# Multimedia Fusion 2 XNA extension SDK

Welcome to the MMF2 XNA extension SDK. Please read this file up to the end. After reading it, you will be able to create new extensions for the MMF2 XNA runtime. If you are fluent with the extension kit, you will see that the XNA extension kit is very similar to Java, iPhone and Flash extension kit.

This extension kit contains all the source code of the XNA runtime. I (Francois) would be really happy to hear any suggestion you have about my source code, as it might help me improve it. If you have an idea about the source code, please write it to me : francois@clickteam.com
Of course, the source code of the Flash runtime is NOT public, or in any kind of GNU licence. It remains the property of Clickteam, and you should not copy it or distribute it in any way.

## Getting ready to program.

### Visual Studio and XNA Game Studio 4.0

You will need a Visual Studio 2010 Express (at least) and XNA Game Studio 4.0 installed on your machine to develop.

### Loading the XNA project

- Unzip one of the zip files (RuntimeXNA.zip, RuntimeXNA_XBOX.zip, RuntimeXNA_Phone.zip) in a directory on your machine
- Open the sln file in Visual Studio

### Creating a new extension.

To create a new extension :

- Right click on the "**Extensions**" folder in the project window.
- Choose "**Add / New Item**"
- Enter for the name of your extension, the name "**CRun**" (respecting the case) followed by the file name of your extension. For example if the extension name is kchisc, then the name of the class should be "CRunkchisc"
- Don't forget to derive your class from “CRunExtension”
- Start programming your extension.

You will have to incorporate your extension in the main XNA runtime. So do so :

- Open the file "Extensions / CExtLoad.as"
- In the loadRunObject method, go to the end of the big "if" structure.
- Enter the following code :

```
if (string.Compare(“Name_Of_Extension"))
{
        object=new CRunName_Of_Extension();
}
```

This line will incorporate your extension in the runtime.  The name in quotes must be the name of the extension as it is on disc.

You are free to create any classes you want for your extension, but the classes must lie in the one extension cs file.

Please note that the sources of MMF Runtime included in the SDK contain the sources of all the extension made at that time. If you need to see example extensions, just have a look at all the files in the extensions folder.

### Running applications with your extension.

To be able to debug your extensions in XNA builder, you need a version of MMF with the XNA exporter. Just select the build type corresponding to the project you are working on, and build the application in it.

### Publishing your extension

Once you are happy with your extension, you need to publish it. To do so :
- Start **ExtXNAMaker** from the Flash SDK Start menu
- Click on the “Create Ext File button”
- Choose your extension cs file in the file selector
- Choose the directory where to create your ext file
- A new .ext file will be created. Copy this file in the Data\Runtime\XNA folder corresponding to the type of runtime you are working on. Please note, that unless you use functions specific to a type of device (Windows, XBOX or Windows Phone), the same ext file can work for all the runtimes.

## The extension class

As stated before, the name of this class should be "CRunNameOfExtension".  It should be in the Extension package.
The class is derived from an class, name CRunExtension.
The class contains functions similar to the C++ extensions, for the runtime. Only more structured. Lets see the functions in details.
In MMF Flash runtime, all object's name start with "C".

### Default variables

Two variables are defined in the  class parent from the object :
- ho : points to the CObject object. Equivalent to the headerObject structure in MMF. Ho is also useful as callback functions are defined to exchange data with the main runtime.
- rh : points to the CRun object. Equivalent to the runHeader structure in MMF.

### Constructor
Nothing special to do in the constructor of the extension. But you can have your own code in it.

### public override int getNumberOfConditions()

This function should return the number of conditions contained in the object (equivalent to the CND_LAST define in the ext.h file).

**public override bool createRunObject(CFile file, CCreateObjectInfo cob, int version)**

This function is called when the object is created. As XNA objects are just created when needed, there is no EDITDATA structure. Instead, I send to the function a CFile object, pointing directly to the data of the object (the EDITDATA structure, on disc).
The CFile object allows you to read the data. It automatically converts PC-like ordering (big or little Indian I cant remember) into Flash ordering. It contains functions to read bytes, shorts, int, colors and strings. Read the documentation about this class at the end of the document.
"Version" contains the version value contained in the extHeader structure of the EDITDATA.

So all you have to do, is read the data from the CBinaryFile object, and initialise your object accordingly.

**public override void destroyRunObject(bool bFast)**

Called when the object is destroyed. Due to garbage collection, this routine should not have much to do, as all the data reserved in the object will be freed at the next GC. bFast is true if the object is destroyed at end of frame. It is false if the object is destroyed in the middle of the application.

