CommonAPI C++ D-Bus in 10 minutes (from scratch)

CommonAPI C++ and bindings have been moved to github, see overview at https://github.com/orgs/GENIVI/teams/someip/repositories and see also the CommonAPI C++ homepage at http://genivi.github.io/capicxx-core-tools/. Please make sure that you use code generators and runtime libraries with the same version by getting the code generator binaries from capicxx-core-tools/releases repository, capicxx-dbus-tools/releases repository and capicxx-someip-tools/releases repository. Otherwise you could get same strange compilation errors. Please note that a newer version of this wiki page can be accessed via capicxx-core-tools/wiki repository.

Valid for CommonAPI 3.1.3 and vsomeip 1.3.0

Step 1: Preparation / Prerequisites

The following description is based on CommonAPI 3.1.3 and assumes that you use a standard Linux distribution (I tested it with Xubuntu 14.04) and that you have installed git and (CMake >=2.8).

Step 2: Build the CommonAPI Runtime Library

Start with fetching the source code of CommonAPI, e.g. with:

Clone CommonAPI Repositories

```
x@ubuntu:~/work$ git clone https://github.com/GENIVI/capicxx-core-runtime.git
Cloning into ...
remote: Counting objects: 984, done.
remote: Compressing objects: 100% (798/798), done.
remote: Total 984 (delta 550), reused 322 (delta 151)
Receiving objects: 100% (984/984), 520.00 KiB | 299.00 KiB/s, done.
Resolving deltas: 100% (550/550), done.
Checking connectivity... done.
x@ubuntu:~/work$ ls
capicxx-core-runtime
```

The CommonAPI runtime library can now be built without any other dependencies:

Build CommonAPI

```
x@ubuntu:~/work$ cd capicxx-core-runtime/
x@ubuntu:~/work/capicxx-core-runtime$ ls
AUTHORS  cmake  CMakeLists.txt  commonapi.spec.in  docx  doxygen.in  include  INSTALL  LICENSE  README  src
x@ubuntu:~/work/capicxx-core-runtime$ mkdir build
x@ubuntu:~/work/capicxx-core-runtime$ cd build/
x@ubuntu:~/work/capicxx-core-runtime$ cmake ..
-- The C compiler identification is GNU 4.8.2
-- The CXX compiler identification is GNU 4.8.2
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Project name: libcommonapi
-- This is CMake for Common API C++ Version 3.1.3.
-- CMAKE_INSTALL_PREFIX set to: /usr/local
-- BUILD_SHARED_LIBS is set to value: ON
-- MAX_LOG_LEVEL is set to value: DEBUG
-- RPM packet version set to r0
-- Build type: Debug
-- Found PkgConfig: /usr/bin/pkg-config (found version "0.26")
-- checking for module 'automotive-dlt >= 2.11'
--   package 'automotive-dlt >= 2.11' not found
-- Found Doxygen: /usr/bin/doxygen (found version "1.8.6")
 -- asciidoc found
-- Configuring done
-- Generating done
-- Build files have been written to: /home/x/work/capicxx-core-runtime/build
x@ubuntu:~/work/capicxx-core-runtime/build$ make
Scanning dependencies of target CommonAPI
[ 11%] Building CXX object CMakeFiles/CommonAPI.dir/src/CommonAPI/ProxyManager.cpp.o
[ 22%] Building CXX object CMakeFiles/CommonAPI.dir/src/CommonAPI/Logger.cpp.o
[ 33%] Building CXX object CMakeFiles/CommonAPI.dir/src/CommonAPI/MainLoopContext.cpp.o
[ 44%] Building CXX object CMakeFiles/CommonAPI.dir/src/CommonAPI/Runtime.cpp.o
[ 55%] Building CXX object CMakeFiles/CommonAPI.dir/src/CommonAPI/Utils.cpp.o
[ 66%] Building CXX object CMakeFiles/CommonAPI.dir/src/CommonAPI/Address.cpp.o
[ 77%] Building CXX object CMakeFiles/CommonAPI.dir/src/CommonAPI/Proxy.cpp.o
[ 88%] Building CXX object CMakeFiles/CommonAPI.dir/src/CommonAPI/ContainerUtils.cpp.o
[100%] Building CXX object CMakeFiles/CommonAPI.dir/src/CommonAPI/IniFileReader.cpp.o
Linking C shared library libCommonAPI.so
[100%] Built target CommonAPI
x@ubuntu:~/work/capicxx-core-runtime/build$ ls
CMakeCache.txt  cmake_install.cmake    CommonAPIConfigVersion.cmake  CommonAPITargets.cmake  libCommonAPI.
so    libCommonAPI.so.3.1.3
CMakeFiles      CommonAPIConfig.cmake  commonapi.spec                Doxyfile                libCommonAPI.so.
3  Makefile
```

