Virtual Piggy for the Marmalade SDK

[Add more info][Copyright and trademarks go here]

Table of Contents

1.0 Introduction

What is Virtual Piggy?

Virtual Piggy is the first e-commerce solution that enables kids to manage and spend money within a parent-controlled environment that also allows merchants to sell to them while complying with the Children's Online Privacy Protection Act (COPPA). You can find out more about Virtual Piggy and create an account by visiting http://www.virtualpiggy.com

Why should I use Virtual Piggy in my Apps?

There are a number of reasons. The first being that usually when children are playing a mobile game or using a mobile app they are restricted to no in-app purchasing or the parent actually has to take over the app and perform the in-app purchase for the child. The second is that when a child is left unattended they can inadvertently rack up hundreds of dollars worth of in-app purchases without even realising it, usually the merchant has to refund these accidental transactions. Virtual Piggy combats both of these problems as well as teaching the child responsible money management along the way. Because the child has an allowance set by the parent the child cannot inadvertently overspend and the child is free to make their own mind up about their purchases. The system is win-win for parent, child and merchant.

What is Virtual Piggy for the Marmalade SDK?

The Marmalade SDK is a cross platform mobile, desktop and emerging technology SDK that enables developers to write apps once and deploy them to a multitude of platforms with very little to no technical changes (see http://www.madewithmarmalade.com for more information). The Virtual Piggy API for Marmalade is an API that sits on top of Marmalade that enables developers to accept in-app purchases for virtual goods and subscriptions safely from children and parents.

2.0 Installation and Setup

To install the Virtual Piggy API for Marmalade, download the API zip file from [Add more info] [Insert download location here]

To install the Virtual Piggy SDK simply unzip the downloaded zip file to a folder and run the VirtualPiggy\Test\test.mkb to open the sample test project. This test project shows the developer how to communicate with Virtual Piggy API to carry out many tasks including authentication, purchasing, subscriptions and user information retrieval.

To install the Virtual Piggy SDK into your Marmalade SDK project, copy the Virtual Piggy SDK to your projects folder and update your projects MKB file like shown below:

```
options
{
        module_path="VirtualPiggy/lib"    # Point module path to Virtual Piggy libraries
}

subprojects
{
        VirtualPiggy                      # Include the Virtual Piggy library project
}
```

You can of course place the Virtual Piggy API anywhere you choose, just remember to change the path to the installation in module_path.

To get Virtual Piggy running in your app you have a number of things that you need to do:

- Initialise Virtual Piggy
- Update Virtual Piggy each game or app frame
- Shut down Virtual Piggy when done with it

The short example below shows how to peform all 3 tasks:

```
int main()
{
        // Initialise Virtual Piggy
        VirtualPiggy::Create();
        VIRTUAL_PIGGY->Init(VIRTUAL_PIGGY_API_KEY);
        VIRTUAL_PIGGY->setMerchantID(VIRTUAL_PIGGY_MERCHANT_ID);

        TEST_PingHeaders();

        // Marmalade main loop
        while (!s3eDeviceCheckQuitRequest())
        {
                VIRTUAL_PIGGY->Update();  // Update Virtual Piggy
                s3eDeviceYield(0);
        }

        // Shut down Virtual Piggy
        VIRTUAL_PIGGY->Release();
        VirtualPiggy::Destroy();

        return 0;
}
```

Note the use of VIRTUAL_PIGGY to refer to the Virtual Piggy instance. For simplicity all calls to Virtual Piggy will be done via the VIRTUAL_PIGGY macro.

