FslBuildGen overview

[1. Summaries 2](#_Toc482711263)

[2. Introduction 2](#_Toc482711264)

[3. Package description 3](#_Toc482711265)

[3.1. Package types 3](#_Toc482711266)

[3.2. UsesFeature <UsesFeature> 4](#_Toc482711267)

[3.3. ImportTemplate < ImportTemplate> 4](#_Toc482711268)

[3.4. Dependency <Dependency> 4](#_Toc482711269)

[3.5. External dependency <ExternalDependency> 6](#_Toc482711270)

[3.6. Define <Define> 6](#_Toc482711271)

[3.7. Platform <Platform> 7](#_Toc482711272)

[3.8. Variant <Variant> 8](#_Toc482711273)

[3.9. Option <Option> 9](#_Toc482711274)

[3.10. Sub packages 9](#_Toc482711275)

[3.11. Additional information 9](#_Toc482711276)

[Appendix 1 - Tool configuration 10](#_Toc482711277)

[A1.1. Package configuration 10](#_Toc482711278)

[A1.2. Package blacklisting 10](#_Toc482711279)

[Appendix 2 - Graphical dependency overview 11](#_Toc482711280)

[Appendix 3 - Experimental Visual Studio 2017 support 12](#_Toc482711281)

[Appendix 4 - Known limitations 13](#_Toc482711282)

# Summaries

This document describes the FslBuildGen tool. The tool is a cross-platform build-file generator. Which main purpose is to keep all build files consistent, in sync and up to date.

# Introduction

The Demo Framework is designed to run on multiple platforms and the exact number of supported platforms will grow over time. Furthermore it has a rather large pool of samples which again grows over time. This means that keeping the build files for all platforms consistent, up to date and in sync is a challenge and prone to human errors.

Today we have eleven OpenGLES2 samples, nineteen OpenGLES3 samples and six OpenVG samples which is 36 sample applications. The demo framework runs on four platforms so we have to maintain 36\*4= 144 build files for the samples alone.

FslBuildGen was created to address that issue it’s a cross-platform build-file generator. It takes a high level package description and from that it generates build files for each supported platform. The tool is in development and its main focus is to solve current build issues for the Demo Framework, it could be used in other places in the future.

The tool operates on software packages. Each package has a type and a set of dependencies. To help ensure that the generated build files are valid it requires that the packages and their dependencies form a directed acyclic graph.

# Package description

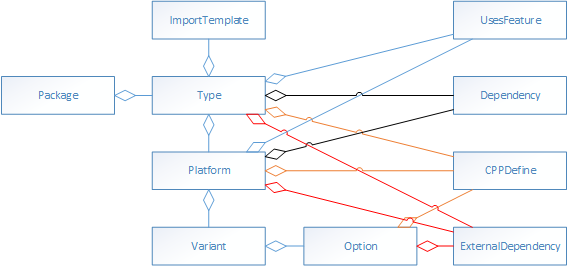
The package description files are designed to require as little maintenance as possible. Which is one of the reasons why a package is required to keep its header and source files in specific directories.

* Include files must be located inside a folder named ‘include’
* Source files must be located inside a folder named ‘source’

During the build file generation process the ‘include’ and ‘source’ folders will be scanned and all found files will be added to the build file. So any platform specific file will have to be properly guarded so it can be compiled on all platforms.

A Package description consist of a

* Type: a type can contain
  + UsesFeature
  + ImportTemplate
  + Dependencies
  + External dependencies
  + Defines
  + Platform(s): a platform can contain
    - UsesFeature
    - Dependencies
    - External dependencies
    - Defines
    - Variant(s): a variant contain
      * Options: an option contain
        + Defines
        + External dependencies



## Package types

The package type decides the output the package generates.

### Executable <Executable>

A package that builds an executable file

|  |  |  |
| --- | --- | --- |
| **Name** | Required | The name of the package (Must match the directory name is saved in) |
| **NoInclude** | Optional | false = A include directory must exist (default)  true = No include directory exist |

### Library <Library>

A package that builds a static library.

|  |  |  |
| --- | --- | --- |
| **Name** | Required | The name of the package (Must match the directory name is saved in) |

### ExternalLibrary < ExternalLibrary>

A package that describes an external precompiled library.

|  |  |  |
| --- | --- | --- |
| **Name** | Required | The name of the package (Must match the directory name is saved in) |

## Requirement <Requirement>

Describes the need for a specific feature or extension to be available for this build to work. Feature requirements are inherited from all dependencies. So the top level package always has a complete feature requirement.

