Visteon Build System (VBuild) Introduction

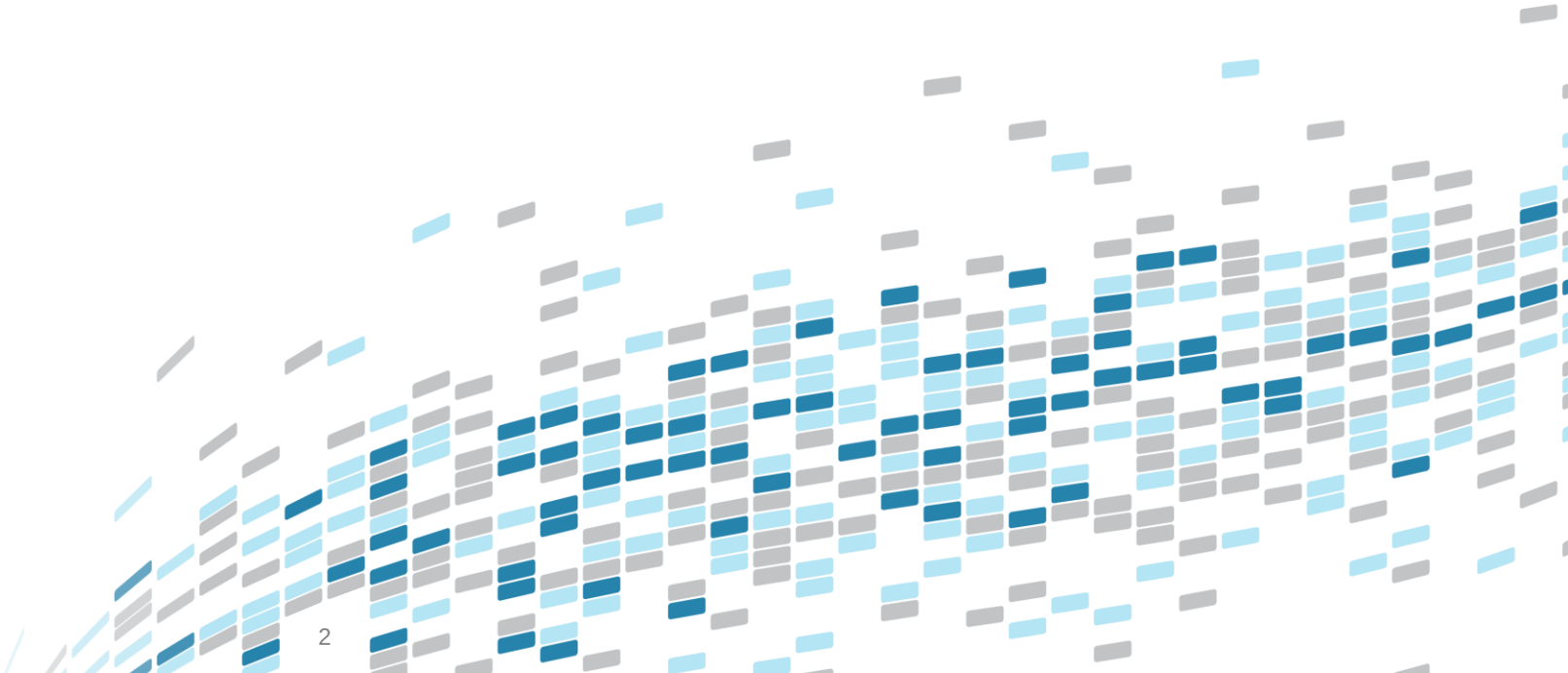
June 2020

Visteon®

Agenda

Visteon

- What is VBuild and Why is was created?
- Current Status
- Roadmap
- How it works?
  - Install
  - Workspace
  - Simple binary
  - Library
  - Dependencies
  - Chained dependencies
  - Multiple binaries



What is VBuild and Why is was required?

Visteon

What is VBuild?

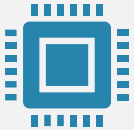
- VBuild is a new build system strategy that can be used to compile any Visteon project independently of the compiler or OS.
- It is is based on CMake and Ninja to provide friendly syntax for build scripts and fast compilation.
- Allows a software component to be compiled for different platforms with minimal changes or even no change at all.
- It is designed to support more modular solutions.
- It works in Windows or Linux (Docker) hosts.
- It has a complete user guide that allows users to implement it in any project.

