successful:

- `RFID_SUCCESS`

Upon failure, it returns one of the following values:

- `RFID_INVALID_HANDLE`
- `RFID_PARAM_ERROR`
- `RFID_CAPNOTSUPPORTED`
- `RFID_CAPREADONLY`
- `RFID_BUFFER_TO_SMALL`
**RFID_SetCapDflts**

*Description*
This function sets the capabilities values to their default values. The default values for some capabilities differ based on device type.

*Function Prototypes*
```
DWORD RFID_SetCapDflts(HANDLE hReader);
```

*Return Values*
If successful:
```
RFID_SUCCESS
```
**RFID_SetCapDfltValue**

**Description**
This function sets the current capability value to the default values. The default values for some capabilities differ based on device type.

**Function Prototypes**
```
DWORD RFID_SetCapDfltValue(HANDLE hReader, DWORD dwCapId);
```

**Parameters**
- `dwCapId`: ID of capability to get.

**Return Values**
If successful:
- `RFID_SUCCESS`

Upon failure:
- `RFID_INVALID_HANDLE`
Reading and Writing Tags

Introduction ................................................................. 5-3
RFID_GetTagID ............................................................. 5-4
RFID_GetTagMask ......................................................... 5-5
RFID_SetTagMask .......................................................... 5-6
RFID_KillTag ................................................................. 5-7
RFID_LockTag ............................................................... 5-9
RFID_ProgramTags ......................................................... 5-11
RFID_EraseTag ............................................................. 5-13
RFID_ReadTagInventory .................................................. 5-14
GPIO support ................................................................. 5-15
Extended version of data structures and functions .................. 5-16
Error in tag reading ........................................................ 5-18
Introduction
This chapter describes tag reading and writing functions.
**RFID_GetTagID**

**Description**
This function attempts to read a single RFID tag into pTag.

**Function Prototype**
```c
DWORD RFID_GetTagID(HANDLE hReader, TYPE_TAG *pTag);
```

**Parameters**
- **HReader**
  Open RFID device handle.
- **pTag**
  Pointer to a TYPE_TAG structure which is filled with the tag if the function returns RFID_SUCCESS.

**Return Values**
If successful:
- RFID_SUCCESS (GetTagID returned a TAG)

Upon failure, it returns one of the following values:
- RFID_ENGINE_BUSY
- RFID_INVALID_HANDLE
- RFID_PORT_NOT_OPEN
- RFID_PARAM_ERROR
- RFID_CMD_NOTAG

**Notes**
The API currently supports two reading modes, On Demand and Autonomous. The Autonomous reading mode is implemented in the API software, however it may not be available using the reader capabilities.

- In On Demand mode (the default mode), this function initiates tag reading and blocks until the reading process completes. If any tags are read, a single tag returns.
- In Autonomous mode, the RFID API is always reading tags. The API saves all tag reads in an internal queue. Calling this function removes a single tag from the queue. If no tags are available, the function returns immediately. If desired, configure the API to generate an event if any tags are queued. By doing so, the application can wait for the event, then use this function to get the tag. For details, see the capability `RFID_READCAP_EVENTNAME` on page A-12.

To change modes, see the capability `RFID_READCAP_READMODE` on page A-21.

**Example**
```c
HANDLE hReader;
TYPE_TAG Tag;
if(RFID_GetTagID(hReader, &Tag) == RFID_SUCCESS)
{
    // Display tag ID bytes
    // Tag.TagID[0] is first byte
    // Tag.dataLength is number of bytes in tag
}
```
**RFID_GetTagMask**

*Description*
This function gets the current TagMask used during RFID tag reading functions. The mask limits the list of tags to read based on the current mask. The tag mask is limited to 64 bits; future releases will support 96-bit and higher masks.

The TagMask is reset whenever the RFID module is reset or powered off.

*Function Prototype*

```c
DWORD RFID_GetTagMask(HANDLE hReader, TAG_MASK *pTagMask);
```

*Parameters*
- `hReader` Handle to open the RFID device.
- `*pTagMask` Pointer to a tag mask structure for getting the tag mask. Tag Mask contains the following:
  - `nBitLength` Number of bits to match.
  - `nStartBit` The bit on which to start matching. The first 16 bits of the tag (0-15) are reserved.
  - `pMaskBits` Array of bytes (bits) representing the bits to match.

*Return Values*
If successful:

- `RFID_SUCCESS`

Upon failure, it returns one of the following values:

- `RFID_ENGINE_BUSY`
- `RFID_INVALID_HANDLE`
- `RFID_PORT_NOT_OPEN`
- `RFID_PARAM_ERROR`

*Example*

```c
HANDLE hReader;
TAG_MASK TagMask;
BYTE *pTagBitMask = "x10\x20\x30\x40\x50\x60\x70\x80";
TagMask.cBitLen = 8*8;    // Match all 64 bits of the TAG ID
TagMask.cBitStartOffset = 16; // Start matching after the first 16 bits. // These bits are reserved by the tag.

// set the tag mask bits
memcpy(TagMask.cTagMask, pTagBitMask, 8);
RFID_SetTagMask(hReader, &TagMask);
```
RFID_SetTagMask

Description
The RFID_SetTagMask function sets the current TagMask used during RFID tag reading functions. The mask limits the list of tags to read based on the current mask.

Function Prototype
DWORD RFID_SetTagMask(HANDLE hReader, TAG_MASK *pTagMask);

Parameters
hReader Handle to open RFID device.
*pTagMask Pointer to a tag mask structure for setting the tag mask. Tag Mask contains the following:
cBitLen Number of bits to match.
cBitStartOffset The starting bit offset is not currently supported.
cTagMask Array of bytes (bits) representing bits to match.

Return Values
If successful:
RFID_SUCCESS
Upon failure:
RFID_XXXX error code

Example
HANDLE hReader;
TAG_MASK TagMask;
BYTE *pTagBitMask = "x10x20x30x40x50x60x70x80";
TagMask.cBitLen = 8*8; // Match all 64 bits of the TAG ID
TagMask.cBitStartOffset = 0;
// set the tag mask bits
memcpy(TagMask.cTagMask, pTagBitMask, 8);
RFID_SetTagMask(hReader, &TagMask);
**RFID\_KillTag**

**Description**
This function kills a tag, making it programmable again. Use this command to reset a locked tag, similar to unlocking and erasing a tag.

**Function Prototype**

```c
DWORD RFID\_KillTag(HANDLE hReader, unsigned char *pTagID,
                     unsigned char cTagLength, unsigned char cTagVerifyCount,
                     unsigned char cKillAttempts, unsigned char cTagKillCode);
```

**Parameters**

- `HReader` Handle to open the RFID reader.
- `*pTagID` TagID of the tag to kill.
- `CTagLength` Number of bytes in the Tag ID killed.
- `CTagVerifyCount` Number of times to verify the tag before locking.
- `CKillAttempts` Number of times to attempt killing the tag.
- `CTagKillCode` Pass code used to kill the tag, set using `RFID\_LockTag`. The `cTagKillCode` has an 8-bit value from 0 to 255.

**Return Values**

If successful:

- `RFID\_SUCCESS`

Upon failure, it returns one of the following values:

- `RFID\_ENGINE\_BUSY`
- `RFID\_INVALID\_HANDLE`
- `RFID\_PORT\_NOT\_OPEN`
- `RFID\_PARAM\_ERROR`

**Notes**

By default, this function targets Class 1 tags. Set the type of tag on which to perform this operation by setting the capability `RFID\_WRITECAP\_TAGTYPE` on page A-23.

The `cTagKillCode` only supports one-byte kill codes. Class 0 tags require a three-byte code. To set a kill code with more than one byte, set the capability `RFID\_TAGCAP\_LOCKCODE` on page A-24 to the proper kill code, and pass `cTagKillCode` to 0.

For Gen2 tags, use `RFID\_TAGCAP\_G2KILL\_PASSWORD`. 


**Example**

HANDLE hRFIDReader;
DWORD dwStatus;
unsigned char cTagLength;
unsigned char cTagVerifyCount;
unsigned char cLockAttempts;
unsigned char cTagKillCode;
unsigned char *pTagID;
pTagID = "\x01\x02\x03\x04\x05\x06\x07\x08";  
cTagLength = 8;
cTagVerifyCount = 3;
cKillAttempts = 3;
cTagKillCode = 255;
dwStatus = RFID_KillTag(hReader, pTagID, cTagLength, cTagVerifyCount, cKillAttempts, cTagKillCode);
if(dwStatus == RFID_SUCCESS)
{
    // Tag Killed
};
RFID_LockTag

Description
This function locks a tag to prevent changes.

Function Prototype
DWORD RFID_LockTag(HANDLE hReader, unsigned char cTagLength,
                     unsigned char cTagVerifyCount, unsigned char cLockAttempts,
                     unsigned char cTagKillCode);

Parameters
- HReader: Handle to open the RFID reader.
- CTagLength: Number of bytes in the Tag ID to lock.
- CTagVerifyCount: Number of times to verify the tag before locking.
- CLockAttempts: Number of times to try locking the tag.
- CTagKillCode: Pass code used to lock the tag. Use this pass code when killing the tag.
  CTagKillCode has an 8-bit value from 0 to 255.

Return Values
If successful:
  RFID_SUCCESS
Upon failure, it returns one of the following values:
  RFID_ENGINE_BUSY
  RFID_INVALID_HANDLE
  RFID_PORT_NOT_OPEN
  RFID_PARAM_ERROR
  RFID_CMD_NOTAG
  RFID_CMD_TAGLOCK
  RFID_CMD_LOCKFAIL

Notes
By default, this function targets Class 1 tags. Set the type of tag on which to perform this operation by setting the capability
RFID_WRITECAP_TAGTYPE on page A-23.

The CTagKillCode only supports one-byte kill codes. Class 0 tags require a three-byte code. To set a kill code with more than one byte,
set the capability RFID_TAGCAP_LOCKCODE on page A-24 to the proper kill code, and pass CTagKillCode to 0.

For Gen2 tags, use RFID_TAGCAP_G2KILL_PASSWORD.
Example

HANDLE hRFIDReader;
DWORD dwStatus;
unsigned char cTagLength;
unsigned char cTagVerifyCount;
unsigned char cLockAttempts;
unsigned char cTagKillCode;
cTagLength = 8;
cTagVerifyCount = 3;
cLockAttempts = 3;
cTagKillCode = 417;
dwStatus = RFID_LockTag(hReader, cTagLength, cTagVerifyCount, cLockAttempts, cTagKillCode);
if(dwStatus == RFID_SUCCESS)
{
    // Tag Locked
}


RFID_ProgramTags

Description
This function programs a tag with a new TagID provided the tag is not locked.

Function Prototype
DWORD RFID_ProgramTags(HANDLE hReader, unsigned char *pTagID,
                        unsigned char cTagLength, unsigned char cTagVerifyCount,
                        unsigned char cEraseAttempts, unsigned char cProgramAttempts);

Parameters
hReader Handle to open the RFID reader.
pTagID Bytes representing the new tag ID.
cTagLength Number of bytes in the Tag ID.
cTagVerifyCount Number of times to verify the tag after programming (performed by the reader module).
cEraseAttempts Number of times to attempt erasing the tag before programming.
cProgramAttempts Number of times to try programming the tag.

