DL RFID API

User Manual

User Manual

Revision n. A

06/07/2017

Scope of Manual

The goal of this user manual is to provide the basic information to work with DL RFID readers.

The DLRFID is a basic library to support an easy communication with different reader types.

Change Document Record

Date	Revision	Changes	Pages
9 Sep 2016	00	Preliminary	22
6 Jul 2017	A	Initial Release	27

Datalogic S.r.l.

Via San Vitalino 13
40012 Calderara di Reno
Italy
Telephone: +39 051 3147011
Fax: +39 051 3147288

Federal Communications Commission (FCC) Notice (Preliminary)

This device was tested and found to comply with the limits set forth in Part 15 of the FCC Rules. Operation is subject to the following conditions: (1) this device may not cause harmful interference, and (2) this device must accept any interference received including interference that may cause undesired operation. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment.

This device generates, uses, and can radiate radio frequency energy. If not installed and used in accordance with the instruction manual, the product may cause harmful interference to radio communications. Operation of this product in a residential area is likely to cause harmful interference, in which case, the user is required to correct the interference at their own expense. The authority to operate this product is conditioned by the requirements that no modifications be made to the equipment unless the changes or modifications are expressly approved by Datalogic.

Index

List of Tables

1 Introduction

This Chapter gives basic information about the SDK architecture. It contains these topics:

- Overview
- SDK Design Concepts

Overview

This guide describes the model, design concepts and the Application Software Interface (API) offered by DL RFID for the development of software for use, integrate and control DL RFID readers and accessories.

DL RFID provides a Software Development Kit (SDK) that includes APIs for the most common programming languages: Java, .NET (C# and Visual Basic) and Visual C/C++.

In details, the SDK package contains:

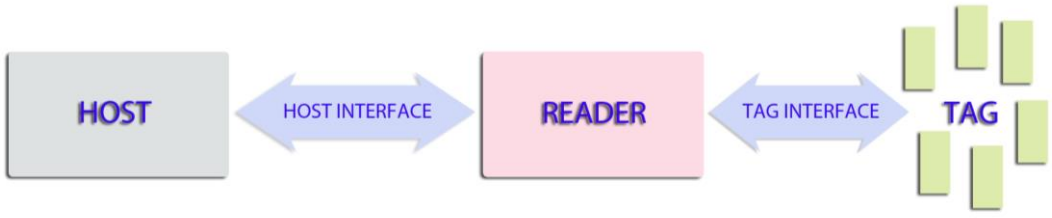
- a .NET library in the form of a DLL file with its own history file, the help file (CHM format), the .NET CF (Compact Framework) version and a demo application (with source code)
- a Java library in the form of a JAR file with its own history file, the corresponding JavaDoc files (HTML format) and a demo application (with source code)
- a Visual C library in the form of a DLL file, with its own history file, the stub file (.lib), the headers and a demo application (with source code)
- the DL RFID API User Manual (this document)
- the DL RFID API Reference Manual

The differences between the APIs are mostly related to the different language syntax and architecture but, from a semantic point of view, the methods/functions are maintained the same in all the languages.

DL RFID provides also the complete documentation of the raw binary protocol that can be used to communicate with DL RFID readers for languages or architectures not supported by the APIs.

SDK Design Concepts

A DL RFID reader can be seen, in a simplified model, as a box with one or more communication interfaces on the host's side and one or more antennas on the tags' side. The reader accepts commands coming from an host (a PC or any other controlling device), it uses the interface to the tag (typically one or more antennas) to perform operations on the tags and it replies to the host.



The API defines a number of classes (emulated in C language by functions and data types) in order to represent this simplified model; two of them define most of the methods and can be considered the core classes of the API itself: *DLRFIDReader* and *DLRFIDLogicalSource*.

DLRFIDReader class provides methods for the general reader configuration, host communication interfaces configuration, HW parameters etc.

DLRFIDLogicalSource class defines the methods for the reader to tag communication and its configuration.

Working with the host interfaces, mostly represented in the *DLRFIDReader* class, is quite straightforward since they are standard communication interfaces (Ethernet, RS232 and similar) while for the *DLRFIDLogicalSource* class a deeper insight is needed.

