Bit Toys Plugin: Integration Guide

Version 1.30 - 6/2/2017
support@bit-toys.com

Overview

This guide walks you through the API basics of the Bit Toys Plugin, including initialization, methods, and events for each of the features. The Bit Toys Plugin supports identifying, authenticating, and ownership management of physical objects through Native NFC (Android only), Bluetooth NFC (Android and iOS), Audio Tags, and QR. Please review **`Bit Toys Plugin Licensing Agreement.PDF`** and refer to **`Bit Toys Plugin Getting Started.pdf`** for initial setup.

Initialization

The Bit Toys Plugin must be initialized using a Unique Identifier, which is used for toy ownership assignments. This Unique Identifier can be a hashed Device ID, a unique User ID within your own systems, a COPPA compliant ID, etc.. For example:

```
	private void Start()
	{
		BitToys.inst.Init("User_ID_Example01");
	}
```

Bit Toys Toy Object

BitToys.Toy is a class defined inside the Bit Toys Plugin and contains information useful for each toy instance. These fields are:

```
string bitToysId; // unique toy identifier in Bit Toys database
string styleId;   // the "type" of toy (T-Rex, Triceratops, etc.)
string ownerId;   // the ID of the owner (the Unique Identifier set from
Initialization)
string skuId;     // the product identifier for the toy (when applicable)
BitToys.CustomData customData;// additional per-toy info (when applicable)
```

Bit Toys ID

Every toy has a unique bitToysId on the Bit Toys Platform, and this value does not change. We strongly recommend you use the same ID inside your code and systems to keep track of the same physical object.

Style ID

Specific types of physical objects are identified using styleId. For example, your application supports 3 types of toys: Blue Car, Red Car, and Green Car. Each type would have its own unique styleId, such as `blueCar`, while each instance of that styleId would have its own unique bitToysId. Work with Bit Toys to define a list of styleId for your application.

SKU ID

If defined with Bit Toys, each BitToys.Toy object can be a child of a skuId. Similar to bitToysId, a skuId is a unique identifier for a group of BitToys.Toy objects. This skuId can be used to keep track of specific toy bundles.

Custom Data

If defined with Bit Toys, each bitToysId can have specific per-toy data bucket. This extra info is application-defined and can be accessed through the toy’s customData member object, which provides an interface for adding, removing, and modifying user-defined fields. See **Custom Data** section.

Time to Live

If defined with Bit Toys, each styleId has its own TTL (time to live) value that allows the BitToys.Toy object to live in the device's encrypted cache. This allows the BitToys.Toy object be accessible during offline / airplane mode during the TTL duration. TTL refreshes when the app connects online and fetches the list of toys again.

Fetch Owned Toys

Upon application launch and application wake, it is recommended that the app retrieves a list of toys owned by the Unique Identifier specified during initialization with this method:

```
        BitToys.inst.FetchOwnedToys();
```

It will trigger the Bit Toys Plugin to contact the Bit Toys Platform and asynchronously download and cache the list of owned BitToys.Toy objects. This operation will either succeed or fail. If successful, it will trigger the **`BitToys.inst.onFetchToyList_OK`** event and return List<BitToys.Toy> containing all the toys owned by the user.

If the device is offline, `FetchOwnedToys()` will check the encrypted cache instead. **`BitToys.inst.onFetchToyList_OK`** will trigger and return a list of all cached BitToys.Toy objects that are still within their TTL. Identify cached BitToys.Toy objects via the val bool. If no valid BitToys.Toy objects are in the cache, **`BitToys.inst.onFetchToyList_OK`** will trigger with an empty list.

If fetching of the list fails, then **`BitToys.inst.onFetchToyList_Fail`** will trigger and return an enum, BitToys.FailReason, and a string.

BitToys.FailReason may assume one of the following values:

```
enum FailReason
{
      TOY_NOT_FOUND, // The toy code does not exist in the database
      NETWORK_ERROR, // A catchall for all network problems
      APP_MISMATCH, // The toy isn't configured for this application Id
      REQUEST_DENIED, // Catchall for toy claiming problems
      API_ERROR // An internal problem with the plugin
      THROTTLED, // Custom Data is being saved too often
      OPERATION_ALREADY_IN_PROGRESS, // The Plugin process is already running
      DATA_VERSION_MISMATCH // Custom Data is out of sync with the server
}
```

The string argument in this event provides additional information on why the operation has failed.