3.0 API Overview

The Virtual Piggy API consists of a number of functions that can be divided into the following categories:

- Housekeeping – Initialisation, release and update methods
- Authentication – Child and parent authentication. Authentication usually returns a textual token that should be stored for the session and pass to other methods that communicate with Virtual Piggy. There are two types of token, the first represents a child user and the second a parent user. Do not pass child tokens to parent methods or parent tokens to child methods, they are not interchangeable.
- Purchase Transactions – Purchase transactions enable users to make purchases from within your app. You create a shopping basket (VpBasket) and fill it with items that your user has purchased then send this basket in a purchase request to Virtual Piggy
- Subscription Transactions – Subscription transactions enable your users to purchase subscriptions for in-app content. This works in a similar way to normal purchase transactions except additional subscription information defined by VpSubscription should also be passed to the API
- Information query – You can query a variety of information about child and parent users including information such as address, age, gender and loyalty points
- Pinging – This enables you to check your connection to Virtual Piggy

Below is a quick overview of the full API:

```
bool     Init(const char* api_key);
void     setMerchantID(const char* merchant_id);
void     Release();
void     Update();
void     AuthenticateChild(const char* user_name, const char* password, VpAuthenticateChildCallback callback);
void     AuthenticateUser(const char* user_name, const char* password, VpAuthenticateUserCallback callback);
void     ProcessTransaction(const char* token, const char* description, VpBasket* basket,
VpProcessTransactionCallback callback);
void     ProcessParentTransaction(const char* token, const char* description, const char* child_id, const char*
payment_id, VpBasket* basket, VpProcessParentTransactionCallback callback);
void     ProcessSubscription(const char* token, const char* description, VpSubscription* subscription,
VpProcessSubscriptionCallback callback);
void     ProcessParentSubscription(const char* token, const char* description, const char* child_id, const char*
payment_id, VpSubscription* subscription, VpProcessParentSubscriptionCallback callback);
void     MerchantCancelSubscription(const char* subscription_id, VpMerchantCancelSubscriptionCallback callback);
void     GetAllChildren(const char* token, VpGetAllChildrenCallback callback);
void     GetPaymentAccounts(const char* token, VpGetPaymentAccountsCallback callback);
void     GetChildAddress(const char* token, VpGetChildAddressCallback callback);
void     GetChildGenderAge(const char* token, VpGetChildGenderAgeCallback callback);
void     GetLoyaltyBalance(const char* token, VpGetLoyaltyBalanceCallback callback);
void     GetParentAddress(const char* token, VpGetParentAddressCallback callback);
void     GetParentChildAddress(const char* token, const char* child_id, VpGetParentChildAddressCallback callback);
void     PingHeaders(VpPingHeadersCallback callback);
const VpString&  getLastResponse() const;
```

The Virtual Piggy API is completely asynchronous so callbacks are used to determine when calls to the Virtual Piggy API have completed and resulting data from those calls is returned to the callback.

4.0 Example Usage

Probably the best form of introduction to using a new API is to see a simple example. The example below shows how to authenticate a child user and make a purchase:

```
//
//
// Set up the basket with a test purchase
//
//
void MySetupBasket(VpBasket* backet)
{
        backet->Currency = "USD";
        backet->removeAllItems();
        backet->addItem(new VpBasketItem("10 Coins", "10 Golden Coins", 0.99f, 1));
        backet->CalculateTotalCost();
}

//
//
// This callback gets called when the transaction is completed
//
//
int MyProcessTransactionCallback(VpProcessTransactionResult* result)
{
        // Display result
        printf(">>>> MyProcessTransactionCallback:\n");
        printf("---- ErrorMessage: %s\n", result->ErrorMessage.c_str());
        printf("---- Status: %d\n", result->Status);
        printf("---- DataXml: %s\n", result->DataXml.c_str());
        printf("---- TransactionIdentifier: %s\n", result->TransactionIdentifier.c_str());
        printf("---- TransactionStatus: %s\n", result->TransactionStatus.c_str());

        return 1;
}

//
//
// This callback gets called when the child user is authenticated
//
//
int MyAuthenticateChildForTransactionCallback(VpAuthenticateChildResult* result)
{
        // Display result
        printf(">>>> MyAuthenticateChildCallback:\n");
        printf("---- ErrorMessage: %s\n", result->ErrorMessage.c_str());
        printf("---- Status: %d\n", result->Status);
        printf("---- TransactionStatus: %s\n", result->TransactionStatus.c_str());
        printf("---- Token: %s\n", result->Token.c_str());


        // Make a purchase
        MySetupBasket(&g_Basket);
        VIRTUAL_PIGGY->ProcessTransaction(result->Token.c_str(), "Test", &g_Basket, MyProcessTransactionCallback);

        return 1;
}

void TEST_ChildPurchase()
{
        printf("******** TEST_ChildPurchase:\n");
        VIRTUAL_PIGGY->AuthenticateChild("ChIldName", "child_password", MyAuthenticateChildForTransactionCallback);
}
```

