TECHNOLOGY SOLUTIONS UK LTD
part of HID

UHF ASCII 2 DESKTOP SDK (.NET) USER GUIDE

www.tsl.com
Design • Development • Manufacture

CONTENT

Overview

This document provides an introduction to the TSL ASCII 2 protocol and the .NET API provided to simplify development of applications using the protocol.

History

Version	Date	Modifications
0.9	09/09/2013	Initial Release of Desktop API
1.0	06/11/2013	Update for V1.0 release of API
1.01	02/12/2013	Code samples and explanation of the commands sample application
1.1	20/11/2014	Update for V1.1 release of API

INTRODUCTION

ASCII 2 is a recreation of the ASCII 1 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 binary API.

The main objectives of the ASCII 2 protocol were 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 protocol is described in the TSL ASCII protocol document available from the website and provided with the SDK.

The ASCII 2 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.

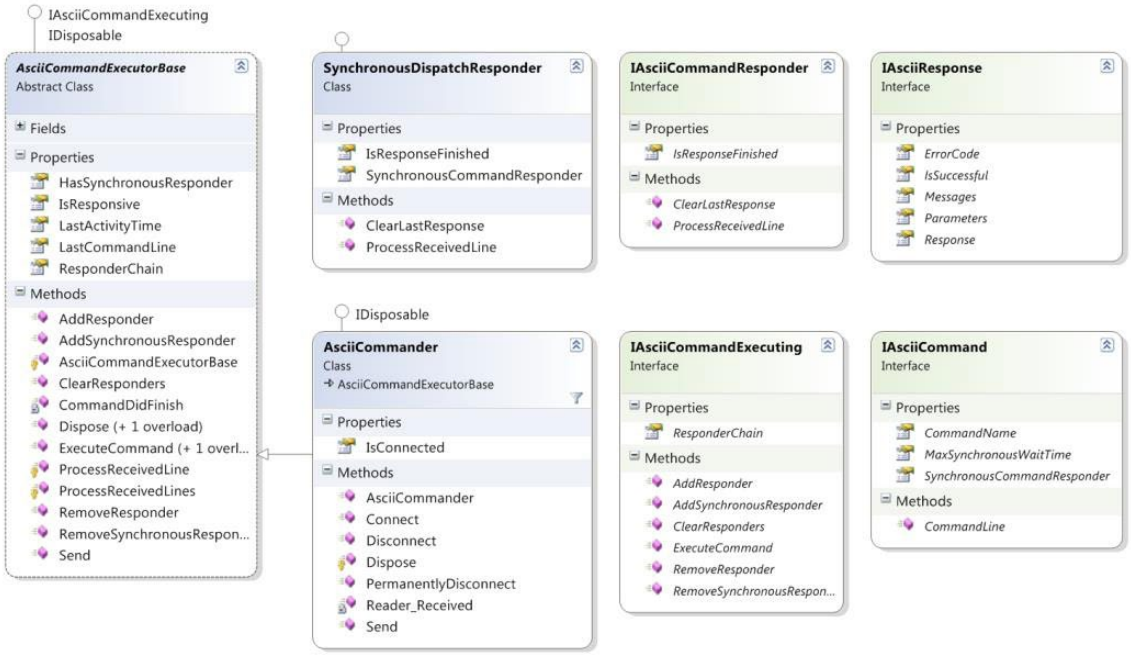
".iv –x " perform an inventory, reset the command parameters to default.

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 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 a help file (.chm) to browse all the classes, interfaces and members and an XML file is provided to support Visual Studio intelli-sense.

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 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. API commands use null-able parameters with the null value indicating that the parameter is not specified. By default, all API command parameters are not specified i.e. *null*.

