Technology
Solutions (UK) Ltd

UHF ASCII 2.0
MOBILE SDK (ANDROID)
USER GUIDE

www.tsl.uk.com
Design • Development • Manufacture

CONTENT

Introduction 3
API Introduction 4
    Command Parameters 4
    Executing a Command and Response Handling 4
Setting up an Android project 5
    Android Developer Tools (ADT) Support 5
    Android Studio Support 6
Sample Applications 7
    Overview 7
    Inventory 8
        Inventory Sample Code Overview 9
    Trigger 9
        Trigger Sample Code Overview 10
    ReadWrite 11
        ReadWrite Sample Code Overview 12
    Licence Key 13
        Licence Key Sample Code Overview 14
About TSL 15
    About 15
    Contact 15

History

Version	Date	Modifications
0.8	29/07/2013	First Release
1.0	18/12/2013	Added sections for the new Inventory, Trigger and ReadWrite sample projects
1.1	11/11/2014	Added Licence Key section – this replaces the obsolete Inventory_Old section, Updated the Inventory section to describe use of DeviceProperties

INTRODUCTION

ASCII 2.0 is a recreation of the ASCII protocol Technology Solutions originally created for their range of UHF readers. The aim of the original protocol was to enable rapid testing of UHF commands from a terminal command prompt connected to a Technology Solutions UHF reader. As more readers implemented the protocol the ASCII protocol became useful to support multiple platform types without having to support the full binaryh API.

The main objectives of the ASCII 2.0 protocol was to maintain the ease of use for experimenting at the command prompt but to add more structure to the commands and responses to make it easier to command from an application. This has been achieved with the following:

- A defined start sequence to every command (e.g. ".iv")
- A consistent way to pass parameters to a command
- Simple framing of all commands and responses with a header and line terminator
- Signalling the termination of a response with an empty line
- Signalling the error for a command with a return code "ER: xxx" or "OK"

The ASCII 2.0 protocol can be implemented in most modern languages very simply. Once a connection is established to the reader a command line is sent to the reader. Lines of response are then read from the reader until an empty line is received signalling the end of the response. The line preceding the empty line will start "OK:" or "ER: xxx" indicating whether the command executed successfully.

All ASCII commands start with a period '.' followed by two characters to identify the command e.g. ".iv" for inventory. The command is terminated with an end of line (Cr, Lf or CrLf). The rest of the command line can contain parameters with or without values. A parameter starts with a minus '-'symbol followed by one or more characters to identify the parameter then followed by the parameter value.

".iv –x " perform an inventory, reset the command parameters to default

An ASCII response is a sequence of lines terminated by an empty line. The each line has a two character header followed by a colon ':' the remainder of the line is the value corresponding to the header.

"ME: this is a message" is a message response as part of an ASCII response.

An ASCII response always starts with the command started "CS:" line and ends with either "OK:" if the command completed successfully or "ER: xxx" if it failed, xxx represents an error code. If the "ER:" is sent is may also be preceded by an "ME:" line with a human readable error.

API INTRODUCTION

The purpose of the API for ASCII 2.0 is to provide a library of commands that enable a developer to rapidly build the commands to send to the reader and also to interpret the responses.

There is Javadoc documentation in the AsciiProtocol Library project to browse all the classes, interfaces and members.

COMMAND PARAMETERS

The *IAsciiCommand* interface represents a command to be sent to the reader and there are classes that implement this interface for all of the common commands. These command classes have properties to set the parameters to send to the reader.

All the parameters for a command are optional, where a parameter is not specified the reader uses its cached value of the parameter. Every parameter has a “Not specified” value, this is the value of the parameter that should be used when that particular parameter should not be sent to the reader.

Where commands have parameters they have a reset to defaults (‘x’) parameter to reset all parameters to their defaults in the reader before executing the command. Note executing the command includes updating any parameter on the command line to the specified value. Commands also have a read parameters parameter (‘p’) to read the current value of all the commands parameters.

Commands can also have a take no action parameter (‘n’). When specified the command parameters can be read or modified without performing the actual command (e.g. an inventory can be configured without actually performing the inventory, the inventory can then be performed by another inventory command without specifying any parameters.

EXECUTING A COMMAND AND RESPONSE HANDLING

An instance of *IAsciiCommandExecuting* is used to execute an *IAsciiCommand* with the *executeCommand* method. *AsciiCommandExecutorBase* is the base implementation of *IAsciiCommandExecuting*. It executes a command by sending a command line to a reader using its *Send* method. Responses from the reader are processed by the *ProcessReceivedLines* method.

