FeatStd Overview

Introduction

FeatStd is the short form for FEAT Standard library. It's purpose is to provide common basic functionalities, like Datatypes, Classes, Macros, Platform/OS abstraction, etc. for various FEAT products. FeatStd is also configurable to the needs of a software product, having feature flags to enable self-contained functionalities, like IPC support, UnitTest framework (based on google test), logging, etc. resp. overall settings like thread-safety support, fractional number handling, etc.

Historically FeatStd was introduced, when the need came up to have a common root for software products in CGI Studio context, which basically need not depend on each other. Precisely it was founded as the request for having CGI Studio Courier running without CGI Studio Candera came at a time where Courier had many dependencies to Candera like Datatypes, Container classes, a.s.o.

FeatStd itself has the requirement to have no CGI Studio context. Generally spoken it shall not have product context at all. It shall be a conglomeration of real common functionalities, which can be re-used in any FEAT software product, which is based on C++.

FeatStd build system

Due to the fact that FeatStd shall have no product context at all, the build system itself was based on plain CMake, including tool-chain sample files to cross-compile for various target platforms (to be found in featstd/cmake/toolchain-files)

As mentioned above FeatStd is configurable and is completely dependent on the configuration file generated via the CMake pass (FeatStd/Config.h in the binary dir). This file is a must-include for any and every file using FeatStd, in any product, as it provides whole configuration for FeatStd. The benefit for this approach is that there is not handicraft work needed to set compiler defines to configure FeatStd, but simple set the include path to the generated source directory (typically called <binary-dir>/gensrc) in the binary output folder. Additionally to the include path to the output directory also a include path to the source directory has to be set. More on that later.

And finally the last and very important setting is the FEATSTD_BUILD_MODE, which unfortunately must be compiler flag dependent on the build configuration. CMake supports four different built-in build configurations, which are *Debug*, *Release*, *MinSizeRel* and *RelWithDebInfo*. For all these four build configuration CMake provides compiler variables called CMAKE_CXX_FLAGS_<build-mode> (e.g. CMAKE_CXX_FLAGS_MINSIZEREL), which all have to be decorated with the FEATSTD_BUILD_MODE define. FeatStd's CMakeLists already provides a macro just for decorating the compiler flags, and can either be called or used as template by the product/application.

See here how it is called and what is does:

```
# ------------------------------------------------------------------------
# Adds FEATSTD_BUILD_MODE setting to compiler flags
# ------------------------------------------------------------------------
macro(FeatStdAddBuildModeDefine)
    set(definitionsFlag "-D")
    if (MSVC)
        string(REPLACE - / definitionsFlag ${definitionsFlag})
    endif()

    set(CMAKE_C_FLAGS_DEBUG "${CMAKE_C_FLAGS_DEBUG} ${definitionsFlag}FEATSTD_BUILD_MODE=
FEATSTD_BUILD_MODE_DEBUG")
    set(CMAKE_C_FLAGS_MINSIZEREL "${CMAKE_C_FLAGS_MINSIZEREL} ${definitionsFlag}FEATSTD_BUILD_MO
FEATSTD_BUILD_MODE_MINSIZEREL")
    set(CMAKE_C_FLAGS_RELEASE "${CMAKE_C_FLAGS_RELEASE} ${definitionsFlag}FEATSTD_BUILD_MODE=
FEATSTD_BUILD_MODE_RELEASE")
    set(CMAKE_C_FLAGS_RELWITHDEBINFO "${CMAKE_C_FLAGS_RELWITHDEBINFO} ${definitionsFlag}FEATSTD_
FEATSTD_BUILD_MODE_RELWITHDEBINFO")

    set(CMAKE_CXX_FLAGS_DEBUG "${CMAKE_CXX_FLAGS_DEBUG} ${definitionsFlag}FEATSTD_BUILD_MODE=
FEATSTD_BUILD_MODE_DEBUG")
    set(CMAKE_CXX_FLAGS_MINSIZEREL "${CMAKE_CXX_FLAGS_MINSIZEREL} ${definitionsFlag}FEATSTD_BUIL
FEATSTD_BUILD_MODE_MINSIZEREL")
    set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS_RELEASE} ${definitionsFlag}FEATSTD_BUILD_MODE
FEATSTD_BUILD_MODE_RELEASE")
    set(CMAKE_CXX_FLAGS_RELWITHDEBINFO "${CMAKE_CXX_FLAGS_RELWITHDEBINFO} ${definitionsFlag}FEAT
FEATSTD_BUILD_MODE_RELWITHDEBINFO")
endmacro()
```

In every CMake run FeatStd produces a pretty informative output, describing under which conditions it was build, which features are enabled and which CMake variables are exported for further use in the product using FeatStd. The latter is quite of interest, as the caller (the software component which is binding FeatStd), might re-use the exported CMake settings produced by FeatStd. All FeatStd exported CMake variables are put into the internal CMake cache and therefore available in the buildsystem after FeatStd was added to the product (using CMake's add_subdirectory()).

Above was mentioned that it is enough to simply add the include directories, which basically are the FeatStd source directory and the generated source directory, but it may happen that FeatStd binds itself a component (e.g. google test), where also the include directory has to be propagated. Additionally FeatStd may add link libraries which have to be bound to the overall software product. To ease this fact of configuration it is recommended to simply propagate values of the *exported FeatStd CMake variables* for include directories(FEATSTD_EXPORTED_INCLUDE_DIRECTORIES), definitions(FEATSTD_EXPORTED_DEFINITIONS) and link libraries(FEATSTD_EXPORTED_LINK_LIBRARIES) to the whole software product, when FeatStd is in the game. All other exported FeatStd variables are ready to be used, in case this information is needed.

For a typical output of a CMake configuration run of FeatStd please refer to the device specific documentation:

- Device Specific Documentation

Revision #3
Created 2023-02-27 07:20:44 UTC by Tsuyoshi.Kato
Updated 2023-06-13 08:10:29 UTC by Tsuyoshi.Kato