The output in your console should be similar to the extract of my output that it is the window above. Do not doubt when you see in your results that Doxygen or asciidoc is not found; it is only needed for the generation of the documentation. The package 'automotive-dlt' is only needed in case you want to get DLT log messages (see http://projects.genivi.org/diagnostic-log-trace/home). As shown in the last two lines you should find `libCommonAPI.so` in your build directory.

The CommonAPI runtime library can be installed to `/usr/local` (or any other directory if you set the installation prefix) by calling `make install`. But I assume that in most of the cases this is not wanted; therefore this description explains the procedure completely without any installations.

Step 3: Build the CommonAPI D-Bus Runtime Library

Start again with cloning the source code of CommonAPI-D-Bus:

Clone CommonAPI D-Bus

```
x@ubuntu:~/work$ git clone https://github.com/GENIVI/capicxx-dbus-runtime
Cloning into ...
remote: Counting objects: 4085, done.
remote: Compressing objects: 100% (1905/1905), done.
remote: Total 4085 (delta 2821), reused 3100 (delta 1914)
Receiving objects: 100% (4085/4085), 1.57 MiB | 741.00 KiB/s, done.
Resolving deltas: 100% (2821/2821), done.
Checking connectivity... done.
x@ubuntu:~/work$ ls
capicxx-dbus-runtime  capicxx-core-runtime
```

The D-Bus runtime library cannot be built without further preparations. The reason is that CommonAPI-D-Bus doesn't use the standard `libdbus` library and needs a patched version. That means that you must first download, patch and build `libdbus` before the CommonAPI D-Bus runtime can be built. Let's start (3 of 10 minutes have already passed 😅). Get and unpack the actual libdbus library:

Get libdbus

```
x@ubuntu:~/work$ wget http://dbus.freedesktop.org/releases/dbus/dbus-1.8.20.tar.gz
x@ubuntu:~/work$ tar -xzf dbus-1.8.20.tar.gz
x@ubuntu:~/work$ ls
capicxx-dbus-runtime  capicxx-core-runtime  dbus-1.8.20  dbus-1.8.20.tar.gz
x@ubuntu:~/work$ cd dbus-1.8.20/
```

I discarded the output in the console. Now apply the patches in the CommonAPI D-Bus source directory:

Patch libdbus

```
x@ubuntu:~/work/dbus-1.8.20$ patch -p1 < ../capicxx-dbus-runtime/src/dbus-patches/capi-dbus-add-send-with-reply-
set-notify.patch
patching file dbus/dbus-connection.c
Hunk #1 succeeded at 3500 (offset 18 lines).
patching file dbus/dbus-connection.h
x@ubuntu:~/work/dbus-1.8.20$ patch -p1 < ../capicxx-dbus-runtime/src/dbus-patches/capi-dbus-add-support-for-
custom-marshalling.patch
patching file dbus/dbus-message.c
Hunk #1 succeeded at 3522 (offset 77 lines).
Hunk #2 succeeded at 4747 (offset 74 lines).
patching file dbus/dbus-message.h
patching file dbus/dbus-string.c
Hunk #1 succeeded at 727 (offset -3 lines).
patching file dbus/dbus-string.h
x@ubuntu:~/work/dbus-1.8.20$ patch -p1 < ../capicxx-dbus-runtime/src/dbus-patches/capi-dbus-correct-dbus-
connection-block-pending-call.patch
patching file dbus/dbus-connection.c
```

At the moment there are three libdbus patches provided. Two of these patches provide additional functions which will be called from CommonAPI.