AsciiCommander is an implementation of *IAsciiCommandExecuting* that extends *AsciiCommandExecutorBase* to send a line to a serial port and received lines are passed to the *processReceviedLines* method.

An ASCII command can either execute synchronously or asynchronously. The *IAsciiCommandExecuting* has a chain of *IAsciiCommandResponders* which get called in sequence to handle each line that is received in the response. Each responder has the opportunity to mark the line as handled so no further responders get notified. There is a *SynchronousDispatchResponder* which when inserted into the chain can relay responses to the executing command. For this to work the command must also implement a responder to capture its own response. This is achieved with the *AsciiSelfResponderCommandBase* class. When a command is its own responder the command executes synchronously and *executeCommand* blocks until the response to the command has been received. For a command to execute synchronously its *synchronousCommandResponder* should be set to itself (this).

If an executing command does not have its *synchronousCommandResponder* set then *executeCommand* will return as soon as the command is sent and the *SynchronousDispatchResponder* in the responder chain will have no action. It is then down to other responders in the chain to handle the response to the command. The command has executed asynchronously.

As each command implements its own responder the commands themselves can be placed into the responder chain to capture the responses of other instances of the same command that are passed to *executeCommand*. In addition custom responders can be implemented like the *LoggerResponder* (inserted at the start of the chain to log but not handle all responses).

SETTING UP AN ANDROID PROJECT

The following sections describe integrating the Ascii Command library into the two Google supported Android development environments.

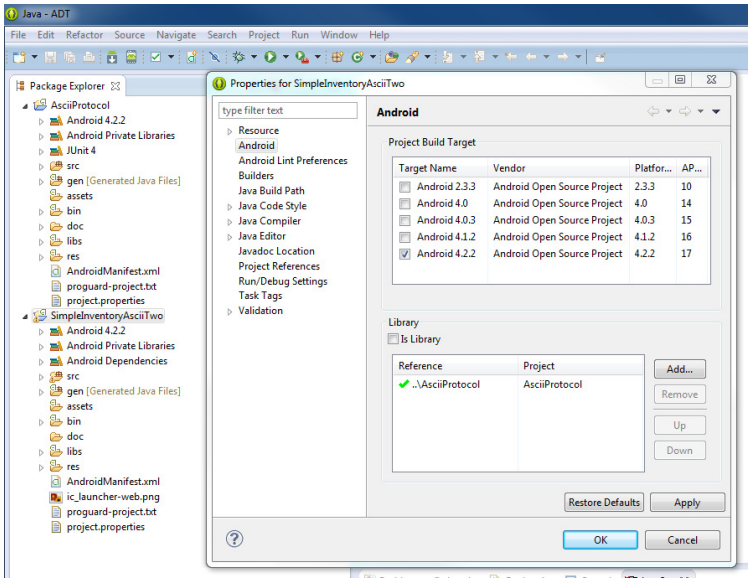
The API currently only supports *Bluetooth*® connections so you will need to add the following permissions to the application's AndroidManifest.xml

```
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
<uses-permission android:name="android.permission.BLUETOOTH"/>
```

ANDROID DEVELOPER TOOLS (ADT) SUPPORT

An Android Library Project is provided to communicate with TSL ASCII 2.0 *Bluetooth*® Devices. It can be used as follows:

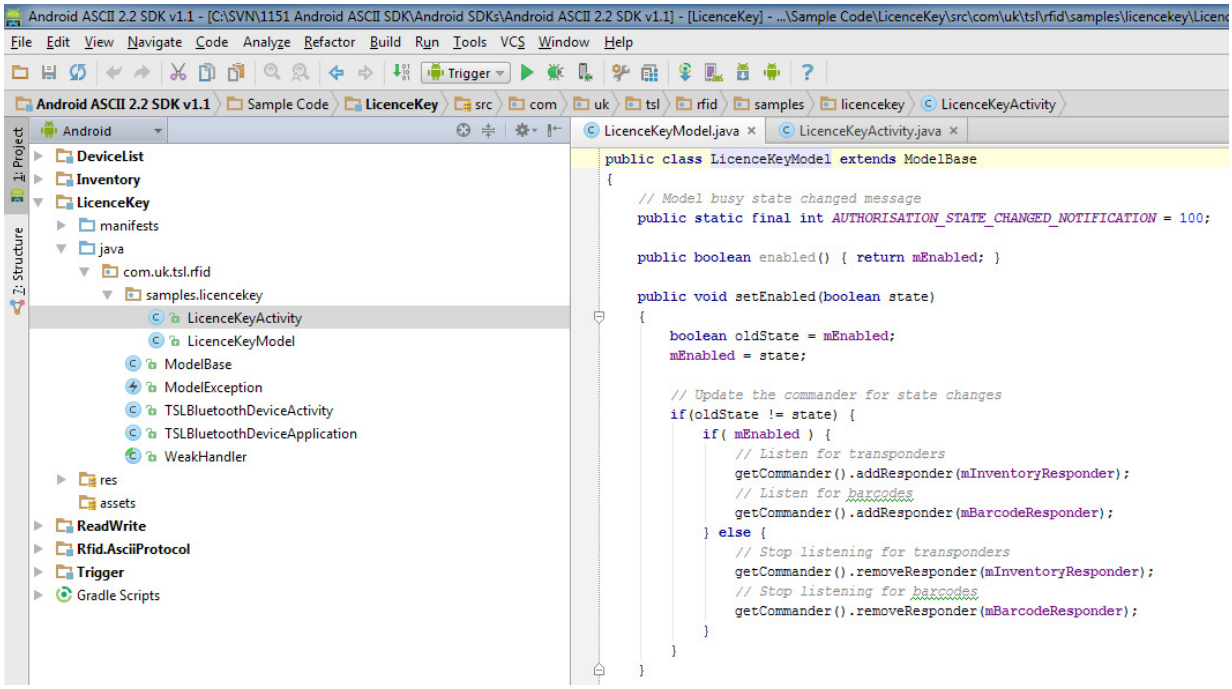
- Import the AsciiProtocol Library project into the Android workspace containing the target project
- In Properties->Android for the target project, add the AsciiProtocol Project



The AsciiProtocol library requires the android-support-v4.jar and, as supplied, this is referenced in the appcompat_v7 folder included with the SDK. All Sample code projects reference this folder too. You will need to adjust the java build path and/or the project dependencies to use the corresponding .jar files in your project.

ANDROID STUDIO SUPPORT

At the time of writing Android Studio had reached Beta v0.8.14.
The supplied SDK can be imported into Android Studio and will allow any of the sample projects to be built and tested.



To use the Rfid.AsciiProtocol library in your project:

- Import Rfid.AsciiProtocol
- Add the imported module as dependencies to your App module.

The next section describes the sample code projects provided to demonstrate the use of the AsciiProtocol Android Library Project.