Claim Toy

Claiming ownership of a toy is accomplished automatically when a valid toy is detected using any of the supported inputs: NFC, Audio Tags, QR, etc. When a toy is detected, the Bit Toys Plugin will attempt to claim it on the Bit Toys Platform. Then one of two events will trigger:

If claim is successful, the `BitToys.inst.onClaimToy_OK` event will trigger and return a BitToys.Toy object representing the toy whose ownership was just transferred to the local user.

If the device is offline, the cache will be checked for a valid BitToys.Toy object that matches the scanned toy and is still within it's TTL. If there's a valid object, `BitToys.inst.onClaimToy_OK` will trigger and the valid BitToys.Toy object will be returned. Identify cached BitToys.Toy objects via the val bool.

In claim fails, the `BitToys.inst.onClaimToy_Fail` event will trigger and return an enum, BitToys.FailReason, and a string similar to `BitToys.inst.onFetchToyList_Fail`.

The APP_MISMATCH BitToys.FailReason might occur here. APP_MISMATCH happens when the toy belongs to the same product line as the application but isn't signed for the current app. For example, **ExampleStudio** is developing 2 applications: **FOO** & **BAR**. Both apps use the Bit Toys Platform, but they each have their own set of toys. If **FOO** scans a toy made for **BAR**, the error APP_MISMATCH will trigger and a URL will be returned in the string parameter of `BitToys.inst.onClaimToy_Fail`. This URL can be used to direct users to download **BAR**, which is the app that the toy belongs to.

For other errors, the string parameter may contain additional information on the failure.

Claim via Native NFC

Currently, only Android allows third party access to its NFC reader module. Enable Uses Native NFC within the Bit Toys Plugin and incorporate necessary Android Manifest permissions. See **`Bit Toys Plugin Getting Started.pdf`** for details. Then simply place the NFC token near the Android device's NFC reader to begin the claim process mentioned above with appropriate callbacks. No additional code is required.

If the device's NFC ability is turned off when the application launches, or when the application resumes after being in the background. The **`BitToys.inst.bittoys_Alert`** callback will trigger and return an enum value USER_MUST_ENABLE_NFC. This can be used to prompt the users to enable NFC.

If NFC is then enabled, the **`BitToys.inst.bittoys_Alert`** callback will trigger again, this time with the enum value NFC_JUST_ENABLED.

Claim via QR Code Scanner

Enable Uses QR within the Bit Toys Plugin and request necessary permission to the device camera. Then use this method to invoke the QR code scanner:

```
BitToys.inst.qr_ScanAndClaim();
```

This will pause the Unity instance, bring up a QR code scanner and start searching for a QR code. If a valid QR code is detected, the Bit Toys Plugin will automatically attempt to claim it on the Bit Toys Platform.

Claim via Audio Tags

Enable Uses Audio Tag within the Bit Toys Plugin and request necessary permission to the device microphone. Then simply plug the Audio Tag into the device's headphone jack to begin the claim process mentioned above with appropriate callbacks. No additional code is required.

Claim via Bluetooth NFC Reader

Enable Uses Bluetooth Reader in the Big Toys Plugin and request necessary permission to the device Bluetooth. Before the Bluetooth Reader can be used by the application. It must first be paired with the device. The Bit Toys Plugin will attempt to discover and connect to a reader with the following method:

```
BitToys.inst.ble_QuickConnectReader();
```

This will discover and attempt to connect to a valid Bluetooth Reader continuously. It will only stop when a reader is connected or is stopped manually. Additionally, this method will return a bool, which is a flag used to know if a discovered reader is currently trying to connect but hasn't finished the connection process yet.

The method:

```
BitToys.inst.ble_CancelQuickConnect();
```

is used to manually stop the quick connect loop.

Once connected, the method:

```
BitToys.inst.ble_Disconnect();
```

will disconnect any connected Bluetooth Readers.