Why develop VBuild?

- Visteon use different build strategies between platforms (AUTOSAR/QNX) and even between groups.
- The current build strategies are not optimized for Visteon projects and the compilation time for some strategies are extremely long.
- Current strategy doesn't provide a decoupled and modularized approach.

Current status

Visteon



VBuild is working with different target platforms

- AUTOSAR project with GHS compiler (Replacing the old GNU make build strategy)
- QNX project (Replacing the QNX generic GNU make build strategy)
- MinGW and Linux GCC for PC compiling (it can be used to compile unit test code)

It is being used in different projects

- T1XX/BT1XX (VP/GP code)
- DI Ford Bookshelf (VP code)
- Platform (BX755, S2.8, U725)

Provide support for different development tools

- Unit Test Frameworks (Unity/CMock)
- Code Coverage (Gcov, Vcast)
- Source code formatting (clang-format)
- Dependency graph generation
- QNX profiling build
- Memory usage analysis
- Warning parsing and filtering

Roadmap

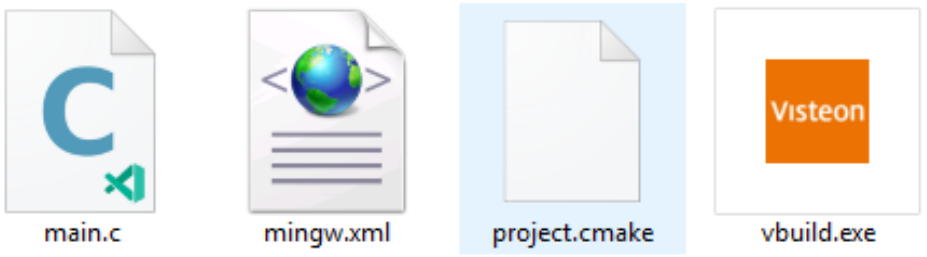
Visteon

Unit Test	Support for more Unit Test Frameworks (Gtest/Gmock, Boost Test Library, etc.)
Static Analysis	Support for static analysis tools (Integrate platform support for Coverity into Vbuild core)
Documentation	Support for documentation generation tool (Doxygen)
New features	Additional features can be requested in https://git.visteon.com/di/tools/vbuild/issues

