# openPASS / World_OSI

Setup guide

September 11, 2020

## Contents

# 1 Document History

| Version | Description of the changes | Author | Date |
|---|---|---|---|
| 0.9 | First tracked version | Reinhard Biegel | 25.11.2019 |
| 1.0 | Extension of the Compile Core chapter | Arun Das | 12.12.2019 |
| 1.1 | Added list of third party libraries, Link to OSI Repo | Arun Das | 14.01.2020 |
| 1.2 | Added FMI Library to dependency list and dependency installation guide | Reinhard Biegel | 24.02.2020 |
| 1.3 | Updated OSI commit reference | Reinhard Biegel | 16.07.2020 |
| 1.4 | Updated Introduction chapter<br>Updated Prerequisites chapter<br>Updated dependency versions<br>Updated FMI Library patch for Linux compatibility<br>Corrected FMI Library CMake parameter<br>Updated OSI build instructions<br>Added CMake instructions<br>Added information about protobuf Arena allocation | Reinhard Biegel | 11.09.2020 |

# 2 Introduction

The World_OSI provides an implementation of the WorldInterface using Open Simulation Interface (OSI) as a "backend storage" [1]. OSI already provides data structures for representation of various objects in traffic simulation environments. As these objects are specified rather sensor centric, the World_OSI module holds additional data that is not stored inside of OSI objects. The currently supported OSI version is 3.2.0. As OSI and openPASS itself are constantly being developed, it is also possible to use experimental OSI versions. This is required for using the Algorithm_FmuWrapper component and OSMP FMUs, which are utilizing TrafficUpdate and TrafficCommand messages not yet defined in standard OSI. See Open Simulation Interface in the chapter describing the dependency installations.

To allow unit-testing of the implementation, an additional software layer (WorldData) has been introduced on top of OSI, whose classes represent the objects needed in the simulator's world. The OSI objects are instantiated there. An architectural diagram showing WorldData and its interactions is currently not available.

Another benefit of the additional WorldData layer is keeping the area of contact to the data backend small, to allow easy adaption if OSI switches from protobuf to another storage technology (open discussion).

The current use of protobuf provides easy serialization and deserialization of all stored data out-of-the-box.

# 3 Prerequisites

You'll find the openPASS sources in Eclipse Gerrit [10]. Please note to check out the **servant** branch, where the most recent development will currently take place. Of course, associated changes on Gerrit are also an option. For the OSI world demonstration an additional project file `OpenPASS_OSI_UseCase.pro` has been

added, which can be opened in QtCreator IDE, or directly be built using `qmake` on the command line. Recently, CMake project files have been added as well. They might be used as a replacement of the QMake project files, but support for old components is not available (Basic use case).

It is recommended to use a **short path for the source code checkout**, due to limitations in the MinGW/Windows setup. If you ever come across compilation errors regarding files not being found, you might have run into problems with path length restrictions. It is advisable to map the repository checkout directory to a drive letter to achieve the shortest path possible. You can use the `subst` command to accomplish that[2].

Before being able to compile and run the World_OSI, make sure to have all dependencies installed.

World_OSI has been developed on Linux x86_64 using Qt 5.12.3, gcc 7.1.0, 7.3.0, 8.1.0 and 9.3.0. On Windows systems a MinGW environment with the same compiler versions up to 8.1.0 has been used. There are no known issues regarding different gcc versions.

To summarize, openPASS with World_OSI has these dependencies:

- MinGW gcc environment (Windows, tested gcc versions are 7.1.0, 7.3.0 and 8.1.0, 64-bit)
- Qt 5.12.3 (incl. QtCreator IDE and qmake build system)
- Open Simulation Interface (OSI) 3.2.0
- Google Protobuf 3.11.4
- CMake 3.9.4
- boost library 1.72.0
    - algorithm
    - math
    - geometry
    - filesystem
    - system (header only since 1.69.0)
- Google Test / Google Mock 1.10.0
- Modelon FMILibrary 2.0.3 (if using Algorithm_FmuWrapper)

**NOTE:** Please stick to the paths as given by this tutorial and the source code – unless you're an expert. Only if the default paths are used, compatibility with the GUI can be guaranteed.