SAMPLE APPLICATIONS

OVERVIEW

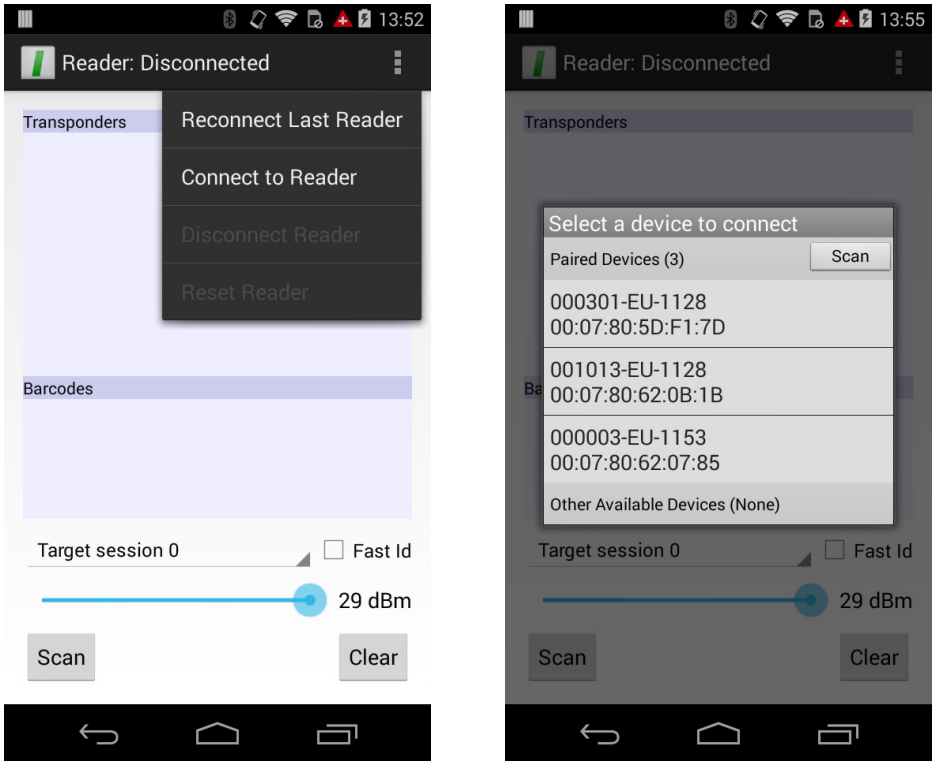


FIGURE 1: the common reader menu (left) and the Connect to Reader dialog (right)

The sample applications shipping with the SDK are all built using the same basic template which provides the reader connection, disconnection and reset operations from the action menu. The operations available are:

- Reconnect Last Reader – Attempts to connect to the last reader that was successfully connected to this App
- Connect to Reader – Presents a device selection dialog from which an existing, paired device can be selected or a new device can be paired
- Disconnect Reader – Disconnects from the currently connected reader
- Reset Reader – Resets the currently connected reader to the factory default settings

All samples use a main activity derived from *BluetoothDeviceActivity* which performs the *Bluetooth®* connection/disconnection logic and instantiates an *AsciiCommander*. The samples also use a custom *BluetoothDeviceApplication* class to hold a reference to the current *AsciiCommander*. The *DeviceList* project provides the logic and UI for discovering and selecting the reader to use.

The main Activity for all sample projects provides the default configuration of the *AsciiCommander* to include a *LoggerResponder* and a *SynchronousResponder* in the *OnCreate()* method. The *Activity* then creates the model which provides any additional responders as needed (often through an *initialise()* method or similar).

The reader specific operations are separated out into a Model class derived from *ModelBase*. The *ModelBase* class uses a Handler property to communicate with the UI thread and provides a *performTask(Runnable task)* method to allow long running operations (including synchronous ASCII commands) to be executed on a non-UI

thread. A *Busy* property is used while *performTask()* runs to signal via the Handler when the Task is active or not. A *ModelException* is provided to indicate failure of a Task.

INVENTORY

The Inventory sample project demonstrates some of the operations available using the *InventoryCommand*. It also illustrates use of the *BarcodeCommand* as a Responder for capturing barcodes from the reader.

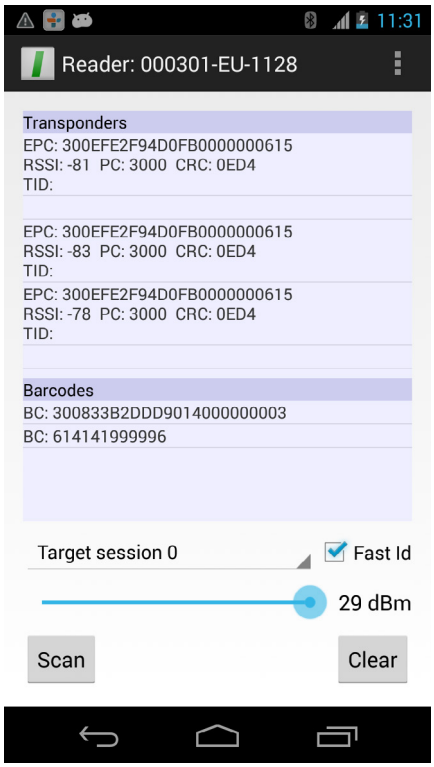


FIGURE 2: The Inventory sample application

The Inventory application uses the default action of the device trigger to produce inventories on single press and barcode scanning from the double press action. An inventory operation can also be initiated from the on-screen *Scan* button.

A pop-up menu can be used to configure the session used for inventories and the slider bar can be used to adjust the antenna output power.

Scanned tags will appear in the upper list area and for each transponder will contain information about the signal strength, the PC control word and the CRC value. If the tag supports it[1] and the *FastId* checkbox is checked then the information from the TID memory will also be displayed.

Barcode scans appear in the lower display area preceded by the prefix 'BC:'.