One or more physical RFID antennas (called ReadPoints in the API) can be connected to a DL RFID, each one able to detect tags and it is typical, in UHF RFID installations, to place multiple antennas in the same place for a better coverage of the reading area. In this case, even if multiple antennas are used, the reading area is the same so, from a logical point of view, it is a single source of homogeneous information. In order to model this concept we introduced the Logical Source concept that is implemented into the DL RFID API with the *DLRFIDLogicalSource* class: it permits to group together ReadPoints (antennas) that are logically related. Each data exchange concerning tags is implemented through the DLRFIDLogicalSource class, the case of a single antenna is a special case where a Logical Source contains a single Read Point.

The DL RFID API permits to add and remove Read Points to/from the Logical Source so that the user can easily represent its installation.

Up to the current revision, the API handles four Logical Source (called “Source_0”, “Source_1”, “Source_2”, “Source_3”) and four Read Points (called “Ant0”, “Ant1”, “Ant2”, “Ant3”) and the default configuration is that each LogicalSource contains only one different ReadPoint; in the future this could change and the number of Logical Sources and Read Points could be different depending on the model of the RFID reader.

On readers with a single antenna connector, the only meaningful Logical Source is the “Source_0” one and it contains the only one available antenna “Ant0”.

Note that after a reader switch off the Logical Sources composition is reset to the default configuration.

2 Getting Started

This chapter describes the minimum steps a programmer should follow in order to operate with a DL RFID reader using the API.

The minimum steps a programmer should follow in order to operate with a DL RFID reader using the API are the following:

- Open a connection with the reader
- Configuration of the logical source (optional if the default configuration is fine)
- Detection of tags and other operations on the tags
- Close the connection with the reader.

Here below a simple but complete code snippet showing the minimum required lines of code to detect RFID tags using a DL RFID reader and the DL RFID API. The code is shown using .NET C# programming language but it can be easily adapted to the other languages supported by the API.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using com.DL.RFIDLibrary;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            DLRFIDReader MyReader = new DLRFIDReader();

            MyReader.Connect(DLRFIDPort.DLRFID_RS232, "COM3");

            DLRFIDLogicalSource MySource = MyReader.GetSource("Source 0");

            DLRFIDTag[] MyTags = MySource.InventoryTag();

            if (MyTags.Length > 0)
            {
                for (int i = 0; i < MyTags.Length; i++)
                {
                    String s = BitConverter.ToString(MyTags[i].GetId());
                    Console.WriteLine(s);
                }
            }
            Console.WriteLine("Press a key to end the program.");
            Console.ReadKey();
            MyReader.Disconnect();
        }
    }
}
```

3 Basic Operations

This chapter shows the most common operations that a programmer can perform using a DL RFID reader and the DL RFID API. All the example will be shown in the C# language; the Visual Basic, Java and C version are omitted since they differ only at syntax level. The chapter contains these topics:

- Importing the libraries
- Reader connection/disconnection
- Getting information about the connected reader
- Setting the power level
- Inventorying RFID tags
- Optimizing the inventory process
- Reading and writing Gen2 tags
- Locking Gen2 tags
- Killing Gen2 tags
- Handling General purpose Inputs/Ouputs (GPIOs)

Importing the libraries

The current version of .Net and Visual C libraries have been developed using Microsoft Visual Studio 2005 and the Java library has been developed using Oracle Netbeans 6.8.

Methods to import the libraries in the application developer's projects strongly depend on the Integrated Development Environment (IDE), please refer to the documentation of the IDE in use for instructions.

Reader connection/disconnection

The first operation to perform in order to start to operate with DL RFID readers is to establish a connection. The connection method depends on the physical interface available on the readers that, at now, could be an Ethernet or a serial interface. Readers with USB interface can be considered as having a standard RS232 serial interface since the readers implement a USB to RS232 converter internally.

The API provides the *Connect* method/function that permits to establish the connection. The method accepts two parameters: the interface type and the address. If the interface type is serial the address will be the name of the serial port (e.g. “COM1”); if the interface is Ethernet (TCP/IP) the address will be the IP address of the reader.

Here below code examples for serial port connection and for TCP/IP connection are shown.

```
MyReader.Connect(DLRFIDPort.RS232,”COM1”);
MyReader.Connect(DLRFIDPort.TCP,”192.168.0.2”);
```

Once connected it is possible to operate on the reader with the other methods. At the end of the operations it is possible to disconnect from the reader using the *Disconnect* method/function.

