V8 Mobile SDK For Windows 10

Rev. 1.0
March 2019

Date	Revision	Changes	Authors
March, 2nd, 2019	1.0	Initial revision	HYPERTIDE SAS

TABLE OF CONTENTS

I. INTRODUCTION TO THE V8M SDK

This document is the developer's documentation related to the V8 Mobile Reader Software Development Kit (SDK) for Windows 10.

The purpose of this SDK is to allow developers to create apps that can interface with the V8 readers and perform the following operations :

- Connect to and disconnect from the reader over Bluetooth
- Start a scan operation
- Receive scan data
- Receive content from the reader's memory
- Read the information stored in chip memory blocks
- Write chip memory block

The SDK itself is distributed as :

- A .NET framework class library that developers can reference in Windows Desktop (WPF) applications (located in the `DesktopV8BTSDK.zip` archive) ;
- A .NET core class library that Universal Windows Platform applications (UWP) can reference (located in the `UWV8BTSDK.zip` archive) ;
- A Windows Runtime Component (WinRT) that managed and unmanaged Win32 applications can reference (located in the `WinRTV8BTSDK.zip` archive.)

The choice of the version of the SDK to use depends on the language and platform the final application is targeting, but it is important to note that since the SDK is using the Windows 10 API to access Bluetooth Low Energy (Bluetooth 4) devices, it is managed code.

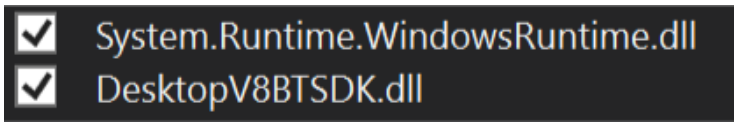
II. PROJECT SETUP

To include and start making use of the SDK, it should be added to the Visual Studio project's references.

The version of the SDK to reference depends of the platform the final app is targeting.

A. Desktop Windows (WPF) app

Right-click on the « References » section of the WPF project in the Visual Studio solution. A dialog box allows to select which files to reference. Click on the « Browse » button and add the following files :



`DesktopV8BTSDK.dll` is the SDK itself and can be located anywhere, typically where the distribution archive has been uncompressed.

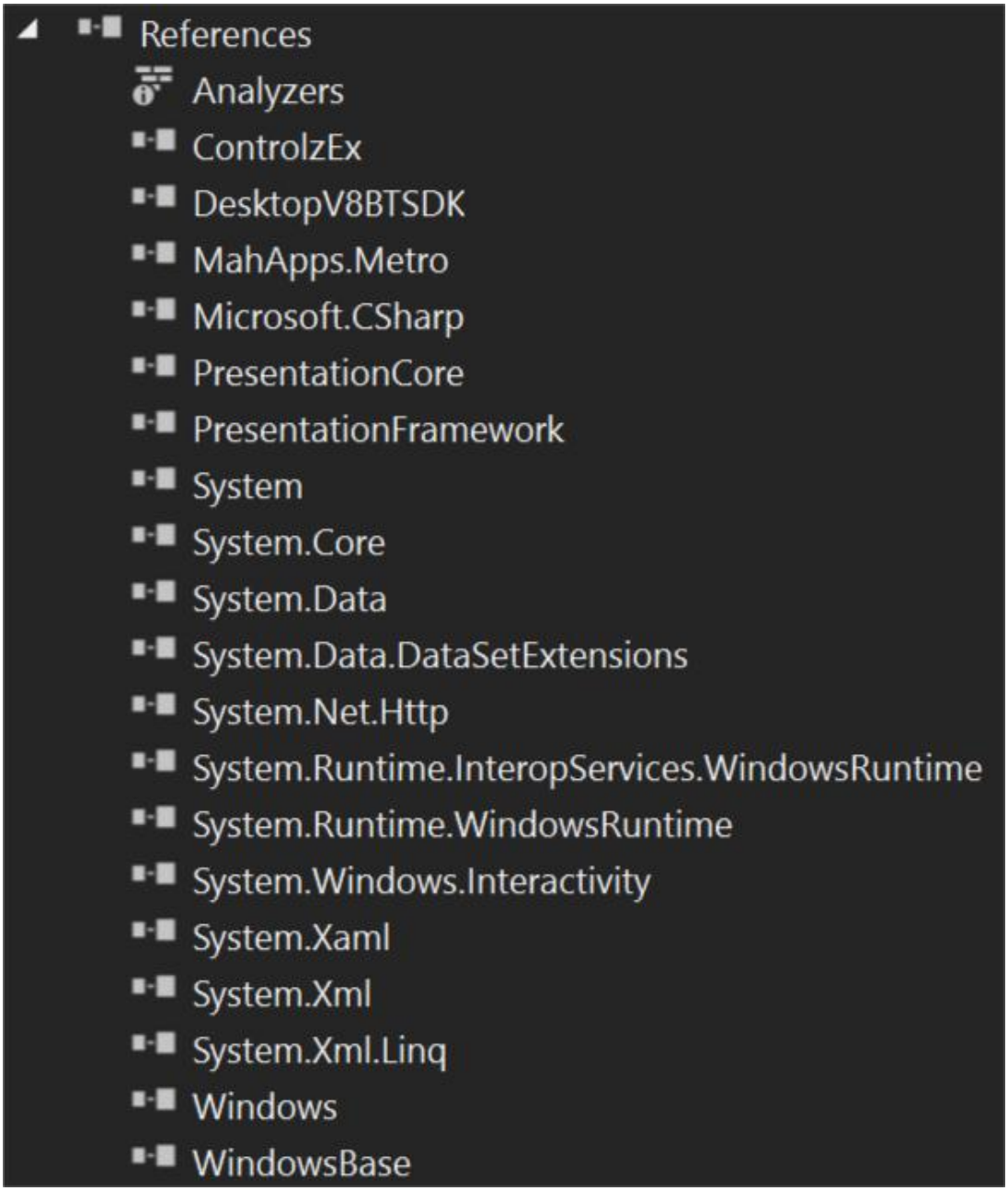
The other included files are also required. `DesktopV8BTSDK.pdb` is useful to debug the referencing app, and could be included in the final package to obtain more insights and named symbols in case of crash.