Return Values
If successful:
RFID_SUCCESS

Upon failure, it returns one of the following values:
RFID_ENGINE_BUSY
RFID_INVALID_HANDLE
RFID_PORT_NOT_OPEN
RFID_PARAM_ERROR
RFID_CMD_NOTAG
RFID_CMD_ERASEFAIL
RFID_CMD_PROGFAIL
RFID_CMD_TAGLOCK

Note
By default, this function targets Class 1 tags. Set the type of tag on which to perform this operation by setting the capability RFID_WRITECAP_TAGTYPE on page A-23.
Example

HANDLE hRFIDReader;
DWORD dwStatus;
unsigned char *pTagID;
unsigned char cTagLength;
unsigned char cTagVerifyCount;
unsigned char cEraseAttempts;
unsigned char cProgramAttempts;
pTagID = "\x01\x02\x03\x04\x05\x06\x07\x08";
cTagLength = 8;
cTagVerifyCount = 3;
cEraseAttempts = 3;
cProgramAttempts = 3;
dwStatus = RFID_ProgramTags(hRFIDReader, pTagID, cTagLength, cTagVerifyCount, cEraseAttempts, cProgramAttempts);
if(dwStatus == RFID_SUCCESS)
{
    // Tag programmed
}

RFID_EraseTag

Description
This function erases a programmed tag, making it unreadable, provided the tag is not locked.

Function Prototype
DWORD RFID_EraseTag(unsigned char cTagVerifyCount, unsigned char cEraseAttempts)

Parameters
- cTagVerifyCount Number of times to verify a successful.
- cEraseAttempts Number of times to attempt erasing the tag.

Return Values
If successful:
RFID_SUCCESS
Upon failure, it returns one of the following values:
RFID_CMD_NOTAG
RFID_CMD_ERASEFAIL
RFID_CMD_TAGLOCK

Notes
By default, this function targets Class 1 tags. Set the type of tag on which to perform this operation by setting the capability RFID_WRITECAP_TAGTYPE on page A-23.

Gen2 does not support erase.
RFID_ReadTagInventory

**Description**
This function supports reading multiple tags. This command attempts to read a tag for a period of time based on programmer-provided parameters. This command also maintains a list of all tags read, the number of times each was read, the last time each was read, and its Tag ID. The caller provides a TAG_LIST structure that the API updates. To reset the tag list, set the clear inventory parameter. If a tag mask was set using RFID_SetTagMask, the inventory does not read tags that do not fit the mask.

**Function Prototype**

```
DWORD WINAPI RFID_ReadTagInventory(HANDLE hReader, TAG_LIST *pTagList, BOOL bClearInventory)
```

**Parameters**

- **hReader**: Handle to open the RFID device.
- **pTagList**: TAG_LIST structure the API fills with results on the RFID_SUCCESS return value. This includes total tags read, new tags read during this inventory request, and a list of tags. The caller can provide a valid tag list which the API updates, or initialize the tag list by setting the bClearInventory flag. This structure is limited to 200 tags. The nNewTags variable contains the number of new tags detected during the current API call. The first new tag is located at index nTotalTags-nNewTags.
- **bClearInventory**: TRUE - Clear out the current list of tags read, and reset the read count.
  FALSE - Continue accumulating tags and keep track of how many times each tag was read.

**Return Values**

If successful:

- RFID_SUCCESS

Upon failure, it returns one of the following values:

- RFID_ENGINE_BUSY
- RFID_INVALID_HANDLE
- RFID_PORT_NOT_OPEN
- RFID_PARAM_ERROR

If the tag list is full of valid tag read results but tags read after the list filled were lost:

- RFID_MAX_TAGS_EXCEEDED

**Notes**

Provide a properly initialized taglist before calling this function. In general, zero the structure the first time used (or call this function with the bClearInventory parameter).

By default, the tag list can hold up to 200 tags. If desired, set the dwMaxTags field in the taglist. The API uses this number to override the maximum number of tags stored in the list. Allocate a TAG_LIST structure of the appropriate size when changing dwMaxTags.

This function returns RFID_MAX_TAGS_EXCEEDED if the tag list is full, and a queued tag cannot be added to the list. Clear the list, then call the function again.

The API currently supports two reading modes, On Demand and Autonomous. The Autonomous reading mode is implemented in the API software, however it may not be available using the reader capabilities.

- In On Demand mode (the default mode), this function initiates tag reading and blocks until the reading process completes. If any tags are read, they are added to the provided tag list, and returned to the caller.
In Autonomous mode, the RFID API is always reading tags. The API saves all tag reads in an internal queue. Calling this function removes all currently queued tags from the queue, and adds these to the provided taglist. If no tags are available, the function returns immediately. If desired, configure the API to generate an event if any tags are queued. By doing so, the application can wait for the event, then use this function to get all queued tags. For details, see the capability RFID_READCAP_EVENTNAME on page A-12.

To change modes, see the capability RFID_READCAP_READMODE on page A-21.

**Example**

```c
if(RFID_ReadTagInventory(hReader, &TagList, TRUE) == RFID_SUCCESS)
{
    DWORD dwI = TagList.dwTotalTags;
    for(dwI = 0; dwI < TagList.dwTotalTags; dwI++)
    {
        // print tag id TagList.Tags[dwI].tagID
    }
};
```

**GPIO Support**

The following functions provide support for GPIO on the XR Series readers.

DWORD WINAPI RFID_GetGPIValue(HANDLE hReader, BYTE *pbValue);

This function gets the current value for the input pins. There are 6 TTL input pins on the XR reader, which will fill in the low 6 bits of the *pbValue variable.

DWORD WINAPI RFID_SetGPOValue(HANDLE hReader, BYTE GPOMask, BYTE bValue);

The readers have 6 TTL output pins. This function can set the value of particular output pins. The GPOMask has the bit mask for selecting pins. The low 6 bits match those output pins respectively. Value of 1 in the mask means that pin is selected. The Value variable tells the desired pin value for the specified pins.

DWORD WINAPI RFID_SetGPIDetection(HANDLE hReader, HANDLE hNotifyEvent, BYTE bMask, BYTE bInterval);

The readers support general purpose input detection and event notification. This function lets user enable this detection and set up the notification event.

The user needs to create a system event for notification, and pass the handle of this event in through the hNotifyEvent parameter. The parameter bMask provides a bit mask specifying which bit the user is interested in and would like the detection to listen to its change. This bit mask has the similar definition as in the previous function. The parameter bInterval specifies the detection interval the reader will use. This value is in terms of 100ms. 0 means the default value as 500ms.
Extended Version of Data Structures and Functions

This version of the XR Series C-API also provides a set of enhanced version of the API function and data structures to provide richer information and functionalities. It is recommended that new applications should use this new extended version of structures and functions.

As a convention, these new structures and functions all have the similar names as their original counter parts with the "EX" extension.

Structures

Following table shows the new extended data structures and their respective original parts:

<table>
<thead>
<tr>
<th>New structure</th>
<th>Original structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE_TAG_EX</td>
<td>TYPE_TAG</td>
</tr>
<tr>
<td>TAG_LIST_EX</td>
<td>TAG_LIST</td>
</tr>
<tr>
<td>TAG_MASK_EX</td>
<td>TAG_MASK</td>
</tr>
</tbody>
</table>

For the structure of TAG_LIST_EX, it should be initialized by the macro of:

```c
TAG_LIST_STRUCTURE_INIT(TagList, MaxTags, TagListStructureFormat)
```

The MaxTags parameter tells the maximum number of tags this structure should hold. 0 means the default value of 200 tags. The TagListStructureFormat parameter should always use the predefined constant value of `TAG_LIST_STRUCTURE_FORMAT_EX1`.

For example:

```c
TAG_LIST_EX NewTagList;
TAG_LIST_STRUCTURE_INIT(  
    NewTagList,  // The new tag list structure
    0,  // 0 means default value of 200.
    TAG_LIST_STRUCTURE_FORMAT_EX1);
```

// Then the NewTagList variable is ready to be used in other function calls
**Functions**

Following shows the pairs of the original functions and their extended counter parts. The only difference is that all extended version of functions use the extended version of data structures. Other than that, their usage is the same.

```
DWORD WINAPI RFID_GetTagID(HANDLE hReader, TYPE_TAG *pTag);
DWORD WINAPI RFID_GetTagIDEX(HANDLE hReader, TYPE_TAG_EX *pTag);

DWORD WINAPI RFID_SetTagMask(HANDLE hReader, TAG_MASK *pTagMask);
DWORD WINAPI RFID_SetTagMaskEX(HANDLE hReader, TAG_MASK_EX *pTagMask);

DWORD WINAPI RFID_GetTagMask(HANDLE hReader, TAG_MASK *pTagMask);
DWORD WINAPI RFID_GetTagMaskEX(HANDLE hReader, TAG_MASK_EX *pTagMask);

DWORD WINAPI RFID_ReadTagInventory(HANDLE hReader, TAG_LIST *pTagList, BOOL bClearInventory);
DWORD WINAPI RFID_ReadTagInventoryEX(HANDLE hReader, void *pTagList, BOOL bClearInventory);

DWORD WINAPI RFID_TagFilterAdd(HANDLE hReader, TCHAR *pFilterName, DWORD dwBitsToMatch, DWORD dwBitMatchOffset, unsigned char *pMatchBits, DWORD dwOptionMask, void (*pMatchCallBack)(TYPE_TAG *pTag, TCHAR *pFilterName, DWORD dwBitsToMatch, DWORD dwBitMatchOffset, const unsigned char *pMatchBits, DWORD dwOptionMask));
DWORD WINAPI RFID_TagFilterAddEX(HANDLE hReader, TCHAR *pFilterName, DWORD dwBitsToMatch, DWORD dwBitMatchOffset, unsigned char *pMatchBits, DWORD dwOptionMask, void (*pMatchCallBack)(TYPE_TAG_EX *pTag, TCHAR *pFilterName, DWORD dwBitsToMatch, DWORD dwBitMatchOffset, const unsigned char *pMatchBits, DWORD dwOptionMask));
```
Error in Tag Reading

When it is reading with the functions of RFID_GetTagIDEX(), RFID_GetTagIDEX(), RFID_ReadTagInventory() /
RFID_ReadTagInventoryEX(), the returned TAG_TYPE or TAG_TYPE_EX structure may contain errors that have occurred in the
background reading.

If the value of the status field of the structure is RFID_SUCCESS, then this structure holds valid tag information in the respective
fields. Otherwise, the status field tells the main error state; the wStatusDetail field holds device specific error information; and the
antennaNum field holds the antenna index where this error occurred; and all other fields are reserved and should not be used.

Code sample:

/// Processes the error from reader tag reads
/// @param[in] hReader The handle of the reader
/// @param[in] Tag The tag structure which holds the status information
/// @returns true if the tag structure is an error and is processed, otherwise false
bool ProcessErrorRead(HANDLE hReader, TYPE_TAG_EX *pTag)
{
    DWORD dwStatus = pTag->status;
    if (dwStatus == RFID_SUCCESS)
    {
        return false;
    }
    printf("RFID_GetTagID Error %S, detail: %2x, antenna: %2u\n",
           RFID_GetCommandStatusText(hReader, pTag->status),
           pTag->wStatusDetail, pTag->antennaNum);

    switch (dwStatus)
    {
    case RFID_CMD_UNKVAL:
        {
            switch (pTag->wStatusDetail)
            {
            case 0xF0:  // invalid command params
                // This antenna (Tag.antennaNum) may need to be removed from the
                // antenna sequence, otherwise it may keep generating the same
                // error
                break;
            } // invalid command params
        } //invalid command params
default:
    break;
}
break;

case RFID_CMD_HWERR:
{
    switch (pTag->wStatusDetail)
    {
    case 0xF3: // Antenna fault
        // This antenna (Tag.antennaNum) may need to be removed from the
        // antenna sequence, otherwise it may keep generating the same
        // error
        break;
    case 0xF4: // DSP Timeout
    case 0xF5: // DSP Error
    case 0xF6: // DSP Idle
    case 0xF7: // Zero Power (invalid RF power specified)
    default:
        break;
    }
}
break;

case RFID_PORT_OPEN_ERROR:
case RFID_PORT_WRITE_ERROR:
case RFID_COMMAND_TIMEOUT:
case RFID_PORT_NOT_OPEN:
case RFID_UNKNOWN_COMM_TYPE:
case RFID_CMD_UNKLEN: //0xF1: // insufficient data
case RFID_CMD_UNKCMD: //0xF2: // Command not supported
case RFID_UNKNOWN_ERROR: //0xFF:
default:

    // enter error handling, may need to stop reading …
    break;

return true;

}
Introduction ......................................................... 6-3
RFID_GetCommandStatusText ........................................ 6-4
RFID_GetStats .......................................................... 6-5
Introduction
This chapter includes commands for retrieving API status codes and module statistics.
RFID_GetCommandStatusText

Description
This function returns RFID API status codes as English text strings. The hRFIDReader handle is not required, but used for consistency with the rest of this API.