# 4 Dependency installation guide

The World_OSI module introduces additional third-party dependency on the OSI library, and as a consequence on Google protobuffers. Furthermore, the boost (geometry) library is required to compile and run World_OSI [6]. Other dependencies have to be set up for the build environment itself.

Note, that the expected folder structure of third-party dependencies is different for qmake and CMake builds of openPASS. If preparing dependencies for qmake, all files have to be installed in a single directory (which in turn has a include and lib subfolder). When building openPASS using CMake, this requirement doesn't hold true anymore. See Using CMake.

## 4.1 MinGW

An installation for 64-bit MinGW environments is provided at [9]. You can download a generic installer binary, which will let you select different Versions of MinGW. Latest tested setup is `x86_64-8.1.0-posix-seh-rt_v6-rev0`.

## 4.2 CMake

For compilation of protobuf and OSI, the CMake build system is required. Please download and install a binary distribution of CMake [5]. Version 3.9.4 has been used during testing. Please feel free to try a more recent version, but note that issues with newer versions have already been reported.

## 4.3 Protobuf

If not already installed on the system, the protobuf library and headers have to be built prior to being able to compile OSI. Protobuf sources are located at [4]. This guide will give some instructions to compile version 3.11.4, but any other version should be compatible (including 2.x).

As most users seem to work on Windows environments, the following instructions will target these systems.

1. Download `protobuf-cpp-3.11.4.zip` and extract the archive. We will use `C:\OpenPASS\thirdParty\sources` as target directory.

2. Open a command line with MinGW environment at the extraction directory

3. Create the build directory

```
> cd cmake
> mkdir build
> cd build
```

4. Run CMake (tests are disabled here due to some compiler warnings being treated as errors, may vary with compiler version)

```
> cmake -G "MinGW Makefiles" -DCMAKE_BUILD_TYPE=Release
  -DCMAKE_INSTALL_PREFIX=C:/OpenPASS/thirdParty
  -Dprotobuf_BUILD_SHARED_LIBS=ON
  -Dprotobuf_BUILD_TESTS=OFF ..
```

5. Compilation

```
> mingw32-make -j3
```

6. Installation

```
> mingw32-make install
```

7. Final steps
    a. Copy the MinGW runtime libraries from the MinGW installation directory to the `thirdParty/bin` directory `(libgcc_s_seh-1.dll, libwinpthread-1.dll and libstdc++6.dll)`. This allows running the protobuf compiler `(protoc.exe)` outside of a full MinGW environment.
    b. Copy the `libprotobuf.dll` from `bin` to `lib` subfolder

## 4.4 Open Simulation Interface

The OSI project is hosted on GitHub. Please check out the OSI sources tagged as **v3.2.0** from [1].

openPASS supports protobuf Arena allocation [3]. This feature is enabled for openPASS by specifying a parameter during build (see Compiling openPASS). In addition, OSI has to be Arena-enabled as well. This is achieved by adding "`option cc_enable_arenas = true;`" to all OSI proto files before compilation.

We will use C:\OpenPASS\thirdParty\sources as target directory. Compile OSI using the following commands executed from the source directory.

1. Create build directory

   ```
   > mkdir build
   > cd build
   ```

2. Run CMake. As I could not make CMake find protobuf in the custom installation directory, we specify the paths manually here.

   ```
   > cmake -G "MinGW Makefiles" -DCMAKE_BUILD_TYPE=Release
     -DCMAKE_INSTALL_PREFIX=C:/OpenPASS/thirdParty
     -DProtobuf_INCLUDE_DIR=C:/OpenPASS/thirdParty/include
     -DProtobuf_PROTOC_EXECUTABLE=C:/OpenPASS/thirdParty/bin/protoc.exe
     -DProtobuf_LIBRARIES=C:/OpenPASS/thirdParty/lib ..
   ```

3. Adding linker flags for protobuf
   Edit the file `C:\OpenPASS\thirdParty\sources\open-simulation-interface\build\CMakeFiles\open_simulation_interface.dir\linklibs.rsp`