How it works? - Install

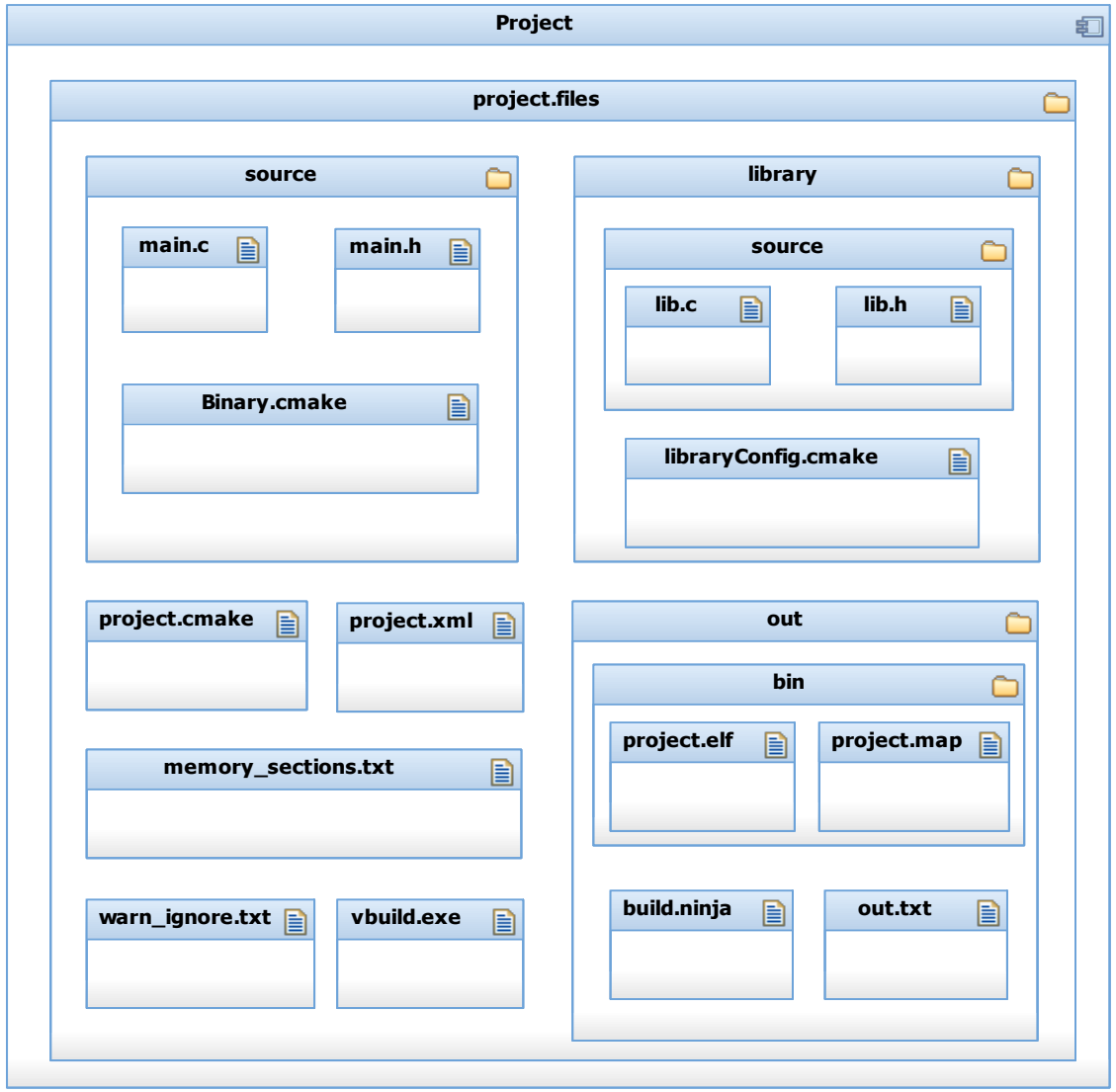
Visteon

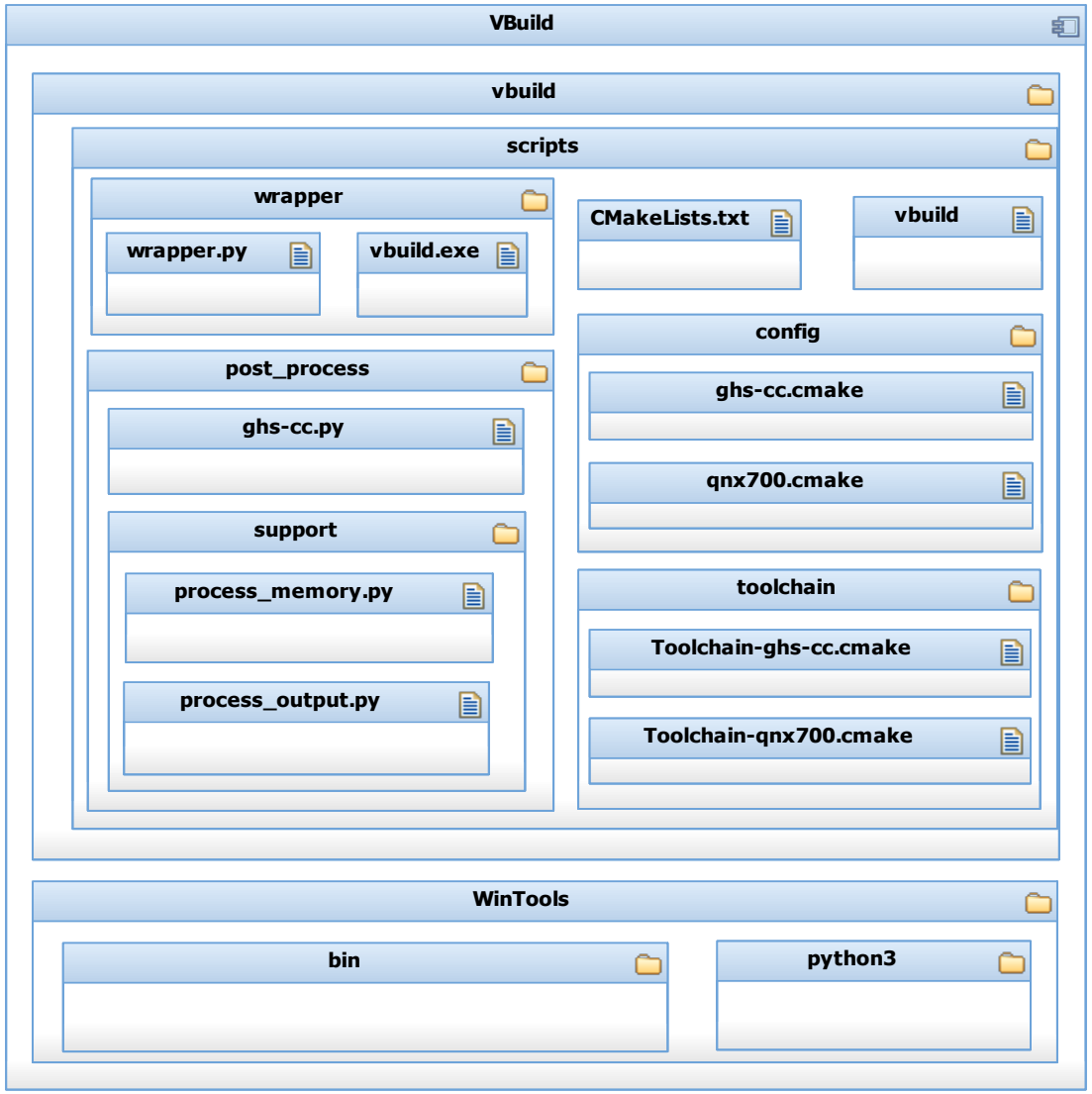
- The minimal required files to use VBuild are:
  - vbuild.exe that can be downloaded from: https://git.visteon.com/di/tools/vbuild/raw/Platform_WinTools/vbuild.exe (This is only required for Windows for Linux Docker container must have it already installed)
  - XML file with the basic project configurations
  - A CMake script file to specify the source files to compile
  - At least one source file



The latest version of the VBuild user guide can be downloaded from:
https://git.visteon.com/di/tools/vbuild/raw/master/develop/doc/VBuild_User_Guide.docx

How it works? - Workspace





How it works? - Simple binary

Visteon

- The XML file must contain at least the name of the project, the out path, compiler to use, main CMake file and tools destination path (based on the project root).