⚠ Please check if all these patches could be applied successfully. I tested it with D-Bus 1.6.x and 1.8.x. For other versions I cannot guarantee that the patches work. If necessary updated patch versions will be provided for D-Bus > 1.8.x.

Now build `libdbus` with autotools and check if the `so` library has been created in the `.libs`-directory.

Build libdbus

```
x@ubuntu:~/work/dbus-1.8.20$ ./configure
x@ubuntu:~/work/dbus-1.8.20$ make
x@ubuntu:~/work/dbus-1.8.20$ ls
dbus/.libs/libdbus-1.so.* dbus/.libs/libdbus-1.so.3  dbus/.libs/libdbus-1.so.3.8.13
```

⚠ Build libdbus with autotools. Otherwise you will miss some necessary files when you try to buld CommonAPI D-Bus.

Now it is possible to build the CommonAPI runtime library. Please note that you only use the uninstalled versions of CommonAPI and `libdbus`. First create the build directory and change to the build directory:

Create Build Directory for CommonAPI D-Bus

```
x@ubuntu:~/work/dbus-1.8.20$ cd ../capicxx-dbus-runtime/
x@ubuntu:~/work/capicxx-dbus-runtime$ mkdir build
x@ubuntu:~/work/capicxx-dbus-runtime$ cd build
x@ubuntu:~/work/capicxx-dbus-runtime/build$
```

For the next step add the path to your D-Bus directory to the actual PKG_CONFIG_PATH environment variable, start CMake and call make as usual:

Build CommonAPI D-Bus

```
x@ubuntu:~/work/capicxx-dbus-runtime/build$ export PKG_CONFIG_PATH="/home/x/work/dbus-1.8.20"
x@ubuntu:~/work/capicxx-dbus-runtime/build$ cmake -DUSE_INSTALLED_COMMONAPI=OFF -DUSE_INSTALLED_DBUS=OFF ..
x@ubuntu:~/work/capicxx-dbus-runtime/build$ make
```

If everything was successful (it should be!) you should find `libCommonAPI-DBus.so` in your build directory. Make sure that there are no empty spaces in the path of your working directory.

Step 4: Write the Franca file and generate code

After all these preparations we start seriously to write a true CommonAPI application (more precisely two applications). We want to write one service which offers the method `sayHello` and one client which calls this method. For CommonAPI services the description of offered interfaces is done by means of Franca IDL. Don't worry if you don't know Franca IDL, you will learn it very fast.

First create some directories and open a new empty ASCII file with the name `HelloWorld.fidl`:

Create Project Directories

```
x@ubuntu:~/work$
x@ubuntu:~/work$ mkdir project
x@ubuntu:~/work$ cd project/
x@ubuntu:~/work/project$ mkdir fidl
x@ubuntu:~/work/project$ cd fidl
x@ubuntu:~/work/project/fidl$ vi HelloWorld.fidl
```

Now create the following interface:

HelloWorld.fidl

```
package commonapi

interface HelloWorld {
  version {major 1 minor 0}
  method sayHello {
    in {
      String name
    }
    out {
      String message
    }
  }
}
```

Ready! A service which instantiates the interface `HelloWorld` provides the function `sayHello` which can be called. The next step is to generate code. For that we need the code generators. We copy them into the new subdirectory `cgen` of our project directory:

Get Code Generators

```
x@ubuntu:~/work/project$ mkdir cgen
x@ubuntu:~/work/project$ cd cgen/
x@ubuntu:~/work/project/cgen$ wget http://docs.projects.genivi.org/yamaica-update-site/CommonAPI/generator/3.1
/3.1.3/commonapi-generator.zip
x@ubuntu:~/work/project/cgen$ wget http://docs.projects.genivi.org/yamaica-update-site/CommonAPI/generator/3.1
/3.1.3/commonapi_dbus_generator.zip
x@ubuntu:~/work/project/cgen$ unzip commonapi-generator.zip -d commonapi-generator
x@ubuntu:~/work/project/cgen$ unzip commonapi_dbus_generator.zip -d commonapi_dbus_generator
x@ubuntu:~/work/project/cgen$ chmod +x ./commonapi-generator/commonapi-generator-linux-x86
x@ubuntu:~/work/project/cgen$ chmod +x ./commonapi_dbus_generator/commonapi-dbus-generator-linux-x86
```