   Add "`-LC:/OpenPASS/thirdParty/lib -lprotobuf`" to the end of the line.
   ***If anybody knows how to avoid this step, please drop a note.***

4. Compilation

   ```
   > mingw32-make -j3
   ```

5. Installation

   ```
   > mingw32-make install
   ```

6. Copy `libopen_simulation_interface.dll` from `lib/osi3` subfolder to `lib`

The OSI class documentation is part of the source code and can be compiled using Doxygen. Instructions are located in the OSI `Readme.md`. A Pre-compiled version is located here: https://opensimulationinterface.github.io/open-simulation-interface/index.html. **Note that this version of the documentation doesn't include the extensions from the openpass-trafficAgents branch.**

## 4.5 boost

Boost consists of multiple libraries, many of them being header-only [6]. At the moment, World_OSI only makes use of the `algorithm`, `math` and `geometry` header-only-libraries. FMU Wrapper and tests have dependencies on the boost `filesystem` library as well.

1. Download and extract boost from [7]. Boost version 1.72.0 was used during development of World_OSI. Newer versions should be compatible, but have not been tested yet.

2. Extract downloaded archive to a temporary directory

3. Copy the `boost` subfolder to your thirdParty/include directory

## 4.6 Google Test and Google Mock

For unit-testing, World_OSI and other modules make use of the gtest/gmock framework. Installation requires `cmake`, just like `protobuf`, which was mentioned above.

1. Download a release from [8].
   Version 1.10.0 has been used during development.

2. Extract the archive to a temporary directory

3. Open a MinGW shell at the extraction directory

4. Create the build directory

   ```
   > mkdir build
   > cd build
   ```

5. Run CMake

   ```
   > cmake -G "MinGW Makefiles"
     -DCMAKE_INSTALL_PREFIX=C:/OpenPASS/thirdParty ..
   ```

6. Compilation

   ```
   > mingw32-make
   ```

7. Installation

   ```
   > mingw32-make install
   ```

## 4.7 Modelon FMI Library

If the component Algorithm_FmuWrapper is to be used, the FMI Library is an additional dependency for compiling and running openPASS. FMI Library uses the CMake build system and requires minor patching:

1. Download release 2.0.3 from [11]

2. Extract archive to a temporary directory

3. Open a MinGW shell at the extraction directory

4. Apply the patch below
   (Algorithm_FmuWrapper uses code from FMU Compliance Checker [12], which has dependencies on some additional symbols in FMI Library. These symbols will be made visible by adding FMILIB_EXPORT to their declarations. On Linux platforms, FMU library loading needs modification. This change doesn't affect Windows platforms.)

```
diff --git a/src/Import/src/FMI1/fmi1_import_capi.c b/src/Import/src/FMI1/fmi1_import_capi.c
index 842c998..807cdae 100644
--- a/src/Import/src/FMI1/fmi1_import_capi.c
+++ b/src/Import/src/FMI1/fmi1_import_capi.c
@@ -67,18 +67,8 @@ jm_status_enu_t fmi1_import_create_dllfmu(fmi1_import_t* fmu, fmi1_callback_func
```

```
                return jm_status_error;
        }

-       if(jm_portability_set_current_working_directory(dllDirPath) != jm_status_success) {
-               jm_log_fatal(fmu->callbacks, module, "Could not change to the DLL directory %s", dllDirPath);
-               if(ENOENT == errno)
-                       jm_log_fatal(fmu->callbacks, module, "The FMU contains no binary for this platform.");
-               else
-                       jm_log_fatal(fmu->callbacks, module, "System error: %s", strerror(errno));
-       }
-       else {
-               /* Allocate memory for the C-API struct */
-               fmu -> capi = fmi1_capi_create_dllfmu(fmu->callbacks, dllFileName, modelIdentifier,
callBackFunctions, standard);
-       }
-
+       /* Allocate memory for the C-API struct */
+       fmu -> capi = fmi1_capi_create_dllfmu(fmu->callbacks, dllFileName, modelIdentifier, callBackFunctions,
standard);

        /* Load the DLL handle */
        if (fmu -> capi) {
diff --git a/src/Import/src/FMI2/fmi2_import_capi.c b/src/Import/src/FMI2/fmi2_import_capi.c
index e794775..a2a68ba 100644
--- a/src/Import/src/FMI2/fmi2_import_capi.c
+++ b/src/Import/src/FMI2/fmi2_import_capi.c
@@ -88,18 +88,8 @@ jm_status_enu_t fmi2_import_create_dllfmu(fmi2_import_t* fmu, fmi2_fmu_kind_enu_
                callBackFunctions = &defaultCallbacks;
        }

-       if(jm_portability_set_current_working_directory(dllDirPath) != jm_status_success) {
-               jm_log_fatal(fmu->callbacks, module, "Could not change to the DLL directory %s", dllDirPath);
-               if(ENOENT == errno)
-                       jm_log_fatal(fmu->callbacks, module, "The FMU contains no binary for this platform.");
-               else
-                       jm_log_fatal(fmu->callbacks, module, "System error: %s", strerror(errno));
-       }
-       else {
-               /* Allocate memory for the C-API struct */
-               fmu -> capi = fmi2_capi_create_dllfmu(fmu->callbacks, dllFileName, modelIdentifier,
callBackFunctions, fmuKind);
-       }
-
+       /* Allocate memory for the C-API struct */
+       fmu -> capi = fmi2_capi_create_dllfmu(fmu->callbacks, dllFileName, modelIdentifier, callBackFunctions,
fmuKind);

        /* Load the DLL handle */
        if (fmu -> capi) {
diff --git a/src/Util/include/JM/jm_portability.h b/src/Util/include/JM/jm_portability.h
index 82e472a..3e50603 100644
--- a/src/Util/include/JM/jm_portability.h
+++ b/src/Util/include/JM/jm_portability.h
@@ -65,6 +65,7 @@ jm_status_enu_t jm_portability_get_current_working_directory(char* buffer, size_
 jm_status_enu_t jm_portability_set_current_working_directory(const char* cwd);

 /** \brief Get system-wide temporary directory */
+FMILIB_EXPORT
 const char* jm_get_system_temp_dir();

 /**
@@ -118,6 +119,7 @@ jm_status_enu_t jm_mkdir(jm_callbacks* cb, const char* dir);
 /**
 \brief Remove directory and all it contents.
 */
+FMILIB_EXPORT
 jm_status_enu_t jm_rmdir(jm_callbacks* cb, const char* dir);

 /**
```

5. Create the build directory

```
> mkdir build
> cd build
```

6. Run CMake

```
> cmake -G “MinGW Makefiles”
  -DFMILIB_INSTALL_PREFIX=C:/OpenPASS/thirdParty
  -DCMAKE_BUILD_TYPE=Release
  -DFMILIB_BUILD_STATIC_LIB=OFF
  -DFMILIB_BUILD_SHARED_LIB=ON ..
```

7. Compilation

   ```
   > mingw32-make
   ```

8. Installation

   ```
   > mingw32-make install
   ```

9. Move the FMI Library headers (located in C:\OpenPASS\thirdParty\include) into a new subfolder C:\OpenPASS\thirdParty\include\FMILibrary. The files/folders to be moved are: `FMI`, `FMI1`, `FMI2`, `JM`, `fmilib.h` and `fmilib_config.h`.

# 5 Compiling openPASS

## 5.1 Using qmake

The variables `EXTRA_LIB_PATH` and `EXTRA_INCLUDE_PATH` are set to `C:/OpenPASS/thirdParty/lib` and `C:/OpenPASS/thirdParty/include` by default. Please adapt the paths in the `OpenPass_Source_Code/defaults.pri` in case you are using different locations. It is also possible to pass these variables on the `qmake` command line, which will then override the default values.

When using the QMake build, protobuf Arena support is always enabled.

**NOTE:** On windows, path length limitations apply during building which often causes the build to fail. To overcome this problem, it is recommended to copy the folder `OpenPass_Source_Code` directly to your drive `C:` and rename it in order to have a short name (e.g. `OP`). Same applies to the build directory, which can be changed under `Projects` → `Build Settings` → `Build directory` after opening the project file. Alternatively, the `subst` command can be used as described in Prerequisites to shorten the path on the filesystem.