This example has 3 stages:

- Authentication – The initial call to AuthenticateChild() sends the supplied user details to Virtual Piggy where they will be authenticated. Once authentication has finished the MyAuthenticateChildForTransactionCallback callback function is called. In MyAuthenticateChildForTransactionCallback you should check to ensure that the user has been authenticated. If a valid token has been returned then user has been successfully authenticated
- Purchase – In MyAuthenticateChildForTransactionCallback we call MySetupBasket which adds an imaginary in-app purchase for 10 in game coins worth $0.99 then calls ProcessTransaction() to send the purchase request to VirtualPiggy. Note how we pass the child users token, the basket and the MyProcessTransactionCallback callback. Once the transaction completes (or fails) MyProcessTransactionCallback is called
- Completing the purchase – In MyProcessTransactionCallback information about the purchase is returned. To ensure that the purchase was valid check that Status has value of 1 and that the TransactionIdentifier is valid. You can also check that DataXml contains the correct item that was purchased. Once the purchase has been confirmed as valid you can enable the in-app content. You may want to add extra checking to ensure that the transaction is valid by checking that the transaction ID is valid with your own server.

5.0 API Reference

The full Virtual Piggy API is shown below:

5.1 Authentication

5.1.1 AuthenticateChild

Prototype:

```
void    AuthenticateChild(const char* user_name, const char* password, VpAuthenticateChildCallback callback);
```

Parameters:

- user_name - Childs user name
- password - Childs password
- callback - A callback that is called when the authentication is completed or fails

Description:
AuthenticateChild will authenticate a child user. If the child is authenticated then a valid token will be returned to the specified callback

Retuns:

```
struct VpAuthenticateChildResult
{
        VpString ErrorMessage;
        bool     Status;
        VpString Token;
        VpString TransactionStatus;
};
```

- ErrorMessage – If there was an error then this property will contain an error message. For additional information regarding the request call getLastReponse()
- Status – The status of the request (a value of 1 represents success)
- Token – The child users session token. You will need to temporarily store this as it is required by other methods
- TransactionStatus – [Add more info]

5.1.2 AuthenticateUser

Prototype:

```
void    AuthenticateUser(const char* user_name, const char* password, VpAuthenticateUserCallback callback);
```

Parameters:

- user_name - Users user name
- password - Users password
- callback - A callback that is called when the authentication is completed or fails

Description:
AuthenticateUser will authenticate a user. If the user is authenticated then a valid token will be returned to the specified callback

Returns:

```
struct VpAuthenticateUserResult
{
        VpString ErrorMessage;
        bool     Status;
        VpString Token;
        VpString TransactionStatus;
};
```

- ErrorMessage – If there was an error then this property will contain an error message. For additional information regarding the request call getLastReponse()
- Status – The status of the request (a value of 1 represents success)
- Token – The users session token. You will need to temporarily store this as it is required by other methods
- TransactionStatus – [Add more info]

5.2 Transactions

5.2.1 ProcessTransaction

Prototype:

```
void    ProcessTransaction(const char* token, const char* description, VpBasket* basket,
VpProcessTransactionCallback callback);
```

Parameters:

- token - Users token (returned from AuthenticateChild)
- description - Description of transaction
- basket - Goods that the child wants to purchase (see notes at the end of this section relating to the basket)
- callback - A callback that is called when the purchase is completed or fails

Description:
ProcessTransaction enables a child user to make a purchase safely.