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.

```
using TechnologySolutions.Rfid.AsciiProtocol;
using TechnologySolutions.Rfid.AsciiProtocol.Commands;

/// <summary>
/// Examples showing how to specify response parameters for a command.
/// <see cref="IResponseParameters"/>
/// </summary>
public class CommandParameterExamples
{
    /// <summary>
    /// Performs reader commands
    /// </summary>
    private IAsciiCommandExecuting executor;

    /// <summary>
    /// Read the current values of a command parameters
    /// </summary>
    public void ReadCurrentValues()
    {
        AlertCommand command;
        AlertDuration alertDuration;
        BuzzerTone buzzerTone;
        bool buzzerEnabled;
        bool vibrateEnabled;

        command = new AlertCommand();

        // specify "-p" to read parameters
        command.ReadParameters = true;

        // specify "-n" to 'take no action'
        // i.e. don't actually perform the alert just read parameters
        command.TakeNoAction = true;

        // as responder specified execute synchronously
        this.executor.ExecuteCommand(command, command.Responder);

        // as '-p' is specified response included a "PR" header with parameter values
        // As command executed synchronously it captured its own response
        // Command parsed the "PR" line and updated the command parameters to match
        alertDuration = command.AlertDuration.Value;
        buzzerEnabled = command.BuzzerEnabled == TriState.Yes;
        buzzerTone = command.BuzzerTone.Value;
        vibrateEnabled = command.VibrateEnabled == TriState.Yes;
    }

    /// <summary>
    /// Reset the parameters of a command to the reader's defaults
    /// </summary>
    public void ResetToDefaults()
    {
        AlertCommand command;

        command = new AlertCommand();

        // specify '-x' to reset parameters to defaults
        command.ResetParameters = true;

        // speicift '-n' to take no action
        command.TakeNoAction = true;

        // execute synchronously
        this.executor.ExecuteCommand(command, command.Responder);
    }

    /// <summary>
    /// Performs an alert as the alert is currently configured
    /// </summary>
    public void PerformTheCommandWithCurrentReaderValues()
    {
        AlertCommand command;

        // will all parameters null this is simply '.al'
        // (with library command and index)
        command = new AlertCommand();

        // perform asynchronously
        this.executor.ExecuteCommand(command, null);
    }

    /// <summary>
    /// Performs an alert modifying how the alert is executed
    /// </summary>
    public void PerformTheCommandWithSpecifiedValues()
    {
        AlertCommand command;

        // specify a low low beep and no vibrate
        command = new AlertCommand();
        command.AlertDuration = AlertDuration.Long;
        command.BuzzerTone = BuzzerTone.Low;
        command.BuzzerEnabled = TriState.Yes;
        command.VibrateEnabled = TriState.No;

        // command the reader to alert. long low beep no vibrate
        this.executor.ExecuteCommand(command, command.Responder);

        // new command so default back to '.al' only
        command = new AlertCommand();

        // command the reader to alert.
        // another long low beep as parameters specified on previous command
        this.executor.ExecuteCommand(command, null);
    }
}
```

FIGURE 1: Command Parameter Examples

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 *ProcessReceivedLines* method.

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).

```
public void TypicalResponderChain()
{
    BarcodeCommand barcodeCommand;
    InventoryCommand inventoryCommand;
    LoggerResponder loggerResponder;

    // using a barcode command in the responder chain
    // to capture barcode events when a barcode command is
    // executed asychronously or performed from a trigger press
    barcodeCommand = new BarcodeCommand();
    barcodeCommand.BarcodeReceived += this.Asynchronous_BarcodeReceived;

    // using an inventory command in the responder chain
    // to capture transponder events when a inventory command is
    // executed asynchronously or performed from a trigger press
    inventoryCommand = new InventoryCommand();
    inventoryCommand.TransponderReceived += this.Asynchronous_TransponderReceived;

    // logger responder outputs all responses to debugger
    loggerResponder = new LoggerResponder();

    // reset the current responders;
    this.executor.ClearResponders();

    // add first to ensure it sees all messages
    this.executor.AddResponder(loggerResponder);

    // add syncrhonous responder to allow commands
    // executed synchronously to capture a response
    // to itself before it gets handle by other
    // responders in the chain
    this.executor.AddSynchronousResponder();

    // capture asychronous barcode responses
    this.executor.AddResponder(barcodeCommand.Responder);

    // capture asynchronous transponder responses
    this.executor.AddResponder(inventoryCommand.Responder);

}

/// <summary>
/// Raised from the barcode command added to the responder chain
/// </summary>
/// <param name="sender">The event source</param>
/// <param name="e">Data provided for the event</param>
private void Asynchronous_BarcodeReceived(object sender, BarcodeEventArgs e)
{
    string barcode;
    DateTime timestamp;

    // barcode scanned
    barcode = e.Barcode;

    // requires barcode command to be executed with IncludeDateTime = TriState.Yes
    timestamp = e.Timestamp;
}

/// <summary>
/// Raised from the inventory command added to the responder chain
/// </summary>
/// <param name="sender">The event source</param>
/// <param name="e">Data provided for the event</param>
private void Asynchronous_TransponderReceived(object sender, TransponderDataEventArgs e)
{
    TransponderData transponder;
    string epc;
    bool more;

    transponder = e.Transponder;
    epc = transponder.Epc;
    more = e.MoreAvailable;
}
```

