APPLE IN-APP PURCHASES MANUAL

Contents

Introduction

This PDF manual is designed for you to use as a reference to the different Apple IAP (In App Purchase) functions for iOS, tvOS and macOS, and as such does *not* contain tutorials on how to set up the API in your games. If you wish information on setting up, general use, etc… then please see the following YoYo Games Knowledge Base articles:

- iOS and tvOS: Using The IAP Extension
- macOS: Using The IAP Extension

We also recommend that before doing *anything* with this extension, you take a moment to look over the official Apple In-App Purchase API documentation, as it will familiarise you with many of the terms and concepts required to use the extension correctly, and many of the functions in the extension are practically 1:1 mappings of the methods described there:

- Apple Developer: In App Purchases

Note that this manual covers *both* the macOS and iOS/tvOS IAP functions, as they **work exactly the same way and have only been separated into two extensions so that you can use one or the other or both as required**. This means that you may need to do certain checks using the `os_type` variable to call the correct function for the current platform the game is running on, but the bulk of the code will be the same regardless. The examples in this manual are based on the iOS/tvOS functions and constants, so for macOS you would simply swap (or duplicate, if developing for both platforms) the function/constant names for the macOS versions and the IAP functionality should work exactly the same.

The Asynchronous IAP Event

When using the Apple IAP extension in your projects, you will be calling different functions that will trigger "callbacks" from the Apple API. What this means is that certain functions will be run but won't return a result until sometime in the future - which could be the next step, or it could be a few seconds later.

This result, when it comes, is called the "callback" and is Apple's IAP API responding to something you've done. This callback is dealt with in the **Asynchronous IAP Event**.

This event will always have a DS map in the GML variable **`async_load`**, and this map can be parsed to get the required information. Each function will generate different callbacks, but they will all have the following key in common:

- **"id"** – This is the event ID key and it will hold a CONSTANT with the ID of the event that has been triggered. For example, if it's an event for a product query, then the constant will be `ios_product_update / mac_product_update`. See the different functions for details about the constants returned for each.

The rest of the key/value pairs in the map will depend on the function that triggered the Async Event and the ID of the event, and you should check the individual functions listed in the rest of this manual for exact details.

Extension Functions

The rest of this manual contains a reference guide to all the functions used by the Apple IAP Extension, along with any constants that they may use or return and examples of code that use them. Some of the examples are **Extended Examples** that also show code from callbacks in the Asynchronous IAP Event.

It is worth noting that in some cases the function description will mention the use of a private server to verify purchases. This is not strictly required, as the extension supplies a verification method that verifies purchases locally with Apple, and purchases can be made and finalised even without server verification. However, YoYo Games and Apple both ***highly recommend private server verification for all IAPs***. Setting up the server to deal with purchase verification is outside of the scope of this documentation and, instead, we refer you to the Apple docs here:

- Apple Developer Docs: Validating Receipts Locally
- Apple Developer Docs: Validating Receipts With The App Store

IMPORTANT! *In order for the function `ios_iap_ValidateReceipt()` to return a `true` response, users must download the Apple Inc.* ***Root Certificate*** *and include it with their project in the Included Files (using included files covers iOS/tvOS and macOS). See:*

- https://www.apple.com/certificateauthority/
- https://www.apple.com/appleca/AppleIncRootCertificate.cer

Please see the section below on **Validation** for more information.

General workflow for using this extension is as follows:

- At the start of the game, check the user is authorised to buy in-app products
- If they are not, disable the possibility for purchases in your game UI and code
- If purchases are permitted, add the different products to the internal products list
- After adding the products but before accepting purchases, query existing purchases and if there are any unfinished transactions then deal with them, and enable any features based on durable or subscription transactions.
- Permit the game to run as normal and let the user purchase/consume products as required, verifying each purchase, then querying them, and then finalising them.

It is important to note that with the Apple purchase API there is no function or method for consuming a consumable IAP, therefore all consumables must be given to the user the moment the purchase receipt is validated.

Also note that Apple want ALL purchase requests to be "finalised", regardless of whether the purchase was actually a success or not (see the function `ios_iap_FinishTransaction()` for more details).