Now you find the executables of the code generators in `cgen/commonapi-generator` and `cgen/commonapi_dbus_generator`, respectively. There are four versions (Linux, Windows and 64bit variants). Type `uname -m` if you don't know if you have a 32bit or 64bit version of Linux (you get `i686` or `x86_64`). For the further description we assume that you have the 32bit version.

Do not complain about details, such as that you have to call `chmod` or that the names contain sometimes underscores and sometimes hyphen. It will change in future versions.

Finally you can generate code (CommonAPI code with the `commonapi-generator` and CommonAPI D-Bus code with the `commonapi-dbus-generator`):

Generate Code

```
x@ubuntu:~/work/project$ ./cgen/commonapi-generator/commonapi-generator-linux-x86 -sk ./fidl/HelloWorld.fidl
x@ubuntu:~/work/project$ ./cgen/commonapi_dbus_generator/commonapi-dbus-generator-linux-x86 ./fidl/HelloWorld.
fidl
x@ubuntu:~/work/project$ ls src-gen/v1_0/commonapi
HelloWorldDBusDeployment.cpp  HelloWorldDBusProxy.cpp  HelloWorldDBusStubAdapter.cpp  HelloWorld.hpp
HelloWorldProxy.hpp        HelloWorldStubDefault.hpp HelloWorldDBusDeployment.hpp  HelloWorldDBusProxy.hpp
HelloWorldDBusStubAdapter.hpp  HelloWorldProxyBase.hpp  HelloWorldStubDefault.cpp  HelloWorldStub.hpp
```

If everything worked, the generated code will be in the new directory `src-gen`. The option `-sk` generates a default implementation of your interface instance in the service.

Step 5: Write the client and the service application

Now we can start to write the Hello World application. Create new subdirectories `src` and `build` in the project directory and change to `src`.

Create src and build directories

```
x@ubuntu:~/work/project$ mkdir src
x@ubuntu:~/work/project$ mkdir build
x@ubuntu:~/work/project$ ls
build  cgen  fidl  src  src-gen
x@ubuntu:~/work/project$ cd src
```

Now we have to create 4 files: The client code (`HelloWorldClient.cpp`), one file for the main-function of the service (`HelloWorldService.cpp`) and 2 files (header and source) for the implementation of the generated skeleton for the stub (we call it `HelloWorldStubImpl.hpp` and `HelloWorldStubImpl.cpp`).

Let's begin with the client. Create a new file `HelloWorldClient.cpp` with the editor of your choice and type:

HelloWorldClient.cpp

```
#include <iostream>
#include <string>
#include <unistd.h>
#include <CommonAPI/CommonAPI.hpp>
#include <v1_0/commonapi/HelloWorldProxy.hpp>

using namespace v1_0::commonapi;

int main() {
    std::shared_ptr < CommonAPI::Runtime > runtime = CommonAPI::Runtime::get();
    std::shared_ptr<HelloWorldProxy<>> myProxy = runtime->buildProxy<HelloWorldProxy>("local", "test");

    std::cout << "Checking availability!" << std::endl;
    while (!myProxy->isAvailable())
            usleep(10);
    std::cout << "Available..." << std::endl;

    CommonAPI::CallStatus callStatus;
    std::string returnMessage;
    myProxy->sayHello("Bob", callStatus, returnMessage);
    std::cout << "Got message: '" << returnMessage << "'\n";

    return 0;
}
```

At the beginning of each CommonAPI application it is necessary to get a pointer to the generic runtime object. The runtime is necessary to create proxies and stubs. A client application has to call functions of an instance of an interface in the service application. In order to be able to call these functions we must build a proxy for this interface in the client. The interface name is the template parameter of the `buildProxy` method; furthermore we build the proxy for a certain instance; the instance name is the second parameter in the `buildProxy` method. In principle there is the possibility to distinguish between instances in different so-call domains (first parameter), but we don't want to discuss this in depth at the moment and take always the domain "local".