FIGURE 2: Setup responder chain

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 an *IAsciiCommandSynchronousResponder* must also be supplied at the time of execution. Each command in the API implements a responder to capture its own response and allows properties of the command to be updated by the response from the reader. 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.

If a command is executed without an *IAsciiCommandSynchronousResponder* 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.

```
using TechnologySolutions.Rfid.AsciiProtocol;
using TechnologySolutions.Rfid.AsciiProtocol.Commands;

/// <summary>
/// Show the difference between synchronous and asynchronous commands
/// </summary>
public class ExecutingCommandExamples
{
    /// <summary>
    /// Performs reader commands
    /// </summary>
    IAsciiCommandExecuting executor;

    /// <summary>
    /// Performs a command synchronously
    /// </summary>
    public void PerformCommandSynchronously()
    {
        BarcodeCommand command;
        string barcode;
        string error;

        command = new BarcodeCommand();

        // This call will block until the command is complete
        this.executor.ExecuteCommand(command, command.Responder);

        // test if command executed successfully
        if (command.Response.IsSuccessful)
        {
            // get the barcode scanned
            barcode = command.Barcode;
        }
        else
        {
            // get the error code why the command failed
            error = command.Response.ErrorCode;

            // Human readable message MAY be available for reason for error
            error = command.Response.Messages.FirstOrDefault();
        }
    }

    /// <summary>
    /// Performs a command asynchronously
    /// </summary>
    public void PerformCommandAysynchronously()
    {
        BarcodeCommand command;

        command = new BarcodeCommand();

        // This call will return as soon as the command is sent
        this.executor.ExecuteCommand(command, null);

        // cannot use BarcodeReceived event or Response property
        // as response is not passed back to executing command
    }
}
```

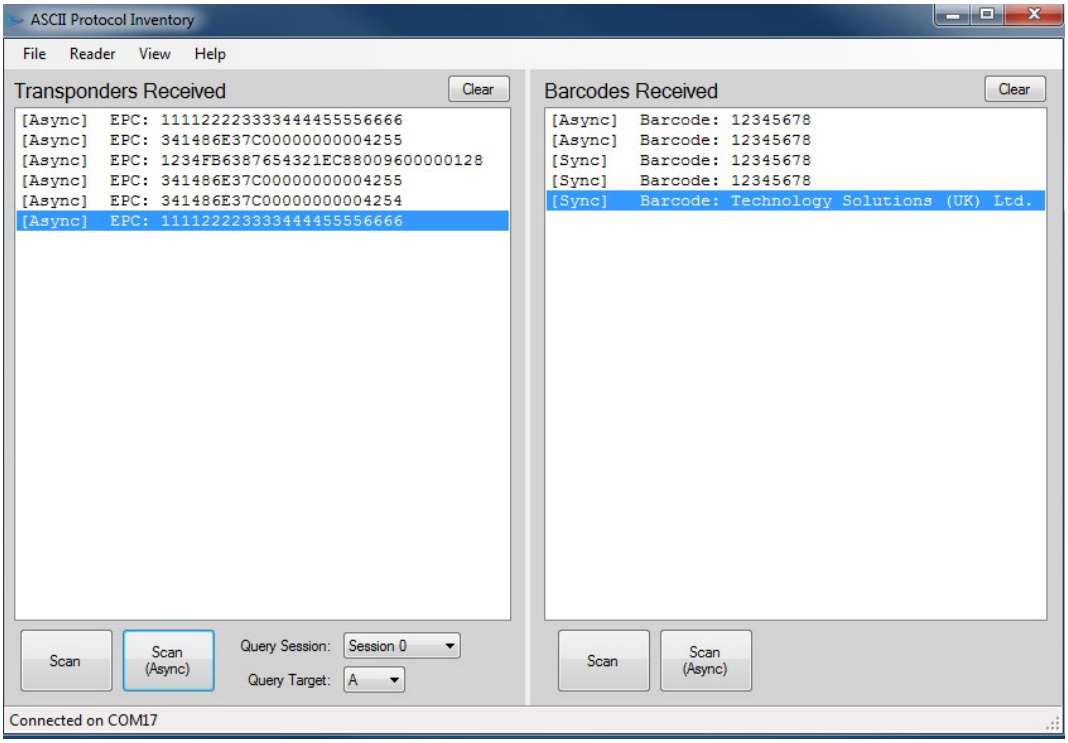
FIGURE 3: Executing Synchronous and Asynchronous commands