`Windows.winmd` and `System.Runtime.WindowsRuntime.dll/.xml` are required by the SDK to access Windows 10 Bluetooth API from a .NET framework application. Although those files are included in the SDK distribution archive, they should also reside in the following locations :

- `Windows.winmd` should be located in : `C:\Program Files (x86)\Windows Kits\10\UnionMetadata\10.0.17134.0` (the last folder may vary depending on the release of Windows the computer is running.)

- `System.Runtime.WindowsRuntime.dll` can be found in : `C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETCore\v4.5`

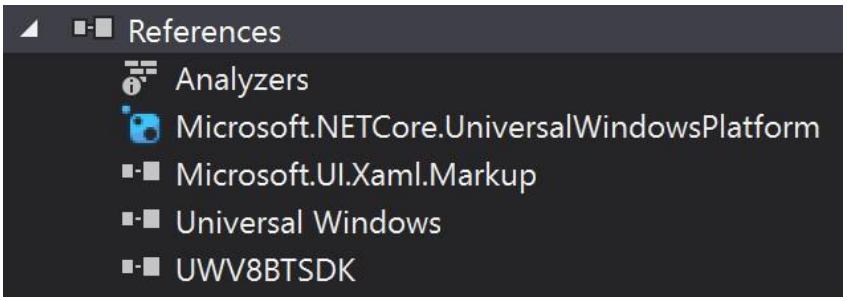
When done the project should show (at least) the following references :