|  |  |  |
| --- | --- | --- |
| **Name** | Required | The name of the requirement that is being used |
| **Type** | Required | feature, extension |
| **Extends** | Optional for feature.  Required for extension | The feature that it extends |
| **Version** | Optional | A version string. |

## ImportTemplate < ImportTemplate>

Imports a template from one of the template import directories specified in the config file. It allows you to group common package setup into one file that can then be pulled in from various packages as needed.

|  |  |  |
| --- | --- | --- |
| **Name** | Required | The name of the template that should be imported. |

## Dependency <Dependency>

Describes a dependency on another package

|  |  |  |
| --- | --- | --- |
| **Name** | Required | The name of the dependency |
| **Access** | Optional | Public = (default) all dependencies are inherited to everyone that pulls in this package.  Private = This package gains access to defines, include directories etc.  Link = We only link against this dependency, no defines or include directories are added. |

The following table illustrates how the package dependencies that were defined resolves to actual dependencies by FslBuildGen.

|  |  |  |  |
| --- | --- | --- | --- |
| **Package dependencies** | | **Resolved result** | |
| P2 public dependency on P1  P3 public dependency on P2 |  |  | P2 public dependency on p1  P3 public dependency on P2  *P3 public dependency on p1* |
| P2 public dependency on P1  P3 public dependency on P1  P4 public dependency on p2  P4 public dependency on p3 |  |  | P2 public dependency on P1  P3 public dependency on P1  P4 public dependency on p2  P4 public dependency on p3  *P4 public dependency on p1* |
| P2 link dependency on P1  P3 public dependency on P2 |  |  | P2 link dependency on p1  P3 public dependency on P2  *P3 link dependency on p1* |
| P2 link dependency on P1  P3 public dependency on P1  P4 public dependency on p2  P4 public dependency on p3 |  |  | P2 link dependency on P1  P3 public dependency on P1  P4 public dependency on p2  P4 public dependency on p3  *P4 public dependency on p1* |
| P2 link dependency on P1  P3 public dependency on P1  P4 public dependency on p2  P4 public dependency on p3  P4 link dependency on p1 |  |  | P2 link dependency on P1  P3 public dependency on P1  P4 public dependency on p2  P4 public dependency on p3  *P4 public dependency on p1* |
| P2 private dependency on P1  P3 public dependency on P2 |  |  | P2 private dependency on p1  P3 public dependency on P2  *P3 link dependency on p1* |
| P2 private dependency on P1  P3 public dependency on P1  P4 public dependency on p2  P4 public dependency on p3 |  |  | P2 private dependency on P1  P3 public dependency on P1  P4 public dependency on p2  P4 public dependency on p3 |
| P2 private dependency on P1  P3 public dependency on P1  P4 public dependency on p2  P4 public dependency on p3  P4 private dependency on p1 |  |  | P2 private dependency on P1  P3 public dependency on P1  P4 public dependency on p2  P4 public dependency on p3  *P4 public dependency on p1* |

## External dependency <ExternalDependency>

Is used to quickly describe an external library. If that library might be used by other packages a package of the type ExternalLibrary should be created instead.

Using this tag basically requires you to be sure that his is the only place this dependency will be required!

An external dependency tag contains

|  |  |  |
| --- | --- | --- |
| **Name** | Required | The name of the dependency. This is the library named that will be used to link against it. |
| **Include** | Optional | The include path to its header files (if any). Environment variables can be used. |
| **Location** | Optional | The library location (if relevant). Environment variables can be used. |
| **Access** | Optional | Public = All dependent packages will inherit this dependency. (default)  Private = The dependency is private to this package. The include path is not inherited. |
| **Type** | Required | Headers = a header file only dependency  StaticLib = a dependency to a static lib (with optional header files) |