RELEASE HISTORY

SDK Version	ASCII API	Notes
0.9	2.0	First release. Supports ASCII Protocol 2
1.0	2.1	Updated to support all commands and additional features added for ASCII Protocol 2.1. Some changes to comply with Microsoft naming guidelines.
1.1	2.2	Updated to support commands and additional features added for ASCII Protocol 2.2 - Added the licence key command (.lk) - Switch action command (.sa) gains repeat delay parameters - Version information command gains Bluetooth address parameter (BA:)

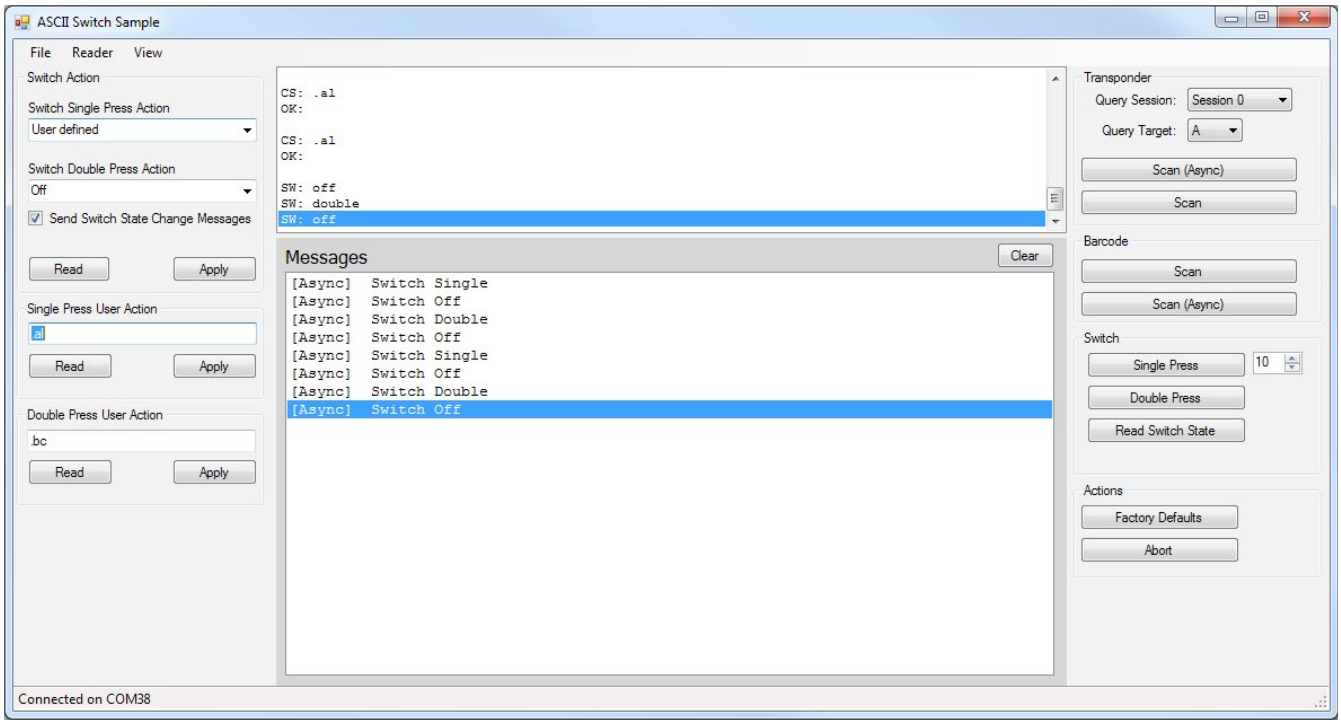
SAMPLE APPLICATIONS

ASCII PROTOCOL INVENTORY



The ASCII protocol Inventory sample application demonstrates using the .NET API to command devices that support the Technology Solutions ASCII 2 protocol to perform inventory and barcode operations. It is described in the document: *ASCII Protocol Inventory Sample Application (Desktop) User Guide* provided with this SDK.