The proxy provides the API function `isAvailable`; if the we start the service first then `isAvailable` returns always `true`. It is also possible to register a callback which is called when the service becomes available; but we try to keep it here as simple as possible.

Now we call the function `sayHello` which we have defined in our fidl-file. The function has one in-parameter (string) and one out-parameter (also string). Have a look into `HelloWorldPorxy.hpp` to get the information how exactly the function `sayHello` must be called. Here it is important to know that it is possible to call the synchronous variant of this function (what we do here) or to call the asynchronous variant (`sayHelloAsync`) which is slightly more complicated. One return value is the so-called `CallStatus`, which gives us the information if the call was successful or not. Again, to keep it simple we do not check the CallStatus and hope that everthing worked fine.

We continue now to write the service. Create a new file `HelloWorldService.cpp`:

HelloWorldService.cpp

```
#include <iostream>
#include <thread>
#include <CommonAPI/CommonAPI.hpp>
#include "HelloWorldStubImpl.hpp"

using namespace std;

int main() {
    std::shared_ptr<CommonAPI::Runtime> runtime = CommonAPI::Runtime::get();
    std::shared_ptr<HelloWorldStubImpl> myService = std::make_shared<HelloWorldStubImpl>();
    runtime->registerService("local", "test", myService);
    std::cout << "Successfully Registered Service!" << std::endl;

    while (true) {
        std::cout << "Waiting for calls... (Abort with CTRL+C)" << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(30));
    }

    return 0;
 }
```

The main function of the service is even simpler as the main function of the client because the implementation of the interface functions is in the stub implementation. Again we need the pointer to the runtime environment; then we instantiate our implementation of the stub and register this instance of the interface by calling `registerService` with an instance name. The service shall run forever and answer to function calls until it becomes killed; therefore we need the while loop at the end of the main function.

At the end we need the stub implementation; we realize it by creating a stub-implementation class which is inherited from the stub-default implementation. The header file is:

HelloWorldStubImpl.hpp

```
#ifndef HELLOWORLDSTUBIMPL_H_
#define HELLOWORLDSTUBIMPL_H_

#include <CommonAPI/CommonAPI.hpp>
#include <v1_0/commonapi/HelloWorldStubDefault.hpp>

class HelloWorldStubImpl: public v1_0::commonapi::HelloWorldStubDefault {
public:
    HelloWorldStubImpl();
    virtual ~HelloWorldStubImpl();
    virtual void sayHello(const std::shared_ptr<CommonAPI::ClientId> _client, std::string _name,
sayHelloReply_t _return);
};

#endif /* HELLOWORLDSTUBIMPL_H_ */
```

The implementation is in the cpp-file:

HelloWorldStubImpl.cpp

```
#include "HelloWorldStubImpl.hpp"

HelloWorldStubImpl::HelloWorldStubImpl() { }
HelloWorldStubImpl::~HelloWorldStubImpl() { }

void HelloWorldStubImpl::sayHello(const std::shared_ptr<CommonAPI::ClientId> _client, std::string _name,
sayHelloReply_t _reply) {
    std::stringstream messageStream;
    messageStream << "Hello " << _name << "!";
    std::cout << "sayHello('" << _name << "'): '" << messageStream.str() << "'\n";

    _reply(messageStream.str());
};
```

If the function `sayHello` is called it gets the name (which is supposed to be the name of the developer of this application) and returns it with an added "Hello" in front. The return parameter is not directly the string as it is defined in the fidl-file; it is a standard function object with the return parameters as in-parameters. The reason for this is to provide the possibility to answer not synchronously in the implementation of this function but to delegate the answer to a different thread.