Returns:

```
struct VpProcessTransactionResult
{
        VpString DataXml;
        VpString ErrorMessage;
        bool     Status;
        VpString TransactionIdentifier;
        VpString TransactionStatus;
};
```

- DataXml – The items that were purchased
- ErrorMessage – If there was an error then this property will contain an error message. For additional information regarding the request call getLastReponse()
- Status – The status of the request (a value of 1 represents success)
- TransactionIdentifier – The ID of the transaction
- TransactionStatus – [Add more info]

5.2.2 ProcessParentTransaction

Prototype:

```
void    ProcessParentTransaction(const char* token, const char* description, const char* child_id, const char*
payment_id, VpBasket* basket, VpProcessParentTransactionCallback callback);
```

Parameters:

- token - Users token (returned from AuthenticateUser)
- child_id - ID of child that user is purchasing goods for (retrieved from GetAllChildren)
- payment_id - ID of payment that user is using to make purchase (retrieved from GetPaymentAccounts)
- description - Description of transaction
- basket - Goods that the child wants to purchase (see notes at the end of this section relating to the basket)
- callback - A callback that is called when the purchase is complete or fails

Description:
ProcessParentTransaction enables a user to make a purchase on behalf of a child user. This is useful for when the child either runs out of allowance as a parent can still make the purchase.

Returns:

```
struct VpProcessParentTransactionResult
{
        VpString DataXml;
        VpString ErrorMessage;
        bool     Status;
        VpString TransactionIdentifier;
        VpString TransactionStatus;
};
```

- DataXml – The items that were purchased
- ErrorMessage – If there was an error then this property will contain an error message. For additional information regarding the request call getLastReponse()
- Status – The status of the request (a value of 1 represents success)
- TransactionIdentifier – The ID of the transaction
- TransactionStatus – [Add more info]

5.2.3 Users Basket

When a transaction takes place the items that are being purchased should be placed into the Basket. A basket is represented by the VpBasket structure and the items that can be placed into the basket are represented by a VpBasketItem. A basket can contain many items which enables the user to make multiple purchases in the same purchase request.

A short example that shows how to add items to a basket is shown below:

```
void MySetupBasket(VpBasket* backet)
{
        backet->Currency = "USD";
        backet->addItem(new VpBasketItem("10 Coins", "10 Golden Coins", 0.99f, 1));
        backet->addItem(new VpBasketItem("LevelsA", "Levels 11 to 20", 1.99f, 1));
        backet->CalculateTotalCost();
}
```

In this example we set the currency format to US dollars then add two items to the basket. Once all items are added we call CalculateTotalCost() to calculate the total cost of the items in the basket. VpBasket provides a number of methods that enable you to interact with the basket, including:

- addItem() - Adds an item to the basket
- removeItem() - Removes an item from the basket
- removeAllItens() - Removes all items from the basket
- CalculateTotalCost() - Calculates the total cost of all items in the basket (query the TotalCost variable to find the basket total)
- getItemCount() - Returns the total number of basket items

5.3 Subscriptions

5.3.1 ProcessSubscription

Prototype:

```
void    ProcessSubscription(const char* token, const char* description, VpSubscription* subscription,
VpProcessSubscriptionCallback callback);
```

Parameters

- token - Users token (returned from AuthenticateChild)
- description - Description of Subscription
- subscription - Subscription that the child wants to purchase
- callback - A callback that is called when the subscription purchase is complete or fails

Description:
ProcessSubscription enables a child user to purchase a subscription based in-app purchase. Note that besides providing a basket that describes the item(s) that the user has subscribed to, an additional VpSubscription must be supplied that described additional information about the subscription. Also note that the basket is provided within the VpSubscription structure. Use the Basket variable to access the basket.