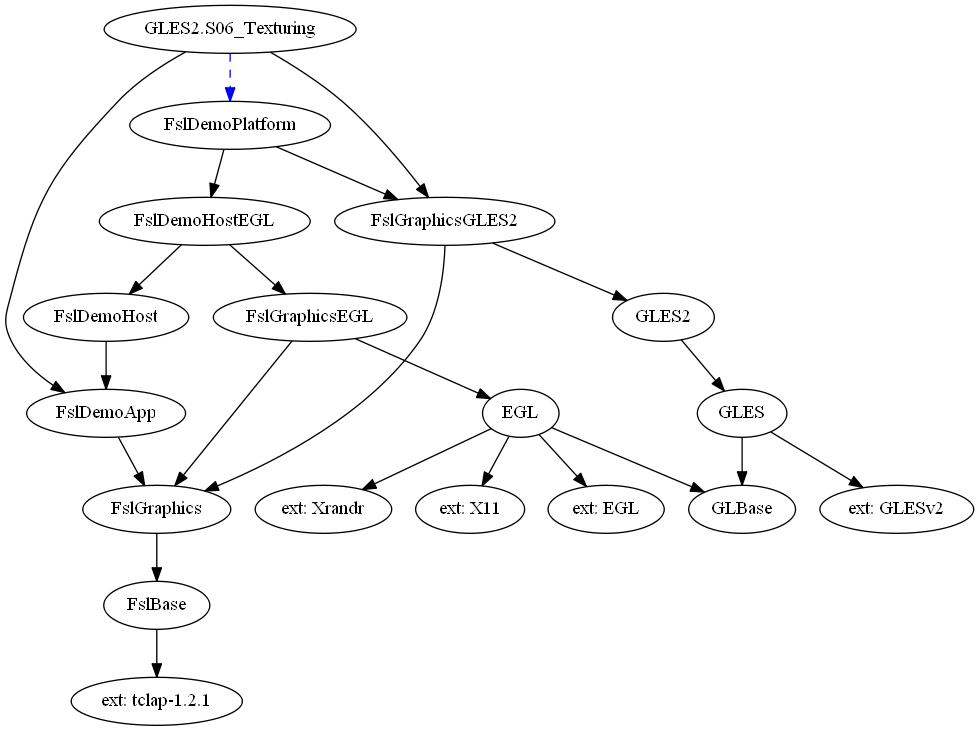
## Define <Define>

Defines a C++ #define that will be set when building.

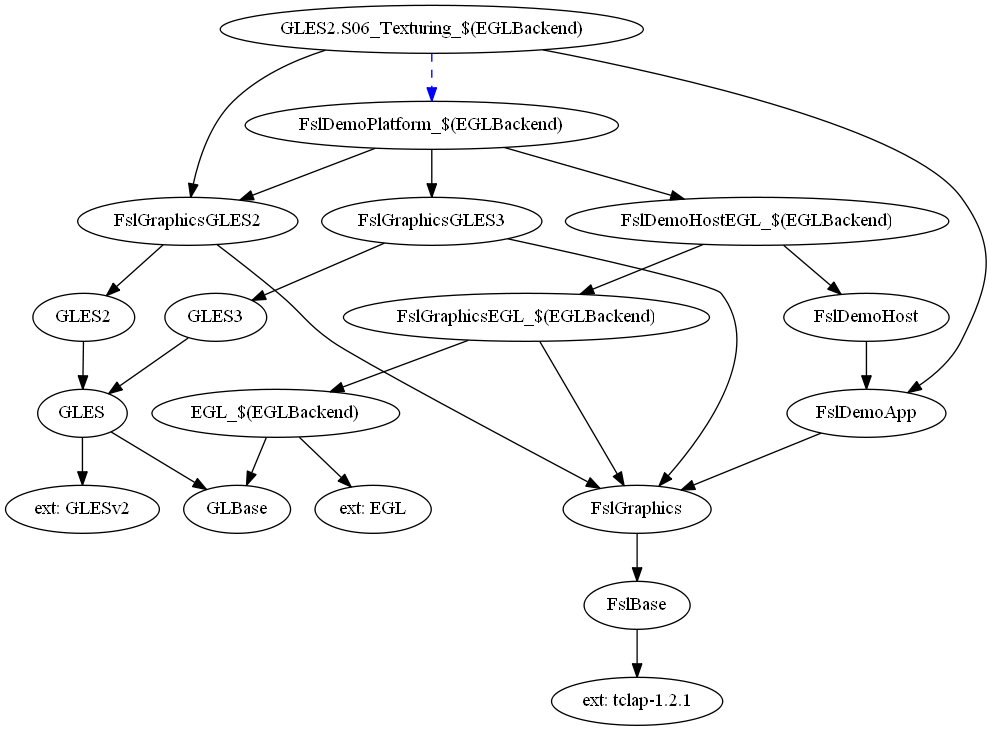
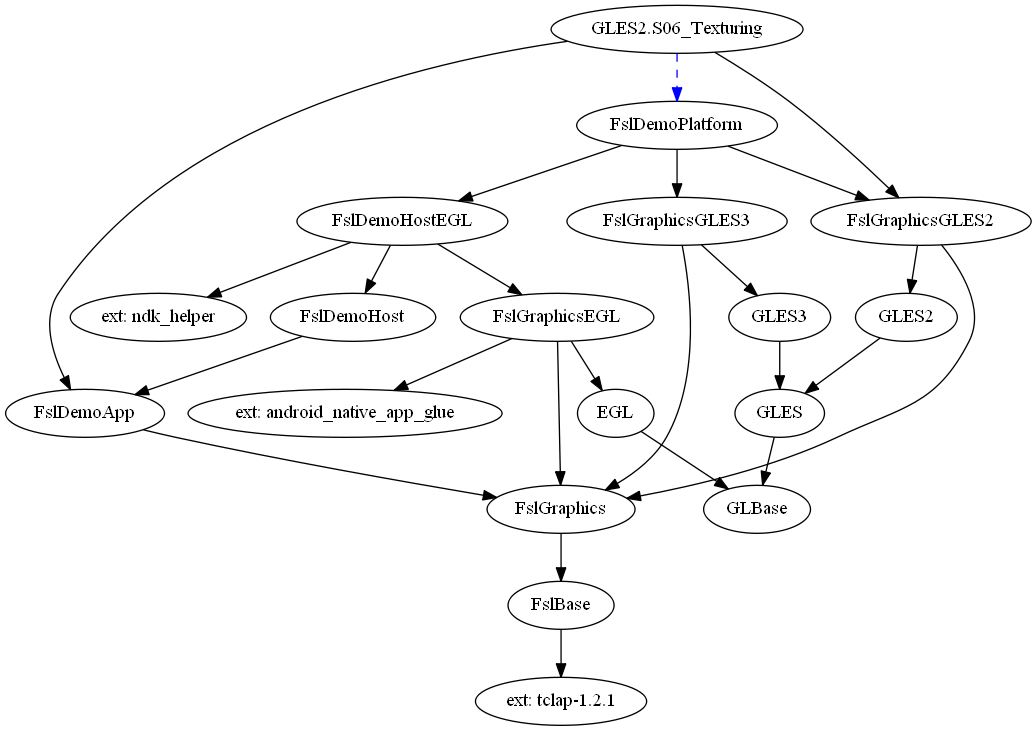
|  |  |  |
| --- | --- | --- |
| **Name** | Required | The name of the dependency |
| **Access** | Required | Public = All dependent packages will inherit the define.  Private = The define is private to this package. |