**public override int handleRunObject()**

Same as the C++ function. Perform all the tasks needed for your object in this function. As the C function, this function returns value indicating what to do :

- CRunExtension.REFLAG_ONESHOT : handleRunObject will not be called anymore
- CRunExtension.REFLAG_DISPLAY : displayRunObject is called at next refresh.
- Return 0 and the handleRunObject method will be called at the next loop.

**public override void displayRunObject(SpriteBatch batch)**

Called to display the object. If your object needs to be displayed on the screen,. Use the SpriteBatch object to paste your textures or text on the screen.

**public override void pauseRunObject()**

Called when the application goes into pause mode.

**public override void continueRunObject()**

Called when the application restarts.

**public override CFontInfo getRunObjectFont()**

Equivalent to the C++ version. This function returns a CFontInfo object, a small object equivalent to the LOGFONT structure in C. See at the end of the document the definition of the CFontInfo object.

**public override void setRunObjectFont(CFontInfo fi, CRect rc)**

Called when the font of the object needs to be changed. The rc parameter is null when no resize is needed.

**public override int getRunObjectTextColor()**

Returns the current color of the text as an Integer.

**public override void setRunObjectTextColor(int rgb)**

Sets the current color of the text.

**public Cmask getRunObjectCollisionMask(int flags)**

If implemented, this function should return a CMask object that contains the collision mask of the object. Return null in any other case.

**public override bool condition(int num, CCndExtension  cnd)**

The main entry for the evaluation of the conditions.
• num : number of the condition (equivalent to the CND_ definitions in ext.h)
• cnd : a pointer to a CCndExtension object that contains useful callback functions to get the parameters.
This function should return true or false, depending on the condition.

**public override void action(int num, CActExtension act)**

The main entry for the actions.
• num : number of the action, as defined in ext.h
• act : pointer to a CActExtension object that contains callback functions to get the parameters.

**public override CValue expression(int num)**

The main entry for expressions.
•num : number of the expression

To get the expression parameters, you have to call the getExpParam() method defined in the "ho" variable, for each of the parameters. This function returns a CValue which contains the parameter. You then do a getInt(), getDouble() or getString() with the CValue object to grab the actual value.

This method should return a CValue object containing the value to return. The content of the CValue can be a integer, a double or a String. There is no need to set the HOF_STRING flags if your return a string : the CValue object contains the type of the returned value.

## Callback functions

The "ho" variable in the extension object gives you access to the CExtension object, which is a derivative of the main CObject class. I have programmed a few callback functions in the CExtension object. Here are these functions :

**public int getX()**

Returns the current X co-ordinate of the object (hoX).

**public int getY()**

Returns the current Y coordinate of the object (hoY)

**public int getWidth()**

Returns the current width of the object (hoImgWidth).

**public int getHeight()**

Returns the current height of the object (hoImgHeight).

**public void setPosition(int x, int y)**

Changes the position of the object, taking the movement into account (much better than poking into hoX and hoY).

**public void setX(int x)**

Changes the position of the object (hoX), and takes care of the movement and refresh. Same remark as setPosition.

**public void setY(int y)**

Same as setX for Y co-ordinate. Same remark as setPosition.

**public void setWidth(int width)**

Change the width of the object, taking care of the hoRect fields.

**public void setHeight(int height)**

Same as setWidth, for height.

**public void loadImageList(short[] imageList)**

This method should be called in the createRunObject method of your object. If your object uses images stored in the image bank, you must call this method so that the proper images are loaded.
Just make an array with all the handles of the images, the size should be the exact number of images to load. Call this method (it may take some time to return). All the images will be loaded in the runtime.

**public CImage getImage(int handle)**

Call this function to retrieve an image from a handle. The image must have been previously loaded with loadImageList. The value returned could be null if the image could not be loaded, but this is very unlikely to occur.
CImage is part of the "Banks" package. So you should import it with "import Banks.*".

**public void reHandle()**

If you returned a REFLAG_ONESHOT value in your handleRunObject method, this will reinforce the method to be called at each loop.

**public void generateEvent(int code, int param)**

Generate an event with the specific code. The parameter can be recuperated with the ho.getEventParam method. You should prefer the push_event method, specially if your event is generated in a listener (MouseListener, EventListener) as the listener events occurs in another thread and the event routines are not multi-thread proof.

**public void pushEvent(int code, int param)**

This is the method of choice to generate an event.