[1]For example, Impinj Monza 5 Transponders

INVENTORY SAMPLE CODE OVERVIEW

The *InventoryModel* uses two instances of *InventoryCommand*, one (*mInventoryCommand*) is used to issue inventory commands and configure the reader and the other (mInventoryResponder) is used as a *Responder* to capture inventory responses. A *BarcodeCommand* is also used in the role of *Responder* to capture incoming barcode scans.

The model adds and removes the responders from the current *AsciiCommander* to enable/disable (respectively) the reporting of scanned transponders/barcodes to the *InventoryActivity* via the model's *Handler*.

The inventory responder sets an *ITransponderReceivedDelegate* to handle each of the transponders received, extracts some of the returned information and uses the model's *sendMessageNotification()* method to present this to the UI. The barcode responses are handled in a similar manner by the barcode responder.

The *InventoryModel's updateConfiguration()* method uses the *InventoryCommand's TakeNoAction* property to ensure that the trigger-initiated inventory matches that of the UI Scan button. As the session, output power or fast id setting changes in the UI the *updateConfiguration()* method is called and the reader's inventory (.iv) command parameters are set without performing an (inventory) action.

The *InventoryActivity* also demonstrates the use of the *AsciiCommander getDeviceProperties()* method. The returned *DeviceProperties* instance contains information about the device that differs from the default ASCII protocol such as the antenna power limits. This is used to adjust the limits of the UI's power level trackbar when the connected reader changes e.g. maximum of 25 dBm for an 1153 reader or 29 dBm for the 1128 reader.

TRIGGER

The Trigger sample project demonstrates monitoring and responding to the change of state of the trigger switch. It also demonstrates changing of the default trigger behaviour.

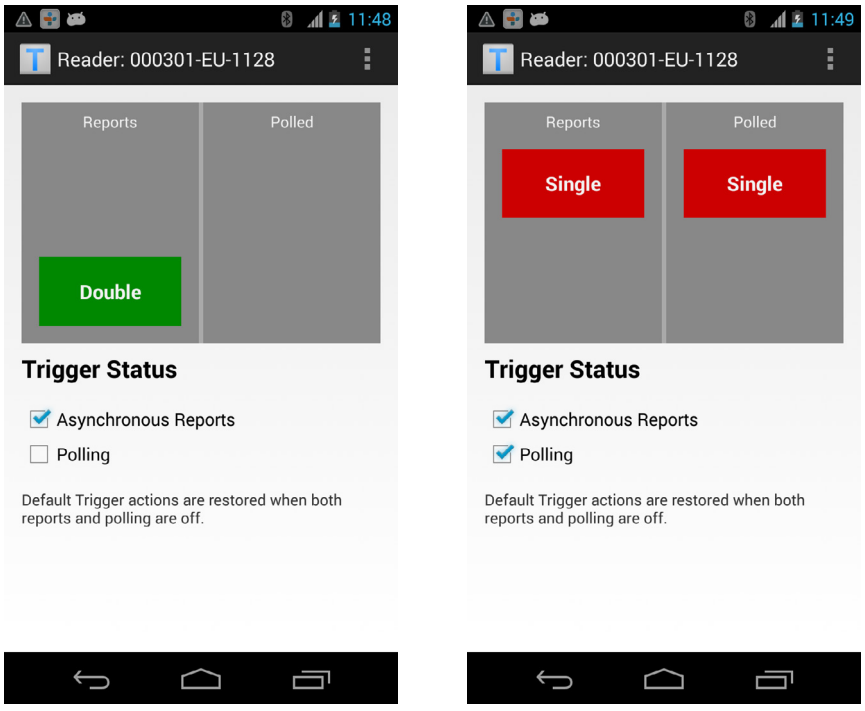


FIGURE 3: The Trigger sample application

The Trigger application provides an on-screen display of the state (Single, Double or Off) of the device trigger switch and determines the switch state using two methods:

- asynchronous switch state reports
- continuous polling of the switch state

Asynchronous reports are also indicated by an accompanying beep from the device.

When any of the trigger status options are selected the default actions of the trigger are disabled. When both of the monitoring options are disabled the default actions of the trigger are restored.

TRIGGER SAMPLE CODE OVERVIEW

The Trigger application reader specific code is in the *TriggerModel* which determines changes of switch state using two methods:

- Via asynchronous switch state reports, set with the *SwitchActionCommand*
- By polling the switch state using the *SwitchStateCommand*.