## Platform <Platform>

The platform tag is used to specify platform specific configuration and variants.



|  |  |  |
| --- | --- | --- |
| **Name** | Required | Android = Android specific configuration  Ubuntu = Ubuntu specific configuration  Yocto = Yocto specific configuration  Windows = Windows specific configuration |
| **Supported** | Optional | Defaults to ‘true’ if ‘false’ this package is not supported on the given platform. |
| **ProjectId** | Optional | This is only relevant for the windows build generator. It is the project GUID that is inserted into the project, it has to be unique for the package (if other packages dependent upon it) |

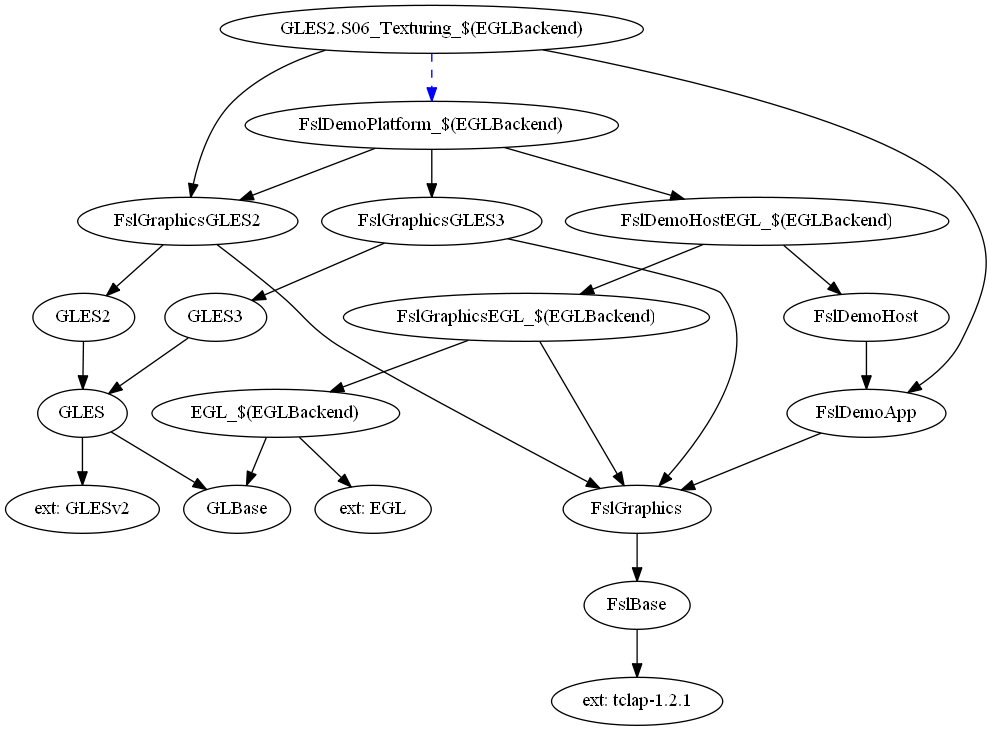


The three graphs above show the package configurations for the Android, Ubuntu and Yocto build of the same sample. The blue arrow in the graph shows a ‘link’ only dependency.

## Variant <Variant>

A variant introduces a specific variant of the package that is inherited by everyone that depends upon the package. So be very careful when introducing one.

We currently use a variant to allow different EGL backends to be used under the Yocto build[[1]](#footnote-1).



As you can see above the EGL package introduces the EGLBackend variant and all dependent packages are affected by it.

|  |  |  |
| --- | --- | --- |
| **Name** | Required | The name of the variant |
| **Extend** | Optional | false = not allowed to extend an existing variant (default)  true = extending a existing variant |
| **Type** | Optional | Normal = a normal variant with a fixed name  Virtual = the variant name is defined by an environment variable like this $(MY\_VARIANT)[[2]](#footnote-2). |

The extend flag can be utilized to extend an option of an existing variant. However the variant and option name must exist, no new options can be added.

## Option <Option>

An option specifies the exact configuration to use for a given variant.

The option can define CPPFlags and pull in external dependencies.

|  |  |  |
| --- | --- | --- |
| **Name** | Required | The name of the option |

## Sub packages

Sub package names allow you to group related items. It is only enabled for Executable packages.

We use it to group all GLES2, GLES3 and OpenVG samples.

So all GLES2 samples are named ‘GLES2.’

This ensures that samples with identical names for GLES2 and GLES3 never collide as their full executable name will always be prefixed with the GLES2/3 tag.

A package named ‘GLES2.Test’ is expected to be stored under the path ‘GLES2/Test’.

## Additional information

* All path mentioned in Fsl.gen files are expected to be unix style and case sensitive (use slash not backslash).
* Environment variables can be used in some locations. An environment variable is always of the format $(NAME). Build file generators will transform them as necessary to fit the target environment.

1. Tool configuration

The tool expects the file ‘FslBuildGen.xml’ to contain its configuration.

It contains information about where

* the build generator template folder is located
* The root directories for packages
* Template import directories.
* Various package configurations. A package configuration specifies a list of package root directories used to locate packages in the system.
  1. Package configuration

We currently have two package configurations

* default (-t default). Which is the default used when running the tool without any ‘-t’ parameter. It sets up the ThirdParty and DemoFramework package locations. So that the loaded package can find its dependencies.
* sdk (-t sdk). Sets up the location of all known packages that is considered part of the sdk. That is currently: ThirdParty, DemoFramework, DemoApps/GLES2, DemoApps/GLES3, DemoApps/OpenVG
  1. Package blacklisting

Each package location listed in the configuration can contain a blacklist tag like this:

<Blacklist Name="T3DStressTest"/>

To blacklist a specific package (in this case the package named T3DStressTest). A blacklisted package is never loaded.

1. Graphical dependency overview

From V0.5.2 of the tool it’s possible to generate a graphical dependency overview picture for the processed packages. It requires that the ‘dot’ executable from the [graphviz](http://www.graphviz.org/)[[3]](#footnote-3) package is present in the path.

To generate the png files just add ‘—graph’ to the command line and it will generate a dependency png for each platform that was processed.

It can be used to gain a quick overview of the package structure.

1. Experimental Visual Studio 2017 support

* If fslbuildgen is run with “—VSVersion 2017” it will generate visual studio 2017 project files. However, this is still an experimental feature.

1. Known limitations

* Most generators are very basic for now.
* Platform Ubuntu Supported still generates the build file, however the root build.sh files are protected so they don’t compile it.

1. The blue arrow in the graph shows a ‘link’ only dependency. [↑](#footnote-ref-1)
2. The virtual variant is currently used under windows to allow a batch file to configure which OpenGLES emulation layer that is being used. [↑](#footnote-ref-2)
3. <http://www.graphviz.org/> [↑](#footnote-ref-3)