Returns:

```
struct VpProcessSubscriptionResult
{
        VpString Identifier;
        VpString Name;
        VpString Type;
};
```

- Identifier – Subscription identifier
- Name – Name of subscription
- Type – Subscription class type

5.3.2 ProcessParentSubscription

Prototype:

```
void    ProcessParentSubscription(const char* token, const char* description, const char* child_id, const char*
payment_id, VpSubscription* subscription, VpProcessParentSubscriptionCallback callback);
```

Parameters

- token - Users token (returned from AuthenticateUser)
- description - Description of Subscription
- child_id - ID of child that user is purchasing goods for (retrieved from GetAllChildren)
- payment_id - ID of payment that user is using to make purchase (retrieved from GetPaymentAccounts)
- subscription - Subscription that the user wants to purchase for the child
- callback - A callback that is called when the subscription purchase is complete or fails

Description:
ProcessParentSubscription enables a user to purchase a subscription based in-app purchase on behalf of the child. Note that besides providing a basket that describes the item(s) that the user has subscribed to, an additional VpSubscription must be supplied that described additional information about the subscription. Also note tha the basket is provided within the VpSubscription structure. Use the Basket variable to access the basket.

Returns:

```
struct VpProcessSubscriptionResult
{
        VpString Identifier;
        VpString Name;
        VpString Type;
};
```

- Identifier – Subscription identifier
- Name – Name of subscription
- Type – Subscription class type

5.3.3 MerchantCancelSubscription

Prototype:

```
void    MerchantCancelSubscription(const char* subscription_id, VpMerchantCancelSubscriptionCallback callback);
```

Parameters

- subscription_id- A subscription identifier as returned by subscription transaction methods (ProcessSubscription and ProcessParentSubscription)
- callback - A callback that is called when the purchase is complete or fails

Description:
Cancels an existing subscription

Returns:

```
struct VpMerchantCancelSubscriptionResult
{
        VpString Result;                        // Cancel subscription result
};
```

- Result – The result of the subscription cancel

5.3.4 User Subscription

To make a successful subscription purchase you will need to fill out a VpSubscription structure that describes the subscription and item(s) that are being subscribed to before passing it to the subscription purchase calls. The VpSubscription structure looks like this:

```
struct VpSubscription
{
        VpString InitialCostCurrency;
        float    InitialCostValue;
        VpString Period;
        int      ExpiryInstances;
        VpBasket Basket;
};
```

- InitialCostCurrency - Name of currency that will be used for initial cost
- InitialCostValue - The initial cost
- Period - How long the subscription lasts (e.g. Weekly, Monthly)
- ExpiryInstances – [Add more info]
- Basket - Details of the subscription item

Below is an example showing how to set up a monthly subscription for a magazine:

```
void MySetupSubscription(VpSubscription* subscription)
{
        subscription->InitialCostCurrency = "USD";
        subscription->InitialCostValue = 2.99f;
        subscription->Period = "Monthly";
        subscription->ExpiryInstances = 1;

        subscription->Basket.Currency = "USD";
        subscription->Basket.addItem(new VpBasketItem("MyMagazine", "My cool magazine", 2.99f, 1));
        subscription->Basket.CalculateTotalCost();
}
```

5.4 Information Query

5.4.1 GetAllChildren

Prototype:

```
void    GetAllChildren(const char* token, VpGetAllChildrenCallback callback);
```

Parameters

- token - Users token (returned from AuthenticateUser)
- callback - A callback that is called when the process is complete or fails

Description:
GetAllChldren returns all child users that belong to the users account. This method should be used to obtain a child_id for methods where the parent has to perform actions on behalf of the child, such as purchasing an item or subscription when the child has no credit on their account

Returns:

```
struct VpGetAllChildrenResult
```

VpGetAllChildrenResult contains a list of children that can be searched for specific child users by name using VpGetAllChildrenResult::findChild(const char* child_name). The structure also contains an iterator that enables you to manually iterate through the list of children, e.g.:

```
int MyGetAllChildrenCallback(VpGetAllChildrenResult* result)
{
        // Display list of children
        printf(">>>> MyGetAllChildrenCallback:\n");
        for (VpGetAllChildrenResult::_Iterator it = result->begin(); it != result->end(); ++it)
                printf("---- Chlld: %s with ID = %s\n", (*it)->Name.c_str(), (*it)->Identifier.c_str());

        // Find the Mat child and make a purchase for them
        VpChild* child = result->findChild("Mat");
        if (child != NULL)
        {
        }

        return 1;
}
```