Function Prototype
```c
const TCHAR * RFID_GetCommandStatusText(int stat);
```

Parameters
- `stat` Status code returned by an RFID API function call.

Return Values
Returns a human readable string representing the RFID API's integer error code.

Example
```c
const TCHAR *szStatus;
stat = RFID_Reboot(hRFIDReader);
szStatus = RFID_GetCommandStatusText(hRFIDReader, stat);
```
RFID_GetStats

Description
This function fills the RFID_STATS structure with RFID module statistics. Statistics include Total Characters Transmitted, Total Characters Received, and Total Tags Read.

Function Prototype
DWORD RFID_GetStats(HANDLE hReader, RFID_STATS *pStats);

Parameters

HReader Open RFID device handle.
PStats Pointer to RFID_STATS structure that receives the latest stats.

Return Values
If successful:
RFID_SUCCESS
Upon failure, it returns one of the following values:
RFID_ENGINE_BUSY
RFID_PARAM_ERROR
RFID_INVALID_HANDLE

Example
RFID_STATS RFID_Stats;
HANDLE hReader;
RFID_GetStats(hReader, &RFID_Stats);
RFID API Capabilities

Introduction ................................................................. A-3
RFID_INFICAP_SUPPORTEDCAPS ........................................ A-4
RFID_DEVCAP_IP_NAME .................................................. A-5
RFID_DEVCAP_IP_PORT .................................................. A-6
RFID_DEVCAP_ANTENNA_SEQUENCE .................................. A-8
RFID_READCAP_RF_ATTENUATION ..................................... A-10
RFID_READCAP_EVENTTAGPTR ......................................... A-11
RFID_READCAP_EVENTNAME ........................................... A-12
RFID_READCAP_EVENT_ALLTAGS ....................................... A-16
RFID_READCAP_METHOD ................................................ A-17
RFID_READCAP_OUTLOOP ............................................... A-18
RFID_READCAP_INLOOP ............................................... A-20
RFID_READCAP_READMODE ............................................. A-21
RFID_WRITECAP_RF_ATTENUATION .................................... A-22
RFID_WRITECAP_TAGTYPE ............................................... A-23
RFID_TAGCAP_LOCKCODE ............................................... A-24
RFID_TAGCAP_SUPPORTED_TYPES ..................................... A-25
RFID_TAGCAP_ENABLED_TYPES ......................................... A-26
RFID_TAGCAP_C0_SINGULATION_FIELD ............................... A-26
RFID_WRITECAP_ANTENNA ............................................. A-27
RFID_TAGCAP_G2_KILL_PASSWORD ...................................... A-27
RFID_TAGCAP_G2_ACCESS_PASSWORD ................................. A-28
RFID_DEVCAP_VALID_ANTENNA_LIST ................................ A-28
RFID_DEVCAP_ANTENNA_GROUP ....................................... A-29
RFID_TAGCAP_MASK_BITS ................................................................. A-30
RFID_TAGCAP_MASK_NUMBITS_TOMATCH ........................................... A-30
RFID_TAGCAP_MASK_STARTPOS ......................................................... A-31
RFID_TAGCAP_MASK_STARTPOS ......................................................... A-31
Introduction
This appendix provides the description of the definition and usage of RFID API capabilities.
RFID_INFCAP_SUPPORTEDCAPS

Description
This capability can hold a complete list of all capability IDs.

Capability Type
CAPUINT32: Each item is a 32-bit value, 4 bytes.

Maximum Number of Items
This capability can hold 40 items

Container Type
CONT_ARRAY: This capability contains an array of value

Container Size
The size is 40 words, (160 bytes)

Default Value
This capability uses 160 bytes memory space, 40 items * 4 bytes
0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07
0x08 0x09 0x0A 0x0B 0x0C 0x0D 0x0E 0x0F
0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17
0x18 0x19 0x1A 0x1B 0x1C 0x1D 0x1E 0x1F
0x20 0x21 0x22 0x23 0x24 0x25 0x26 0x00
RFID DEVCAP_IP_NAME
This is the IP address to use when calling RFID_FindFirst.

Description
This capability represents the IP address of the reader (Unicode string)

Capability Type
CAPUINT16: Each item is a 16-bit value, 2 bytes.

Maximum Number of Item
This capability can hold 14 items

Container Type
CONT_ARRAY: This capability's container contains an array.

Container Size
The size is 28 bytes

Default Value
"127.0.0.1"
**RFID_DEVCAP_IP_PORT**

This is the IP port number to use when calling `RFID_FindFirst`.

**Description**

This capability represents the TCP port of the reader.

**Capability Type**

CAPUINT32: Each item is a 32-bit value, 4 bytes.

**Maximum Number of Item**

This capability can hold 1 item.

**Container Type**

CONT_ONEVALUE: This capability's container contains one value.

**Container Size**

The size is 4 bytes.

**Default Value**

0x0BB8 (3000)

**Example**

By setting `RFID_DEVCAP_IP_NAME` and `RFID_DEVCAP_IP_PORT`, the API searches for an RFID device on the provided TCP information when `RFID_FindFirst` or `RFID_FindNext` are called. If a device is not found at the supplied address:port, the API searches local com ports for a valid RFID device.

```c
void ConfigureSocket(void)
{
    DWORD dwNumItems;
    DWORD dwSize;

    szIPaddress = L"localhost";
    dwPort = 3000; // default XR port number

    dwNumItems = 1;
    // Start by setting IP port number
    if(RFID_SetCapCurrValue(hReader, RFID_DEVCAP_IP_PORT, &dwNumItems, sizeof(dwPort), &dwPort) == RFID_SUCCESS)
    {
        // Use local host as the host name. (try and find reader locally)
        dwNumItems = wcslen(szIPaddress) + 1; // + 1 for null
        dwSize = dwNumItems * sizeof(szIPaddress[0]);

        if(RFID_SetCapCurrValue(hReader, RFID_DEVCAP_IP_NAME, &dwNumItems, dwSize, szIPaddress) == RFID_SUCCESS)
        {
```

**RFID_DEVCAP_ANTENNA_SEQUENCE**

Use this to set the antenna sequence when reading tags. The ReadTag functions issue one set of read commands for each antenna specified in the sequence.

**Description**

Use this to set the antenna sequence when reading tags. The ReadTag functions issue one set of read commands for each antenna specified in the sequence.

**Capability Type**

CAPUINT8: Each item is a 8-bit value, 1 byte.

**Maximum Number of Item**

This capability can hold 16 items

**Container Type**

CONT_ARRAY: This capability's container contains an array.

**Container Size**

The size is 28 bytes

**Default Value**

0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07
0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF 0xFF

**Note**

0xFF means the end of the antenna sequence.

**Example**

// Create a small list of antennas to use when reading.
// Read functions will cycle through the antenna sequence
BOOL ConfigureAntennas(void)
{
    BOOL bSuccess = FALSE;
    DWORD dwNumItems;
    char cAntennaSequence[2];
    cAntennaSequence[0] = 1;// First antenna
    cAntennaSequence[1] = 0;// Second antenna
    dwNumItems = 2; // antenna sequence has only 2 entries
    // Now tell the API to use this list of antennas
    if(RFID_SetCapCurrValue(hReader, RFID_DEVCAP_ANTENNA_SEQUENCE,
    &dwNumItems, sizeof(cAntennaSequence),
    cAntennaSequence) == RFID_SUCCESS)
    {
        bSuccess = TRUE;
    }
    return(bSuccess);
}
Example

// Create a small list of antenna’s to use when reading.
// Read functions will cycle through the antenna sequence
BOOL ConfigureAntennas(void)
{
    BOOL bSuccess = FALSE;
    DWORD dwNumItems;
    char cAntennaSequence[2];

    cAntennaSequence[0] = 1; // First antenna
    cAntennaSequence[1] = 0; // Second antenna

    dwNumItems = 2; // antenna sequence has only 2 entries

    // Now tell the API to use this list of antennas
    if(RFID_SetCapCurrValue(hReader, RFID_DEVCAP_ANTENNA_SEQUENCE, &dwNumItems, sizeof(cAntennaSequence),
                               cAntennaSequence) == RFID_SUCCESS)
    {
        bSuccess = TRUE;
    }
    return(bSuccess);
}
RFID_READCAP_RF_ATTENUATION

Description
This function represents the attenuation level to use for all tag read operations.

Description
This capability represents the attenuation level to use for tag read operations.

Capability Type
CAPUINT8: Each item is a 8-bit value, 1 byte.

Maximum Number of Items
This capability can hold 1 item

Container Type
CONT_ONEVALUE: This capability's container contains one value

Container Size
The size is 1 byte

Default Value
0

Example
void ConfigureAttenuation(void)
{
    unsigned char cReadAttenuation;
    unsigned char cWriteAttenuation;
    DWORD dwNumItems;

    // Attenuation of 0 is full power, 255 is no power...
    cReadAttenuation = 25;
    dwNumItems = 1;
    // Now tell API to set read attenuation
    RFID_SetCapCurrValue(hReader, RFID_READCAP_RF_ATTENUATION, &dwNumItems, sizeof(cReadAttenuation), &cReadAttenuation);

    // set to low power for tag writes...
    cWriteAttenuation = 220;
    dwNumItems = 1;
    // Now tell API to set read attenuation
    RFID_SetCapCurrValue(hReader, RFID_WRITECAP_RF_ATTENUATION, &dwNumItems, sizeof(cWriteAttenuation), &cWriteAttenuation);
}

RFID_API_Capabilities

RFID_READCAP_EVENTTAGPTR

Description
This capability contains the address of a TYPE_TAG structure owned by the application. To use this feature, also set the RFID_READCAP_EVENTNAME capability.

This capability operates as follows:

In On Demand mode, when a tag is read, the API checks to see if the RFID_READCAP_EVENTNAME event is reset. If so, the API places tag data into the user supplied TYPE_TAG structure, and sets the event RFID_READCAP_EVENTNAME. If the event is already set, the API queues the tag for later.

When the application receives the event, it processes the tag data and resets the event. The API then re-starts this process. (The API takes the next queued tag in the TYPE_TAG structure, and resets the event.)
**RFID_READCAP_EVENTNAME**

**Description**
The name of the windows event object user provided for signaling tag read

**Capability Type**
CAPUINT16: Each item is a 16-bit value, 2 byte (Unicode string).

**Maximum Number of Items**
This capability can hold 1 item

**Container Type**
CONT_ARRAY: This capability contains an array of Unicode characters

**Container Size**
The size is 64 Unicode characters, 128 bytes

**Default Value**
NULL string

**Notes**
The eventname is stored as a Unicode null terminated string. Include the terminating 16-bit NULL character. By implementing this capability, an application can efficiently wait for tags using a window event.

The API currently supports two reading modes, On Demand and Autonomous. The Autonomous reading mode is implemented in the API software, however it may not be available using the reader capabilities.

- In On Demand mode, by default, this event is signaled when a new tag is added to the supplied taglist.
- In Autonomous mode, this event is signaled for all tags.