ASCII PROTOCOL SWITCH SAMPLE APPLICATION

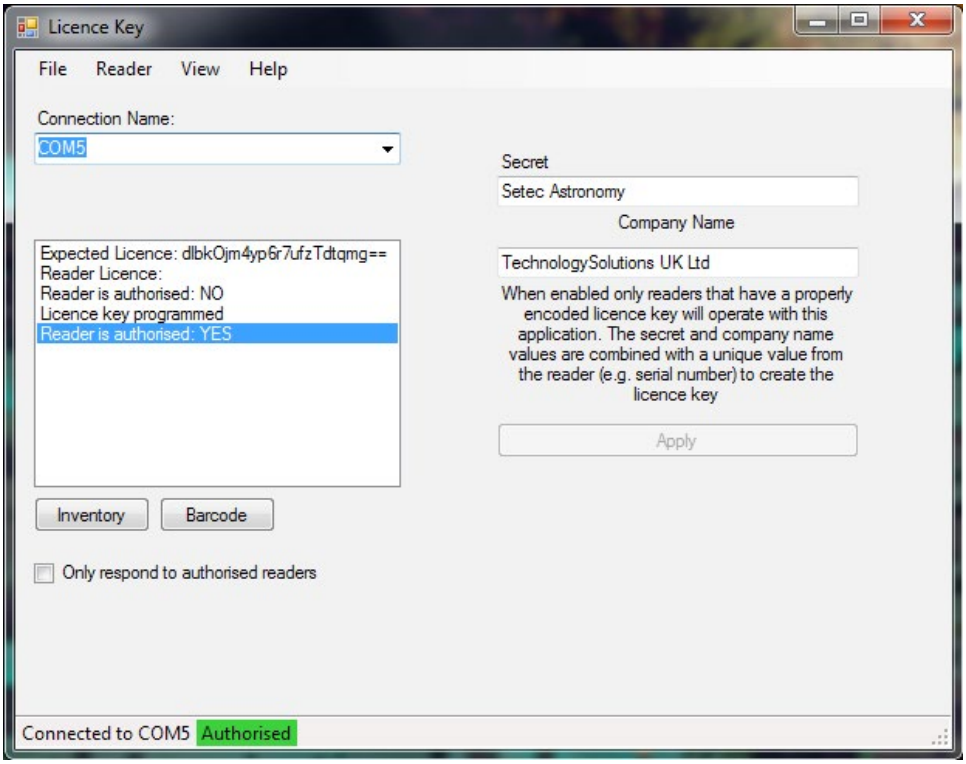


The ASCII protocol switch sample application builds on the inventory sample and demonstrates using the .NET API to command devices that support the Technology Solutions ASCII 2 protocol to customise the actions of the trigger.

- Change the default actions of the single and double trigger press
- Customise the commands used for each trigger press
- Perform soft trigger actions (emulate a hardware trigger press)
- Switch off trigger actions and respond to changes in trigger state in a custom manner