The *TriggerModel* provides two properties to control the switch monitoring *AsyncReportingEnabled* and *PolledReportingEnabled*.

When either monitoring method is enabled the *SwitchActionCommand* is used to change the default trigger switch behaviour to do nothing. This command is also used to enable the asynchronous reporting when needed.

The *TriggerModel* base class messaging mechanism is extended to provide unique notifications to the UI for asynchronous and polled switch state change notifications.

For monitoring asynchronous reports an instance of *SwitchResponder* is used which captures the reader's switch changed notifications and passes them to its *ISwitchStateReceivedDelegate's switchStateReceived()* method. When the *switchStateReceived()* method is executed the UI is notified using the model's message system and an (asynchronous) *AlertCommand* is configured and executed to provide audible feedback of the switch state. This responder is added to the current commander's responder chain from the model's *initialise()* method.

The polling method uses the model's *Handler* to execute a *Runnable* periodically. The *Runnable* instance uses a synchronous *SwitchStateCommand* to interrogate the reader for the current switch state. When the command completes the *SwitchStateCommand's* State property holds the trigger state which is passed to the UI via the model message system. Note that this approach to periodic polling causes the *SwitchStateCommand* to execute on the UI thread and, while the command will usually execute very quickly, this may introduce some lag to the UI responses.

READWRITE

The ReadWrite sample project demonstrates reading from and writing to transponders. The commands can operate on one or more transponders simultaneously depending on the EPC specified.

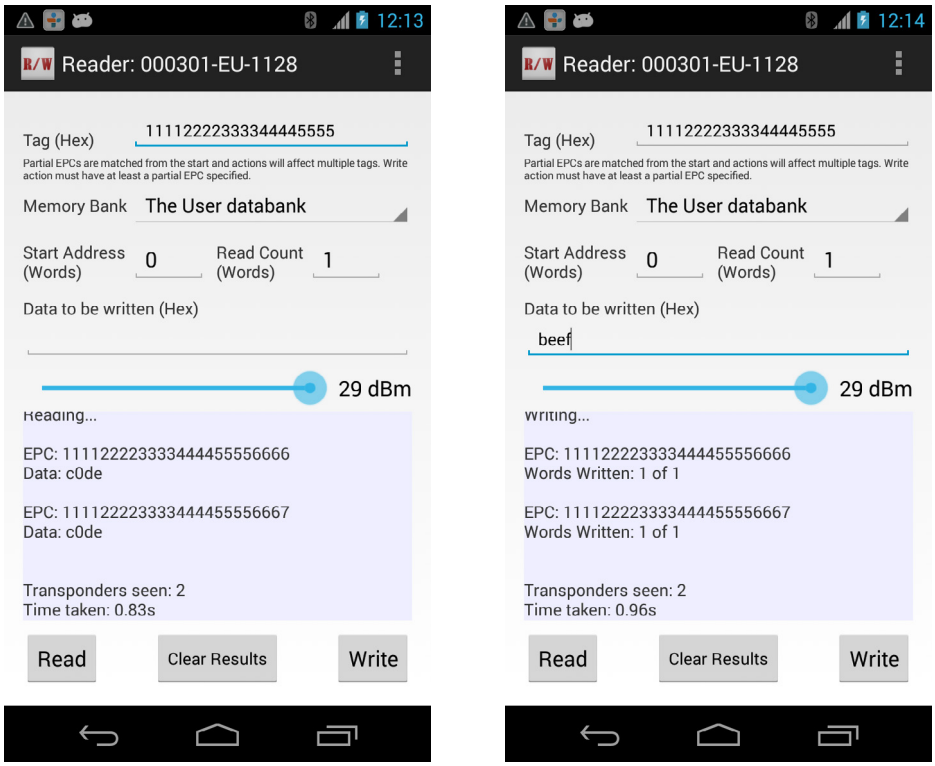


FIGURE 4: The ReadWrite sample application - Reading (left) - Writing (right)

The *ReadWrite* application provides an interface for configuring and executing operations that demonstrate the use of the *ReadTransponderCommand* and *WriteTransponderCommands*.

The interface fields and their usage are described below:

- **Tag (Hex)** – Enter either a full or partial EPC here. Partial EPCs are matched from the start of the EPC. As a safety precaution, Write commands must have at least a partial value entered.
- **Memory Bank** – Select the memory bank for the operation using the pop-up menu.
- **Start Address** – Specify the starting offset into the chosen memory bank in words.
- **Read Count** – Specify the number of words to be read here.
- **Data to be written** – Enter the data to be written here as a hex value.
- **Power Slider** – Adjust the Antenna power using the slider as desired.