**Example**

**Autonomous Mode Tag Event Handling**

```c
DWORD WINAPI TagEventThread(LPVOID pvarg)
{
    TCHAR *szReadEvent = L"MyReadTagEvent";
    HANDLE hTagEvent;
    DWORD dwNumItems;
    DWORD dwSize;
    DWORD dwReadMode = RFID_READCAP_READMODE_AUTONOMOUS;

    // create a named event. Use this event for listening to tag read events from api
    hTagEvent = CreateEvent(NULL, TRUE, FALSE, szReadEvent);
    if(hTagEvent)
    {
```


// Register for tag read events using a named event
// include the null character
dwNumItems = wcslen(szReadEvent) + 1;
dwSize = dwNumItems * sizeof(szReadEvent[0]);

RFID_SetCapCurrValue(hReader, RFID_READCAP_EVENTNAME, &dwNumItems, dwSize, szReadEvent);

dwNumItems = 1;
// Tell API to configure autonomous read mode
// Once this is called, reading starts…
RFID_SetCapCurrValue(hReader, RFID_READCAP_READMODE, &dwNumItems, sizeof(dwReadMode), &dwReadMode);

// wait for api to signal that there are tags
// the API will keep event signalled as long as tags are available...
while(WaitForSingleObject(hTagEvent, INFINITE) == WAIT_OBJECT_0)
{
    // Ask API for tag…
    if(RFID_GetTagID(hReader, &Tag) == RFID_SUCCESS)
    {
        // process tag
    }
};

return(0);
**On Demand Tag Event Handling**

In On Demand mode, calls to `RFID_ReadTagInventory()` and `RFID_GetTagID()` block until the read process completes. Since the application thread is blocked, the application cannot see tags until the call completes. However, the developer can process read events while blocked by listening to events on another thread as follows:

```c
DWORD WINAPI OnDemandModeTagEventThread(LPVOID pvarg)
{
    DWORD dwTotalReads = 0;

    // wait for api to signal that there are tags
    // the API will keep event signalled as long as tags are available...
    while(WaitForSingleObject(hTagEvent, INFINITE) == WAIT_OBJECT_0)
    {
        // we read another tag...
        // By default, we only get these events when a new tag is added to the TagList.
        // When RFID_ReadTagInventory returns, dwTotalReads will be the same as TagList.dwNewTags.
        dwTotalReads++;
        // We can see tag, and process it now...

        // On demand mode requires we reset the event...
        // this way, API knows when to overwrite the tag structure we provided.
        ResetEvent(hTagEvent);
        // On Demand mode does not allow us to call tag read functions on tag event.
        // Main thread will get entire list when call completes

    }
    return(0);
}

void OnDemandEventTest(void)
{
    TCHAR *szReadEvent = L"MyReadTagEvent";
    DWORD dwNumltems;
    DWORD dwSize;
    HANDLE hThread;
    TYPE_TAG *pTag;

    // create a named event. Use this event for listening to tag read events from api
    hTagEvent = CreateEvent(NULL, TRUE, FALSE, szReadEvent);
    if(hTagEvent)
```
{ 
    // Register for tag read events using a named event
    // include the null character
    dwNumItems = wcslen(szReadEvent) + 1;
    dwSize = dwNumItems * sizeof(szReadEvent[0]);
    RFID_SetCapCurrValue(hReader, RFID_READCAP_EVENTNAME, &dwNumItems, dwSize, szReadEvent);

    // Give the API a place to store tag
    dwNumItems = 1;
    dwSize = sizeof(&Tag); // Size of pointer to a tag
    // pass the address of the tag pointer. The API will use the tag pointer to store read tags
    pTag = &Tag;
    RFID_SetCapCurrValue(hReader, RFID_READCAP_EVENTTAGPTR, &dwNumItems, dwSize, &pTag);

    // create the listening thread
    hThread = CreateThread(NULL, 64*1024, OnDemandModeTagEventThread, (LPVOID)hReader, 0, NULL);

    if(hThread)
    {
        // start the read using OnDemand Mode.
        // This call will block until the read completes it's inventory
        if(RFID_ReadTagInventory(hReader, &TagList, TRUE) == RFID_SUCCESS)
        {
            PrintTagList();
        };
    };
};
**RFID_READCAP_EVENT_ALLTAGS**

Set this to generate events (`RFID_READCAP_EVENTNAME`) on all tag reads while in On Demand mode. By default, the API signals only new tags (i.e., it does not signal duplicates).
**RFID_READCAP_METHOD**

*Description*
This function represents the tag reading method to use when performing the `RFID_ReadTagInventory` call.

*Capability Type*
CAPUINT8: Each item is a 8-bit value, 1 byte.

*Maximum Number of Item*
This capability can hold 1 item

*Container Type*
CONT_ONEWVALUE: This capability's container contains one value.

*Container Size*
The size is 1 byte

*Default Value*
0

*Note*
The API supports two read methods for the Class 1 Reader only:
- Method 0 performs RFID tag reading using a binary tree method.
- Method 1 performs RFID tag reading using a masked scroll technique.
RFID_READCAP_OUTLOOP

**Description**
This function represents the outer loop value used when performing the RFID_ReadTagInventory call.

**Capability Type**
CAPUINT8: Each item is a 8-bit value, 1 byte.

**Maximum Number of Item**
This capability can hold 1 item

**Container Type**
CONT_ONEVALUE: This capability’s container contains one value.

**Container Size**
The size is 1 byte

**Default Value**
5

**Note**
Calling RFID_ReadTagInventory() performs a tag inventory, which consists of one or more actual read attempts. The number of read attempts depends on a number of capabilities, including:

- RFID_READCAP_OUTLOOP
- RFID_READCAP_INLOOP
- RFID_TAGCAP_ENABLED_TYPES
- RFID_DEVCAP_ANTENNA_SEQUENCE

The following pseudo code illustrates the tag reading sequence for each read attempt:

```c
While RFID_READCAP_OUTLOOP
    While RFID_READCAP_INLOOP
        For each antenna in RFID_DEVCAP_ANTENNA_SEQUENCE
            For each RFID_TAGCAP_ENABLED_TYPES
                Read Tags
```
Example

```c
void ReadTagInventoryTest(void)
{
    DWORD dwNumItems = 1;
    BYTE bParam;

    ZeroMemory(&TagList, sizeof(TagList));

    // Set Outer loop to 3
    bParam = 3;
    dwNumItems = 1;
    RFID_SetCapCurrValue(hReader, RFID_READCAP_OUTLOOP, &dwNumItems, sizeof(bParam), &bParam);

    // Set inner loop to 5
    bParam = 5;
    dwNumItems = 1;
    RFID_SetCapCurrValue(hReader, RFID_READCAP_INLOOP, &dwNumItems, sizeof(bParam), &bParam);

    // Now do a tag inventory
    if(RFID_ReadTagInventory(hReader, &TagList, TRUE) == RFID_SUCCESS)
    {
        // print tag list
    };
};
```
RFID_READCAP_INLOOP

Description
This function represents the inner loop value used when performing the RFID_ReadTagInventory call.

Capability Type
CAPUINT8: Each item is a 8-bit value, 1 byte.

Maximum Number of Item
This capability can hold 1 item

Container Type
CONT_ONEVALUE: This capability's container contains one value.

Container Size
The size is 1 byte

Default Value
3

Note
The API uses this capability value only when calling RFID_ReadTagInventory with the innerLoop parameter set to zero.
RFID_READCAP_READMODE

Description
The API currently supports two reading modes, On Demand and Autonomous. The Autonomous reading mode is implemented in the API software, however it may not be available using the reader capabilities.

Use this capability to set the read mode to On Demand or Autonomous mode. The default mode is On Demand. This capability can have one of the following values:

- RFID_READCAP_READMODE_ONDEMAND
- RFID_READCAP_READMODE_AUTONOMOUS

Capability Type
CAPUINT32: Each item is a 32-bit value, 4 bytes.

Maximum Number of Item
This capability can hold 1 item

Container Type
CONT_ONEVALUE: This capability's container contains one value.

Container Size
The size is 4 byte

Default Value
RFID_READCAP_READMODE_ONDEMAND
**RFID_WRITECAP_RF_ATTENUATION**
This function represents the power level when writing to a tag.

*Description*
This capability represents the attenuation level to use for tag write operations.

*Capability Type*
CAPUINT8: Each item is a 8-bit value, 1 byte.

*Maximum Number of Items*
This capability can hold 1 item

*Container Type*
CONT_ONEVALUE: This capability's container contains one value

*Container Size*
The size is 1 byte

*Default Value*
0
**RFID_WRITECAP_TAGTYPE**
This is the tag type to use during a write operation.

*Description*
This capability represents the tag type tag write operations.

*Capability Type*
CAPUINT8: Each item is a 8-bit value, 1 byte.

*Maximum Number of Item*
This capability can hold 1 item

*Container Type*
CONT_ONEVALUE: This capability's container contains one value

*Container Size*
The size is 1 byte

*Default Value*
0
RFID_TAGCAP_LOCKCODE
This is the tag lock code to use for lock and killtag commands (used when setting lockcode to 0 when calling a lock/kill function).

Description
This capability represents the lock code for tag lock or kill operations. This capability only applies to Class 0 and Class 1 tags.

Capability Type
CAPUINT8: Each item is a 8-bit value, 1 byte.

Maximum Number of Item
This capability can hold 3 items

Container Type
CONT_ARRAY: This capability's container contains an array.

Container Size
The size is 3 bytes

Default Value
0xFF, 0xFF, 0xFF
RFID_TAGCAP_SUPPORTED_TYPES
This is a list of supported tag types.

Description
This capability represents the supported tag types for tag read operations.

Capability Type
CAPUINT8: Each item is a 8-bit value, 1 byte.

Maximum Number of Item
This capability can hold 5 items

Container Type
CONT_ARRAY: This capability's container contains an array.

Container Size
The size is 5 bytes

Default Value
0x00, 0x01, 0x00, 0x01, 0x01

- Value[RFID_TAG_TYPE_EPC_CLASS0] == 0x01, Support Class 0
- Value[RFID_TAG_TYPE_EPC_CLASS1] == 0x01, Support Class 1
- Value[RFID_TAG_TYPE_EPC_CLASSG2] == 0x01, Support Gen 2
RFID_TAGCAP_ENABLED_TYPES
This is a list of enabled tag types.

Description
This capability represents the enabled tag types for tag read operations.

Capability Type
CAPUINT8: Each item is a 8-bit value, 1 byte.

Maximum Number of Items
This capability can hold 5 items

Container Type
CONT_ARRAY: This capability's container contains an array.

Container Size
The size is 5 bytes

Default Value
0x00, 0x01, 0x00, 0x01, 0x01
- Value[RFDI_TAG_TYPE_EPC_CLASS0] == 0x01, Support Class 0
- Value[RFDI_TAG_TYPE_EPC_CLASS1] == 0x01, Support Class 1
- Value[RFDI_TAG_TYPE_EPC_CLASSG2] == 0x01, Support Gen 2

RFID_TAGCAP_C0_SINGULATION_FIELD
This is the type of class 0 simulation field to use when reading tags. (For multiprotocol handhelds only.)

RFID_TAGCAP_C0_SINGULATION_ID0
RFID_TAGCAP_C0_SINGULATION_ID1
RFID_TAGCAP_C0_SINGULATION_ID2
**RFID_WRITECAP_ANTENNA**

*Description*
Use this capability to set the antenna for tag write operation.

*Capability Type*
CAPUINT8: Each item is a 8-bit value, 1 byte.

*Maximum Number of Item*
This capability can hold 1 item

*Container Type*
CONT_ONEVALUE: This capability’s container contains one value.

*Container Size*
The size is 1 byte

*Default Value*
0

**RFID_TAGCAP_G2_KILL_PASSWORD**

*Description*
This capability represents the kill password for Gen 2 kill operation. This capability only applies Gen 2 tags.

*Capability Type*
CAPUINT8: Each item is a 32-bit value, 4 bytes.

*Maximum Number of Items*
This capability can hold 1 item

*Container Type*
CONT_ONEVALUE: This capability’s container contains one value.

*Container Size*
The size is 4 bytes

*Default Value*
0x00000000
RFID_TAGCAP_G2_ACCESS_PASSWORD

**Description**
This capability represents the access password for Gen 2 access operations. This capability only applies Gen 2 tags.