B. Universal Windows (UWP) app

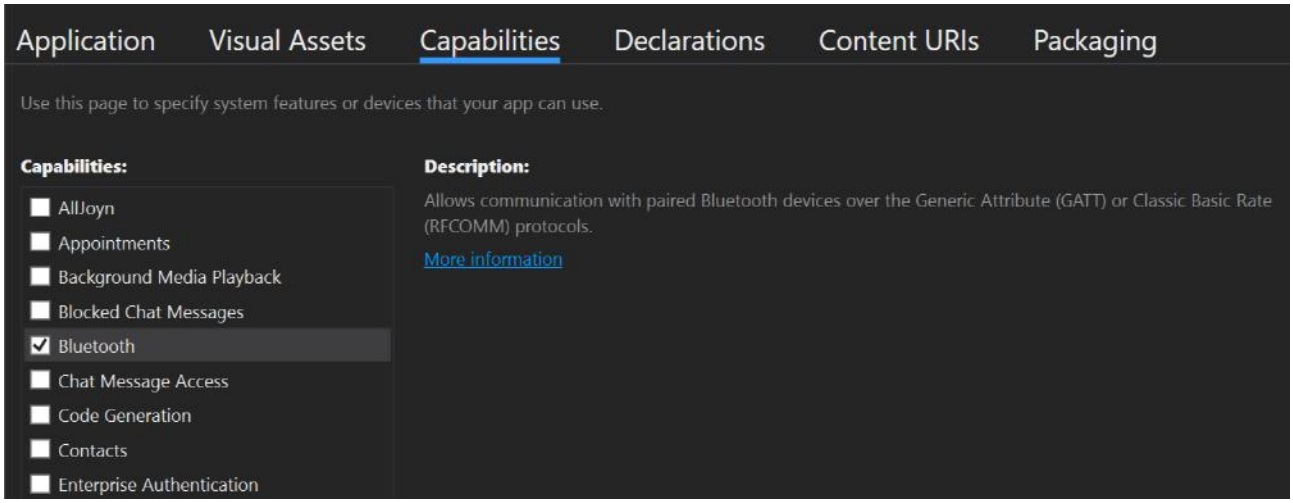
Unlike with WPF apps, the class library is the only reference needed to use the SDK in UWP apps. Add `UWV8BTSDK.dll` to the project references by opening the `Add References…` dialog in the project's « `References` » section contextual menu.

The minimum set of references should include the following items :



It should be noted that it is also possible to reference the WinRT component by referencing the `V8BT.winmd` meta-data file. It is however not recommended in terms of optimization (size, assembly, simplicity) and speed.

One very important step for any UWP app is to declare the Bluetooth capability in the app manifest. Open the « `Properties` » section of the project and click on the « `Package Manifest…` » button. Once open, click on the « `Capabilities` » tab of the manifest sheet and check « `Bluetooth` » :



C. Windows Runtime (WinRT)

The SDK is also provided as a WinRT component to be used in various languages (C++, CX, …) in Win32 apps.

This component is the `V8BT.winmd` file.

D. Reader Micro App

To support reading and writing RFID tags' internal memory, the reader should be updated with a specific micro code. The SDK does this automatically when it connects to a reader and detects that it does not have the correct micro code version.

For the SDK to update the reader, the micro app data should be available to the SDK. Therefore, it is necessary to include the `v8m.rt8` file in the final app distribution.

The `v8m.rt8` file has a size of 416 bytes and the following checksums :

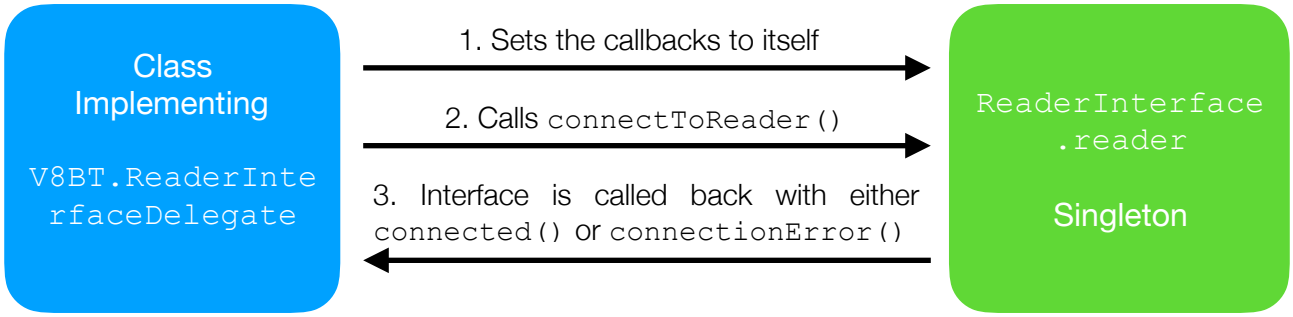
SHA1 : `7e2e345c5d2b10cb03e89f25bce510e73434ab54`

MD5 : `540eb6a6c32edce6a2925d11e0dfd1a8`

III. USING THE FRAMEWORK

A. GENERAL CONCEPT

All exchanges with the Framework, and subsequently with the reader, are done through the `ReaderInterface` class.