Step 6: Build and run

Since you must have installed CMake in order to build the CommonAPI runtime libraries, the fastest way to compile and build our applications is to write a simple CMake file. Create a new file `CMakeLists.txt` directly in the project directory:

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -pthread -std=c++0x")

include_directories(
    src-gen
    ../capicxx-core-runtime/include
    ../capicxx-dbus-runtime/include
    ../dbus-1.8.20
)

link_directories(
    /home/x/work/capicxx-core-runtime/build
    /home/x/work/capicxx-dbus-runtime/build
    /home/x/work/dbus-1.8.20/dbus/.libs
)

add_executable(HelloWorldClient
    src/HelloWorldClient.cpp
    src-gen/v1_0/commonapi/HelloWorldDBusProxy.cpp
    src-gen/v1_0/commonapi/HelloWorldDBusDeployment.cpp
)
target_link_libraries(HelloWorldClient CommonAPI CommonAPI-DBus dbus-1)

add_executable(HelloWorldService
    src/HelloWorldService.cpp
    src/HelloWorldStubImpl.cpp
    src-gen/v1_0/commonapi/HelloWorldDBusStubAdapter.cpp
    src-gen/v1_0/commonapi/HelloWorldStubDefault.cpp
    src-gen/v1_0/commonapi/HelloWorldDBusDeployment.cpp
)
target_link_libraries(HelloWorldService CommonAPI CommonAPI-DBus dbus-1)
```

As include paths we need the include directories of CommonAPI and bindings and D-Bus; the directory of the generated code must be also added. We link everything together and do not discuss here questions concerning the configuration with different bindings. Therefore we tell CMake where to find the CommonAPI libraries and libdbus (replace the absolute paths with the paths on your machine). At the end we build two executables, one for the service and one for the client.

Now call CMake (remember that we created the build directory before:

Build everything

```
x@ubuntu:~/work/project$ cd build
x@ubuntu:~/work/project/build$ cmake ..
-- The C compiler identification is GNU 4.8.2
-- The CXX compiler identification is GNU 4.8.2
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/x/work/project/build
x@ubuntu:~/work/project/build$ make
Scanning dependencies of target HelloWorldClient
[ 12%] Building CXX object CMakeFiles/HelloWorldClient.dir/src/HelloWorldClient.cpp.o
[ 25%] Building CXX object CMakeFiles/HelloWorldClient.dir/src-gen/v1_0/commonapi/HelloWorldDBusProxy.cpp.o
[ 37%] Building CXX object CMakeFiles/HelloWorldClient.dir/src-gen/v1_0/commonapi/HelloWorldDBusDeployment.cpp.o
Linking CXX executable HelloWorldClient
[ 37%] Built target HelloWorldClient
Scanning dependencies of target HelloWorldService
[ 50%] Building CXX object CMakeFiles/HelloWorldService.dir/src/HelloWorldService.cpp.o
[ 62%] Building CXX object CMakeFiles/HelloWorldService.dir/src/HelloWorldStubImpl.cpp.o
[ 75%] Building CXX object CMakeFiles/HelloWorldService.dir/src-gen/v1_0/commonapi/HelloWorldDBusStubAdapter.
cpp.o
[ 87%] Building CXX object CMakeFiles/HelloWorldService.dir/src-gen/v1_0/commonapi/HelloWorldStubDefault.cpp.o
[100%] Building CXX object CMakeFiles/HelloWorldService.dir/src-gen/v1_0/commonapi/HelloWorldDBusDeployment.cpp.
o
Linking CXX executable HelloWorldService
[100%] Built target HelloWorldService
x@ubuntu:~/work/project/build$
```

Your output should look similiar. In the build direcory there should be two executables now: HelloWorldClient and HelloWorldService.

And start:

Start

```
x@ubuntu:~/work/project/build$ ./HelloWorldService &
[1] 23066
Successfully Registered Service!
Waiting for calls... (Abort with CTRL+C)
x@ubuntu:~/work/project/build$ ./HelloWorldClient
Checking availability!
Available...
sayHello('Bob'): 'Hello Bob!'
Got message: 'Hello Bob!'
x@ubuntu:~/work/project/build$
```