Overview

Using batch processing, the Resource Compiler (RC) is able to batch process all of the asset files for a specific platform in one go only.

To do this, you invoke the RC with the **/job** switch, where you specify a path to a Job XML file.

For example, to build the main part of the SDK assets as shipped, we use:

```
Bin64\RC\rc.exe /p=PC /job=Bin64\RC\RCJob_Build_SDK.xml
```

For more example Job XML files you can look in the `Bin64\RC` folder of the SDK package.

Job XML Format

The Job XML format is specified using an XML file, which you can edit using any Text Editor of your choosing.

The RC will execute the file in a top-down manner, parsing and executing (where appropriate) the content of the batch script.

The root tag of a Job XML file is **<RCJobs>**:

```
<RCJobs>
        <!-- contents of the file -->
</RCJobs>
```

Properties

It is important to understand that a Job XML is a template for a build, which is specialized using parameters passed on the command line. Generally speaking the most important is the '**p**' parameter (platform).

During the batch execution each command-line parameter is converted to a property, just like every other RC property.

For a list of properties used by the RC, run 'rc.exe /help'.

Property values can be accessed inside the Job XML script using the '${propertyname}' construct, which can be used to set other properties, or to conditionally execute parts of the batch.

Default Properties

Depending on the complexity of a batch script, the number of properties used may become significant and typing values on the command line may become a nuisance.

Therefore, you can specify default values for properties at the start of the job XML file. For this, use a **<DefaultProperties>** XML block.

In the SDK build we specify default folder names (amongst others) for the assets using default properties:

```
<DefaultProperties game="GameSDK" engine="Engine" loc="Localization" />
```

Default properties can be used to initialize values before the rest of the batch script executes to reasonable defaults.

Command-line parameters override properties specified in a DefaultProperties block.

Computed Properties

You may wish to dynamically compute properties depending on the values of other properties.

It is possible to pass in a "feature flag" and adjust conversion-properties accordingly (i.e, a property that controls other properties). For this, use a **<Properties>** XML block.

In the SDK we can pass a "streaming" property, or if it's not specified we set it according to the platform-property '**p**'.

We dynamically deduce the property 'do_streaming' which we use to control texture streaming later.

```
<DefaultProperties streaming="auto" />
<if streaming="auto">
        <if p="PC">
                <Properties do_streaming="0" />
        </if>
        <if p="X360">
                <Properties do_streaming="1" />
        </if>
</if>
<ifnot streaming="auto">
        <Properties do_streaming="${streaming}" />
</ifnot>
```

Unlike default properties, properties defined through a **<Properties>** block are not conditional, the property value will be updated regardless of any command-line parameters with the same name.

Conditionals (if & ifnot)

Sometimes, it makes sense to only execute a certain processing step for a specific platform, or alternate processing of specific assets depending on some property.

For this, use an **<if>** XML block:

```
<if propertyname='compare'>
        <!-- only evaluated if the value of property 'propertyname' is equal to 'compare' -->
</if>

<ifnot propertyname='compare'>
        <!-- only evaluated if the value of property 'propertyname' is not equal to 'compare' -->
</ifnot>

<!-- it's also possible to compare to another property value -->
<if propertyname='${anotherproperty}' />
```

Scope

Properties in an RC script have a certain scope. In general the XML is evaluated sequentially and property values are persistent.

However, the following tags will introduce a new scope for properties:

- Job: Any property values specified (as XML attributes) inside the 'Job tag' are scoped to that job invocation
- Run: Any property values specified (as XML attributes) inside the 'Run tag' are scoped to that invocation, and any properties introduced or modified by the invoked XML will be forgotten once the "Run" returns

Using these constructs, you can write a "subroutine" inside the XML that can be called multiple times with different variables (where the variable values are stored in named properties).
For example:

```
<MySubRoutine>
  <!-- Takes parameters "what" and "destination" -->
  <Job Input="${What}" Zip="${Destination}/MySubRoutine.pak" />
</MySubRoutine>

<MainRoutine>
  <!-- Call subroutine for packing *.foo into C:/Some/Path/MySubRoutine.pak -->
  <Run Job="MySubRoutine" What="*.foo" Destination="C:/Some/Path" />
</MainRoutine>
```

In this example, let's suppose 'MainRoutine' was executed (either using command-line `/jobtarget=MainRoutine` or a `<Run Job="MainRoutine"/>` as a child of the root XML).
The **<Run>** tag will be used to "call" the "subroutine" 'MySubRoutine' by name. The attributes What and Destination are specified inline in the example and will be visible inside 'MySubRoutine'.
Once the 'MySubRoutine' "call" has "returned" to the 'MainRoutine', the value of What and Destination will go back to their previous value (if any).

Any property values currently visible can be used and modified in the 'MySubRoutine', however any modification or new property values introduced will not survive the "return".

For reasons of readability or re-use, you can also use **<Properties>** or **<DefaultProperties>** tags when using "subroutines".
For example:

```
<RCJobs>
  <Foo>
    <Properties A="a" />
    <!-- Do something with A B and C -->
  </Foo>
  <Bar>
    <DefaultProperties B="b" />
    <Properties A="x" />
    <Run Job="Foo" C="c" />
    <!-- Here, A==x -->
  </Bar>
  <Run Job="Bar" />
</RCJobs>
```

Lets suppose you run this example file, the values of A, B and C will be 'a', 'b' and 'c' respectively at the commented line.
The value 'x' of A is passed into Foo, but then overwritten. However, once Foo returns, the value 'x' will be restored.
Also, it would be possible to change the value of B by passing the command-line parameter "/B=x" when running the job to modify from the default behavior.