Build the `OpenPASS_OSI_UseCase.pro` project to build the simulation core with all components and the graphical user interface.

## 5.2 Using CMake

For compiling openPASS using CMake, the following command line instructions shall be provided as a template. Start from MinGW shell at the repository checkout directory.

1. Create build directory

   ```
   > mkdir build
   > cd build
   ```

2. Run CMake (the CMAKE_PREFIX_PATH variable has to list all dependency installation directories, separated by semicolon).

   ```
   > cmake -G "MinGW Makefiles" -D WITH_GUI=OFF -D
   CMAKE_PREFIX_PATH=C:\Qt\Qt5.12.2\5.12.2\mingw73_64;
   D:\os\deps\thirdParty\FMILibrary;
   D:\os\deps\thirdParty\boost;
   ```

```
D:\os\deps\thirdParty\osi;
D:\os\deps\thirdParty\protobuf;
D:\os\deps\thirdParty\googletest
-D CMAKE_INSTALL_PREFIX=C:\OpenPASS\bin\Slave
-D CMAKE_BUILD_TYPE=Release
-D USE_CCACHE=ON
-D OPENPASS_ADJUST_OUTPUT=OFF
-D WITH_DEBUG_POSTFIX=OFF
..
```

   a. To disable protobuf Arena allocation, add "`-D WITH_PROTOBUF_ARENA=OFF`" to the cmake call.
   b. `OPENPASS_ADJUST_OUTPUT` and `WITH_DEBUG_POSTFIX` provide mechanisms to control build file placement and debug binary naming. The settings shown should be changed by developers only.

3. Compilation

```
> mingw32-make -j3
```

4. Installation

```
> mingw32-make install
```

5. **IMPORTANT** Recent changes to the CMake build system will put the `OpenPassSlave` executable inside a `'bin'` subfolder in the `CMAKE_INSTALL_PREFIX`. The core itself is not yet adjusted to this layout, so we need to move the files.

```
> cd C:\OpenPASS\bin\Slave
> move bin\*.* .
```

# 6 Simulation

There are multiple demo scenarios located at `sim/contrib/examples`. You can copy the configs folder of a scenario to the openPASS installation directory and run `OpenPassSalve.exe` without additional command line arguments. Results will be placed in `'results'`.

Alternatively, you can provide the location of the config files to the simulator on the command line using the `--configs` switch.

**NOTE:** Remember to copy the required third-party libraries (see table below) to the directory, where the `OpenPassSlave.exe` is located. Dependency locations during runtime are (mostly) independent from compile-time locations. Alternatively, you can add a directory containing these libraries to the PATH environment variable. *When using the cmake build, the dependencies used during linking will be copied to the installation folder automatically.*

| Required third party libraries | Location to get the libraries from |
|---|---|
| `libprotobuf.dll`<br>`libopen_simulation_interface.dll`<br>`libfmilib_shared.dll (optional)` | `C:\OpenPASS\thirdParty\lib` |

| | |
|---|---|
| `Qt5Xml.dll`<br>`Qt5Core.dl` | Qt install directory (e.g. `C:\Qt\Qt5.12.2\Tools\QtCreator\bin` ) |
| `libwinpthread-1.dll`<br>`libstdc++-6.dll`<br>`libgcc_s_seh-1.dll` | MinGW install directory (e.g. `C:\Qt\mingw-w64\x86_64-8.1.0-posix-seh-rt_v6-rev0\mingw64\bin` ) |

# 7 References


[1] https://github.com/OpenSimulationInterface/open-simulation-interface

[2] https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/subst

[3] https://developers.google.com/protocol-buffers/docs/reference/arenas

[4] https://github.com/protocolbuffers/protobuf/releases

[5] https://cmake.org/files/v3.9/

[6] https://www.boost.org/

[7] https://sourceforge.net/projects/boost/files/boost/

[8] https://github.com/google/googletest

[9] https://sourceforge.net/projects/mingw-w64/

[10] https://git.eclipse.org/r/#/admin/projects/simopenpass/simopenpass

[11] https://github.com/modelon-community/fmi-library

[12] https://github.com/modelica-tools/FMUComplianceChecker