If the device's Bluetooth ability is turned off when the application launches, or when the application resumes after being in the background, the **`BitToys.inst.bittoys_Alert`** callback will trigger and return an enum value USER_MUST_ENABLE_BLUETOOTH. This can be used to prompt the users to enable Bluetooth.

If Bluetooth is then enabled, the **`BitToys.inst.bittoys_Alert`** callback will trigger again, this time with the enum value BLUETOOTH_JUST_ENABLED.

Once the Bluetooth Reader is successfully paired, place a NFC token on top of the reader will start the claim process mentioned above with appropriate callbacks.

Bluetooth LED

Each Bluetooth Reader is fitted with a RGB LED light. This LED light can be configured to various brightness and pulse speed for each color channel, thus resulting in an array of different effects. Additionally, the LED can be configured to change its behavior based on the state of the device, such as when it has detected a NFC tag or if there's an error reading a tag.

Examples of each of the configurable states:

```
//Method: SetLED_Connected(Red, Green, Blue, Red Pulse, Green Pulse, Blue
Pulse).
//When device is connected and idle, the light is solid bright blue.
BitToys.inst.ble_SetLED_Connected(0, 0, 255, 0, 0, 0);
//When a tag is detected and being read, change the light to a bright
pulsing green
BitToys.inst.ble_SetLED_nfcTagDetected( 0, 255, 0, 0, 15, 0);
//If a tag fails to read properly, change it to bright pulsing Red.
BitToys.inst.ble_SetLED_nfcTagError( 255, 0, 0, 15, 0, 0);
//If a tag is read correctly, change the light to bright solid green.
BitToys.inst.ble_SetLED_nfcTagOK( 0, 255, 0, 0, 0, 0);
```

Each of these methods have the same parameters, from left to right:
Red: Red channel brightness from 0 - 255
Green: Green channel brightness from 0 - 255
Blue: Blue channel brightness from 0 - 255

Setting each **Red**, **Green** and **Blue** channel to 0 will turn off the LED light.

Red Pulse: Red channel blink speed from 0-100
Green Pulse: Green channel blink speed from 0-100
Blue Pulse: Blue channel blink speed from 0-100

Each increment of 1 in Pulse value represents an 100-millisecond blink cycle from off to on to off. The higher the Pulse value, the slower the blink. 0 represents a solid color. 100 represents a 10-second blink cycle. By changing Pulse value for each of the color channels, you can create various rainbow effects as different channels come on and off over time.

Low Battery Warning

If the Bluetooth Reader's battery dips below an acceptable voltage, the callback **`BitToys.inst.ble_onBatteryLow`** will trigger and return a string ID for the reader. Prolonged low battery state will impact the Bluetooth Reader's ability to properly detect NFC tags and result in shorter read distance, longer read time, and even complete read failures. Therefore it is strongly recommended that you inform the end-users to change their batteries.

Remove Toy

Removing a toy will simply remove a claimed toy from the user's owned toy list. It will no longer be available to the user and no longer returned when fetching the toy list. A toy can be removed by either using the Toy Code, which is an identifier physically embedded inside the toy, or by using a BitToys.Toy bitToysId.

To remove a toy, simply use the method:

```
//1st parameter is bitToysId or Toy Code and 2nd Parameter is a bool
"isBitToysId"
//When the bool is true, it'll read the first parameter as a bitToysId,
when false, it'll read it as a Toy Code.

//Remove the toy with the bitToysId: "exampleBitToysId".
BitToys.inst.RemoveToy("exampleBitToysId", true);

//Remove the toy with the Toy Code: "294FOO"
BitToys.inst.RemoveToy("294FOO", false);
```

Custom Data

BitToys.CustomData is a class defined inside the Bit Toys Plugin and is used to set and retrieve additional info for each toy. When a BitToys.Toy object is retrieved, either by claiming it as a new toy or by fetching a toy list, it will also contain the BitToys.CustomData object. Use the BitToys.CustomData object to create new data or read/update existing data. Additionally, `BitToys.inst.GetCustomData(btId)` can be used to pull a toy's Custom Data at any time.