Validation

This extension includes a method of validating in-app purchases that does not require the setting up or use of external servers. However, the extension code also includes a warning about the potential for hacking intrinsic in this method of validation. This warning is in place as a means of highlighting the sensitive nature of the Receipt Validation code which is based on an open source repository (and is credited as such).

The intention of the inclusion of local receipt validation was to give you (the user) a means of allowing your project to get IAP code running quickly and be easily testable.

It should be understood, however, that **there is risk involved with running it in production**. Since the source code is open source and is widely available and readable, using this code will make your receipt validation more vulnerable to potential attackers, and Apple themselves state that "it's important that you employ a solution that is unique to your application":

https://developer.apple.com/library/archive/releasenotes/General/ValidateAppStoreReceipt/Introduction.html

In this case, your available options are as follows:

- Leave this code in place and use local validation, having assessed and understood the risks.

- Alter the code in question (VerifyStoreReceipt.h/mm) to create your own custom solution for validating receipts, in which case you should study the following documentation:

  https://developer.apple.com/library/archive/releasenotes/General/ValidateAppStoreReceipt/Chapters/ValidateLocally.html#//apple_ref/doc/uid/TP40010573-CH1-SW2

  In doing so, you should create your own solution for parsing and validating the iOS IAP receipt.

- Validate the receipt with the Apple Appstore using the apple end-points (not recommended) as described in the documentation:

  https://developer.apple.com/documentation/storekit/original_api_for_in-app_purchase/validating_receipts_with_the_app_store?language=objc

  If you decide to do so please refer to the function **RequestServerValidation** on the demo project, for an implementation example.

- Run a server that validates IAP receipts. This is Apple's preferred and suggested method, as it removes the ability for tampered-with iOS devices to spoof your validation code (since it is not executed on said compromised device).

Function Reference

This section contains detailed information on all the functions provided with this extension.

NOTE: The example used here in this document use **json_decode** on the json formatted strings that are returned from the function the resulting in the creation of a ds_map/ds_list, if instead you want to use structs/arrays (recommended))you can utilise the **json_parse** function instead (for a fully working example on how to do it refer to the included demo project).

ios_iap_Init / mac_iap_Init

Description

This function will initialise the Apple In-App Purchase API and **is called automatically by the extension**. As such, you should *not* be using it ever in your game code as it is not required.

Syntax

```
ios_iap_Init();
mac_iap_Init();
```

Returns

N/A

Example

N/A

ios_iap_IsAuthorisedForPayment / mac_iap_IsAuthorisedForPayment

Description

This function will check whether the user currently signed in on the device has authorised the payment process or not. The function will return `true` if payment can be completed, or `false` otherwise, in which case you should disable all purchase options in the game. Normally, you'd want to check this return value at Game Start.

Syntax

```
ios_iap_IsAuthorisedForPayment();
mac_iap_IsAuthorisedForPayment();
```

Returns

Boolean

Example

```
global.IAP_Enabled = ios_iap_IsAuthorisedForPayment();
global.ProductID[0, 0] = "ios_consumable";
global.ProductID[1, 0] = "ios_durable";
global.ProductID[2, 0] = "ios_subscription";
if (global.IAP_Enabled)
{
    ios_iap_AddProduct(global.ProductID[0, 0]);
    ios_iap_AddProduct(global.ProductID[1, 0]);
    ios_iap_AddProduct(global.ProductID[2, 0]);
    ios_iap_QueryProducts();
}
```

ios_iap_AddProduct / mac_iap_AddProduct

Description

This function can be used to add a product to the internal IAP product list, preparing it for purchase. The function takes a string, which is the product ID as defined in the App Store Connect console for your game. The function will return a constant (listed below) to inform you of the success or failure of the addition, and you should call this at the start of your game *before* querying or permitting purchases.

Syntax

```
ios_iap_AddProduct(product_id);
mac_iap_AddProduct(product_id);
```

Argument	Description	Data Type
product_id	The product ID string of the product being added	String

Returns

Constant

Constant	Actual Value	Description
ios_no_error mac_no_error	0	The product was successfully added to the internal product list.
ios_error_extension_not_initialised mac_error_extension_not_initialised	1	This indicates that there was an issue with initialising the extension itself and you should check it has been set up correctly and set to export for the given platform.
ios_error_duplicate_product mac_error_duplicate_product	3	This indicates that the IAP product ID has already been added to the internal product list.