Once the appropriate configuration has been set, the **Read** or **Write** buttons will execute the command.

The outcome of the command is reported in the display area above the buttons. This area can be cleared using the **Clear** button

READWRITE SAMPLE CODE OVERVIEW

The ReadWrite application's reader-specific code is in the *ReadWriteModel* which provides access to both a *ReadTransponderCommand* and a *WriteTransponderCommand*. Changes to the UI field values are used to change the appropriate command properties.

Pressing the Read or Write button will cause the appropriate command's *ISelectMaskParameters* properties to be set for the given EPC. The *ISelectControlParameters* and the *IQueryParameters* properties are set to use a session with a long persistence time and to select the target tags into the B state if an EPC is specified otherwise (for read command only) an inventory only of session 0, state A is specified.

Each command uses the *ITransponderReceivedDelegate* to report progress via the Model's messaging system. An external field is used to count the number of transponders seen. Note that the delegate will never be called if no transponders respond.

Since there are potentially many transponders that could respond to any given operation the *ModelBase.PerformTask()* method is used to execute the commands on a separate thread. This will send notification of the Busy property changes via the model's messaging system. As the operations are using synchronous commands the model will be 'busy' for the duration of the read or write.

Upon completion of the execution of the command the *reportErrors()* method is used to detect any command configuration errors and forward them to the UI's result area. Note that the isSuccessful command property will be false when the command does not report any transponders.

LICENCE KEY

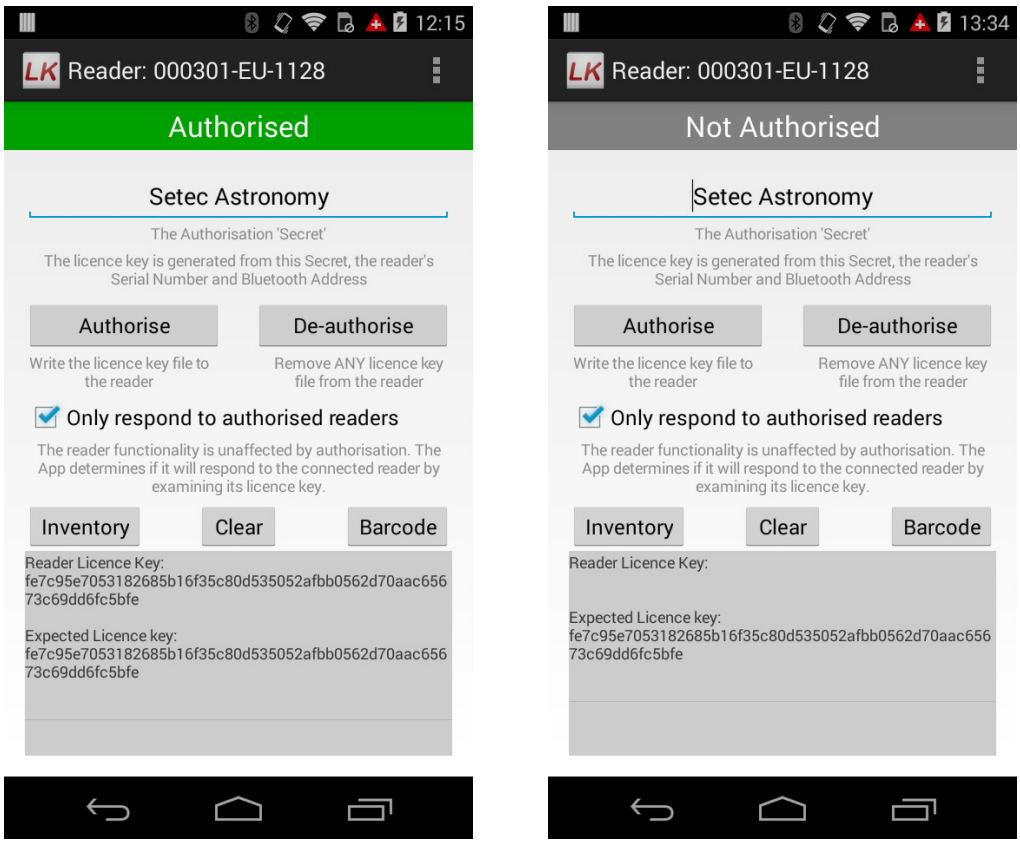


FIGURE 5: the Licence key Sample Project

The Licence Key sample application demonstrates setting and deleting a licence key from a reader using the *LicenceKeyCommand* which is available in *ASCII Protocol 2.2* and above. It also shows that the addition of a licence key to the reader does not affect how the reader operates it simply provides a method for software Apps to only respond to *authorised* readers.