MyGetAllChildrenCallback contains a list VpChild structures. The definition of this structure is shown below:

```
struct VpChild
{
        VpString Identifier;      // Child identifier
        VpString Name;            // Name of entity
        VpString Type;            // Type
};
```

5.4.2 GetPaymentAccounts

Prototype:

```
void    GetPaymentAccounts(const char* token, VpGetPaymentAccountsCallback callback);
```

Parameters

- token - Users token (returned from AuthenticateUser)
- callback - A callback that is called when the process is complete or fails

Description:
GetPaymentAccounts returns all payment accounts that belong to the users account. This method should be used to obtain a payment_id for methods where the parent has to perform actions on behalf of the child, that requite payment such as purchasing an item or subscription when the child has no credit on their account

Returns:

```
        struct VpGetPaymentAccountsResult
```

VpGetPaymentAccountsResult contains a list of payment accounts that can be searched for specific methods of payment by name using VpGetPaymentAccountsResult::findAccount (const char* account_name). The structure also contains an iterator that enables you to manually iterate through the list of payment accounts, e.g.:

```
int MyGetPaymentAccountsCallback(VpGetPaymentAccountsResult* result)
{
        // Display list of children
        printf(">>>> MyGetPaymentAccountsCallback:\n");
        for (VpGetPaymentAccountsResult::_Iterator it = result->begin(); it != result->end(); ++it)
                printf("---- Account: %s with ID = %s\n", (*it)->Name.c_str(), (*it)->Identifier.c_str());

        // Get first available payment account
        VpPaymentAccount* account = *(result->begin());
        if (account != NULL)
        {
                // Make a purchase for the child
                MySetupBasket(&g_Basket);
                VIRTUAL_PIGGY->ProcessParentTransaction(g_Token.c_str(), "Test", g_ChildID.c_str(), account-
>Identifier.c_str(), &g_Basket, MyProcessParentTransactionCallback);
        }

        return 1;
}
```

MyGetPaymentAccountsCallback contains a list VpPaymentAccount structures. The definition of this structure is shown below:

```
struct VpPaymentAccount
{
        VpString Identifier;      // Payment identifier
        VpString Name;            // Name of entity
        VpString Type;            // Type
};
```

5.4.3 GetChildAddress

Prototype:

```
void    GetChildAddress(const char* token, VpGetChildAddressCallback callback);
```

Parameters

- token - Childs token (returned from AuthenticateChild)
- callback - A callback that is called when the process is complete or fails

Description:
GetChildAddress returns the address details of the child

Returns:

```
struct VpGetChildAddressResult
{
        VpString Address;
        VpString AttentionOf;
        VpString City;
        VpString Country;
        VpString ErrorMessage;
        VpString Name;
        VpString ParentEmail;
        VpString ParentPhone;
        VpString State;
        VpString Status;
        VpString Zip;
};
```

- Address – Main address line
- AttentionOf - Attention of Name
- City - City of residence
- Country - Country of residence
- ErrorMessage - Error message returned, empty if none
- Name – [Add more info]
- ParentEmail - Parents email address
- ParentPhone - Parents telephone number
- State - State of residence
- Status - Status
- Zip - Zip code

5.4.4 GetChildGenderAge

Prototype:

```
void    GetChildGenderAge(const char* token, VpGetChildGenderAgeCallback callback);
```

Parameters

- token - Childs token (returned from AuthenticateChild)
- callback - A callback that is called when the process is complete or fails

Description:
GetChildGenderAge returns the address details of the child

Returns:

```
struct VpGetChildGenderAgeResult
{
        int     Age;
        VpString ErrorMessage;
        VpString Gender;
        bool    Status;
};
```

- Age - Childs age
- ErrorMessage - Error message returned, empty if none
- Gender - Gender of child
- Status - true if success, false if not

5.4.5 GetLoyaltyBalance

Prototype:

```
void    GetLoyaltyBalance(const char* token, VpGetLoyaltyBalanceCallback callback);
```

Parameters

- token - Childs token (returned from AuthenticateChild)
- callback - A callback that is called when the process is complete or fails