BitToys.CustomData supports single and array entries of the following types:

string	bool	float	int

Global and Local Custom Data

BitToys.CustomData supports both local and global Custom Data. Local Custom Data is data that is only retrieved on an app-by-app basis and only pertains to a single application. Global Custom Data is data that can be used across multiple applications. For example:

ExampleStudio is building 2 applications: **FOO** and **BAR**. Both apps can scan and use the toy named **ExampleToy**. Local Custom Data created in **FOO** for **ExampleToy** will only be usable in **FOO**. Scanning **ExampleToy** via **BAR** would not return the Local Custom Data created in **FOO**. However, if Global Custom Data is created for **ExampleToy** using **FOO**. It can be retrieved, used and updated using **BAR** and vise versa.

Updating and Creating Data

The following section is an example of how to create new data or update data that already exist in Custom Data.

```
BitToys.Toy toyExample; //The retrieved BitToys.Toy object.

//Creating single entries for local data. If entry already exists, overwrite.
toyExample.customData.SetBool("Bool Example", true);
toyExample.customData.SetFloat("Float Example", 3.14f);
toyExample.customData.SetInt("Int Example", 1);
toyExample.customData.SetString("String Example", "string");

//Creating single entries for global data. If entry exists, overwrite.
toyExample.customData.SetFloat_Global("Global Float", 11.11f);
toyExample.customData.SetString_Global("Global Name", "Nickname");
```

In the above example, we use a **Set<type>** method. The 1st parameter is a string type. This is the field identifier used to retrieve or overwrite data. The 2nd parameter is whatever type of data you're trying to set. So for `SetBool()`, the 2nd parameter is a bool type, while using `SetString()`, the 2nd parameter is a string type, and so on. **Set** will never create array entries. If **Set** is used on a field that has multiple entries, **Set** will overwrite the first value (index 0).

In this next section, we will add multiple array entries. If data already exists, the Bit Toys Plugin will add a new entry to the end of the array. If data doesn't exist, it will create an entry.

```
BitToys.Toy toyExample; //The retrieved BitToys.Toy object.

//Adding array entries for both global and local data

//Two local data entries under "boolMultiple", 1 false and 1 true
toyExample.customData.AddBool("boolMultiple", false);
toyExample.customData.AddBool("boolMultiple", true);

//Two local data entries under "floatMultiple".
toyExample.customData.AddFloat("floatMultiple", 1.14f);
toyExample.customData.AddFloat("floatMultiple", 2.14f);

//Two global data entries under "intMultiple".
toyExample.customData.AddInt_Global("intMultiple", 2);
toyExample.customData.AddInt_Global("intMultiple", 3);

//Two global data entries under "stringMultiple".
toyExample.customData.AddString_Global("stringMultiple", "string1");
toyExample.customData.AddString_Global("stringMultiple", "string2");
```

Retrieving Data

Simply using **Get<type>** will retrieve the data if it exists. An optional index parameter can be used to retrieve a value in the array. If no index is presented, the first value is in the array is retrieved. Additionally, an additional parameter is used and returned if the **Get** fails. For example:

```
//Grab local data string at index 1 for field "stringMultiple"
//If field doesn't exist, or index doesn't exist, return "string2" instead.
toyExample.customData.GetString("stringMultiple", 1, "string2");

//Grab global data string at index 0 for field "globalInts"
//If field doesn't exist, return -1 instead.
toyExample.customData.GetInt_Global("globalInts", -1);
```

Removing Data

Each field can be removed using `Remove(fieldName)`. An entry in a field array can be removed using `RemoveAt(fieldName)`. Or the entire BitToys.CustomData object can be cleared using `ClearAll_Local()`. For example:

```
//Remove the entire "boolSingle" and "intMultple" fields from local data.
toyExample.customData.Remove("boolSingle");
toyExample.customData.Remove("intMultiple");

//Remove the first array entry of "stringSingle"
// and the 2nd entry from "boolMultiple" from local data.
toyExample.customData.RemoveAt("stringSingle", 0);
toyExample.customData.RemoveAt("boolMultiple", 1);

//Remove all fields and values from local customData
toyExample.customData.ClearAll_Local();

//Remove the entire "intGlobal" fields from global data
toyExample.customData.Remove_Global("intGlobal");

//Remove the 2nd entry from the global data field "floatGlobal"
toyExample.customData.RemoveAt_Global("floatGlobal",1);

//Remove all fields and values from global customData
toyExample.customData.ClearAll_Global();
```