The interface fields and their usage are described below:

Authorisation Indicator – At the top of the screen is an indicator that will show the authorisation state of the currently connected reader. An *authorised* reader is a reader that has a licence key generated from the current Secret, the device's serial number and the device's Bluetooth address. The authorisation indicator is grey for non-authorised devices and green if the reader is authorized.

Secret Text – This value is the salt for the licence key generated from the reader serial number and Bluetooth address using a SHA-256 hash

Authorise Button – Tap this button to write a licence key to the current reader

De-authorise Button – Tap this button to remove ANY licence key from the current reader

Only respond to authorised readers – When this is checked the *Inventory* and *Barcode* operations will only function if the current reader has a licence key derived from the current *Secret value*. If it is unchecked then the *Inventory* and *Barcode* actions works as expected irrespective of the readers licence key (if any). This check box also applies to responses generated using the devices trigger.

Inventory Button – Tap this button to perform an inventory of any tags in range

Barcode Button – Tap this button to initiate a barcode scan

Clear Button – Tap this button to clear the results area

Various messages are shown in the grey area at the bottom of the screen.

LICENCE KEY SAMPLE CODE OVERVIEW

The Licence Key application's reader-specific code is in the *LicenceKeyModel* which uses the *LicenceKeyCommand* to manipulate the licence key stored on the currently connected reader. The code also demonstrates the use of the *VersionInformationCommand*, the *InventoryCommand* and the *BarcodeCommand*.

The *LicenceKeyModel.authoriseReader()* method uses the *LicenceKeyCommand* to delete any existing key and to write the new value generated by the *createLicenceKey()* method (which produces a SHA-256 salted hash from the model's *getSecret()* property and the reader's serial number and Bluetooth address). The *LicenceKeyCommand* can only write a licence key when the existing key is blank so the *setDeleteLicenceKey()* property is used to achieve this.

The *deAuthoriseReader()* method uses the *LicenceKeyCommand* command to simply delete any existing licence key.

The *validateReader()* method sets the model's *isReaderAuthorised()* property by comparing the licence key read from the reader with one generated using the model's current *Secret* property value, the reader's serial number and its Bluetooth address. The latter two values are obtained using the *VersionInformationCommand* which now has a *getBluetoothAddress()* method which returns a valid value only when used with readers implementing ASCII protocol 2.2 or greater.

The remainder of the code shows how an App may make use of the *isReaderAuthorised()* property to restrict the options available within the App. In this App the *isOnlyAuthorisedReaderAllowed()* property also allows this restricted behaviour to be switched on and off for demonstration purposes.

Note: The licence key generated is a simple salted hash key which is in no way intended to represent security 'best practice'. This approach is susceptible to reverse engineering of the apk since the 'secret' must be in the app to generate the comparison licence key. Investigate asymmetric (public key) cryptographic algorithms for more secure solutions.

ABOUT TSL

ABOUT

TSL designs and manufactures both standard and custom embedded, snap on and standalone peripherals for handheld computer terminals. Embedded technologies include:

- RFID - Low Frequency, High Frequency & UHF
- *Bluetooth*® wireless technology
- Contact Smartcard
- Fingerprint Biometrics
- 1D and 2D Barcode Scanning
- Magnetic Card Readers
- OCR-B and ePassport

Utilizing class leading Industrial design, TSL develops products from concept through to high volume manufacture for Blue Chip companies around the world. Using the above technologies TSL develops innovative products in a timely and cost effective manner for a broad range of handheld devices.

CONTACT

Address:	Technology Solutions (UK) Limited, Suite C, Loughborough Technology Centre, Epinal Way, Loughborough, Leicestershire, LE11 3GE. United Kingdom.
Telephone:	+44 (0)1509 238248
Fax:	+44 (0)1509 220020
Email:	enquiries@tsl.uk.com
Website:	www.tsl.uk.com

ISO 9001: 2008