Description:
GetLoyaltyBalance returns the childs loyalty balance

Returns:

```
struct VpGetLoyaltyBalanceResult
{
        int     Balance;
};
```

- Balance – The childs loyalty balance

5.4.6 GetParentAddress

Prototype:

```
void    GetParentAddress(const char* token, VpGetParentAddressCallback callback);
```

Parameters

- token - Parents token (returned from AuthenticateUser)
- callback - A callback that is called when the process is complete or fails

Description:
GetParentAddress returns the address details of the parent

Returns:

```
struct VpGetParentAddressResult
{
        VpString Address;
        VpString AttentionOf;
        VpString City;
        VpString Country;
        VpString ErrorMessage;
        VpString Name;
        VpString ParentEmail;
        VpString ParentPhone;
        VpString State;
        VpString Status;
        VpString Zip;
};
```

- Address – Main address line
- AttentionOf - Attention of Name
- City - City of residence
- Country - Country of residence
- ErrorMessage - Error message returned, empty if none
- Name – [Add more info]
- ParentEmail - Parents email address
- ParentPhone - Parents telephone number
- State - State of residence
- Status - Status
- Zip - Zip code

5.4.7 GetParentChildAddress

Prototype:

```
void    GetParentChildAddress(const char* token, const char* child_id, VpGetParentChildAddressCallback callback);
```

Parameters

- token - Parents token (returned from AuthenticateUser)
- child_id - ID of child that address query si for
- callback - A callback that is called when the process is complete or fails

Description:
GetParentChildAddress returns the address details of the specified child

Returns:

```
struct VpGetParentChildAddressResult
{
        VpString Address;
        VpString AttentionOf;
        VpString City;
        VpString Country;
        VpString ErrorMessage;
        VpString Name;
        VpString ParentEmail;
        VpString ParentPhone;
        VpString State;
        VpString Status;
        VpString Zip;
};
```

- Address – Main address line
- AttentionOf - Attention of Name
- City - City of residence
- Country - Country of residence
- ErrorMessage - Error message returned, empty if none
- Name – [Add more info]
- ParentEmail - Parents email address
- ParentPhone - Parents telephone number
- State - State of residence
- Status - Status
- Zip - Zip code

5.5 Housekeeping

5.5.1 Create

Prototype:

```
static void Create();
```

Description:
Creates an instance of the VirtualPiggy singleton class. Note that the Virtual Piggy API uses a singleton pattern to ensure that only one instance of the API is running and to give global access to Virtual Piggy.

5.5.2 Destroy

Prototype:

```
static void Destroy();
```

Description:
Destroys the Virtual Piggy instance. Virtual Piggy should not be used again until the singleton instance has been recreated.

5.5.3 Init

Prototype:

```
bool    Init(const char* api_key);
```

Description:
Initialises Virtual Piggy for use by a specific merchant. This is usually used in conjunction with setMerchantID() to set the ID of the merchant. Note that the api_key is obtained from Virtual Piggy when creating an account

5.5.4 Release

Prototype:

```
void    Release();
```

Description:
Release Virtual Piggy's internals, do not use once released. To make usable again, call Init()

5.5.5 Update

Prototype:

```
void    Update();
```

Description:
Updates Virtual Piggy internals. This method needs to be called every game or app frame update.

5.5.6 setMerchantID

Prototype:

```
void    setMerchantID(const char* merchant_id)
```

Description:
Sets the Merchant ID to use (obtained from Virtual Piggy by creating an account)

5.6 Other Methods

5.6.1 PingHeaders

Prototype:

```
void    PingHeaders(VpPingHeadersCallback callback);
```

Parameters

- callback - A callback that is called when the process is complete or fails

Description:
PingHeaders is a test function that can be used to test for a connection with the Virtual Piggy web server

Returns:

```
struct VpPingHeadersResult
{
        bool    Success;
};
```

- Success – true if successful, false if not

5.6.2 getLastResponse

Prototype:

```
const VpString&         getLastResponse() const
```

Description:
getLastResponse returns the last SOAP request response. Examining this information can be useful if you encounter unexpected errors or data.

Returns:

- Last SOAP response