Cont.../

Example

```
global.IAP_Enabled = ios_iap_IsAuthorisedForPayment();
global.ProductID[0, 0] = "ios_consumable";
global.ProductID[1, 0] = "ios_durable";
global.ProductID[2, 0] = "ios_subscription";
if (global.IAP_Enabled)
{
    ios_iap_AddProduct(global.ProductID[0, 0]);
    ios_iap_AddProduct(global.ProductID[1, 0]);
    ios_iap_AddProduct(global.ProductID[2, 0]);
    ios_iap_QueryProducts();
}
```

ios_iap_QueryProducts / mac_iap_QueryProducts

Description

This function can be used to query the status of products from the Apple store. The function will trigger an **Asynchronous IAP Event** with the data from the products query. This event will fill the `async_load` DS map with the following keys:

- "**id**" – This will be the constant **`ios_product_update`** or **`mac_product_update`**`, depending on the current platform you're targeting`.
- "**response_json**" – This will be a JSON string object which will contain the product details.

The "response_json" string can be converted into a DS map using the `json_decode()` function, and the map will contain two keys: "**valid**" and "**invalid**". These keys will in turn contain a DS list ID which can then be parsed to get information about each of the individual products.

The "invalid" DS list will simply be a list of strings, where each string relates to an invalid product ID (note that a product can be invalid if it is not configured or configured incorrectly on the App Store Connect console, or even if there is a connection issue between the device and App Store Connect).

The "valid" list will contain a DS map for each list entry, where each map corresponds to a single valid product. This map will have the following keys:

- "**productId**" – The unique product ID for the product as a string, for example "mac_consumable".
- "**price**" – The localised price of the product as a string, for example "£0.99".
- "**localizedDescription**" – This will hold the description of the product as a string, and localised.
- "**localizedTitle**" – This will hold the title of the product as a string, and localised.
- "**locale**" – A string representing the user's region settings (see here for more information).
- "**isDownloadable**" – This will be a boolean true or false, depending on whether App Store has downloadable content for this product.
- "**discounts**" – This will hold a DS list ID where each list entry corresponds to a discount value.

Cont…/

- "**ISOCountryCode**" – This is ISO 3166-1 country code, as a string (example: USA, EUR, JPY)
- "**ISOLanguageCode**" – This is the ISO 639-2 language code, as a string (example: en, es, zh)
- "**ISOCurrencyCode**" – This is the ISO 4217 currency code as a string (example: $, €, ¥)
- "**subscriptionPeriod**" – This will be a DS map ID where the returned map will have the following keys:
    - "**numberOfUnits**" – The number of "units" that the subscription is for.
    - "**unit**" – The unit being used to calculate the duration of the subscription. This will be one of the following constants:

Constant	Actual Value	Description
ios_product_period_unit_day mac_product_period_unit_day	22101 32105	Each unit represents a day.
ios_product_period_unit_week mac_product_period_unit_week	22102 32106	Each unit represents a week.
ios_product_period_unit_month mac_product_period_unit_month	22103 32107	Each unit represents a month.
ios_product_period_unit_year mac_product_period_unit_year	22104 32108	Each unit represents a year.

Syntax

```
ios_iap_QueryProducts();
mac_iap_QueryProducts();
```

Returns

N/A

Example

This function would normally be called straight after adding the desired products to the internal product list, as shown in the example for the function [ios_iap_AddProduct() / mac_iap_AddProduct()](). Once called it will trigger an Asynchronous IAP Event which would be parsed with something like the following code (note that we show combined code for both macOS and iOS/tvOS here, but for one or the other you would simply remove the irrelevant cases):

```
var _eventId = async_load[? "id"];
switch (_eventId)
{
    case ios_product_update:
        // Decode the returned JSON
        var _map = json_decode(async_load[? "response_json"]);
        var _plist = _map[? "valid"];
        var _sz = ds_list_size(_plist);
        // Loop through all valid products and store any data that you require
        for (var i = 0; i < _sz; ++i)
        {
            var _pmap = _plist[| i];
            switch (_pmap[? "productId"])
            {
                case "ios_consumable":
                    global.ProductID[0, 1] = _pmap[? "price"];
                    global.ProductID[0, 2] = _pmap[? "localizedDescription"];
                    global.ProductID[0, 3] = _pmap[? "localizedTitle"];
                    break;
                case "ios_durable":
                    global.ProductID[1, 1] = _pmap[? "price"];
                    global.ProductID[1, 2] = _pmap[? "localizedDescription"];
                    global.ProductID[1, 3] = _pmap[? "localizedTitle"];
                    break;
                case "ios_subscription":
                    global.ProductID[2, 1] = _pmap[? "price"];
                    global.ProductID[2, 2] = _pmap[? "localizedDescription"];
                    global.ProductID[2, 3] = _pmap[? "localizedTitle"];
                    break;
            }
        }
        // Parse any invalid responses here if required
        ds_map_destroy(_map);
        // Query purchases here and react appropriately
        break;
}
```

