TEGO OS
API GUIDE

January 26th, 2018

Version 0.1.1

460 Totten Pond Road

Waltham, MA 02451 USA

T: +1.781.547.5680

Document Revisions

Date	Version Number	Document Changes

Table of Contents

1 Introduction

1.1 Scope and Purpose

This document is the introductory guide to the Tego OS, a platform for enabling smart asset solutions. This platform comprises various functional areas which are used internally within applications, but the extent of the functionality supported externally is the reader framework discussed here.

The purpose of the document is not to be a tutorial re how to use the API – there are demonstration projects that do that – but to provide an overview of the main aspects of the API for orientation purposes. For detailed implementation, the demonstration projects and interface documentation should be used.

2 OS libraries

The OS comprises a number of portable and platform specific libraries compiled against a .NET 4.5 target API level.

The main library is a profile 7 portable class library named Tego.dll and which is usable across all supported platforms (UWP, Android, iOS).

It is accompanied by a platform specific library, named Tego.Platform.X where X is the name of the target, for example iOS. A bait library named Tego.Platform.Bait can be used if the application needs to reference Tego.Platform functionality directly for compilation purposes using the bait and switch pattern.

Although Tego.Platform functionality is intended for internal use only, it is required to allow the Tego assembly to operate as it provides functionality that is not otherwise available in portable class libraries. A separate Tego.Platform assembly will likely be eliminated in a subsequent .NET Standard 2 based release.

Additionally, the OS may provide one or more reader libraries as follows:

- Tego.Devices.Zeti (plus a platform specific Tego.Devices.Connection.X library)
- Tego.Devices.TechnologySolutions
- Tego.Devices.Llrp
- Tego.Devices.Simulated

The following third-party libraries should be referenced in the application via Nuget as they are used internally within the OS:

- PCLCrypto version 2.0.147
- Newtonsoft.Json version 10.0.3
- Rda.SocketsForPCL version 2.0.2

3 Gen 2 RFID

3.1 Interface to readers

Reader and other Gen 2 functionality may be found in the Tego.dll assembly and resides primarily in the Tego.Rfid.Gen2 namespace.

3.1.1 Purpose of Reader and IReaderImplementation

The OS provides a common interface that allows many different types of reader to be interchanged without necessitating any code change within the application. This is similar to the concept behind the LLRP interface, but many readers (especially hand-held readers) do not implement LLRP, and LLRP is much more complex to use, requiring many lines of code for even simple operations.

The Reader class provides the primary application interface to readers and provides value-adding functionality over a typical reader API that includes:

- Thread safe access during execution;
- Reliable read and write operation via chunking and retry algorithms;
- Progress and activity reporting;
- Optimization algorithms;
- Reader interchangeability without code change.

The IReaderImplementation interface is used to implement a new reader. Its purpose is to translate the manufacturer API into a reader independent interface and to report the capabilities of the implemented reader, e.g. the granularity of selection that the reader can support from EPC only on an individual tag through to compound selects with complex bit masks.

The Reader class used by the application wraps an inner reader implementation and changes its algorithms according to the capabilities reported by each reader. As a result, both interfaces have substantial commonality, for example both use identical interface paradigms for naming banks, etc.

In later releases, applications will only need to use the Reader interface. For now, a very small subset of IReaderImplementation may need to be accessed to perform functions such as accessing hand-held triggers as shown in the examples solution, but external use of the IReaderImplementation beyond those examples is not supported.

3.1.2 Reader

3.1.2.1 Connection, initialization and disconnection.

Each Reader object manages an individual reader and may be maintained for the life of the application irrespective of reader connection state. Instantiation of the reader is not a ‘heavy’ or slow process and maintaining the same instance gives implementers the

ability to preserve state and avoid unnecessary set-up if e.g. the reader is connected and disconnected on multiple occasions.

There are a number of methods that can be used to instantiate a reader, but the easiest way is to use ReaderManager to create the type of reader being used. For example:

```
var rm = new ReaderManager();
Reader reader1 = rm.CreateReader(typeof(ZetiReader));
```

Post-instantiation, a call to the Connect method should be is made, which may include connection data if needed, such as a reader IP address or Bluetooth identity.

```
reader1.Connect(address);
```

After a successful call to Connect, the reader can report its capabilities (via the Capabilities property) and expose its settings (via the Settings property).

Capabilities are primarily used within the OS and of less interest to applications. Settings are reader specific and will vary from reader to reader. After connection, settings will either be preserved from the last use or will be in a default state if this is the first run. At this stage, settings may be changed but are not actually applied.

Then a call to Initialize should be made, which will apply the settings to the reader and perform any other set up needed for the reader to operate.

```
reader1.Initialize();
```

Thereafter, the reader is used as needed until finished, whereupon the Disconnect method is called, which will disconnect the reader and save its settings.

N.B. some mobile devices will deactivate Bluetooth links or other capabilities when the app is minimized. In such cases it is desirable to connect and disconnect the reader when restore and minimize events are seen in order to ensure proper operation.

3.1.3 IReaderImplementation

Reader implementation is outside the scope of this release.

3.1.4 Execution

Execution commands take four parameters as follows:

- Selection criteria;
- List of operations;
- Execution settings;
- Expected tag population.

The OS provides for two modes of operation: synchronous and asynchronous.