- The main cmake file will contain the command to generate the binaries. Standard Cmake commands like add_executable are used to create a binary. Variables like **BINARY_NAME** that contains the project name and **CMAKE_CURRENT_DIR** that contains the path where this script is located are available to use. Other variables are explained in the user guide.
- To compile the code type: **vbuild mingw**

Where **mingw** is the xml that is desired to be used to build the code. By default it will use the debug profile otherwise release must be specified.

Output folder where binary code will be located and tool folder where Vbuild is download to are created during compilation.

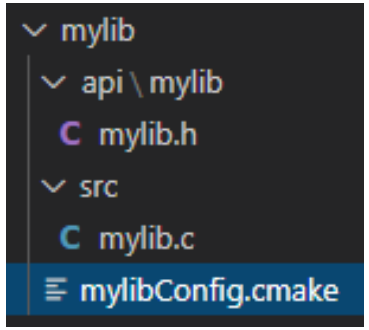
```xml
<?xml version="1.0" encoding="UTF-8"?>
<project root="./">
    <name>Example1</name>
    <compiler toolchain="mingw-gcc"/>
    <out>out</out>
    <script>project.cmake</script>

    <tools dest="./" />
</project>
```

```
# Create Binary
add_executable(${BINARY_NAME}
    ${CMAKE_CURRENT_SOURCE_DIR}/main.c
)
```