ios_iap_QueryPurchases / mac_iap_QueryPurchases

Description

This function can be used to query the status of all un-finalised purchases. This function can be called anytime and in any place in your game code, as the purchase status details are retrieved when the API is initialised and after any change has been made (i.e., something has been purchased). However, we recommend that you initially call it *after* adding product IDs to the internal product list and generally it's better to call it after having queried product details too.

IMPORTANT! *This function will only return items that have not been finalised. So, any products that are returned by this function will need to be finalised using the* `ios_iap_FinishTransaction() / mac_iap_FinishTransaction()` *function.*

The function will return a JSON string object that can be decoded into a DS map using `json_decode()` function. This map will have a single key "**purchases**" which in turn is a DS list ID. Each entry in the DS list will be a DS map corresponding to a single purchase, and will contain the following keys:

- "**productId**" – The ID string of the purchased product.
- "**purchaseState**" – The state of the purchase when the function was called. Will be one of the following constants:

Constant	Actual Value	Description
ios_purchase_success mac_purchase_success	23001 33101	This indicates that the product has been successfully purchased.
ios_purchase_failed mac_purchase_failed	23002 33102	The product purchase has failed in some way, for example, it was cancelled by the user.
ios_purchase_restored mac_purchase_restored	23003 33103	The purchase has been restored. This usually only occurs when a purchase was made on one device, and then restored on another *before* being finalised.

`ios_iap_QueryPurchases() / mac_iap_QueryPurchases` **Cont…/**

- "**responseCode**" – This is the Apple response code, an integer value, where:
    - purchasing = 0
    - purchased = 1
    - failed = 2
    - restored = 3
    - deferred = 4
- "**purchaseToken**" – The purchase token string.
- "**receipt**" – The receipt string. This is deprecated and should *not* be used for anything. It is only included in this documentation as it is still part of the return payload from Apple. To get the correct receipt string, please use the function `ios_iap_GetReceipt / mac_iap_GetReceipt()`.

Generally, you would want to call this function once at the start of the game, and then again after any purchase receipt validation so that you know which items have been purchased and need to be finalised and awarded to the user.

Syntax

```
ios_iap_QueryPurchases();
mac_iap_QueryPurchases();
```

Returns

String (JSON)

Cont…/

`ios_iap_QueryPurchases() / mac_iap_QueryPurchases` **Cont…/**

Example

The following code example shows how to use this function to finalise products after the receipt has been correctly validated. Generally, this would be done in the **Asynchronous HTTP Event** which would be triggered by the return of the validation from either the game server or Apple:

```
var _json = ios_iap_QueryPurchases();
if (_json != "")
{
    var _map = json_decode(_json);
    var _plist = _map[? "purchases"];
    var _sz = ds_list_size(_plist);
    for (var i = 0; i < _sz; ++i)
    {
        var _pmap = _plist[| i];
        if (_pmap[? "purchaseState"] != ios_purchase_failed)
        {
            switch (_pmap[? "productId"])
            {
                case global.ProductID[0, 0]: global.Gold += 100; break;
                case global.ProductID[1, 0]: global.NoAds = true; break;
                case global.ProductID[2, 0]: global.Subs = true; break;
            }
        }
        var _ptoken = _pmap[? "purchaseToken"];
        ios_iap_FinishTransaction(_ptoken);
    }
    ds_map_destroy(_map);
}
```

ios_iap_PurchaseProduct / mac_iap_PurchaseProduct

Description

This function is what is used to purchase a product within your game. You supply the product ID as a string (which should match the ID of the product on the App Store Connect console), and the function will immediately return one of the constants shown below to inform you of the initial status of the purchase request, and if that is **`ios_no_error / mac_no_error`** then it will also trigger an Asynchronous IAP Event. In this event the `async_load` DS map will have an "**id**" key which will be either **`ios_payment_queue_update`** or **`mac_payment_queue_update`** depending on the platform being run.