```
MyReader.Disconnect();
```

Connect and *Disconnect* methods are members of the *DLRFIDReader* class.

Getting information about the connected reader

The same API can be used to control different type of readers with different capabilities. It is often useful to know which type of reader is connected to the application in order to access to the right features.

The API provides two methods for getting information about the connected reader: *GetReaderInfo* and *GetFWRelease*. The first returns information about the reader's model and serial number, the latter returns the version of the firmware running into the reader itself.

Here below code examples for getting information about the connected reader are shown.

```
DLRFIDReaderInfo Info = MyReader.GetReaderInfo();

String Model = Info.GetModel();
String SerialNumber = Info.GetSerialNumber();
```

```
String FWRelease = MyReader.GetFWRelease();
```

GetReaderInfo and *GetFWRelease* methods are members of the *DLRFIDReader* class.

Setting the power level

Most of the DL RFID readers allow to regulate the emitted power. This setting is useful in order to limit the read range of the reader, to limit the power as stated by the local regulations or for adapt the power depending on the antenna characteristics.

The API provides two methods, one for setting the power (*SetPower*) and one for getting the current power level (*GetPower*). The value passed to *SetPower* and returned by *GetPower* is expressed in milliWatt (mW) and refers to the power generated by the reader at the antenna's connectors.

The effective radiate power in mW ERP (*Perp*) is related to the conducted RF power (*Pw*) provided at the reader's connector by the following formula:

$$Pw = \frac{Perp}{10^{\frac{(G-2.14-L)}{10}}}$$

where G is the antenna gain expressed in dBi and L the cable attenuation expressed in dB.

So, if you require (as often is) to set a ERP power level, the above formula has to be implemented in your software in order to obtain the conducted RF power.

Here below a code example that permits to obtain the desired ERP power from the antenna using the above formula and the *SetPower* method:

```
double Gain = 8.0;
double Loss = 1.5;
double ERPPower = 2000.0;
int OutPower;

OutPower = (int)(ERPPower/Math.Pow(10,((Gain-Loss-2.14)/10)));
MyReader.SetPower(OutPower);
```

and here a code example that permits to know the current setting of the ERP power using the inverse form of the formula and the *GetPower* method:

```
double Gain = 8.0;
double Loss = 1.5;
double ERPPower;
int OutPower;