In general, you should always keep the following order of blocks in your Job XML file to keep the behavior of the RC in line with expectations when reading the XML file:

- DefaultProperties blocks
- Properties blocks
- Job blocks
- Run blocks

Asset Conversion Jobs

Asset conversion is defined through **<Job>** XML blocks, and that execute serially (although the sub-parts of some jobs can be executed in parallel - see the RC property threads).

Jobs iterate over a set of files (in the local file system), thus invoking RC processing for each file in the set and as if the RC was specifically invoked for that file (while using the properties currently set through the batch script for execution).

In general, conversion of assets is a lossy process and you should always keep the original files separate from the conversion results.
A converted file can usually not be transformed back to the original file.
The RC is designed to take data from one location (specified using 'sourceroot') and write converted data to another location (specified using 'targetroot').

The syntax of a <Job> XML block is like the following:

```
<Job sourceroot="path to folder inside which to search for files" input="file mask of files to select for
processing" targetroot="path to folder in which to store the processed files" conversion_parameter="some value"
/>
```

In general, it makes sense to only execute a single conversion type inside a single job block, thus "input" is typically set to a file extension mask, for example: *.tif

Here is an example of how to process textures from CryTiff (*.tif) files to platform-specific files (typically *.dds for PC):

```
<ConvertJob>
        <Job sourceroot="${source_game_folder}" input="*.tif" targetroot="${target_game_folder}"
imagecompressor="CTsquish" streaming="0" />
</ConvertJob>
```

This job will search for all CryTiff files in the value of the property with name 'source_game_folder' and convert them using CTsquish and not preparing for streaming assets. The result will be stored in the value of the property with the name 'target_game_folder'.

The set of meaningful conversion parameters that can be set depends on the set of filetypes that pass the input-set restrictions.

Thus, it is sensible to limit the file set to a single filetype and only specify conversion parameters for that specific conversion to keep your Job XML file understandable.

The List of meaningful conversion parameters for a given conversion can be found by calling 'rc.exe /help'

In general, conversion will take into account the relative path from the 'sourceroot' property to the matched file and use the same relative path to store the 'targetfile', but relative to the 'targetroot' property.

Please note that any job that is run with SourceRoot==TargetRoot will not match any files (and thus will do nothing) regardless of actual files. The RC has several transformations that do not change the filename of the source or the target (as this would cause problems), and even though there are several transformations that could work, for simplicity this is treated as an error.

Pass-through (copy)

Sometimes it makes sense to not do any processing for a given set of files.

In that case, you can set the special conversion parameter 'copyonly' to 1, which will just copy all the files matching the input-set restrictions to that target folder:

```
<CopyJob>
        <Job sourceroot="${source_folder}" input="*.xml" targetroot="${target_folder}" copyonly="1" />
</CopyJob>
```

Input Set

The input set can be further controlled by the following special properties:

- **recursive:** (default is 1).
  When set to 0 the search will only consider files in the 'sourceroot ' folder, but not in subfolders.
- **input:** (default is '*').
  Matches the relative path to the specified expression(s) and only passes files that match one or more of the expression(s).
  Use '*' to indicate zero or that more characters are accepted in that location.
  Split multiple expressions using semicolons, '*.a;*.b' will pass all filenames ending with '.a' or '.b'
- **exclude:** (default is none).
  Like "input", except any file matching the expression(s) is not passed.
  Exclude overrides "input", so a file matching both expressions will NOT be in the input set.
- **listfile** (default is none).
  Opens the specified expression as a text file and restricts the input set to any file in the list.
  The list file is a a text file, with each file on a separate line.
- **exclude_listfile** (default is none).
  Like 'listfile' exception, any file matching the list is not passed.
  Exclude overrides 'listfile', so a file in both lists will NOT be in the input set.

File Type Override

Sometimes, you might want to interpret a specific file with a specific type, even though it doesn't have a matching extension.

In that case, you can use the property **overwriteextension** to force the RC to pick the converter for the overridden extension i.e rather than the converter it would have picked for the actual extension.

For example, to force a .mtl file to be converted as if they were XML files you can use:

```
<Job input="*.mtl" overwriteextension="xml" ...rest of the job... />
```

Packing Jobs

Once you have converted and/or copied all the files that should be provided on the end user file system you can pack them into *.pak files.

For this use the **<PakJob>** group and add **<Job>** blocks with input sets that should be added to a *.pak.

Here is an example for generating a Textures.pak file containing all *.dds files previously converted from CryTIFF:

```
<PakJob>
	<Job sourceroot="${processed_files_folder}" input="*.dds" zip="${target_folder}\Textures.pak" />
</PakJob>
```

You can optionally specify the 'zip_compression' property which controls compression.

When set to 0 the *.pak file will not be compressed, this might be suitable for certain files that frequently require file seeking at runtime.

Running Jobs

Once you have specified all the job groups that you need, you can run them using a **<Run>** block:

```
<ConvertJob>
	<!-- conversion jobs -->
</ConvertJob>

<CopyJob>
	<!-- copy jobs -->
</CopyJob>

<PakJob>
	<!-- packing jobs -->
</PakJob>

<Run Job="ConvertJob"/>
<Run Job="CopyJob"/>
<Run Job="PakJob"/>
```


Since the file is parsed top down, make sure your job groups are specified before the <Run> blocks that reference them.