The `async_load` map will also have another key "**json_response**" which will contain a JSON object string with the details of the purchase. This string can be decoded into a DS map using the `json_decode()` function, and the resulting DS map will have a single key "**purchases**". This in turn will be a DS list ID in which each entry is a DS map corresponding to a single purchase, containing the following keys:

- "**productId**" – The ID string of the purchased product.
- "**responseCode**" – This is the Apple response code, an integer value, where:
  - purchasing = 0
  - purchased = 1
  - failed = 2
  - restored = 3
  - deferred = 4
- "**purchaseToken**" – The purchase token string.

Cont…/

- "**purchaseState**" – The state of the purchase which will be one of the following constants:

Constant	Actual Value	Description
`ios_purchase_success` `mac_purchase_success`	23001 33101	This indicates that the product has been successfully purchased.
`ios_purchase_failed` `mac_purchase_failed`	23002 33102	The product purchase has failed in some way, for example, it was cancelled by the user.
`ios_purchase_restored` `mac_purchase_restored`	23003 33103	This indicates that the purchase has been restored.

- "**receipt**" – The receipt string. This is deprecated and should *not* be used for anything. It is only included in this documentation as it is still part of the return payload from Apple. To get the correct receipt string, please use the function `ios_iap_GetReceipt / mac_iap_GetReceipt()`.

If the purchase state comes back as a success or a restored purchase, then you should go ahead and validate the purchase with either your own server (recommended) or with Apple, and then finalise the purchase. If the purchase has failed, then you should still finalise the purchase, but no other action needs to be taken. For more information on finalising purchases please see the function `ios_iap_FinishTransaction() / mac_iap_FinishTransaction()`.

Syntax

```
ios_iap_PurchaseProduct(product_id);
mac_iap_PurchaseProduct(product_id);
```

Argument	Description	Data Type
product_id	The product ID string of the product being purchased	String

Cont.../

Returns

Constant

Constant	Actual Value	Description
ios_no_error mac_no_error	0	The product was successfully added to the internal product list.
ios_error_extension_not_initialised mac_error_extension_not_initialised	1	This indicates that there was an issue with initialising the extension itself and you should check it has been set up correctly and set to export for the given platform.
ios_error_no_skus mac_error_no_skus	2	There are no products added to the internal product list.
ios_error_duplicate_product mac_error_duplicate_product	3	This indicates that the IAP product ID has already been added to the internal product list.

Extended Example

The following code would be used to create a purchase request for the given product, and would be placed anywhere in the game (like a button object):

```
if (global.IAP_Enabled)
{
    ios_iap_PurchaseProduct(global.ProductID[0, 0]);
}
```

Cont…/

`ios_iap_PurchaseProduct / mac_iap_PurchaseProduct` **Cont…/**

This will then trigger an Asynchronous IAP Event which can be dealt with something like this:

```
var _eventId = async_load[? "id"];
switch (_eventId)
{
    case ios_payment_queue_update:
        // Decode the returned JSON
        var _json = async_load[? "response_json"];
        if (_json != "")
        {
            var _map = json_decode(_json);
            var _plist = _map[? "purchases"];
            var _sz = ds_list_size(_plist);
            // loop through purchases
            for (var i = 0; i < _sz; ++i)
            {
                var _pmap = _plist[| i];
                // Check purchases
                var _ptoken = _pmap[? "purchaseToken"];
                if (_pmap[? "purchaseState"] != ios_purchase_failed)
                {
                    var _receipt = ios_iap_GetReceipt();
                    // CALL SERVER CHECK WITH RECEIPT HERE
                    // or validate, finalise and award the product
                    if (ios_iap_ValidateReceipt() == true)
                    {
                        switch (_pmap[? "productId"])
                        {
                            case global.ProductID[0, 0]:
                                global.Gold += 100;
                                break;
                            case global.ProductID[1, 0]:
                                global.NoAds = true;
                                break;
                            case global.ProductID[2, 0]:
                                global.Subs = true;
                                break;
                        }
                        ios_iap_FinishTransaction(_ptoken);
                    }
                    else
                    {
                        // Validation failed, so deal with it here
                    }
                }
                else
                {
                    // Purchase failed, so finalise it.
                    ios_iap_FinishTransaction(_ptoken);
                }
                ds_map_destroy(_pmap);
            }
        }
        break;
}
```