In synchronous mode, the Execute method is used to initiate operation by the reader, and all results are returned when the operation has ceased. Execute is therefore a blocking command which will block the calling thread during execution.

```
IList<TagResponse> tagResponses =
```

```
reader1.Execute(selection, operations, settings, 10);
```

In asynchronous mode, the StartExecution method is used to initiate operation, and results are returned as each tag responds via the TagResponse event. StartExecution is a non-blocking command which returns immediately and in this release of the OS, the events are raised in a background thread. The ExecutionComplete event is raised at the end of execution.

```
reader1.TagResponse += HandleTagResponse;
reader1.ExecutionComplete += HandleExecutionComplete;
reader1.StartExecution(selection, operations, settings, 10);
```

This execution API is more complete and hence more complex than typical reader API's which target one operation at one tag: it is designed to facilitate N operations being executed on M tags, which facilitates and optimizes reading user memory from large tag populations, for example, the maintenance history of parts within a facility.

A simpler and more conventional API known as the "One Tag" API is available for use with single tags.

3.1.4.1 Multithreaded operation

The Reader execution API is thread safe, allowing multiple threads to make concurrent calls to the reader without causing an error as would ordinarily occur. The OS does not allow the reader to execute the operations simultaneously as Gen 2 technology does not allow this, but will synchronize the various execution requests so that only one is running at a time.

Multithreaded operation can simplify application development where the read or write activities for multiple tags are interleaved with slow activities such as getting data from a server, or running computationally intensive algorithms such as compression or encryption. Reader 'dead' time, which would ordinarily occur while the slow non-reader activity is running, is now no longer dead but is available for other tags being processed in other threads to use without needing to 'schedule' the reader within the application.

3.1.5 One Tag Methods

The one tag methods are special methods designed to provide reliable operation and progress reporting when communicating with individual tags. In the Reader interface the one tag methods are:

- OneTagRead;
- OneTagReverseRead (not supported in this release);
- OneTagWrite.

The one tag methods are called synchronously and will reliably read and write tags, retrying when an error is encountered. Should it be necessary to stop this retry loop because the read or write cannot resolve the error, then execution can be canceled by calling the RequestCancel method.

The OneTagRead method ordinarily returns the data read, but will return null if execution was canceled. The OneTagWrite method ordinarily returns true if the data was correctly written, and false if execution was canceled.

Both methods will cause the Progress event to be raised, which provides an ongoing progress report including number of words complete, together with received signal strength, operation results, error counts, and other information.

3.1.6 ISelection

Selection allows the reader to target one or more tags when executing.

The selection can be a simple individual EPC or more complex selection of tag(s) that match defined patterns in EPC, User or TID memory. Examples of use include:

- Matching the first byte of EPC codes to select e.g. all SGTIN-96 tags;
- Matching the first word of user memory to select all unprogrammed tags;
- Matching the manufacturer mask in the TID to select all Tego tags.

Additionally, it is possible to compound these selections via ‘and’, ‘or’ and ‘not’ operators, allowing selection of e.g. all unprogrammed Tego tags.

Examples of various selections are shown in the example solution. Note though that the RFD8500 (Zeti) reader has a minimum select requirement of nine bits and so matching the EPC first byte is not shown.

3.1.7 Operations

The OS allows multiple operations to be applied to the tags that are selected by the selection criteria.

The simplest operation is the search operation (Operation.Search) which finds the individual tags using an inventory process. As search is the extent of any operations performed, it should appear on its own within the list of operations.

The two most important operations other than search are the ReadOperation and the WriteOperation (although other operations will be supported in future releases). Multiple read and write operations (to the extent allowed by the reader capabilities) can be executed on each tag seen and are run in order until an operation fails, whereupon subsequent operations in the list are discontinued until the next tag is found.

3.1.8 Execution settings

Execution settings include the following options:

- Antennas;
- Session;
- Trigger;
- Filter.

In this release, only session S0 and the trigger should be used. The trigger is used to specify when reader execution stops (execution starts immediately when one of the execute methods is called).

Supported triggers include:

- RoundsTrigger;
- TimedTrigger;
- TimedRoundsTrigger.

Rounds refers to the number of times the tag is queried, after which execution stops. Time is the time in mS after which execution stops.

In asynchronous mode, execution may be stopped by calling the StopExecution method if an extremely long stop trigger was provided.

3.2 EPC code support

The Tego.Rfid.Tds namespace include various classes to encode decode EPC's in accordance with the Tag Data Standard (version 1.9).

An EPC code may be held within an EpcCode instance, which comprises two important properties: Hex and Encoder. Hex is the hexadecimal representation of the EPC code and encoder is the encoder used. Both of these properties are mutually dependent, e.g. setting a hex value may cause the encoder to change and visa-versa. Changes are reported through the conventional INotifyPropertyChanged interface (also INotifyDataErrorInfo).

For example, setting the Hex value to a value beginning with 0x30 will cause the Encoder property to change to an Sgtin96Encoder as 0x30 is the header byte for an sgtin-96 EPC code.

The OS includes encoders for all EPC codes defined in the Tag Data Standard, and each encoder has properties representing the various EPC fields defined for that encoding scheme. Encoders also report changes via the INotifyPropertyChanged and INotifyDataErrorInfo interfaces.

N.B. For binding purposes the EpcCode will remain the same instance even when the EPC is changed, but its Encoder property may change instance when the encoder type changes.

EpcCode objects may be output in a variety of GS1 URI formats based on a selector argument to the ToString method as follows:

- 'R' or 'r' outputs the raw URI format;
- 'T' or 't' outputs the Tag URI format;
- 'U' or 'u' outputs the ID URI format;
- 'X' or 'x' outputs the hexadecimal value of the code.

The EpcCode may be set by setting its Hex property or by calling the SetFromTagUri method which decodes a tag URI and sets the encoder, encoder fields and hex accordingly.