OutPower = MyReader.GetPower();
ERPPower = ((double)power)*((double)Math.Pow(10,((Gain-Loss-2.14)/10)));
```

Typically DL RFID readers approximate automatically the power to the nearest available power level and the same is done for the minimum and maximum power level. For the available power levels please refer to the reader's user manual.

There could be a difference between the power level set and the power level read just after the setting, this effect is normal and it is due to one or more of the following reasons:

- the set value was not exaclty one of the available power level on the reader so the real power level is the nearest available;

- the formula used to convert ERP to conducted power introduced a mathematical approximation;
- on some DL RFID readers the value returned by the *GetPower* method is a measurement of the power that can be affected by the accuracy of the measurement and the caracteristics of the connected antenna.

GetPower and *SetPower* methods are members of the *DLRFIDReader* class.

Inventorying RFID tags

The fundamental operation of a UHF RFID system is the inventory of the population of tags inside the reading zone of the reader's antennas. This operation, for the Gen2 protocol as for other UHF protocols, consists of a sequence of commands and replies exchanged between the reader and the tags, typically with stringent timings between them. DL RFID readers hide the complexity of the inventory algorithm implementing the algorithm in the firmware and providing a macro command as interface for the user.

The DL RFID API provides methods in order to activate the inventory process, the simplest method that just tries to collect all the tags inside the reading zone, a more complex one with a list of options and a method to start a cycle of inventories.

In the simplest form the inventory process can be activated simply by calling the *InventoryTag* method of the *DLRFIDLogicalSource* class. This method has no parameters at all and returns an array of DLRFIDTag objects once a complete run of the inventory algorithm is performed inside the reader (a DLRFIDTag object is a software representation of the physical tag carrying the data associated to it like the EPC code, its length, the type of the tag and others). For a code sample look at the Getting Started pag. 9 of this manual.

A more complete version of the *InventoryTag* methods takes parameters for filtering tags and to activate some optional features. The parameters used to filter the tags that have to be detected are: the memory bank (Bank), the mask (Mask), the length of the mask (MaskLength) and the start address (Position) for the matching. Using those parameters a Gen2 Select command is issued before starting the inventory process in order to match only the interesting tags. In the matching process the Mask parameter is compared to the memory bank content starting from the address Position for MaskLength bits. Only the matching tags will be involved in the inventory process and returned back to the user by the *InventoryTag* method.

The user can also choose the result of the matching mechanism, i.e. can choose to return the matching tags, the not matching tags or all the tags (ignoring indeed the filter). This setting can be changed using the *SetSelected_EPC_C1G2* method of the *DLRFIDLogicalSource* class; possible values for its only parameter are:

- EPC_C1G2_SELECTED_YES: for matching tags;
- EPC_C1G2_SELECTED_NO: for non-matching tags;
- EPC_C1G2_SELECTED_ALL: for all tags (no filter).

An additional parameter (Flags) permits to activate special features of the inventory process, it is a bit mask where only the 5 less significant bits are used.

Bit 0 enables (1) or disables (0) the Return Signal Strength Indicator (RSSI) reading for each tag for those readers supporting it.

Bit 1 enables (1) or disables (0) the so called framed mode: if the framed mode is not enabled (default behavior) all the tags collected during the inventory process are stored into the reader's

memory and returned back to the user at the end of the process. With the framed mode enabled as soon as a tag is detected is returned immediately to the user. This behavior results in a better responsiveness of the application especially with large population of tags and it is suggested when a small embedded reader with limited memory is used. It is mandatory to enable the framed bit when the continuous mode is enabled (see next bit description).

Bit 2 enables (1) or disables (0) the continuous mode: when this bit is enabled the reader implements internally a cycle of inventories. The number of executed inventories is determined by the ReadCycle parameter that can be set with the *SetReadCycle* method of the *DLRFIDLogicalSource* class. When ReadCycle is 0 the cycle is repeated indefinitely until an abort command is sent to the reader.

Bit 3 enables (1) or disable(0) the compact data mode: when this flag is enabled, the inventory method will return back to the caller only the EPC code of the tag and all the other information like the timestamp and the type of the tag are filled with fake values. This flag is useful when it is necessary to reduce the data exchanged on the host interface, typically when the interface is slow (low baud rate serial interfaces).

Bit 4 enables (1) or disables (0) the readout of the TID during the inventory process.

A further option is to use an event-based inventory handling that means start a continuous and autonomous inventorying getting immediately the control of the thread flow to the caller. All the readings will be received by the application as software events. The user needs to define an event handler that will take care of handling the data coming from the tags.

InventoryTag, EventInventoryTag and *InventoryAbort* methods are members of the *DLRFIDLogicalSource* class.

EventInventoryTag

Here below a code sample about the *EventInventoryTag* method: the code shows how to setup an event handler for processing data sent by the reader, start a continuous inventory operation and at the end stop it using the *InventoryAbort* method.

```
static void Main(string[] args)
{
 DLRFIDReader Reader = new DLRFIDReader();
 DLRFIDLogicalSource LS0;
 byte[] Mask = new byte[4];

 Reader.DLRFIDEvent += new DLRFIDEventHandler(Reader EventHandler);
 Reader.Connect(DLRFIDPort.DLRFID TCP, "10.0.32.125");

 LS0 = Reader.GetSource("Source 0");

 LS0.SetReadCycle(0);
 LS0.EventInventoryTag(Mask, 0x0, 0x0, 0x06);
 Thread.Sleep(2*1000);
 Console.WriteLine("Main Task awake");
 Console.WriteLine("Tags read : " + ntag.ToString());
 Reader.InventoryAbort();
 Console.ReadLine();

 Reader.Disconnect();
}

static void Reader_EventHandler(object Sender, DLRFIDEventArgs Event)
{
 foreach (DLRFIDNotify n in Event.Data)
 {
  ntag++;
 }
```

```
}
```

In the readers equipped with button (for example in the R1240I qID reader) you can perform the inventory just by pressing the scan button (for more info, refer to the technical specifications of your reader), setting the flag parameter of the *EventInventoryTag* as follows:

```
LS0.EventInventoryTag (Mask, 0x0, 0x0, 0x26);
```

Note the use of the flag parameter in the method EventInventoryTag: enabling a continuous cycle is achieved through the activation of the FRAMED and CONTINUOUS fields while the suspension of the cycle by pressing the button requires the activation of the field TRIGGER EVENT.

For more info about the flag value meaning of the *EventInventoryTag* method, please refer to *DL RFID API Reference Manual*.

Optimizing the inventory process

EPC Class1 Gen2 protocol defines a set of parameters useful for the optimization of the tags' detection, in the current paragraph we give a brief explanation of the most useful ones and we show how to work with those parameters using the DL RFID API.

The Q parameter

The Gen2 protocol inventory method is based on the so called “Slotted ALOHA” algorithm; without going into the details of the algorithm, it is important to know that it foresees a division of the time in discrete slots. Only one tag can be detected for each slot, if two or more tags reply on the same slot a collision is generated and the tags are not detected so a further iterations of the algorithm is needed.

The number of time slots is defined in the Gen2 protocol as 2Q where Q is a parameter ranging from 0 to 15 that can be set by the user.

The optimal Q value for a certain application is related to the average number of tags that are simultaneously present in the reading zone. A few tags require only a few slots, whereas many tags require many slots. Left to its own, the reader doesn’t have any way of knowing how many tags are under the antenna's field until it counts them, which may be difficult if its initial “guess” is wildly wrong. DL RFID reader will work faster and more efficiently if you provide an accurate starting value for Q corresponding to the expected tag population: the more are the collisions the worse are the performance of the reader in detecting the tags. The detection method implemented inside DL RFID readers has also an auto-adaptive mechanism in order to generate more or less time slots when needed but, starting with an adequate number of time slots, helps this mechaism to avoid wasting time.

For reference, the following table gives recommended values of Q that produce reasonably efficient inventories for varying numbers of tags in the read zone.

Estimated number of tags	Starting Q value
1	0
2	1
3 – 6	2
7 – 15	3
16 – 30	4
30 – 50	5
50 – 100	6
100 – 200	7

Tab. 3.1: Recommended Q values

DL RFID API provides the methods to set and read back the starting Q value: *SetQ_EPC_C1G2* and *GetQ_EPC_C1G2*.

Here below a code sample where the Q value is set to 3 before to start the inventory process:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using com.DL.RFIDLibrary;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            DLRFIDReader MyReader = new DLRFIDReader();

            MyReader.Connect(DLRFIDPort.DLRFID RS232, "COM3");

            DLRFIDLogicalSource MySource = MyReader.GetSource("Source_0");

            MySource.SetQ EPC C1G2(3);

            DLRFIDTag[] MyTags = MySource.InventoryTag();

            if (MyTags.Length > 0)
            {
                for (int i = 0; i < MyTags.Length; i++)
                {
                    String s = BitConverter.ToString(MyTags[i].GetId());
                    Console.WriteLine(s);
                }
            }
            Console.WriteLine("Press a key to end the program.");
            Console.ReadKey();
            MyReader.Disconnect();
        }
    }
}
```

SetQ_EPC_C1G2 and *GetQ_EPC_C1G2* methods are members of the *DLRFIDLogicalSource* class.

Sessions

When a tag is singulated by the “slotted ALOHA” algorithm described in the previous paragraph, it switches automatically from an internal state A to another internal state B.

In the EPC C1G2 terminology these states are called “target A” and “target B” respectively; for our purposes we can refer to target A as the non-inventoried state and to target B as the inventoried state: singulation makes therefore switch a tag from the non-inventoried state to the inventoried state.

By default DL RFID readers looks for non-inventoried tags so once a tag has been inventoried it is no more detectable by the reader until it returns back in the non-inventoried state.

Tags return in the non-inventoried state autonomously and the times needed to return detectable can be regulated using the session parameter. This tuning can have a big impact on the inventory process performances especially in case of large tags' population.

Gen2 protocol provides four different sessions: S0, S1, S2, S3. Each session has its own independent inventoried and non-inventoried states with its specific persistence time as shown in the following table:

Flag	Required persistence
S0 inventoried flag	Tag energized: Indefinite Tag not energized: None
S1 inventoried flag	Tag energized: Nominal temperature range: 500ms < persistence < 5s Extended temperature range: Not specified Tag not energized: Nominal temperature range: 500ms < persistence < 5s Extended temperature range: Not specified
S2 inventoried flag	Tag energized: Indefinite Tag not energized: Nominal temperature range: 2s < persistence Extended temperature range: Not specified
S3 inventoried flag	Tag energized: Indefinite Tag not energized: Nominal temperature range: 2s < persistence Extended temperature range: Not specified

Tab. 3.2: Persistence Time

When session S0 is used, each time the reader switches off the radiofrequency field, the tags return back in the original not-inventoried status so that they are again detectable with a successive inventory algorithm round. Since DL RFID readers switch off the field at the end of each inventory round, all the tags are detected again each time the inventory is repeated. This behavior can be desirable for testing purposes or when a certain level of redundancy is needed but it is typically not efficient and generate a lot of useless information.