ios_iap_ValidateReceipt / mac_iap_ValidateReceipt

Description

This function can be used for local receipt validation with Apple. In general, you'd want to use a private server for validation of all purchases (especially subscriptions), but if that is not possible then you can use this function, *after* calling the `ios_iap_GetReceipt()/mac_iap_GetReceipt()` function to validate purchases. The function will return `true` if validation has been successful, or `false` otherwise, in which case you should attempt to refresh and revalidate the receipt using `ios_iap_RefreshReceipt()` (on iOS *only*) or exit the app using the function `mac_iap_exit()` (on macOS *only*).

Syntax

```
ios_iap_ValidateReceipt();
mac_iap_ValidateReceipt();
```

Returns

Boolean

Example

For an example of using this function, please see:

- ios_iap_PurchaseProduct / mac_iap_PurchaseProduct

ios_iap_RestorePurchases / mac_iap_RestorePurchases

Description

This function can be used to restore any previous purchases and Apple require you to have a button in your game that calls this function so that users that have changed or refreshed their device can still access previously made purchases. Calling this function will immediately return one of the constants shown below to inform you whether the restore request has been made or not, and then a successful request *may* trigger an Asynchronous IAP Event with the restored purchase details. We say "may", as under the following circumstances no Async Event will be triggered:

- All transactions are unfinished.
- The user did not purchase anything that is restorable.
- You tried to restore items that are not restorable, such as a non-renewing subscription or a consumable product.
- Your app's build version does not meet the guidelines for the CFBundleVersion key.

If an Asynchronous IAP Event is triggered, the async_load DS map "id" key will be either **`ios_payment_queue_update`** or **`mac_payment_queue_update`** depending on the platform being run.

The `async_load` map will also have another key "**json_response**" which will contain a JSON object string with the details of the purchase. This string can be decoded into a DS map using the `json_decode()` function, and the resulting DS map will have a single key "**purchases**". This in turn will be a DS list ID in which each entry is a DS map corresponding to a single purchase, containing the following keys:

- "**productId**" – The ID string of the purchased product.
- "**responseCode**" – This is the Apple response code, an integer value, where:
    - purchasing = 0
    - purchased = 1
    - failed = 2
    - restored = 3
    - deferred = 4
- "**purchaseToken**" – The purchase token string.

Cont…/

- "**purchaseState**" – The state of the purchase which will be one of the following constants:

Constant	Actual Value	Description
`ios_purchase_success` `mac_purchase_success`	23001 33101	This indicates that the product has been successfully purchased.
`ios_purchase_failed` `mac_purchase_failed`	23002 33102	The product purchase has failed in some way, for example, it was cancelled by the user.
`ios_purchase_restored` `mac_purchase_restored`	23003 33103	This indicates that the product has been restored.

- "**receipt**" – The receipt string. This is deprecated and should *not* be used for anything. It is only included in this documentation as it is still part of the return payload from Apple. To get the correct receipt string, please use the function `ios_iap_GetReceipt / mac_iap_GetReceipt()`.

If the purchase state comes back as a success or a restored purchase, then you should go ahead and validate the purchase with either your own server (recommended) or with Apple, and then finalise the purchase and award any features or products to the user. If the purchase has failed, then you should still finalise the purchase, but no other action needs to be taken. For more information on finalising purchases please see the function `ios_iap_FinishTransaction() / mac_iap_FinishTransaction()`.

Syntax

```
ios_iap_RestorePurchases();
mac_iap_RestorePurchases();
```

Cont.../

`ios_iap_RestorePurchases / mac_iap_RestorePurchases` **Cont.../**

Returns

Constant

Constant	Actual Value	Description
`ios_no_error` `mac_no_error`	0	The restore request has been sent successfully.
`ios_error_extension_not_initialised` `mac_error_extension_not_initialised`	1	This indicates that there was an issue with initialising the extension itself and you should check it has been set up correctly and set to export for the given platform.
`ios_error_no_skus` `mac_error_no_skus`	2	There are no products added to the internal product list and so nothing can be restored.