The `ReaderInterface` class can't be instantiated and all calls are made through its singleton (unique instance of the class) `ReaderInterface.reader` in the `V8BT` namespace. This singleton is created and initialized when the application starts.

B. LIFECYCLE

The lifecycle consists in three different stages :

- Phasing from disconnected to connected
- Executing read and write operations
- Reverting back to disconnected

Each phase is achieved by calling the relevant methods of the `ReaderInterface.reader` singleton and asynchronously waiting for the result through the current `ReaderInterface`'s callbacks.

It is important to note that all operations are done asynchronously, meaning that :

- `ReaderInterface`'s methods never block the current execution flow (ie, return immediately.)
- `ReaderInterface`'s methods never provide the caller with any result or answer directly. One of the call-related `ReaderInterfaceDelegate` interface method will be called back by the `ReaderInterface` later, with the result of the operation.
- Caller should never block its main thread (which is also the UI thread) waiting for its last call result.

C. CONNECTING TO THE READER

To connect to a reader, the caller must first set the `ReaderInterface`'s callbacks to any of its own classes implementing the `ReaderInterfaceDelegate` interface.

Failure to do so will result in the caller never get the results of its calls.

Step 1 & 2 :

```
using V8BT;
…
public sealed partial class MainPage : Page,
ReaderInterfaceDelegate {
      …
      public MainPage() {
            this.InitializeComponent();
            …
            ReaderInterface.reader.Callbacks = this;
            ReaderInterface.reader.connectToReader();
      }
```

Step 3, 4 & 5 :

```
public sealed partial class MainPage : Page,
ReaderInterfaceDelegate {
      …
      void ReaderInterfaceDelegate.connected() {
            // Reader is connected and is ready to perform
            // scan, read and write operations.
      }

      void ReaderInterfaceDelegate.connecting(DeviceInformation
      device) {
            // We're currently trying to connect to the
            // reader identified in the device parameter.
      }

      void ReaderInterfaceDelegate.connectionError(ConnectionError
      error) {
            // Connection failed, warn the user.
      }
}
```

IMPORTANT : Calls made by the `ReaderInterface` to the callbacks might not be on the UI thread. Most of the time, those calls are made from a background thread the SDK uses to handle Bluetooth operations, in order not to affect the application's main thread, as the main thread should remain as light as possible to manage the UI.

If it is needed to interact with the UI from the `ReaderInterface`'s callbacks, then tasks should be submitted to the UI Dispatcher with calls like `Dispatcher.RunAsync()` in UWP apps or `Application.Current.Dispatcher.InvokeAsync()` in WPF apps.

D. READ OPERATIONS

There are three kind of read operations :

- Scans initiated by the App
- Scans initiated by the user from the reader itself
- Scans transmitted by the reader when dumping its memory

These events can only happen when the `ReaderInterface` is in the connected state.

1. Initiate a scan from the App

This is done by calling the `scan()` method of the `ReaderInterface`. As every other `ReaderInterface` methods, `scan()` will not immediately return any result or block the caller while waiting for the user to make an actual scan.

Upon successful scan, the `scan(ScanResult, boolean)` method of the current `ReaderInterface`'s callbacks will be called.

The first parameter is a `ScanResult` object that may contain the actual data read by the reader.

Second parameter is a `boolean` :

- If `true` : this is the initial scan, other data may come later (for example data read from the chip memory) ;
- If `false` : this is the last call for this scan, the `ScanResult` object contains everything the reader has been able to read.

If the scan operation failed, `scanFailure()` will be called back instead of `scan(ScanResult, boolean)`.

2. Scan initiated from the reader

The callback class should be prepared to have its `scan(ScanResult, boolean)` method called at any time, whenever the user initiates a scan by pressing the scan button on the reader itself.

Some reader models signal the currently connected party that the user pressed the scan button. In this case, the `scanFeedback(boolean)` callback might be called before `scan(ScanResult, boolean)`. The parameter is a `boolean` indicating if the reader is starting a scan (`true`) or stopping to scan (`false`).

A `scanFeedback(boolean)` call, especially with its parameter passed with `false`, does not indicate that a scan will be made or has failed. This is an indication that the reader's state has changed. Typically, app will give the user some visual feedback upon `scanFeedback(boolean)` calls.