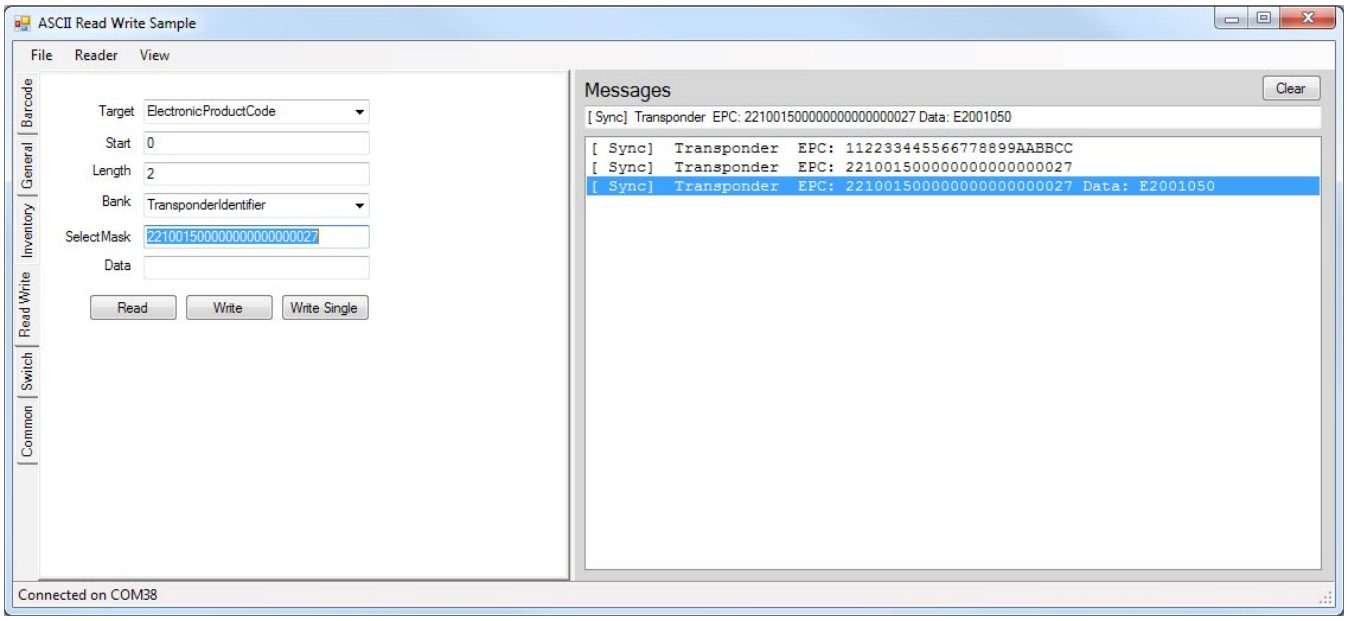
It is described in the document: *ASCII Protocol Switch Sample Application (Desktop) User Guide* provided with this SDK.

ASCII PROTOCOL LICENCE KEY SAMPLE APPLICATION



The ASCII Protocol licence key sample application demonstrates how an application can use a licencing model to operate only with particular ASCII Protocol compatible readers. Using the Licence key command added in ASCII Protocol v 2.2 a value can be stored and retrieved to each reader the application is used with. If the application derives the value stored for the licence key from a value unique to that reader (e.g. the serial number or Bluetooth address) then an application can verify whether the reader is licenced for use with that application.