Extended Example

The following code would be used to create a restore request, and would be placed anywhere in the game (like a button object):

```
if (global.IAP_Enabled)
{
    ios_iap_ RestorePurchases();
}
```

Cont…/

`ios_iap_RestorePurchases / mac_iap_RestorePurchases` **Cont…/**

This may then trigger an Asynchronous IAP Event which can be dealt with something like this:

```
var _eventId = async_load[? "id"];
switch (_eventId)
{
    case ios_payment_queue_update:
        // Decode the returned JSON
        var _json = async_load[? "response_json"];
        if (_json != "")
        {
            var _map = json_decode(_json);
            var _plist = _map[? "purchases"];
            var _sz = ds_list_size(_plist);
            // loop through purchases
            for (var i = 0; i < _sz; ++i)
            {
                var _pmap = _plist[| i];
                // Check purchases
                if (_pmap[? "purchaseState"] != ios_purchase_failed)
                {
                    var _receipt = ios_iap_GetReceipt();
                    // CALL SERVER CHECK WITH RECEIPT HERE
                    // or validate, finalise and award the product
                    if (ios_iap_ValidateReceipt() == true)
                    {
                        switch (_pmap[? "productId"])
                        {
                            case global.ProductID[0, 0]:
                                global.Gold += 100;
                                break;
                            case global.ProductID[1, 0]:
                                global.NoAds = true;
                                break;
                            case global.ProductID[2, 0]:
                                global.Subs = true;
                                break;
                        }
                        ios_iap_FinishTransaction(_ptoken);
                    }
                    else
                    {
                        // Validation failed, so deal with it here
                    }
                }
                else
                {
                    var _ptoken = _pmap[? "purchaseToken"];
                    // Purchase failed, so finalise it
                    ios_iap_FinishTransaction(_ptoken);
                }
                ds_map_destroy(_pmap);
            }
        }
        break;
    }
```

ios_iap_FinishTransaction / mac_iap_FinishTransaction

Description

Once a purchase request, restore request or purchase query has been sent, any products returned in the Asynchronous IAP event for these calls should be validated and then finalised before awarding any products to the player. Finalising a product means that you are telling Apple that the transaction has been completed and the product awarded, and this function should be called on **all** transactions, even those that have failed (for example, cancelled by the user). Any transaction that has not been finalised will appear in the above-mentioned purchase/restore/query data and should be finalised before any further purchases of the same product are processed.

When calling this function, you need to supply the product token string (as returned in the Asynchronous IAP Event for the associated function call), and the function will return one of the constants listed below.

Syntax

```
ios_iap_FinishTransaction(purchase_token);
mac_iap_FinishTransaction(purchase_token);
```

Argument	Description	Data Type
Purchase_token	The purchase token returned by the purchase request.	String

Returns

Constant

Constant	Actual Value	Description
ios_no_error mac_no_error	0	The restore request has been sent successfully.
ios_error_unknown mac_error_unknown	-1	This indicates that there is an issue with finalising the product.

Example

For examples of using this function, please see:

- ios_iap_QueryPurchases / mac_iap_QueryPurchases
- ios_iap_PurchaseProduct / mac_iap_PurchaseProduct
- ios_iap_RestorePurchases / mac_iap_RestorePurchases

ios_iap_GetReceipt / mac_iap_GetReceipt

Description

This function can be used to retrieve the receipt string for all purchases currently in progress. This string can then be sent as part of the payload to your server (or to Apple) to verify the purchases in the receipt.

> **IMPORTANT!** *The receipt string can contain* ***multiple transaction receipts at once*** *as Apple sends back all pending receipts in one string. For more information, including how to check the information provided in the receipt, please see the Apple Developer Documentation.*