How it works? - Library

Visteon

- What makes VBuild modular is how the dependency libraries are handled.
- A library requires to have a cmake file with the following structure **<lib_name>Config.cmake** where **<lib_name>** is the name of the library that will be created.
- It also required to have the c files and header files.
- Header files can be exported to other modules or be private and only visible to library source files.
- The library is created using the **add_library** function. The variable **MODULE_NAME** is the same as **<lib_name>** part of the cmake file.
- The created library can be **STATIC, SHARED, OBJECT** or event an interface for header only libraries
- All of his is done using standard Cmake functions.




```
cmake_minimum_required (VERSION 3.14)

# Create library target
add_library(${MODULE_NAME} STATIC
    ${CMAKE_CURRENT_SOURCE_DIR}/src/mylib.c
)

# Directories with header files to be shared
target_include_directories(${MODULE_NAME}
    PUBLIC
    ${CMAKE_CURRENT_SOURCE_DIR}/api
)
```

How it works? - Dependencies

Visteon

- A library is only compiled if it is included into a binary.
- To add a library to a binary it is necessary to be added as a dependency by using the custom function t**arget_add_dependencies** (details of this custom function can be obtained from the user guide).
- But the library will not be found if the search path where this library can be found is not updated. To add a search path the function **add_search_path** can be used (details of this custom function can also be found in the user guide).
- The path doesn't need to be exact to where the library resided. A search path will scan all the subfolders and detect all the **<library_name>Config.cmake** files it founds, and they will be available to be added as dependencies.

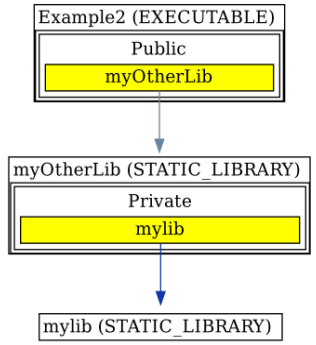
```
# Add all the required modules paths
add_search_paths(
    INCLUDE
    ${CMAKE_CURRENT_SOURCE_DIR}/libraries
)

# Create Binary
add_executable(${BINARY_NAME}
    ${CMAKE_CURRENT_SOURCE_DIR}/main.c
)

# Add dependency libraries
target_add_dependencies(${BINARY_NAME}
    PUBLIC
    mylib
)
```

How it works? - Chained dependencies

Visteon

- The custom function **target_add_dependencies** can be used also on libraries (except on **INTERFACE** libraries).
- The libraries should have their dependencies solved by adding the proper libraries that it depends on.
- The dependencies for a library can also be **PRIVATE** or **PUBLIC**. It will have the same behavior as the **PRIVATE** and **PUBLIC** headers.
- These are chained dependencies because the binary doesn't need to add these libraries directly. These dependencies are added through other libraries.
- A dependency graph can be generated by using the argument option **--gen-dep** on the vbuild command.

```
cmake_minimum_required (VERSION 3.14)

# Create library target
add_library(${MODULE_NAME} STATIC
    ${CMAKE_CURRENT_SOURCE_DIR}/src/myOtherLib.c
)

# Directories with header files to be shared
target_include_directories(${MODULE_NAME}
    PUBLIC
    ${CMAKE_CURRENT_SOURCE_DIR}/api
)

# Add dependency libraries
target_add_dependencies(${MODULE_NAME}
    PRIVATE
    mylib
)
```