Saving Changes

Up until now, all changes have been made on the local system. If the app were to shut off, none of these changes will have been saved to the server and therefore, the next data retrieval will be the old data set. It's important to remember to push the changes to the server so they get saved. Both Local Custom Data and Global Custom Data need to be saved, but are handled from a single call: `SendAsync()`.

```
toyExample.customData.SendAsync(); //Save all customData changes
```

Once called, the application will send the data to be saved to the server. This process takes some time and will notify the app via 1 of 2 callbacks when completed. If your application needs to know when this process is completed, it must subscribe to these callbacks. For example:

```
BitToys.inst.onPutCustomData_Fail += OnPutData_Fail;
BitToys.inst.onPutCustomData_OK += OnPutData_Success;
private void OnPutData_Fail(string _id, BitToys.FailReason reason, string text)
{
   Debug.Log("Updating customData for id: " + _id + " failed: " + reason + " " + text);
}
private void OnPutData_Success(BitToys.Toy _toy)
{
     Debug.Log("Updating customData succeeded for toy: " + _toy.bitToysId);
}
```

It's important to note that `SendAsync()` should not be used more than once per second. Otherwise it will result in a **`BitToys.inst.onPutCustomData_Fail`** callback with the FailReason being THROTTLED**.**

Events

The Bit Toys Plugin provides events to help keep track of various states. These events are designed to aid and improve your overall end-user experience. It is up to your application to subscribe and handle each of these events:

```
        BitToys.inst.onFetchToyList_OK += OnGotToyList_Success;
        BitToys.inst.onFetchToyList_Fail += OnGotToyList_Fail;
        BitToys.inst.onClaimToy_OK += OnClaimToy_Success;
        BitToys.inst.onClaimToy_Fail += OnClaimToy_Fail;
        BitToys.inst.onGetToy_Fail += OnGetToy_Fail;
        BitToys.inst.onGetToy_OK += OnGetToy_Success;

        BitToys.inst.qr_onSawQR += OnSaw_QR;
        BitToys.inst.nfc_onSawTag += OnSaw_NFC;
        BitToys.inst.audiotag_onSawTag += OnSaw_AudioTag;
        BitToys.inst.audiotag_onTagGone += OnGone_AudioTag;
        BitToys.inst.ble_onSawTag += OnSaw_BLE;
        BitToys.inst.ble_onTagGone += OnGone_BLE;

        BitToys.inst.ble_onDeviceConnected += OnDeviceConnected;
        BitToys.inst.ble_onDeviceLost += OnDeviceLost;
        BitToys.inst.ble_onDeviceConnectFailed += OnDeviceConnectFailed;
        BitToys.inst.ble_onBatteryLow += OnBleBatteryLow;

        BitToys.inst.bittoys_Alert += OnAlert;

        BitToys.inst.onGetCustomData_OK += OnGetData_Success;
        BitToys.inst.onGetCustomData_Fail += OnGetData_Fail;
        BitToys.inst.onPutCustomData_OK += OnPutData_Success;
        BitToys.inst.onPutCustomData_Fail += OnPutData_Fail;
```

`BitToys.inst.onFetchToyList_OK` will trigger after an attempt to get the owned toys list is successful. This will return a list of BitToys.Toy objects that are owned by the Unique Identifier set in initialization. It will also return a bool value used to determine if the list is from the Bit Toys Platform or from the local encrypted cache. See **Fetch Owned Toys** section.

`BitToys.inst.onFetchToyList_Fail` will trigger after an attempt to get the owned toys list fails. This will return a BitToys.FailReason enum that provides information on why it failed. If applicable, it will also return a string value that contains detailed information on the failure. See **Fetch Owned Toys** section.

`BitToys.inst.onClaimToy_OK` will trigger after an attempt to claim ownership of a scanned toy is successful. This will return a BitToys.Toy object of the newly scanned and claimed toy. It will also return a bool value used to determine if the BitToys.Toy object came from the Bit Toys Platform or from the local encrypted cache. See **Claim Toy** section.