**Capability Type**
CAPUINT8: Each item is a 32-bit value, 4 bytes.

**Maximum Number of Item**
This capability can hold 1 item

**Container Type**
CONT_ONEVALUE: This capability’s container contains one value.

**Container Size**
The size is 4 bytes

**Default Value**
0x00000000

RFID_DEVCAP_VALID_ANTENNA_LIST

**Description**
Use this to set the antenna sequence when reading tags. The ReadTag functions issue one set of read commands for each antenna specified in the sequence.

**Capability Type**
CAPUINT8: Each item is a 8-bit value, 1 byte.

**Maximum Number of Item**
This capability can hold 8 items

**Container Type**
CONT_ARRAY: This capability's container contains an array.

**Container Size**
The size is 8 bytes

**Default Value**
0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07

**Note**
For XR480, the valid antenna list has 8 items in the current value.
For XR400, the valid antenna list has 4 items in the current value.
This can be used to differentiate these different reader type.
RFID_DEVCAP_ANTENNA_GROUP

Description
Use this to form logical groups among antennae when reading tags. The ReadTag functions issue one set of read commands for each group specified in this capability.

Capability Type
CAPUINT8: Each item is a 8-bit value, 1 byte.

Maximum Number of Item
This capability can hold 8 items

Container Type
CONT_ARRAY: This capability's container contains an array.

Container Size
The size is 8 bytes

Default Value
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

Note
For XR480, there can be up to 4 different logical groups defined for each reader.
For XR400, there can be up to 2 different logical groups defined for each reader.

The array index maps to the antenna index. Value 0 means that antenna in independent, does not belong to any group. Otherwise, put the group index (1~4) at the appropriate array item to specify which logical group that antenna should belong to. For example,

Value[8] = 0, 1, 0, 2, 0, 1, 0, 2

This means the antenna #0, #2, #4 and #6 are independent antenna. Antenna #1 and #5 belong to logical group #1. Antenna #3 and #7 belong to logical group #2.

By default, each antenna is independent.
RFID_TAGCAP_MASK_BITS

Description
Use this capability to set the tag mask bits.

Capability Type
CAPUINT8: Each item is a 8-bit value, 1 byte.

Maximum Number of Items
This capability can hold 64 items

Container Type
CONT_ARRAY: This capability's container contains an array.

Container Size
The size is 64 bytes

Default Value
0

RFID_TAGCAP_MASK_NUMBITS_TOMATCH

Description
Use this capability to set the number of bits to match in the tag mask bits.

Capability Type
CAPUINT32: Each item is a 32-bit value, 4 byte.

Maximum Number of Items
This capability can hold 1 item

Container Type
CONT_ONEVALUE: This capability's container contains one value.

Container Size
The size is 4 bytes

Default Value
0
RFID_TAGCAP_MASK_STARTPOS

Description
Use this capability to set the start matching position in the tag mask bits.

Capability Type
CAPUINT32: Each item is a 32-bit value, 4 byte.

Maximum Number of Items
This capability can hold 1 item

Container Type
CONT_ONEVALUE: This capability's container contains one value.

Container Size
The size is 4 bytes

Default Value
0

RFID_TAGCAP_MASK_STARTPOS

Description
Use this capability to set the tag mask type.

Capability Type
CAPUINT8: Each item is a 8-bit value, 1 byte.

Maximum Number of Items
This capability can hold 1 item

Container Type
CONT_ONEVALUE: This capability's container contains one value.

Container Size
The size is 1 bytes

Default Value
0
**RFID_TAGCAP_MASK_LENGTH**

**Description**
Use this capability to set the tag mask length.

**Capability Type**
CAPUINT32: Each item is a 32-bit value, 4 bytes.

**Maximum Number of Items**
This capability can hold 1 item

**Container Type**
CONT_ONEVALUE: This capability's container contains one value.

**Container Size**
The size is 4 bytes

**Default Value**
0
Return Values

Introduction
Introduction
Table 2-1 lists and describes the return values that the C API uses when responding to function calls.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFID_SUCCESS</td>
<td>0</td>
<td>Function is successful.</td>
</tr>
<tr>
<td>RFID_PORT_OPEN_ERROR</td>
<td>1</td>
<td>Port is not open.</td>
</tr>
<tr>
<td>RFID_PARAM_ERROR</td>
<td>2</td>
<td>Invalid parameter provided.</td>
</tr>
<tr>
<td>RFID_CRC_ERROR</td>
<td>3</td>
<td>CRC receive error.</td>
</tr>
<tr>
<td>RFID_CANNOT_ALLOC_MEM</td>
<td>4</td>
<td>Cannot allocate memory.</td>
</tr>
<tr>
<td>RFID_PORT_WRITE_ERROR</td>
<td>5</td>
<td>Error writing to port.</td>
</tr>
<tr>
<td>RFID_PORT_READ_ERROR</td>
<td>6</td>
<td>Error reading from port.</td>
</tr>
<tr>
<td>RFID_UNKNOWN_ERROR</td>
<td>7</td>
<td>Unable to determine the error.</td>
</tr>
<tr>
<td>RFID_MAX_TAGS_EXCEEDED</td>
<td>8</td>
<td>Tag list is full, cannot store tag.</td>
</tr>
<tr>
<td>RFID_COMMAND_TIMEOUT</td>
<td>9</td>
<td>Command timed out, no response from RFID module.</td>
</tr>
<tr>
<td>RFID_UPLOADOK</td>
<td>10</td>
<td>Firmware packet successful (internal to API).</td>
</tr>
<tr>
<td>RFID_UPLOADEND</td>
<td>11</td>
<td>Firmware upload complete (internal to API).</td>
</tr>
<tr>
<td>RFID_ENGINE_BUSY</td>
<td>12</td>
<td>Cannot perform function, module or port is busy.</td>
</tr>
<tr>
<td>RFID_PORT_NOT_OPEN</td>
<td>13</td>
<td>Port is not open.</td>
</tr>
<tr>
<td>RFID_UNKNOWN_COMM_TYPE</td>
<td>14</td>
<td>RFID module returned an unknown error code.</td>
</tr>
<tr>
<td>RFID_BUFFER_TOO_SMALL</td>
<td>15</td>
<td>Assigned memory buffer is too small.</td>
</tr>
<tr>
<td>RFID_FIND_COMPLETE</td>
<td>16</td>
<td>Attempt to find RFID device is complete.</td>
</tr>
<tr>
<td>RFID_INVALID_HANDLE</td>
<td>17</td>
<td>Invalid device handle was provided.</td>
</tr>
<tr>
<td>RFID_NO_DEVICE_POWER</td>
<td>18</td>
<td>Function cannot proceed without RFID_SetDevicePower TRUE.</td>
</tr>
<tr>
<td>RFID_INVALID_FLASHFILE</td>
<td>19</td>
<td>Flash file provided is invalid.</td>
</tr>
<tr>
<td>RFID_CAPNOTSUPPORTED</td>
<td>20</td>
<td>Specified capability is not supported.</td>
</tr>
<tr>
<td>RFID_CAPREADONLY</td>
<td>21</td>
<td>Specified capability is read only.</td>
</tr>
<tr>
<td>RFID_TAG_DETECTED</td>
<td>22</td>
<td>An RFID tag was detected.</td>
</tr>
<tr>
<td>RFID_TAGTYPE_DISABLED</td>
<td>23</td>
<td>The tag type specified is disabled.</td>
</tr>
<tr>
<td>RFID_TAGTYPE_NOSUPPORT</td>
<td>24</td>
<td>The tag type specified is not supported.</td>
</tr>
<tr>
<td>RFID_CMD_UNKLEN</td>
<td>0x81</td>
<td>Invalid RFID module command parameter length.</td>
</tr>
<tr>
<td>RFID_CMD_UNKVAL</td>
<td>0x82</td>
<td>Invalid RFID module command value used.</td>
</tr>
<tr>
<td>RFID_CMD_UNKCMD</td>
<td>0x83</td>
<td>Invalid RFID module command used.</td>
</tr>
<tr>
<td>RFID_CMD_UNKTAGCMD</td>
<td>0x84</td>
<td>Invalid RFID module tag command used.</td>
</tr>
<tr>
<td>RFID_CMD_OVERERR</td>
<td>0x85</td>
<td>Data overflow error during the last command.</td>
</tr>
<tr>
<td>RFID_CMD_NOTAG</td>
<td>0x86</td>
<td>No tag found, command cannot proceed.</td>
</tr>
<tr>
<td>RFID_CMD_ERASEFAIL</td>
<td>0x87</td>
<td>Failure to erase the tag.</td>
</tr>
</tbody>
</table>
### Table 2-1. RFID C API Return Values (Continued)

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RFID_CMD_PROGFAIL</td>
<td>0x8B</td>
<td>Failure to program the tag.</td>
</tr>
<tr>
<td>RFID_CMD_TAGLOCK</td>
<td>0x89</td>
<td>Tag is locked, command cannot proceed.</td>
</tr>
<tr>
<td>RFID_CMD_KILLFAIL</td>
<td>0x8A</td>
<td>Failure to kill the tag.</td>
</tr>
<tr>
<td>RFID_CMD_LOCKFAIL</td>
<td>0x8B</td>
<td>Failure to lock the tag.</td>
</tr>
<tr>
<td>RFID_CMD_DATASIZE</td>
<td>0x8C</td>
<td>Command data size error.</td>
</tr>
<tr>
<td>RFID_CMD_HWERR</td>
<td>0x8D</td>
<td>Hardware error, failure to find a valid antenna.</td>
</tr>
<tr>
<td>RFID_CMD_LISTFULL</td>
<td>0x8E</td>
<td>RFID module’s internal tag list is full.</td>
</tr>
<tr>
<td>RFID_CMD_UPLOADERR</td>
<td>0x8F</td>
<td>Failure to upload firmware.</td>
</tr>
<tr>
<td>RFID_CMD_UPLOADINVALID</td>
<td>0x90</td>
<td>Invalid firmware upload buffer.</td>
</tr>
<tr>
<td>RFID_CMD_UPLOADCRC</td>
<td>0x91</td>
<td>CRC error in firmware upload buffer.</td>
</tr>
</tbody>
</table>
Introduction

This sample application can be built both in Visual Studio 2005 for PC and eVC++ 4.0 for XR400/480 device version.