**public void pause()**

Pauses the application, when you have some lengthy work to perform (like opening a file selector).

**public void resume()**

Resumes the application at the end of your work.

**public void redraw()**

Forces a redraw of the object (the displayRunObject routine is called at next refresh).

**public void redisplay()**

Forces a redraw of the background of the application. You should use this if your object is a background object.

**public void destroy()**

Destroys the object at the end of the current loop.

**public int getExtUserData()**

Returns the private field of the extHeader structure.

**public void setExtUserData(int data)**

Changes the private field of the extHeader structure.

**public int getEventCount()**

Returns the rh4EventCount value, used in controls to trigger the events.

**public CValue getExpParam()**

Returns the next expression parameter.

**public CObject getFirstObject()**

Returns the first object currently defined in the frame. Should be used in conjunction with getNextObject().

**public CObject getNextObject()**

Returns the next object in the list of objects of the frame. Returns null if no more objects is available. This method will return the extension object it is called from.

## CRun callback functions

You access the CRun object via the "rh" variable defined in the CRunExtension class. This object contains three methods used to define global data.

Some extensions need to communicate between objects. In C++ this was done simply by defining global variable in the code. I have defined three functions in CRun to allow you to create global classes.

**public void addStorage(CExtStorage data, int id)**

Adds a new object to the storage, with the "id" identifier. "id" is a simple integer number. The storage class must be derived from the class CExtStorage. This function has no effect if an object with the same identifier already exists.

**public CExtStorage getStorage(int id)**

Returns the storage object with the given identifier. Returns null if the object is not found.

**public void delStorage(int id)**

Deletes the object with the given identifier.

## CActExtension callback functions

The CActExtension object is transmitted to the extension "action" method when a action is called. This object contains callback function to gather the parameters of the action.
These functions want two parameters:
• The CRun object (available in the extension as "rh").
• The number of the parameter in the action, starting at 0

**public CObject getParamObject(CRun rhPtr, int num)**

Returns the CObject pointed to by the PARAM_OBJECT.

**public int getParamTime(CRun rhPtr, int num)**

Returns the time value in milliseconds.

**public short getParamBorder(CRun rhPtr, int num)**

Returns the border parameter.

**public int getParamDirection(CRun rhPtr, int num)**

(obsolete, but might be used in old extensions). Returns a direction from 0 to 31.

**public PARAM_CREATE getParamCreate(CRun rhPtr, int num)**

Returns a pointer to the PARAM_CREATE object (for future use maybe).

**public int getParamAnimation(CRun rhPtr, int num)**

Returns the number of the animation.

**public int getParamPlayer(CRun rhPtr, int num)**

Returns the number of the player.

**public PARAM_EVERY getParamEvery(CRun rhPtr, int num)**

Returns a pointer to the Every parameter.

**public Keys getParamKey(CRun rhPtr, int num)**

Return the key code contained in the parameter.

**public int getParamSpeed(CRun rhPtr, int num)**

Returns a speed, from 0 to 100.

**public CPositionInfo getParamPosition(CRun rhPtr, int num)**

Returns a pointer to a CPositionInfo classe that contains the X and Y coordinate.
CPositionInfo is defined in the Params package.

**public int getParamJoyDirection(CRun rhPtr, int num)**

Returns a joystick direction.

**public PARAM_SHOOT getParamShoot(CRun rhPtr, int num)**

Returns a pointer to the PARAM_SHOOT object contained in the action (for future use maybe).

**public PARAM_ZONE getParamZone(CRun rhPtr, int num)**

Returns a pointer to the PARAM_ZONE parameter.

**public int getParamExpression(CRun rhPtr, int num)**

Returns the value contained in the expression.

**public int getParamColour(CRun rhPtr, int num)**

Returns a color, as an integer.

**public int getParamFrame(CRun rhPtr, int num)**

Returns a number of frame.

**public int getParamNewDirection(CRun rhPtr, int num)**

Returns a direction, from 0 to 31.

**public short getParamClick(CRun rhPtr, int num)**

Returns the click parameter (left/middle/right button).

**public string getParamExpString(CRun rhPtr, int num)**

Returns the result of a String expression.

**public double getParamExpDouble(CRun rhPtr, int num)**

Returns as a double the result of the evaluation of the parameter.