`BitToys.inst.onClaimToy_Fail` will trigger after an attempt to claim a scanned toy fails. This will return a BitToys.FailReason enum that provides information on why it failed. If applicable, it will also return a string value that contains detailed information on the failure. See **Claim Toy** section.

`BitToys.inst.onGetToy_OK` will trigger after an attempt to retrieve information for a toy using the method `GetToy()` is successful. This will return a single BitToys.Toy object. See **Get Toy** section.

`BitToys.inst.onGetToy_Fail` will trigger after an attempt to retrieve information for a toy using the method `GetToy()` failed. This will return a BitToys.FailReason enum that provides information on why it failed. If applicable, it will also return a string value that contains detailed information on the failure. See **Get Toy** section.

`BitToys.inst.qr_onSawQR` will trigger when a valid QR code is detected but before the claim process occurs.

`BitToys.inst.nfc_onSawTag` will trigger when a valid NFC tag is detected by the native NFC reader but before the claim process occurs.

`BitToys.inst.audiotag_onSawTag` will trigger when a valid Audio Tag is detected but before the claim process occurs.

`BitToys.inst.audiotag_onTagGone` will trigger when the device can no longer detect a previously detected Audio Tag.

`BitToys.inst.ble_onSawTag` will trigger when the Bluetooth Reader detects a valid NFC tag but before the claim process occurs. It will return a string containing the Bluetooth Reader's ID.

`BitToys.inst.ble_onTagGone` will trigger when the Bluetooth Reader can no longer detect a previously detected tag. It will return a string containing the Bluetooth Reader's ID.

`Bittoys.inst.ble_onDeviceConnected` will trigger if a valid Bluetooth Reader has been paired after the method `BitToys.inst.ble_QuickConnectReader()` is called.

`BitToys.inst.ble_onDeviceLost` will trigger if a connected Bluetooth Reader has lost its connection and is no longer reachable. This event will return a string containing the Bluetooth Reader's ID.

`BitToys.inst.ble_onDeviceConnectFailed` will trigger if a valid Bluetooth Reader has been found but did not connect properly. If applicable, this event will return a string with details on the failure.

`BitToys.inst.ble_onBatteryLow` is an event used to notify the application that the connected Bluetooth Reader has a low battery and will return a string ID for the reader.

`BitToys.inst.bittoys_Alert` can trigger when the app is launched, and/or when the app is resumed after being put in the background, and a device feature is turned off when the app needs it on. This callback will return a BitToys.Alert enum value, and a string with additional details when applicable:

```
USER_MUST_ENABLE_BLUETOOTH,
USER_MUST_ENABLE_NFC,
BLUETOOTH_JUST_ENABLED,
NFC_JUST_ENABLED
```

See **Native NFC** and **Bluetooth NFC Reader** sections.

`BitToys.inst.onGetCustomData_OK` will trigger if an attempt to retrieve a toy's Custom Data using `BitToys.inst.GetCustomData(btId)` is successful. This event will return a single BitToys.Toy that contains the latest saved Custom Data object. See **Custom Data** section.

`BitToys.inst.onGetCustomData_Fail` will trigger if any attempt to get a toy's Custom Data fails. This includes when a toy is retrieved using `ClaimToy()`, `GetToy()` or `FetchOwnedToys()`, as well as `GetCustomData()`. This event will return a string btId to identify which toy failed, a BitToys.FailReason enum that provides information on why it failed and if applicable, a string value that contains detailed information on the failure. See **Custom Data** section.

`BitToys.inst.onPutCustomData_OK` will trigger if an attempt to save **Custom Data** using `SendAsync()` is successful. This event will return a single BitToys.Toy object that contains the latest **Custom Data**. See **Custom Data** section.

`BitToys.inst.onPutCustomData_Fail` will trigger if an attempt to save **Custom Data** using `SendAsync()` fails. This event will return a string btId to identify which toy failed, a BitToys.FailReason enum that provides information on why it failed and if applicable, a string value that contains detailed information on the failure. See **Custom Data** section.