With session S1 the tags, once singulated, remain in the inventoried state for a time in the range between 500ms and 5s regardless if the tag is energized or not; during this period they are no more detectable by the subsequent inventory iterations. On large tags' populations this behavior helps to reduce the number of tags simultaneously detectable by the reader optimizing the speed of the inventory process and reducing the generated data.

Sessions S2 and S3 have a longer and not explicitly limited persistence time giving the opportunity to detect tags only one time during the inventory process repetitions. These sessions are for advanced use only and are out of the scope of this manual.

DL RFID API provides the methods to set and read back the session parameter: *SetSession_EPC_C1G2* and *GetSession_EPC_C1G2*.

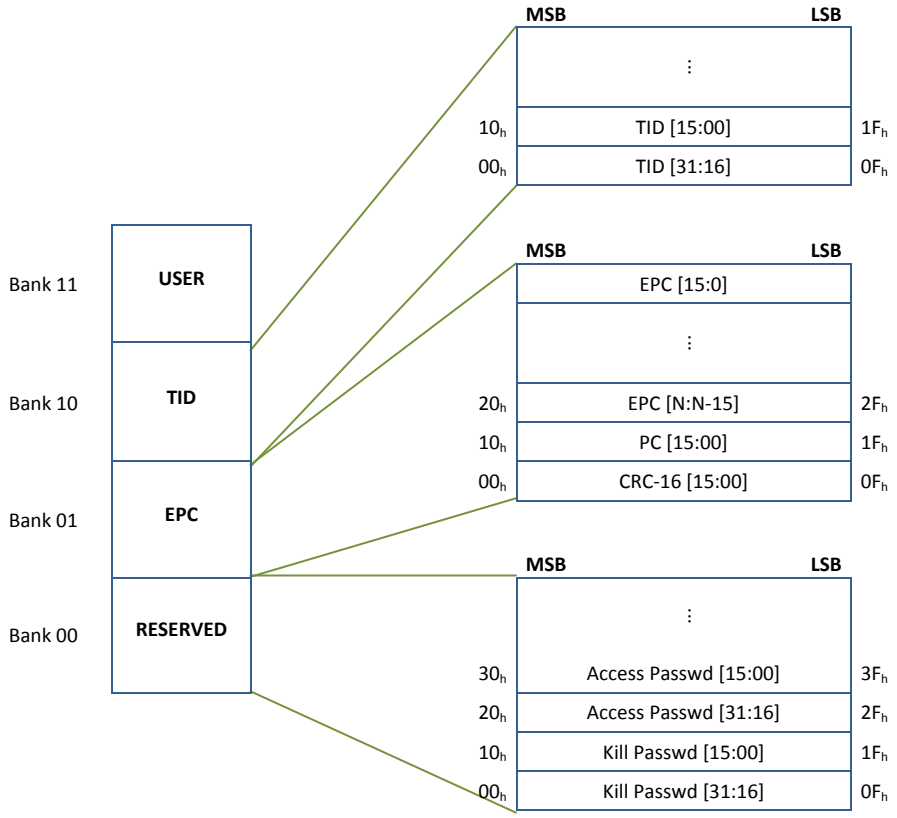
Here below a code sample where the session S1 is used during the inventory process:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using com.DL.RFIDLibrary;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            DLRFIDReader MyReader = new DLRFIDReader();

            MyReader.Connect(DLRFIDPort.DLRFID RS232, "COM3");

            DLRFIDLogicalSource MySource = MyReader.GetSource("Source_0");

            MySource.SetSession EPC C1G2(DLRFIDLogicalSourceConstants.EPC C1G2 SESSION S1);

            DLRFIDTag[] MyTags = MySource.InventoryTag();