#pragma once
#define WIN32_LEAN_AND_MEAN // Exclude rarely-used stuff from Windows headers

#include <stdio.h>
#include <tchar.h>
#include <windows.h>
#include <stdlib.h>
#include "rfiddefs.h"
#include "rfidcapi.h"
#include "rfidcapsengine.h"

// converts a byte array into a hex null terminated string
void MakeHexString(unsigned char *pBytes, int nBytes, char *pOutBuf)
{
    int i;
    char *psz = pOutBuf;

    for(i = 0; i < nBytes; i++)
    {
        psz += sprintf(psz, "%2.2X", (int)*pBytes);
        pBytes++;
    }
};

int MyMakeHexBuffer(char *pBytes, int nBytes, unsigned char *pOutBuf)
{
    int nBytesConverted = 0;
    char *pBuf = pBytes;
    unsigned char *pOutPtr = pOutBuf;
    unsigned int temp;

    for (INT j=0;j<nBytes;j++)
    {
        if (sscanf(pBuf, "%2X", &temp) == 1)
        {
            pBytes = pBytes + 2;
            nBytesConverted++;
        }
        else
        {
            return -1;
        }
    }

    return nBytesConverted;
}
*pOutPtr++ = (unsigned char)temp;
pBuf += 2;
nBytesConverted++;
);
}
return(nBytesConverted);
};

void DisplayCapabilities(HANDLE hReader)
{
    DWORD *pCaps;
    DWORD dwNumCaps;
    CAPINFO CapInfo;
    // ask for the number of capabilities supported
    if(RFID_GetCapList(hReader, 0, &dwNumCaps) == RFID_SUCCESS)
    {
        // allocate a buffer for them
        pCaps = (DWORD *)malloc(dwNumCaps * sizeof(DWORD));
        if(pCaps)
        {
            // now we know how many there are, and we have a valid buffer to hold them
            // Ask for the cap list
            if(RFID_GetCapList(hReader, pCaps, &dwNumCaps) == RFID_SUCCESS)
            {
                DWORD dwCount;
                // now go through the list, and print the human readable name
                for(dwCount = 0; dwCount < dwNumCaps; dwCount++)
                {
                    if(RFID_GetCapInfo(hReader, *(pCaps + dwCount), &CapInfo) ==
                        RFID_SUCCESS)
                    {
                        TCHAR szBuffer[512];
                        swprintf(szBuffer, TEXT("Cap[%d] Name: %s
"), dwCount,
                            (CapInfo.tszName));
                        wprintf(szBuffer);
                    }
                }
            }
        }
    }
}
free(pCaps);
};
};

/// Removes an antenna from a reader's antenna sequence
/// @param[in] hReader The handle of the reader
/// @param[in] AntennaIndex The 0 based index of the antenna
void RemoveAntennaFromSequence(HANDLE hReader, BYTE AntennaIndex)
{
    BYTE NewSequence[RFID_ANTENNA_SEQUENCE_LENGTH];
    BYTE CapSequence[RFID_ANTENNA_SEQUENCE_LENGTH];
    memset(NewSequence, 0, sizeof(NewSequence));
    memset(CapSequence, 0, sizeof(CapSequence));

    DWORD dwNumItems = sizeof(CapSequence);

    DWORD dwStatus = RFID_GetCapCurrValue(hReader, RFID_DEVCAP_ANTENNA_SEQUENCE,
                                                &dwNumItems, dwNumItems, CapSequence);

    int index = 0;
    if (dwStatus == RFID_SUCCESS)
    {
        for (int i=0; i<sizeof(NewSequence); i++)
        {
            if (CapSequence[i] != AntennaIndex)
            {
                NewSequence[index++] = CapSequence[i];
            }
            if (CapSequence[i] == 0xFF)
            {
                break;
            }
        }
        dwNumItems = index;
        RFID_SetCapCurrValue(hReader, RFID_DEVCAP_ANTENNA_SEQUENCE, &dwNumItems,
                                             dwNumItems, NewSequence);
    }
}
bool ProcessError(HANDLE hReader, DWORD dwStatus, WORD wStatusDetail, BYTE antennaNum)
{
    if (dwStatus == RFID_SUCCESS)
    {
        return false;
    }

    printf("RFID_GetTagID Error %S, detail: %2x, antenna: %2u\n", 
           RFID_GetCommandStatusText(0, dwStatus),
           wStatusDetail, antennaNum);

    switch (dwStatus)
    {
    case RFID_CMD_UNKVAL:
    {
        switch (wStatusDetail)
        {
        case 0xF0:// invalid command params
            //RemoveAntennaFromSequence(hReader, antennaNum);
            break;
        default:
            break;
        }
    }
    break;

    case RFID_CMD_HWERR:
    {
        switch (wStatusDetail)
        {
        case 0xF3:// Antenna fault
            //RemoveAntennaFromSequence(hReader, antennaNum);
            break;
        case 0xF4:// DSP Timeout
        case 0xF5:// DSP Error
        case 0xF6:// DSP Idle
        case 0xF7:// Zero Power (invalid RF power specified)
        default:
            break;
        }
    }
    break;
    }
break;

case RFID_PORT_OPEN_ERROR:
case RFID_PORT_WRITE_ERROR:
case RFID_COMMAND_TIMEOUT:
case RFID_PORT_NOT_OPEN:
case RFID_UNKNOWN_COMM_TYPE:
#if 0
    ToggleReadMode(hReader, false);
    // clean up the queue
    do {
        dwStatus = RFID_GetTagIDEX(hReader, &Tag);
    } while (dwStatus != RFID_CMD_NOTAG);
#endif //0
    break;

    #if 0
    case RFID_CMD_UNKLEN://0xF1:// insufficient data
    case RFID_CMD_UNKCMD://0xF2:// Command not supported
    case RFID_UNKNOWN_ERROR://0xFF:
    default:
        break;
    }
#endif //0

Sleep(0);
//Sleep(100);
return true;

} // Processes the error from a reader tag reads
/// @param[in] hReaderThe handle of the reader
/// @param[in] TagThe tag structure which holds the status information
bool ProcessErrorRead(HANDLE hReader, TYPE_TAG &Tag)
{
    return ProcessError(hReader, Tag.dwOperationStatus, Tag.wStatusDetail, Tag.antennaNum);
}

/// Processes the error from a reader’s tag reads
/// @param[in] hReaderThe handle of the reader
/// @param[in] Tag The tag structure which holds the status information
bool ProcessErrorReadEx(HANDLE hReader, TYPE_TAG_EX &Tag)
{
    return ProcessError(hReader, Tag.dwOperationStatus, Tag.wStatusDetail, Tag.antennaNum);
}

/// To detect if the reader is XR480 series, which supports 8 ports
/// @param[in] hReader the handle of the reader
/// @return true if it is a XR480, false for XR400
bool IsXR480(HANDLE hReader)
{
    // buffer to hold the capability value
    BYTE bValueBuffer[MAX_NUMBER_ANTENNA];
    // the size of the buffer holding the value of this capability
    DWORD dwBufferSize = sizeof(bValueBuffer);
    // number of items for this capability
    DWORD dwNumItems = dwBufferSize/sizeof(BYTE);

    // retrieve the capability value for currently enabled tag types from the API
    if(RFID_GetCapCurrValue(hReader,
        RFID_DEV_CAP_VALID_ANTENNA_LIST,
        &dwNumItems,
        dwBufferSize,
        bValueBuffer) == RFID_SUCCESS)
    {
        if (dwNumItems == 8)
        {
            for (unsigned int i=0; i<dwNumItems; i++)
            {
                if (bValueBuffer[i] == 0xFF)
                {
                    return false;
                }
            }
            return true;
        }
        return false;
    }
}
// Prints a tag to the stdout
// @param[in] hReader The handle of the reader
// @param[in] pTag The pointer to a tag structure
void PrintTagEx(HANDLE hReader, TYPE_TAG_EX *pTag)
{
    char szTagID[64];
    char *szProtocol;

    if (ProcessErrorReadEx(hReader, *pTag))
    {
        return;
    }

    MakeHexString(pTag->tagID, pTag->dataLength, szTagID);

    if(pTag->dwType == RFID_TAG_TYPE_EPC_CLASS0 || pTag->dwType ==
        RFID_TAG_TYPE_EPC_CLASS0PLUS)
    {
        szProtocol = "C0";
    }
    else if(pTag->dwType == RFID_TAG_TYPE_EPC_CLASS1)
    {
        szProtocol = "C1";
    }
    else if(pTag->dwType == RFID_TAG_TYPE_EPC_CLASSG2)
    {
        szProtocol = "G2";
    }
    else
    {
        szProtocol = "   ";
    }

    printf("  Tag: %25.25s Type: %s Ant: %2.2u Time: %2.2u:%2.2u:%2.2u.%3.3u\n",
            szTagID,
            szProtocol,
            pTag->antennaNum,
            pTag->lastSeen.wHour,
            pTag->lastSeen.wMinute,
            pTag->lastSeen.wSecond,
void PrintTagList(HANDLE hReader, TAG_LIST *pTagList)
{
    DWORD dwI = pTagList->dwTotalTags;
    char *szProtocol;
    char szTagID[32];

    printf("%s\n", szTagID);

    for(dwI = 0; dwI < pTagList->dwTotalTags; dwI++)
    {

        // convert tag id to ascii text string
        MakeHexString(pTagList->Tags[dwI].tagID, pTagList->Tags[dwI].dataLength, szTagID);

        if(pTagList->Tags[dwI].dwType == RFID_TAG_TYPE_EPC_CLASS0)
        {
            szProtocol = "C0";
        }
        else if(pTagList->Tags[dwI].dwType == RFID_TAG_TYPE_EPC_CLASS1)
        {
            szProtocol = "C1";
        }
        else if(pTagList->Tags[dwI].dwType == RFID_TAG_TYPE_EPC_CLASSG2)
        {
            szProtocol = "G2";
        }
        else
        {
            szProtocol = "  ";
        }

        printf("Tag %3.3u Reads: %3.3u Type: %s Ant: %2.2u:%s\n", dwI+1, pTagList->Tags[dwI].readCount,
            szProtocol, pTagList->Tags[dwI].antennaNum, szTagID);
    }
}
DWORD indexEnd = pTagList->dwTotalTags + pTagList->dwErrorTags;
for( dwI < indexEnd; dwI++ )
{
    ProcessErrorRead(hReader, pTagList->Tags[dwI]);
};
);

/// Prints a tag list to the stdout
/// @param[in] hReader The handle of the reader
/// @param[in] pTagList The pointer to a tag list structure
void PrintTagListEx(HANDLE hReader, TAG_LIST_EX *pTagList)
{
    DWORD dwI = 0;
    char *szProtocol;
    char szTagID[32];

    printf("n");

    for(dwI = 0; dwI < pTagList->dwTotalTags; dwI++)
    {
        if (ProcessErrorReadEx(hReader, pTagList->Tags[dwI]))
        {
            continue;
        }

        // convert tag id to ascii text string
        MakeHexString(pTagList->Tags[dwI].tagID, pTagList->Tags[dwI].dataLength, szTagID);

        if(pTagList->Tags[dwI].dwType == RFID_TAG_TYPE_EPC_CLASS0)
        {
            szProtocol = "C0";
        }
        else if(pTagList->Tags[dwI].dwType == RFID_TAG_TYPE_EPC_CLASS1)
        {
            szProtocol = "C1";
        }
        else if(pTagList->Tags[dwI].dwType == RFID_TAG_TYPE_EPC_CLASSG2)
```c
{
    szProtocol = "G2";
}
else
{
    szProtocol = " ";
};

printf("Tag %3.3u Reads: %3.3u Type: %s Ant: %2.2u:%s\n", dwI+1, pTagList->Tags[dwI].readCount,
        szProtocol, pTagList->Tags[dwI].antennaNum, szTagID);
}

DWORD indexEnd = pTagList->dwTotalTags + pTagList->dwErrorTags;
for(; dwI < indexEnd; dwI++)
{
    ProcessErrorReadEx(hReader, pTagList->Tags[dwI]);
};

/// Set reader capabilities to default values
void SetDefaultCapabilities(HANDLE hReader)
{
    // after the reader has been opened and hReader has been initialized
    // reset all capability to their default values
    RFID_SetCapDflts(hReader);
}

/// Set RF power attenuation for reading tag
/// @param[in] NewValue The new read power attenuation value, 0 means maximum power!
/// @return RFID_SUCCESS for success, otherwise error code for reason
DWORD SetReadPowerAttenuation(HANDLE hReader, BYTE NewValue)
{
    // buffer to hold the capability value
    BYTE bValueBuffer = NewValue; // 0 --> MAXIMUM power!
    // the size of the buffer holding the value of this capability
    DWORD dwBufferSize = sizeof(bValueBuffer);
    // number of items for this capability
    DWORD dwNumItems = dwBufferSize/sizeof(BYTE);
```
return RFID_SetCapCurrValue(hReader,
    RFID_READCAP_RF_ATTENUATION,
    &dwNumItems,
    dwBufferSize,
    &bValueBuffer);
}