Syntax

```
ios_iap_GetReceipt();
mac_iap_GetReceipt();
```

Returns

String

Example

The following code is a very simple example of how to use the function and send a verification request off to a server you have set up. Note, however, that the actual usage will very much depend on the how you've set up the server and this is not a one-size-fits-all example to be copied and used directly:

```
var _receipt = ios_iap_GetReceipt();
if _receipt != ""
{
    var _map = ds_map_create();
    _map[? "apple_receipt"] = receipt;
    var _body = json_encode(_map);
    ds_map_clear(_map);
    _map[?  "Host"] = "10.36.11.105:9999";
    _map[? "Content-Type"] = "application/json";
    _map[? "Content-Length"] = string_length(_body);
    var url = "http://" + _map[?  "Host"] + "/apple-receipt-verify";
    http_request(url, "POST", _map, _body);
    ds_map_destroy(_map);
}
```

ios_iap_RefreshReceipt

Description

With this function you can request a new receipt for all purchases pertaining to a user and app. This function is **iOS only** and should only be called if a previous receipt has been unable to be validated correctly (for Mac apps, please see the function `mac_iap_exit()`). The function will return one of the constants listed below immediately to inform you whether the refresh request has been successful, and if it is successful then an Asynchronous IAP Event will be triggered. In this event the `async_load` DS map will have an "**id**" key, which will be the constant **`ios_receipt_refresh`**, and an additional key "**status**". The status will be one of two constants: **`ios_receipt_refresh_success`** or **`ios_receipt_refresh_failure`**. If the refresh is successful, you can then retrieve the new receipt using the `ios_iap_GetReceipt()` function, but if it fails then you may want to try again at least once before deciding that something is wrong.

Note that failing validation is a rare occurrence and is very indicative that there is something funny going on with the request. As such, you may want to consider locking down and preventing any further purchases – or at least not granting the products that were being validated – should validation fail 2 or more times. Any outstanding purchases should still be finalised at this time.

Syntax

```
ios_iap_RefreshReceipt();
```

Returns

Constant

Constant	Error Code	Description
ios_no_error	1	The refresh request has been sent successfully.
ios_error_extension_not_initialised	2	This indicates that there was an issue with initialising the extension itself and you should check it has been set up correctly and set to export for the given platform
ios_error_no_skus	3	There are no SKUs in the product list and so no receipts to request.

Cont…/

`ios_iap_RefreshReceipt` **Cont…/**

Example

The following example assumes you have received a failed validation attempt from your server or from local validation and have called this function to request a refresh of the IAP receipt. This would then be dealt with in the Asynchronous IAP Event in the following way:

```
var _id = async_load[? "id"];
switch (_id)
{
    case ios_receipt_refresh:
        if (async_load[? "status"] == ios_receipt_refresh_success)
        {
            var _receipt = ios_iap_GetReceipt();
            if (_receipt != "")
            {
                // Send off another validation request to your
                // server (or locally) and try again
                var _receipt = ios_iap_GetReceipt();
                if (ios_iap_ValidateReceipt() == true)
                {
                    switch (_pmap[? "productId"])
                    {
                        case global.ProductID[0, 0]:
                            global.Gold += 100;
                            break;
                        case global.ProductID[1, 0]:
                            global.NoAds = true;
                            break;
                        case global.ProductID[2, 0]:
                            global.Subs = true;
                            break;
                        }
                        ios_iap_FinishTransaction(_ptoken);
                    }
                }
                else
                {
                    // Validation failed, so deal with it here
                }
            }
        }
        else if async_load[? "status"] == ios_receipt_refresh_failure
        {
            global.IAP_Enabled = false;
            // Finalise the purchase here
        }
        break;
}
```

mac_iap_exit

Description

This function will force close the app and send the given error code to the OS. This function is for IAPs on the **macOS only**. When you send a receipt for validation (either locally or via your own servers) and the validation fails, you should always call this function and supply the exit code value as the constant **`mac_invalid_receipt_exit_code`**. After the app has closed, the OS will attempt to obtain a valid receipt and may prompt the user for their iTunes credentials. If the system successfully obtains a valid receipt, it will relaunch the application, otherwise, it will display an error message to the user, explaining the problem.

Syntax

```
mac_iap_exit(error_code);
```

Argument	Description	Data Type
error_code	The exit code value to use, which should be the constant "**mac_invalid_receipt_exit_code**" for invalid receipts.	Integer

Returns

N/A

Example

Simply call the function should validation of the purchase receipt fail.

RegisterCallbacks

Description

This is an internal function for **macOS only**. This function should never be called in your code, and you should not edit or change anything about it otherwise the extension may no longer work.

Syntax

N/A

Returns

N/A

Example

N/A