## CCndExtension callback functions

The CCndExtension object is transmitted to the extension when calling the condition method. It contains callbacks to gather the parameters of the condition. Most of the method are identical to the ones in CActExtension, with the following differences :

**Missing methods:**
getParamPosition
getParamCreate
getParamShoot

**Different returns**
The getParamObject method returns a pointer to the PARAM_OBJECT object (as in the C++).
public PARAM_OBJECT getParamObject(CRun rhPtr, int num)

**public bool compareValues(CRun rhPtr, int num, CValue value)**

In the C++ version, when you had a PARAM_COMPARAISON parameter in the condition, the condition routine returned a long value, and MMF was automatically doing the comparison with the parameter. You have to call this method in the Flash version. For example, a condition with a PARAM_COMPARAISON as first parameter, the end of the condition method should be:

return cnd.compareValues(rh, 0, returnValue);

Where returnValue is a CValue object containing the value to compare with.

**public bool compareTime(CRun rhPtr, int num, int t)**

Same as the previous function, for PARAM_CMPTIME parameters, where "t" is the time to compare to.

## Useful objects

I will document now some of the objects of the runtime you will be using to program your extension.

### The CFile object

This object is sent to you in the createRunObject method. It automatically points to the start of the extension data (EDITDATA structure, in memory). It automatically performs the indian translation between PC values and Flash values.

**byte[] readArray(int size)**

Read a array of bytes.

**void read(byte[] array, int size)**

Read a array of bytes.

**void read(byte[] array)**

Read a array of bytes.

**void skipBytes(int n)**

Skips bytes in the file.

**int getFilePointer()**

Returns the current file pointer position.

**void seek(int pos)**

Change the file pointer position.

**byte readAByte()**

Reads one byte.

**short readAShort()**

Reads one short (two bytes).

**int readAInt()**

Reads an integer (4 bytes).

**char readAChar()**

Reads an char  (2 bytes).

**int readAColor()**

Reads a color making it compatible with iOS color values (inversion of Red and Blue values).

**string readAString()**

Reads a string that finishes with 0. The string is allocated for you, you should release it later in the code.

**string readAString(int size)**

Reads a string of the given size. The string is allocated for you, you should release it later in the code.

**CFontInfo readLogFont()**

Reads a LOGFONT structure into a CFontInfo object. The object is allocated for you, you should release it later in the code.

**CFontInfo readLogFont16()**

Reads a LOGFONT structure into a CFontInfo object in the old 16 bits format. The object is allocated for you, you should release it later in the code.

## <u>The CValue object</u>

The CValue object is used in expression evaluation. It can contain an Integer, a double or a String.

**public CValue(int i)**

Creates the object as type integer with the given value.

**public Cvalue(double d)**

Creates the object as type double with the given value.

**public Cvalue(string s)**

Creates the object as type string with the given value.

**public Cvalue(CValue v)**

Creates the object with the type and value of the given CValue object.

**public int getType()**

Returns the type of the object. The return value can be :

CValue.TYPE_INT : and integer
CValue.TYPE_DOUBLE : a double
CValue.TYPE_STRING : a string

**public int getInt()**

Returns an integer. (if the object is of TYPE_DOUBLE, converts the value to int).

**public double getDouble()**

Returns a double.

**public string getString()**

Returns the string. Warning, if the object is of type int or double, this will NOT convert the value to string.

**public void forceInt(int value)**

Forces the value of type int into the object (if the object was previously double or int, then changes the type).

**public void forceDouble(double value)**

Forces the value of type double.

**public void forceString(string value)**

Forces a string.

**public void forceValue(CValue value)**

Forces the content and type of the CValue.

**public void setValue(CValue value)**

Change the content of the object, respecting its type.

## The CFontInfo object

The CFontInfo object is a replacement of the LOGFONT structure in C++. It contains the name of the font, its height, weight and attributes.

**Fields contained in the object :**

```
public int lfHeight;
public int lfWeight;
public byte lfItalic;
public byte lfUnderline;
public byte lfStrikeOut;
public string lfFaceNameString;
```

**public function copy(CFontInfo source)**

Copies the content of the given CFontInfo object.

**The CRect object**

This object is intended as a replacement of the C++ RECT structure.

**Fields :**

```
public int left;
public int top;
public int right;
public int bottom;
```

public void load(CFile file)

Reads a CRect from the Cfile object.

**public void copyRect(CRect srce)**

Copies the content of the given CRect object in the object.

**public bool ptInRect(int x, int y)**

Returns true if the given point is located in the rectangle.

**public bool intersectRect(CRect rc)**

Returns true if the two rectangle intersect.

## The CPoint object

The CPoint object is a replacement of the C++ POINT structure. Is only contains two fields:

```
public x;
public y;
```