ASCII PROTOCOL READ WRITE SAMPLE APPLICATION

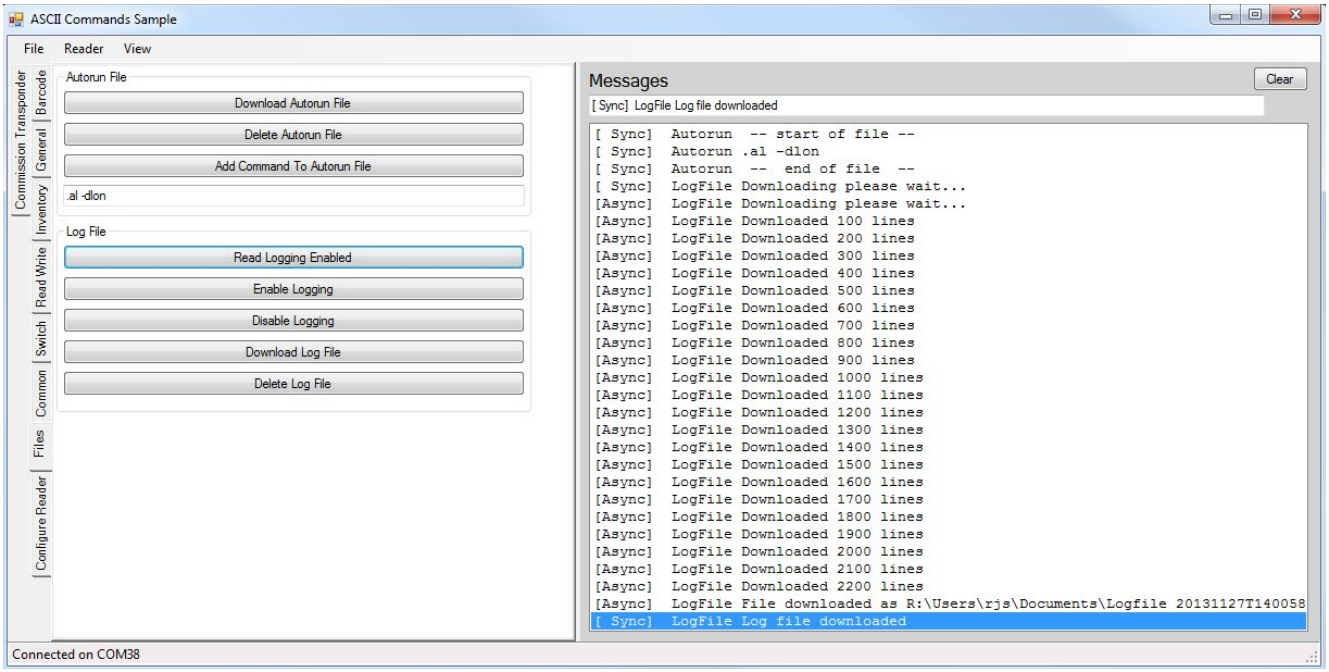


The ASCII protocol read write sample application builds on the switch sample and demonstrates using the .Net API to command devices that support the Technology Solutions ASCII 2 protocol to read and write transponders. It also demonstrates using a few other commands in the command set

- Read and write transponders
  - By TID
  - By EPC
  - Any in range
- Configure how transponders are reported by the reader (PC, EPC, CRC, Index, Timestamp)
- Incorporates the inventory and barcodes commands
- Incorporates the switch configuration and action commands

It is described in the document: *ASCII Protocol Read Write Sample Application (Desktop) User Guide* provided with this SDK.

ASCII PROTOCOL COMMANDS SAMPLE APPLICATION



The ASCII protocol commands sample application builds on the previous samples and demonstrates using the .NET API to command devices that support the Technology Solutions ASCII 2 protocol. If demonstrates almost all of the available ASCII commands.

- Download, append to, delete and execute the Autorun file
- Download, delete the log file. Enable and disable command logging
- Configure reader parameters (echo, sleep, Bluetooth)
- Change transponder EPC
- Lock transponder memory
- Incorporates reading and writing transponders
- Incorporates inventory and barcode commands
- Incorporates switch configuration and action commands

ABOUT

ABOUT TSL®

Technology Solutions UK Ltd (TSL®), part of HID Global, is a leading manufacturer of high performance mobile RFID readers used to identify and track products, assets, data or personnel.
For over two decades, TSL® has delivered innovative data capture solutions to Fortune 500 companies around the world using a global network of distributors and system integrators. Specialist in-house teams design all aspects of the finished products and software ecosystems, including electronics, firmware, application development tools, RF design and injection mould tooling.
TSL® is an ISO 9001:2015 certified company.

ISO 9001: 2015

CONTACT

Address:	Technology Solutions (UK) Ltd, Suite A, Loughborough Technology Centre, Epinal Way, Loughborough, Leicestershire, LE11 3GE, United Kingdom.
Telephone:	+44 1509 238248
Fax:	+44 1509 214144
Email:	enquiries@tsl.com
Website:	www.tsl.com

ABOUT HID GLOBAL

HID Global powers the trusted identities of the world's people, places and things. We make it possible for people to transact safely, work productively and travel freely. Our trusted identity solutions give **people** convenient access to physical and digital **places** and connect **things** that can be identified, verified and tracked digitally. Millions of people around the world use HID products and services to navigate their everyday lives, and billions of things are connected through HID technology. We work with governments, educational institutions, hospitals, financial institutions, industrial businesses and some of the most innovative companies on the planet. Headquartered in Austin, Texas, HID Global has over 4,000 employees worldwide and operates international offices that support more than 100 countries. HID Global is an ASSA ABLOY Group brand.
For more information, visit www.hidglobal.com.