How it works? - Multiple binaries

Visteon

- A project can have more than one binary.
- The common approach is to have the main CMake project to include all the binaries that the project must build.
- To add a binary the custom function **binary_script** must be used (details on the custom function can be found in the user guide).
- If several binaries need to be added, then the custom function **binary_scripts** can be used to specify all of them at once.
- An specific binary can be compiled by adding the library name after the profile to use (**vbuild mingw release <binary_name>**) but not only binaries but any library or custom target also.

```
# Add binary scripts
binary_script(${CMAKE_SOURCE_DIR}/gm.di.t1.bsp.delivery/di.build.2022.gm.t1.vip/tgt/app.cmake        GM_${PROJECT_NAME}_VP_APP)
binary_script(${CMAKE_SOURCE_DIR}/bsp-delivery-gm-t1-fbl/T1_FBL/T1Fbl/fbl.cmake                    GM_${PROJECT_NAME}_VP_FBL)
```

```
binary_scripts(${PROJ_BINARIES}
    ${CMAKE_SOURCE_DIR}/Build/Applications/components/AlertManager/AlertManager.cmake
    ${CMAKE_SOURCE_DIR}/Build/Applications/components/AlertOwner/AlertOwner.cmake
    ${CMAKE_SOURCE_DIR}/Build/Applications/components/DeviceInformation/DeviceInformation.cmake
    ${CMAKE_SOURCE_DIR}/Build/Applications/components/DisplayFeatures/DisplayFeatures.cmake
    ${CMAKE_SOURCE_DIR}/Build/Applications/components/EarlyHMI/EarlyHMI.cmake
    ${CMAKE_SOURCE_DIR}/Build/Applications/components/RemoteHMI/RemoteHMI.cmake
    ${CMAKE_SOURCE_DIR}/Build/Applications/components/RttManager/RttManager.cmake
    ${CMAKE_SOURCE_DIR}/Build/Infrastructure/BacklightService/BacklightService.cmake
    ${CMAKE_SOURCE_DIR}/Build/Infrastructure/ButtonManager/ButtonManager.cmake
    ${CMAKE_SOURCE_DIR}/Build/Infrastructure/DataCollector/DataCollector.cmake
    ${CMAKE_SOURCE_DIR}/Build/Infrastructure/DiagnosticService/DiagnosticService.cmake
    ${CMAKE_SOURCE_DIR}/Build/Infrastructure/DisplayService/DisplayService.cmake
    ${CMAKE_SOURCE_DIR}/Build/Infrastructure/GraphicsStateService/GraphicsStateService.cmake
    ${CMAKE_SOURCE_DIR}/Build/Infrastructure/HmiMountService/HmiMountService.cmake
    ${CMAKE_SOURCE_DIR}/Build/Infrastructure/HealthInfoExtraction/HealthInfoExtraction.cmake
    ${CMAKE_SOURCE_DIR}/Build/Infrastructure/IOMan/IOMan.cmake
    ${CMAKE_SOURCE_DIR}/Build/Infrastructure/LinService/LinService.cmake
    ${CMAKE_SOURCE_DIR}/Build/Infrastructure/NA_CRE/ArbitrationService/ArbitrationService.cmake
    ${CMAKE_SOURCE_DIR}/Build/Infrastructure/NA_CRE/iix/iix.cmake
    ${CMAKE_SOURCE_DIR}/Build/Infrastructure/NA_CRE/HealthMonitor/HealthMonitor.cmake
    ${CMAKE_SOURCE_DIR}/Build/Infrastructure/NA_CRE/TimeMetricsDump/TimeMetricsDump.cmake
    ${CMAKE_SOURCE_DIR}/Build/Infrastructure/NetworkManager/NetworkManager.cmake
    ${CMAKE_SOURCE_DIR}/Build/Infrastructure/NvMService/NvMService.cmake
    ${CMAKE_SOURCE_DIR}/Build/Infrastructure/OdiService/OdiService.cmake
    ${CMAKE_SOURCE_DIR}/Build/Infrastructure/SafetyService/SafetyService.cmake
    ${CMAKE_SOURCE_DIR}/Build/Infrastructure/SecurityService/SecurityService.cmake
    ${CMAKE_SOURCE_DIR}/Build/Infrastructure/SystemInfoClient/SystemInfoClient.cmake
    ${CMAKE_SOURCE_DIR}/Build/Infrastructure/SystemPowerMode/SystemPowerMode.cmake
    ${CMAKE_SOURCE_DIR}/Build/UI/components/T1UiModelHud/T1UiModelHud.cmake
    ${CMAKE_SOURCE_DIR}/Build/UI/components/T1UiModelMainDisplay/T1UiModelMainDisplay.cmake
    ${CMAKE_SOURCE_DIR}/Build/UI/safety/T1UiModelEarlyHmi/T1UiModelEarlyHmi.cmake
    ${CMAKE_SOURCE_DIR}/Build/UI/UIController/UIController.cmake
)
```

Visteon®