            if (MyTags.Length > 0)
            {
                for (int i = 0; i < MyTags.Length; i++)
                {
                    String s = BitConverter.ToString(MyTags[i].GetId());
                    Console.WriteLine(s);
                }
            }
            Console.WriteLine("Press a key to end the program.");
            Console.ReadKey();
            MyReader.Disconnect();
        }
    }
}
```

SetSession_EPC_C1G2 and *GetSession_EPC_C1G2* methods are members of the *DLRFIDLogicalSource* class.

Reading and writing Gen2 tags

Gen2 tags contains a memory with the following structure:



The inventory process returns the EPC code that is part of the content of the EPC memory bank as a side-effect of the singulation process. All the memory content, anyway, can be read using the Gen2 Read command and the rewritable memory can be written using the Gen2 Write command.

The DL RFID API provides methods to read (*ReadTag_EPC_C1G2*) and write (*WriteTag_EPC_C1G2*) data into the tags' memory; the methods implement internally a Select command that is used to identify the tag on which execute the read or write command permitting to read/write into a specific tag even if multiple tags are inside the reading zone. The identification of the tag is executed by matching the EPC code that can be previously obtained by an inventory process. The other parameters needed to execute the read and write command are the memory bank, the starting address, the data length and, in case of the write command, the data that have to be written.

The API provides also a “secured” version of those commands that require a password as an additional parameter. These variants of the methods have to be used in the case the Access password is set into the tag. Tags with a non-zero Access password are “protected” (in Gen2 terminology: tags with a non-zero Access password are in Open state) and the user needs to know and passes to the tag the Access password in order to write into the tag's memory (in Gen2 terminology: the tag has to switch in Secured state using the password before to allow to write).

Here below a code sample of reading/writing data from/to the user memory:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using com.DL.RFIDLibrary;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            DLRFIDReader MyReader = new DLRFIDReader();

            MyReader.Connect(DLRFIDPort.DLRFID RS232, "COM3");

            DLRFIDLogicalSource MySource = MyReader.GetSource("Source_0");

            DLRFIDTag[] MyTags = MySource.InventoryTag();

            if (MyTags.Length > 0)
            {
                for (int i = 0; i < MyTags.Length; i++)
                {
                    String EPCString = BitConverter.ToString(MyTags[i].GetId());
                    Console.WriteLine(EPCString);

                    byte[] DataToWrite;
                    ASCIIEncoding Enc = new ASCIIEncoding();

                    DataToWrite = Enc.GetBytes("Hello!");
                    MySource.WriteTagData_EPC_C1G2(MyTags[i], 3, 0, 6, DataToWrite);
                    Console.WriteLine("Tag written!");

                    byte[] DataToRead;
                    DataToRead = MySource.ReadTagData EPC C1G2(MyTags[i], 3, 0, 6);
                    Console.WriteLine("Tag read, value = " + Enc.GetString(DataToRead));
                }


            }
            Console.WriteLine("Press a key to end the program.");
            Console.ReadKey();
            MyReader.Disconnect();
        }
    }
}
```

The read or write command will be executed on the first tag that replies to the reader chosen from those matching the filtering criterion.

ReadTag_EPC_C1G2 and Write*Tag_EPC_C1G2* methods are members of the *DLRFIDLogicalSource* class.

Locking Gen2 tags

The EPCGlobal Class1 Gen2 protocol provides a mechanism to lock temporarily or permanently blocks of tag's memory. The user can lock an entire memory bank with the only exception of the Reserved memory bank where it is allowed to lock independently the Access Password and the Kill Password. The lock mechanism prevents the possibility of further modifications on locked memory banks and, only for the passwords in the Reserved memory bank, it prevents also the possibility to read back the data (the passwords). The lock command works with a single parameter called *payload* that includes both the lock's type (permanent or not) and the bank to lock. The format of the payload is described by the following tables:

Lock-Command Payload

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Kill Mask		Access Mask		EPC Mask		TID Mask		User Mask		Kill Action		Access Action		EPC Action		TID Action		User Action	

Masks and Associated Action Fields

	Kill pwd		Access pwd		EPC memory		TID memory		User memory	
	0	1	2	3	4	5	6	7	8	9
Mask	skip/ write	skip/ write	skip/ write	skip/ write	skip/ write	skip/ write	skip/ write	skip/ write	skip/ write	skip/ write
	10	11	12	13	14	15	16	17	18	19
Action	pwd read/ write	perma lock	pwd read/ write	perma lock	pwd write	perma lock	pwd write	perma lock	pwd write	perma lock

Tab. 3.3: Lock payload and usage

pwd-write	permalock	Description
0	0	Associated memory bank is writeable from either the **open** or **secured** states
0	1	Associated memory bank is permanently writeable from either the **open** or **secured** states and may never be locked
1	0	Associated memory bank is writeable from the **secured** state but not from the **open** state
1	1	Associated memory bank is not writeable from any state
pwd-read/write	**permalock**	**Description**
0	0	Associated password location is readable and writeable from either the **open** or **secured** states
0	1	Associated password location is permanently readable and writeable from either the **open** or **secured** states and may never be locked
1	0	Associated password location is readable and writeable from **secured** state but not from **open** state
1	1	Associated password location is not readable or writeable from any state

Tab. 3.4: Lock Action-field functionality

The DL RFID API provides a method (*LockTag_EPC_C1G2*) to lock the tag's memory contents that mimic exactly the behaviour of the protocol command. The payload parameter is implemented as a bitmask with the meaning described by the above tables. The API provides also the secured version of the *LockTag_EPC_C1G2* method to be used when the Access password is set.