3. Memory dump

The user can operate the reader to dump its memory to the connected party. When the reader does that, the `ReaderInterface` callbacks will have its `export(ScanResult, int)` method called once for every exported scan.

As with `scan(ScanResult, boolean)`, the app should be prepared to have `export(ScanResult, int)` called any time, as this can happen while performing any operation (such as a network connection or transfert,) when the user manipulates the reader to dump the memory.

The first parameter passed to `export(ScanResult, boolean)` is a `ScanResult` object that contains only a chip ID, and no additional information, because the reader does not store scan time or chip memory data when an offline scan has been made by the user (ie, while not connected to any terminal.)

The second parameter is an `int` intended for future use, and currently has no meaning.

E. WRITE OPERATIONS

Write operations are used to change chip memory blocks' content using the reader as a writing device.

It must be stressed that write operations are far more delicate by nature and thus error prone, because they take longer than reads and should be performed by the user at a shorter distance on non-moving targets.

Therefore, it is vital for apps allowing users to perform writes to handle errors and be very informative to the users. They will encounter more failures.

1. Chip's ID verification

The `ReaderInterface` will ensure of the chip identity before performing the actual write operation.

It is the responsibility of the app to provide the `ReaderInterface` with this ID in the call. Typically, this is the ID of the last chip read by the user.

2. Initiate the write operation and handle the result

A write operation consists in a call to the `writeContact(string, string, ContactUpdaterCompletionDelegate, string)` method of the `ReaderInterface`.

The first parameter is a `string` that contains the « name » part of the contact information to write in the chip memory. This fields can only contains 25 characters, restricted to:

- Letters: a to z, A to Z
- Digits: 0 to 9
- The following symbols: @ . _ + -
- Spaces

The second parameter is a `string` that contains the « phone » part of the contact information. This string is limited to 13 digits if prefixed with a « + » or only 12 digits without the leading « + ».

The third parameter is a `ContactUpdaterCompletionDelegate` that with two parameters :

- A `bool` indicating if the operation was successfully.
- A `ContactStatus` holding an error code in case of failure.

The last parameter is a `string` holding the ID of the chip to write. If the actual chip presented to the reader has a different ID, the write operation will be canceled.

F. DISCONNECTING

This is a very straightforward operation, consisting in calling `disconnectFromReader()`.

It should be noted that this call always succeed, and no callback will be called.

The app may clear the callbacks field of the `ReaderInterface` if needed :

```
ReaderInterface.reader.setCallbacks(null);
```

IV. DEMO APP

The SDK is distributed with a demonstration app along with this documentation and the Framework's bundle itself as a Visual Studio solution consisting of two projects: an UWP and a WPF version of the same app.

The demo app is a single `Page/Window` that allows the user to connect to and disconnect from the reader, receive and initiate scans, handle reader's memory export and read/write chips' memory.

The main page or window (`MainPage / MainWindow`) constructor is in charge of UI initialization with `InitializeComponent()`, sets the `ReaderInterface.reader` singleton's callbacks to itself (`this`), as it implements the `ReaderInterfaceDelegate` interface.

When the user clicks on the « Connection » button, a call to `connectToReader()` is made. The result of this call will be handled by one of the two `connected()` or `connectionError(ConnectionError)`. In the first case, the user is then allowed to make scans, by enabling the « Scan » button, and the UI is modified to reflect the new state (« Connection » button is changed to « Disconnect » and the status label changed to « Connected ».)

When a scan is received, the `scan(ScanResult, boolean)` method of the callbacks is called by the `ReaderInterface`. The UI is modified to display the scanned information. If `waitForContact` is `true`, this means this is the first batch of data, that includes the chip's ID. If `false`, we got all the scanned data in the `ScanResult` object, and could update the contact frame's name and phone fields with any of those data available in the `ScanResult`'s `Contact` field.

If the user pressed the « Write » button, a write operation is started, using the last scanned chip ID.