/// Set RF power attenuation for programming tag
/// @param[in] hReader The handle of the reader
/// @param[in] NewValue The new write power attenuation value, 0 means maximum power!
/// @return RFID_SUCCESS for success, otherwise error code for reason
DWORD SetWritePowerAttenuation(HANDLE hReader, BYTE NewValue)
{
    // buffer to hold the capability value
    BYTE bValueBuffer = NewValue; // 0 --> MAXIMUM power!
    // the size of the buffer holding the value of this capability
    DWORD dwBufferSize = sizeof(bValueBuffer);
    // number of items for this capability
    DWORD dwNumItems = dwBufferSize/sizeof(BYTE);

    return RFID_SetCapCurrValue(hReader,
        RFID_WRITECAP_RF_ATTENUATION,
        &dwNumItems,
        dwBufferSize,
        &bValueBuffer);
}

/// This function demonstrates how to check what tag types are supported by a reader
/// @param[in] hReader The handle of the reader
void GetSupportedTagTypes(HANDLE hReader)
{
    // buffer to hold the supported tag type information
    BYTE bValueBuffer[RFID_TAG_TYPE_MAXINDEX];
    // the size of the buffer holding the value of this capability
    DWORD dwBufferSize = sizeof(bValueBuffer);
    // number of items for this capability
    DWORD dwNumItems = dwBufferSize/sizeof(BYTE);
// retrieve the capability value for supported tag types from the API
if(RFID_GetCapCurrValue(hReader, RFID_TAGCAP_SUPPORTED_TYPES, &dwNumItems, 
   dwBufferSize, bValueBuffer) == RFID_SUCCESS)
{
    if (bValueBuffer[RFID_TAG_TYPE_EPC_CLASS0]) {
        // EPC Gen 1 Class 0 is supported by the reader
    }

    if (bValueBuffer[RFID_TAG_TYPE_EPC_CLASS0PLUS]) {
        // EPC Gen 1 Class 0+ is supported by the reader
    }

    if (bValueBuffer[RFID_TAG_TYPE_EPC_CLASS1]) {
        // EPC Gen 1 Class 1 is supported by the reader
    }

    if (bValueBuffer[RFID_TAG_TYPE_EPC_CLASSG2]) {
        // EPC Gen 2 is supported by the reader
    }
}

/// Check if a tag type is supported by this reader
/// @param[in] TagTypeIndex The tag type that we want to check
/// @return true if it is supported, false if it is not supported
bool IsTagTypeSupported(HANDLE hReader, RFID_TAG_TYPE_INDEX TagTypeIndex)
{
    if ((TagTypeIndex > RFID_TAG_TYPE_OTHER) &&
        (TagTypeIndex <= RFID_TAG_TYPE_MAXINDEX))
    {
        // buffer to hold the supported tag type information
        BYTE bValueBuffer[RFID_TAG_TYPE_MAXINDEX];
        // the size of the buffer holding the value of this capability
        DWORD dwBufferSize = sizeof(bValueBuffer);
        // number of items for this capability
        DWORD dwNumItems = dwBufferSize/sizeof(BYTE);

        // retrieve the capability value for supported tag types from the API
        if(RFID_GetCapCurrValue(hReader, RFID_TAGCAP_SUPPORTED_TYPES, &dwNumItems, 
            dwBufferSize, bValueBuffer) == RFID_SUCCESS)
return (bValueBuffer[TagTypeIndex] != 0);
}
}
return false;
}

/// Enable tag types for reading
/// @param[in] TagTypeIndex The index of the tag type
/// @param[in] Enable True to enable the specified type, false to disable this type
/// @return RFID_SUCCESS for success,
/// RFID_TAGTYPE_NOSUPPORT if the type is not supported,
/// or other error code for reason
DWORD SetTagTypeForReading(HANDLE hReader, RFID_TAG_TYPE_INDEX TagTypeIndex, bool Enable)
{
    // buffer to hold the capability value
    BYTE bValueBuffer[RFID_TAG_TYPE_MAXINDEX];
    // the size of the buffer holding the value of this capability
    DWORD dwBufferSize = sizeof(bValueBuffer);
    // number of items for this capability
    DWORD dwNumItems = dwBufferSize/sizeof(BYTE);

    DWORD dwStatus = RFID_TAGTYPE_NOSUPPORT;

    if (!IsTagTypeSupported(hReader, TagTypeIndex))
    {
        return dwStatus;
    }

    // retrieve the capability value for currently enabled tag types from the API
    if((dwStatus = RFID_GetCapCurrValue(hReader, RFID_TAGCAP_ENABLED_TYPES, &dwNumItems, dwBufferSize, bValueBuffer)) == RFID_SUCCESS)
    {
        // enable/disable the specified tag type
        bValueBuffer[TagTypeIndex] = Enable;

        dwStatus = RFID_SetCapCurrValue(hReader, RFID_TAGCAP_ENABLED_TYPES, &dwNumItems, dwBufferSize, bValueBuffer);
    }
}
return dwStatus;
}

/// Enable tag types for writing
/// @param[in] TagTypeIndex The tag type for writing
/// @return RFID_SUCCESS for success,
/// RFID_TAGTYPE_NOSUPPORT if the type is not supported,
/// or other error code for reason
DWORD SetTagTypeForWriting(HANDLE hReader, RFID_TAG_TYPE_INDEX TagTypeIndex)
{
    // buffer to hold the capability value
    BYTE bValueBuffer = TagTypeIndex;
    // the size of the buffer holding the value of this capability
    DWORD dwBufferSize = sizeof(bValueBuffer);
    // number of items for this capability
    DWORD dwNumItems = 1;
    DWORD dwStatus = RFID_TAGTYPE_NOSUPPORT;

    if (!IsTagTypeSupported(hReader, TagTypeIndex))
    {
        return dwStatus;
    }

    dwStatus = RFID_SetCapCurrValue(hReader, RFID_WRITECAP_TAGTYPE, &dwNumItems,
                                     dwBufferSize, &bValueBuffer);
    return dwStatus;
}

/// Configure a reader work in auronomoue reading mode
/// @param[in] hReader The handle of the reader
void ConfigureAutonomousMode(HANDLE hReader)
{
    DWORD dwParamValue = RFID_READCAP_READMODE_AUTONOMOUS;
    DWORD dwNumItems = 1;

    // Now tell API to configure the read mode
    RFID_SetCapCurrValue(hReader, RFID_READCAP_READMODE, &dwNumItems, sizeof(dwParamValue),
                         &dwParamValue);

/// Configure a reader work in on demand reading mode
/// @param[in] hReader The handle of the reader
void ConfigureOnDemandMode(HANDLE hReader)
{
    DWORD dwParamValue = RFID_READCAP_READMODE_ONDEMAND;
    DWORD dwNumItems = 1;

    // Now tell API to configure the read mode
    RFID_SetCapCurrValue(hReader, RFID_READCAP_READMODE, &dwNumItems, sizeof(dwParamValue),
        &dwParamValue);
}

/// Read tags in the on demand reading mode
/// @param[in] hReader The handle of the reader
void ReadOnDemand(HANDLE hReader, TAG_LIST *pTagList) {
    printf("On demand read from Reader %8x\n", (WORD)hReader);
    ConfigureOnDemandMode(hReader);

    DWORD dwStatus = RFID_ReadTagInventory(hReader, pTagList, TRUE);
    if(dwStatus == RFID_SUCCESS)
    {
        PrintTagList(hReader, pTagList);
    }
    else
    {
        printf("%S\n", RFID_GetCommandStatusText(hReader, dwStatus));
    }
}

/// Read tags in the on demand reading mode
/// @param[in] hReader The handle of the reader
void ReadOnDemandEx(HANDLE hReader, TAG_LIST_EX *pTagList) {
    printf("On demand read EX from Reader %8x\n", (WORD)hReader);
    ConfigureOnDemandMode(hReader);
DWORD dwStatus = RFID_ReadTagInventoryEX(hReader, pTagList, TRUE);

if(dwStatus == RFID_SUCCESS)
{
    PrintTagListEx(hReader, pTagList);
}
else
{
    printf("%S\n", RFID_GetCommandStatusText(hReader, dwStatus));
}

/// Read tags in the autonomous reading mode
/// @param[in] hReader The handle of the reader
void ReadAutonomous(HANDLE hReader, TAG_LIST_EX *pTagList) {
    printf("Autonomous read from Reader %8x\n", (WORD)hReader);

    DWORD dwStatus = RFID_SUCCESS;
    ConfigureAutonomousMode(hReader);

    Sleep(1000);
    dwStatus = RFID_ReadTagInventoryEX(hReader, pTagList, true);

    if(dwStatus == RFID_SUCCESS)
    {
        PrintTagListEx(hReader, pTagList);
    }
    else
    {
        printf("%S\n", RFID_GetCommandStatusText(hReader, dwStatus));
    }
    ConfigureOnDemandMode(hReader);

}

/// Create the read tag read event for the reader
/// @param[in] hReader The handle of the reader
/// @return the handle of the read tag event, 0 for failure
HANDLE SetReadTagEvent(HANDLE hReader) {
{
TCHAR tszReadTagEventName[MAX_PATH];
// prepare the unique tag read event name for this reader
swprintf(tszReadTagEventName, TEXT("ReadTagEvent%8x"),(WORD)hReader);
// create a tag read event for this reader
HANDLE hReadTagEvent = CreateEvent(NULL, TRUE, FALSE, tszReadTagEventName);

if (!hReadTagEvent)
{
    return 0;
}

DWORD dwNumItems = (DWORD)wcslen(tszReadTagEventName) + 1;// include the null character
DWORD dwStatus = RFID_SetCapCurrValue(hReader, RFID_READCAP_EVENTNAME, &dwNumItems,
     dwNumItems * sizeof(tszReadTagEventName[0]),
     tszReadTagEventName);
if (dwStatus != RFID_SUCCESS)
{
    printf("nFailed to set read event name capability.");
    CloseHandle(hReadTagEvent);
    return 0;
}
return hReadTagEvent;

/// Enable the tag types for reader
/// @param[in] hReader The handle of the reader
void EnableTagTypesForReading(HANDLE hReader)
{
    // to see if the reader is a XR480
    // XR400 supports all protocols. XR480 only supports Gen 2 protocol!
    // Trying to enable other air protocols on XR480 will cause error!
    bool isXR480 = IsXR480(hReader);
    if (isXR480)
    {
        SetTypeForReading(hReader, RFID_TAG_TYPE_EPC_CLASS0, false);
        SetTypeForReading(hReader, RFID_TAG_TYPE_EPC_CLASS1, false);
        SetTypeForReading(hReader, RFID_TAG_TYPE_EPC_CLASSG2, true);
    } else
    {
        SetTypeForReading(hReader, RFID_TAG_TYPE_EPC_CLASS0, true);
    }
SetTagTypeForReading(hReader, RFID_TAG_TYPE_EPC_CLASS1, true);
SetTagTypeForReading(hReader, RFID_TAG_TYPE_EPC_CLASSG2, true);
}

/// Configure the parameters for the reader
/// @param[in] hReader the handle of the reader
/// @return true for success, false for failure
bool ConfigureReader(HANDLE hReader)
{
    if (!hReader)
    {
        return false;
    }

    EnableTagTypesForReading(hReader);

    return true;

    //DisplayCapabilities(hReader);
};

/// Configure the TCP/IP parameter of the reader
/// @param[in] hReader The handle of the reader
/// @param[in] ptszIPAddress Pointer to the IP address TCHAR string of the reader
/// @param[in] wPort The TCP port of the reader
/// @return true for success, false for failure
bool ConfigureTCPIP(HANDLE hReader, TCHAR *ptszIPAddress, WORD wPort)
{
    bool bSuccess = false;
    DWORD dwItems = 1;
    DWORD dwStatus = RFID_SUCCESS;

    dwStatus = RFID_SetCapCurrValue(hReader, RFID_DEV_CAP_IP_PORT, &dwItems, sizeof(wPort), &wPort);
    if (dwStatus == RFID_SUCCESS)
    {
        dwItems = (DWORD)wcslen(ptszIPAddress) + 1; // + 1 for null
        dwStatus = RFID_SetCapCurrValue(hReader, RFID_DEV_CAP_IP_NAME, &dwItems,
        dwItems * sizeof(ptszIPAddress[0]), ptszIPAddress);
    }
    return bSuccess;
}
if(dwStatus == RFID_SUCCESS)
{
    bSuccess = true;
}
else
{
    printf("RFID_DEVCAP_IP_NAME Set Cap Error %S\n", RFID_GetCommandStatusText(hReader, dwStatus));
}
else
{
    printf("RFID_DEVCAP_IP_PORT Set Cap Error %S\n", RFID_GetCommandStatusText(hReader, dwStatus));
}

return(bSuccess);