Here below a code example of the *LockTag_EPC_C1G2* method utilization.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using com.DL.RFIDLibrary;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            DLRFIDReader MyReader = new DLRFIDReader();

            MyReader.Connect(DLRFIDPort.DLRFID RS232, "COM3");

            DLRFIDLogicalSource MySource = MyReader.GetSource("Source_0");

            DLRFIDTag[] MyTags = MySource.InventoryTag();

            if (MyTags.Length > 0)
            {
                for (int i = 0; i < MyTags.Length; i++)
                {
                    String EPCString = BitConverter.ToString(MyTags[i].GetId());
                    Console.WriteLine(EPCString);

// +-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+
// | Kill  | Acces | EPC   | TID   | User  | Kill  | Acces |  EPC  |  TID  |  User |
// | Mask  | Mask  | Mask  | Mask  | Mask  | Act.  | Act.  |  Act. |  Act. |  Act. |
// +---+---+---+---+---+---+---+---+---+---+---+---+-------+-------+-------+-------+
// | W | P | W | P | W | P | W | P | W | P | W | P | W | P | W | P | W | P | W | P |
// +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
// | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
// +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
//
// Non permanent locking of User Memory Bank
//
                    int Payload = 0x00802;
                    MySource.LockTag_EPC_C1G2(MyTags[i], Payload);

                }
            }
            Console.WriteLine("Press a key to end the program.");
            Console.ReadKey();
            MyReader.Disconnect();
        }
    }
```

LockTag_EPC_C1G2 method is a member of the *DLRFIDLogicalSource* class.

Killing Gen2 tags

The EPCGlobal Class1 Gen2 protocol provides a way to kill tags, that means tags, after the kill process, are no more readable. This process is password protected: you can kill only tags with a non-zero kill password. So, once you have set the kill password using the *WriteTag_EPC_C1G2* method on the RESERVED memory bank at address 0, you can use the *KillTag_EPC_C1G2* method of the *DLRFIDLogicalSource* class to kill the tag.

The tag that has to be killed is selected just by matching the complete EPC along with the Kill Password.

Differently from the previously described methods, there is not a secured version of the kill method since the kill command is always protected by a specialized kill password.

Handling General purpose Inputs/Ouputs (GPIOs)

Almost all DL RFID readers are provided along with some programmable GPIOs (for further details on GPIOs for a specific DL reader please refer to the correspondent reader's manual).

On each GPIO you can perform two basic operations:

- Selects its direction (INPUT, OUTPUT).
- Sets its value (HIGH, LOW).

GPIO direction can be set and read by means of a 4 byte long bitmask used in conjunction with the *GetIODirection* and *SetIODirection* methods. Each bit in the bitmask represents the GPIO direction: a '0' value means INPUT, a '1' value means OUTPUT.

A 4 byte long bitmask in conjunction with GetIO and SetIO functions is also used to set/get GPIO's OUTPUT/INPUT values:

- Each upset bit in the bitmask sets the correspondent GPIO to a HIGH logic voltage.
- Each cleared bit in the bitmask sets the correspondent GPIO to a LOW level voltage.

High and low voltage value varies with the specific DL's reader used. Please refer to the specific DL reader's manual to know the effective voltage values adopted.

Let's suppose we have a reader with 4 GPIOs available and we want to program GPIO0, GPIO1 as OUTPUT and GPIO2, GPIO3 as INPUT : the bitmask associated to this settings is 0000..0011b and the code is as follows:

```
int MyDirections = 0x3; //hex format

MyReader.SetIODirection(MyDirections);
```

Coming to the GetIO and SetIO methods, we can get GPIO's input status and set GPIOs' output status as follows:

```
int InputVal = 0x0;
int OutputVal = 0x2; //GPIO0 output value : 0,GPIO1 output value : 1

MyReader.GetIO(&InputVal);
MyReader.SetIO(OutputVal);
```