V. SDK API REFERENCE

A. ReaderInterface

1. Methods

a) setCallbacks(ReaderInterfaceDelegate)

Sets the `ReaderInterface.reader` singleton's callback to a class implementing the `ReaderInterfaceDelegate` interface.

b) connectToReader()

This method shall be called when the application is ready to connect to the reader over Bluetooth, meeting the following requirements :

- The `ReaderInterface.reader` singleton's callbacks have been set ;
- For UWP, the Bluetooth capability has been granted (see section II.B.)

c) scan()

Ask the reader to initiate a scan. This is the equivalent of the user pressing the scan button of the reader, at the difference that the reader will not emit a beep when a chip has been successfully scanned.

The following delegate's methods will be called, depending on the scan operation results :

- `scan(ScanResult, boolean)` if a chip has been scanned ;
- `scanFailed()` if no chip have been scanned before the timeout of 20 seconds, or if the presented chip was not readable.

d) disconnectFromReader()

Disconnect from the reader. This will not shutdown the Bluetooth module, so it is not needed to call `initializeBluetooth()` again before any other connection attempt.

This call always succeeds. No delegate method will be called back, meaning that the callbacks of the `ReaderInterface.reader` singleton can be nullified after calling `disconnectFromReader()`.

e) `writeContact(string name, string phone, ContactUpdaterCompletionDelegate @delegate, string xpdrId)`

Write the chip memory with the name and phone data provided in the call parameters. The user must present the chip whose ID is the same than the one passed in the `xpdrId` parameter, or the write will be denied.

Unlike other calls in `ReaderInterface`, this one won't call a method of the current callbacks back to pass the operation result, but instead uses a specific completion delegate. This is to ensure that the result will not be lost in case of a change of callbacks in-between.

Parameters :

- `name` : the name part of the data to write with the following restrictions :
  - Maximum size of 25 characters
  - Letters : a to z, A to Z
  - Digits : 0 to 9
  - The following symbols : @ . _ + -
  - Spaces
- `phone` : the phone part of the data to write, a maximum of 12 digits with an optional leading '+' symbol (not counted in the total digit number.)
- `xpdrId` : the ID of the chip to write.
- `@delegate` : a `ContactUpdaterCompletionDelegate`, taking two parameters, that will be called to signal the result of the operation :
  - 1st parameter : a `boolean` set to `true` if the operation was successful or `false` in case of the contrary.
  - 2nd parameter : a `ContactStatus` enum value indicating the reason of the failure.

B. ReaderInterfaceDelegate

1. Methods

a) `connecting(DeviceInformation device)`

Signals that the `ReaderInterface` is currently trying to connect to a reader over Bluetooth. Called after a call to `connectToReader()`.

Parameter :

- `device` : A `DeviceInformation` object identifying the Bluetooth device the `ReaderInterface` is trying to connect to.

b) `connectionError(ConnectionError error)`

Called when a connection attempt has failed. Called after a call to `connectToReader()`.

Parameter :

- `error` : the cause of the connection failure (see `ConnectionError` for reference.)

c) `connected()`

Called when a connection attempt succeeded and a usable reader is ready to receive scan request and send scan results.

d) `disconnected(DisconnectionCause reason)`

Signals that the currently connected reader has been disconnected. This can be the consequence of a call to `disconnectFromReader()` or an external condition such as the reader powering off, the reader going out of range, the device's Bluetooth hardware module turning off, the reader not answering to keepalives or the connection state going out of sync.

Parameter :

- `reason` : the reason why the connection was terminated (see `DisconnectionCause` for reference.)

e) `scan(ScanResult scan, boolean waitForContact)`

Called upon successful scan. This method may be called multiple times, depending on how much data has been received from the reader. A successful scan garanties at least one call with the read chip's ID. Subsequent calls may contains data read from the chip's memory.

Parameters :

- `scan` : a `ScanResult` object holding scanned data.
- `waitForContact` : a `bolean` indicating :
  - `true` : this is not the last batch of data. Generally the `ScanResult` object only contains the chip ID in its `Scan` object ;
  - `false` : this is the last call. All that could have been read is in the `scan` parameter.

f) `scanFeedback(boolean started)`

Some readers (not all) signal when the user pressed the scan button, before the actual scan happens. Some other readers don't. When this method is called, it only gives an indication of the user's intent to initiate a scan, not that a scan is or will actually be made.

The principal goal of this call is to make visual or audio feedback to the user. All actual scan operation results should be managed by `scan(ScanResult, boolean)`, `export(ScanResult, int)` and `scanFailure()`.