/// Opens and establishes connection to a reader
/// @param[in] tszNewIPAddress The IP address of the reader
/// @param[in] wPort The TCP port of the reader
HANDLE OpenReader(TCHAR tszNewIPAddress[32], WORD wPort)
{
    HANDLE hReader = 0;

    // open the API object for the reader
    if(RFID_Open(&hReader) == RFID_SUCCESS)
    {
        // configure TCP/IP parameter for the reader
        if(ConfigureTCPIP(hReader, tszNewIPAddress, wPort))
        {
            // connect to the reader
            if(RFID_OpenReader(hReader, 0) == RFID_SUCCESS)
            {
                printf("Found reader\n");
                // configure other parameters of the reader
                ConfigureReader(hReader);
            }
        }
    }
}
RFID_CAPS Caps;
if(RFID_GetCaps(hReader, &Caps) == RFID_SUCCESS)
{
    printf("RFIDAPI Version %S Firmware Version %S\n", Caps.szAPIVersionString, Caps.szFirmwareVersion);
    printf(" Serial # Info: %S\n", Caps.szSerialInfo);
}
else
{
    printf("Failed to open RFIDAPI\n");
}

return hReader;

/// @class XR400APITestReader
/// A class encapsulates the common functionalities of a test reader
class XR400APITestReader {
public:
    /// Default constructor
    XR400APITestReader() {
        hReader = 0;
        hReadTagEvent = 0;

        // initialize the tag list structure
        TAG_LIST_EX1_INIT(tagList, 0);
    }

    /// Destructor
    ~XR400APITestReader() {
        // clean up
        if(hReadTagEvent) {
            SetEvent(hReadTagEvent);
            CloseHandle(hReadTagEvent);
        }
    }
};
if (hReader) {
    RFID_CloseReader(hReader);
    RFID_Close(&hReader);
}

public:
    HANDLE hReader;
    HANDLE hReadTagEvent;
    TAG_LIST_EX tagList;
    TCHAR tszReadTagEventName[32];// = L"TagReadEvent";
    TCHAR tszIPAddress[32];
};

int main(int argc, char* argv[])
{
    printf("Test programm (%s %s) started...
", __DATE__, __TIME__);
    printf("Create instance...
");
    XR400APITestReader reader1;
    #ifndef _WIN32_WCE
        reader1.hReader = OpenReader(TEXT("157.235.88.44"), 3000);
    #else
        printf("Open reader...
");
        reader1.hReader = OpenReader(TEXT("127.0.0.1"), 3000);
    #endif // _WIN32_WCE
    printf("SetReadTagEvent...
");
    reader1.hReadTagEvent = SetReadTagEvent(reader1.hReader);
    DisplayCapabilities(reader1.hReader);

    #ifndef _WIN32_WCE
        XR400APITestReader reader2;
        reader2.hReader = OpenReader(TEXT("157.235.88.41"), 3000);
        reader2.hReadTagEvent = SetReadTagEvent(reader2.hReader);
    #endif // _WIN32_WCE
DisplayCapabilities(reader2.hReader);
#endif //_WIN32_WCE

printf("Tag list instance...
");
TAG_LIST tagList;
printf("Clear Tag list instance...
");
memset(&tagList, 0, sizeof(tagList));
printf("ReadOnDemand...
");
ReadOnDemand(reader1.hReader, &tagList);

#ifndef _WIN32_WCE
memset(&tagList, 0, sizeof(tagList));
ReadOnDemand(reader2.hReader, &tagList);
#endif //_WIN32_WCE

TAG_LIST_EX1_INIT(reader1.tagList, 0);
ReadOnDemandEx(reader1.hReader, &reader1.tagList);

#ifndef _WIN32_WCE
TAG_LIST_EX1_INIT(reader2.tagList, 0);
ReadOnDemandEx(reader2.hReader, &reader2.tagList);
#endif //_WIN32_WCE

memset(&reader1.tagList, 0, sizeof(reader1.tagList));
TAG_LIST_EX1_INIT(reader1.tagList, 0);
ReadAutonomous(reader1.hReader, &reader1.tagList);

#ifndef _WIN32_WCE
memset(&reader2.tagList, 0, sizeof(reader2.tagList));
TAG_LIST_EX1_INIT(reader2.tagList, 0);
ReadAutonomous(reader2.hReader, &reader2.tagList);
#endif //_WIN32_WCE

printf("\nType in anything and enter to exit...
");
getchar();
printf("\nClosing down...
");

return 0;
Index

A
API capabilities ........................................ A-3
RFID_DEVCAP_ANTENNA_SEQUENCE ........ A-8
RFID_DEVCAP_IP_NAME ................. A-5
RFID_DEVCAP_IP_PORT .......... A-6
RFID_INF_CAP_SUPPORTEDCAPS ... A-4
RFID_READCAP_EVENTNAME ........ A-12
RFID_READCAP_EVENTTAGPTR ........ A-11
RFID_READCAP_EVENT_ALLTAGS .... A-16
RFID_READCAP_INLOOP .......... A-20
RFID_READCAP_METHOD .......... A-17
RFID_READCAP_OUTLOOP ........ A-18
RFID_READCAP_READMODE .......... A-21
RFID_TAGCAP_RF_ATTENUATION .... A-10
RFID_TAGCAP_C0_SINGULATION_FIELD .. A-26
RFID_TAGCAP_ENABLED_TYPES ........ A-26
RFID_TAGCAP_LOCKCODE .......... A-24
RFID_TAGCAP_SUPPORTED_TYPES .... A-25
RFID_WRITECAP_RF_ATTENUATION ... A-22
RFID_WRITECAP_TAGTYPE .......... A-23
API return values ................. B-3
autonomous mode .................... 5-4, 5-15, A-12
setting ........................................ A-21
tag event handling ................. A-12

B
bullets ........................................ xi

C
capabilities ...................................... 4-3, A-3
RFID_DEVCAP_ANTENNA_SEQUENCE ... A-8
RFID_DEVCAP_IP_NAME ................. A-5
RFID_DEVCAP_IP_PORT .......... A-6
RFID_GetCapCurrValue ............. 4-7
RFID_GetCapDfltValue ............. 4-10
RFID_GetCapInfo ................. 4-5
RFID_GetCapList .......... 4-4
RFID_INF_CAP_SUPPORTEDCAPS ... A-4
RFID_READCAP_EVENTNAME ........ A-12
RFID_READCAP_EVENTTAGPTR ........ A-11
RFID_READCAP_EVENT_ALLTAGS .... A-16
RFID_READCAP_INLOOP .......... A-20
RFID_READCAP_METHOD .......... A-17
RFID_READCAP_OUTLOOP ........ A-18
RFID_READCAP_READMODE .......... A-21
RFID_READCAP_RF_ATTENUATION ... A-10
RFID_WRITECAP_RF_ATTENUATION ... A-22
RFID_WRITECAP_TAGTYPE .......... A-23

conventions
notational ................................... xi
## D
device capabilities ........................................ 4-3

## F
functions
- RFID_Close ........................................... 3-8
- RFID_DEVCAP_ANTENNA_SEQUENCE .................. A-8
- RFID_DEVCAP_IP_NAME ................................ A-5
- RFID_DEVCAP_IP_PORT ................................ A-6
- RFID_EraseTag ......................................... 5-13
- RFID_GetCapCurrValue ................................ 4-7
- RFID_GetCapDfltValue ................................ 4-10
- RFID_GetCapInfo ....................................... 4-5
- RFID_GetCapList ....................................... 4-4
- RFID_GetCommandStatusText ......................... 6-4
- RFID_GetStats ......................................... 6-5
- RFID_GetTagID ......................................... 5-4
- RFID_GetTagMask ....................................... 5-5
- RFID_INFNCAP_SUPPORTEDCAPS ......................... A-4
- RFID_KillTag ........................................... 5-7
- RFID_LockTag .......................................... 5-9
- RFID_Open ................................................ 3-5
- RFID_ProgramTags ..................................... 5-11
- RFID_READCAP_EVENTNAME ............................ A-12
- RFID_READCAP_EVENTTAGPTR ......................... A-11
- RFID_READCAP_EVENT_ALLTAGS ....................... A-16
- RFID_READCAP_INLOOP ................................ A-20
- RFID_READCAP_METHOD ................................ A-17
- RFID_READCAP_OUTLOOP ................................ A-18
- RFID_READCAP_READMODE .............................. A-21
- RFID_READCAP_RF_ATTENUATION ..................... A-10
- RFID_ReadTagInventory ................................ 5-14
- RFID_SetCapCurrValue ................................ 4-11
- RFID_SetCapDflts ..................................... 4-12
- RFID_SetCapDfltValue ................................ 4-13
- RFID_SetTagMask ....................................... 5-6
- RFID_TAGCAP_C0_SINGULATION_FIELD ............... A-26
- RFID_TAGCAP_ENABLED_TYPES ....................... A-26
- RFID_TAGCAP_LOCKCODE ................................ A-24
- RFID_TAGCAP_SUPPORTED_TYPES ..................... A-25
- RFID_WRITECAP_RF_ATTENUATION .................... A-22
- RFID_WRITECAP_TAGTYPE ................................ A-23

## H
helper functions ............................................ 6-3
- RFID_GetCommandStatusText ......................... 6-4
- RFID_GetStats ......................................... 6-5

## I
information, service ....................................... xii
- initialization .......................................... 3-3
- RFID_Close ............................................. 3-3
- RFID_Open ................................................ 3-5

## O
on demand mode ............................................ 5-4, 5-14, A-12
- setting .................................................. A-21
- tag event handling .................................... A-14

## R
reading tags ................................................ 5-3
return values ................................................ B-3

## S
service information ........................................ xii

## T
tag functions ............................................ 5-3
- RFID_EraseTag ......................................... 5-13
- RFID_GetTagID ......................................... 5-4
- RFID_GetTagMask ....................................... 5-5
- RFID_KillTag .......................................... 5-7
- RFID_LockTag .......................................... 5-9
- RFID_ProgramTags ..................................... 5-11
- RFID_ReadTagInventory ................................ 5-14
- RFID_SetTagMask ....................................... 5-6

## W
writing tags ................................................. 5-3
Tell Us What You Think...

We’d like to know what you think about this Manual. Please take a moment to fill out this questionnaire and fax this form to: (631) 738-3318, or mail to:

Symbol Technologies, Inc.
One Symbol Plaza   M/S B-4
Holtsville, NY  11742-1300
Attention: Technical Publications Manager

IMPORTANT: If you need product support, please call the appropriate customer support number provided. Unfortunately, we cannot provide customer support at the fax number above.

Manual Title:___________________________________________
(please include revision level)

How familiar were you with this product before using this manual?

☐  Very familiar  ☐  Slightly familiar  ☐  Not at all familiar

Did this manual meet your needs? If not, please explain.
_______________________________________________________
_______________________________________________________
_______________________________________________________
_______________________________________________________

What topics need to be added to the index, if applicable?
_______________________________________________________
_______________________________________________________
_______________________________________________________
_______________________________________________________

What topics do you feel need to be better discussed? Please be specific.
_______________________________________________________
_______________________________________________________
_______________________________________________________
_______________________________________________________

What can we do to further improve our manuals?
_______________________________________________________
_______________________________________________________
_______________________________________________________
_______________________________________________________

Thank you for your input—We value your comments.