Parameter :

- `started` : set to `true` if the user just pressed the scan button, or `false` if the scan attempt has ended (either successfully or not.)

g) `export(ScanResult scan, int id)`

Called when the reader is dumping its memory to the connected party. This happens when the user operates its reader to proceed with an export, and thus can result in this method being called any time.

Parameter :

- `scan` : the recorded scan being exported. The `ScanResult` object will only contains a `Scan` object and no `Contact` object.
- `id` : this `int` is intended for future use, and can be ignored.

h) `scanFailure()`

Signals that a scan attempt has failed. This can be the result of a timeout, or the reader not able to read a chip.

i) `version(string version)`

Called at some point in the connection process, after a call to `connectToReader()`. This call reports the detected version of the reader the `ReaderInterface` is connecting to. This could not be called at all, If the reader does not support the `ReaderInterface` querying for its version.

Parameter :

- `version` : a `string` object representing the version of the reader.

C. BluetoothError

This enum is used to report a failure trying to initialize the device's Bluetooth hardware module.

Value	Description
unavailable	The device does not have Bluetooth capabilities.

D. ConnectionError

This enum is used by the connectionError(ConnectionError) callback to report that a connection attempt failed.

Value	Description
connectionTimeout	The timeout period has expired before the actual connection could be established.
badAnswer	The remote device did not answer correctly. Could be the consequence of a misbehaving reader or an unknown device (ie, device is not a reader.)
connectionOutOfState	The remote device did not conform to the protocol or has sent unexpected data.
connectionRefused	The remote device refused the connection attempt.
bluetoothServicesUnavailable	The Bluetooth GATT server of the remote device did not report any service.
bluetoothServicesDiscoveryFailed	Discovery of available Bluetooth services offered by the remote device failed.
bluetoothCharacteristicsUnavailable	The Bluetooth GATT server of the remote device did not report any characteristic for the intended service.
bluetoothCharacteristicsDiscoveryFailed	Discovery of available characteristics for the designated Bluetooth GATT service failed.

E. DisconnectionCause

This enum is used by the `disconnected(DisconnectionCause)` callback to signal that the active connection has been terminated.

This is also called after a call to the `disconnectFromReader()` method of the `ReaderInterface` to confirm the disconnection.

Value	Description
`bluetoothStatusChanged`	The connection has been terminated because the device's Bluetooth hardware module state has changed.
`connectionInterrupted`	The connection has been terminated because the remote device disconnected, moved out of range, or after a call to `disconnectFromReader()`.
`communicationError`	Impossible to send data to the reader.

F. ContactStatus

This enum is used by the `writeContact(string, string, ContactUpdaterCompletionDelegate, string)` third parameter, a `ContactUpdaterCompletionDelegate` with two parameters : a `boolean` indicating the operation was successful if set to `true` and a `ContactStatus` holding the reason of the failure if set to `false`.

Value	Description
`success`	The chip memory has been written.
`contactReadTimeout`	The operation could not have been conducted within time.
`writeOperationFailed`	The reader reported it was not able to correctly write the chip memory.
`wrongChip`	The ID of the chip presented to the reader does not match the `xpdrId` parameter.
`protocolError`	Error while communicating with the reader in order to perform the write operation.

G. Contact

This object is used to store data read from the memory of a chip.

Field	Type	Description
`Phone`	`string`	A phone number. Maximum of 13 characters : 12 digits with or without a leading '+' sign.
`Name`	`string`	A contact name. Maximum of 25 characters : • Letters: a to z, A to Z • Digits: 0 to 9 • The following symbols: @ . _ + - • Spaces

H. Transponder

This object is used to store a chip ID and its type.

Field	Type	Description
`Id`	`string`	The chip ID.
`Type`	`byte`	Chip type. One of the `Types` enum.
`typename()`	`Method`	Returns a `string` representing the type of the chip.

I. Scan

A `Scan` object encapsulates a `Transponder` and the `Date` at which this scan was performed.

Field	Type	Description
`Xpdr`	`Transponder`	The scanned chip ID.
`date`	`DateTimeOffset`	The date at which the scan was made.

J. ScanResult

This object is used to associate a scanned chip with the data read from its memory.

Field	Type	Description
`Scan`	`Scan`	Holds the chip's ID (see I.)
`Contact`	`Contact`	Optional that holds the chip memory contents (see G.)