CommonAPI C++ Tutorial

Contents

1 Introduction

This tutorial has the following content:

- installation instructions for CommonAPI and CommonAPI-DBus including the tools
- a step by step tutorial on how you can write your first Hello World program
- some examples with description which show the usage of CommonAPI in conjunction with Franca IDL
- some special topics like deployment or the communication with legacy D-Bus applications

Common API and its mechanism specific bindings (e.g. Common API D-Bus) provide a set of libraries and tools to work with RPC communication in a way independent of wich mechanism is used. Once you have implemented your services and clients and tested it with D-Bus you later can switch D-Bus for any other communication layer (provided it has Common API support) *without the need to touch your code or your binary at all!*.

Further information on Common API and Common API D-Bus is provided in the individual README files accompanying both packages.

2 Getting started

CommonAPI is a GENIVI project. Source code and latest news can be found at http://projects.genivi.org/commonapi/.

For documentation please visit the GENIVI document page http://docs.projects.genivi.org/.

CommonAPI currently consists of four sub-projects:

CommonAPI	This is the base C++ library, which provides the application interface for users and can load runtime bindings such as dbus.
CommonAPI-Tools	The eclipse based tools for CommonAPI. This is essentially the code generator for Franca → Common API C++ code. (This is the current package.)
CommonAPI-D-Bus	This is the D-Bus binding C++ library, which provides the necessary code to communicate over D-Bus. This is invisible to the application code, and simply needs to be linked against.
CommonAPI-D-Bus-Tools	The eclipse based tools for CommonAPI D-Bus. This is the code generator for Franca → Common API D-Bus C++ code.

Closely related to CommonAPI is the yamaica project which provides a full integration of all Franca IDL and CommonAPI plug-ins and some more enhanced features like the import and export of Franca files to Enterprise Architect (see http://projects.genivi.org/-yamaica/). The yamaica project provides eclipse update-sites for CommonAPI and yamaica (see http://docs.projects.genivi.org/-yamaica-update-site/) ready for installation.

Before we proceed you should clarifiy whether you are a user, in the sense that you want to write applications based on CommonAPI or whether you want to contribute to CommonAPI yourself or write your own middleware specific binding.

2.1 Requirements

- Code generator and Franca tooling are based on Eclipse. Please make sure that you have an appropriate Eclipse version installed.
- Make sure all requirements to build the CommonAPI Runtime are installed and in the correct version. CommonAPI was developed using gcc 4.6 and gcc 4.7, but is feature compatible to gcc 4.5 and compiler compatible to gcc 4.4.

2.2 Set up your environment as CommonAPI user

2.2.1 CommonAPI Tools

The CommonAPI Tools are available as Eclipse update-site or (if you prefer not to use Eclipse) as commandline tool.

First we assume that you use an Eclipse CDT development environment. Please make sure that you have installed Franca IDL first before you continue with installing the CommonAPI Tools. Please find the installing instructions for Franca IDL at https://code.google.com/a/eclipselabs.org/p/franca/.

The simplest way to use the CommonAPI Tools is to add the update site available on the GENIVI project servers to your Eclipse. This is available under:

Help→Install New Software→Add Button

Enter the following URL: http://docs.projects.genivi.org/yamaica-update-site/CommonAPI/updatesite/ and confirm. This provides CommonAPI, CommonAPI-D-Bus, CommonAPI Validator and all dependencies. Then select the newly added site in the site selection dropdown box, and in the Software selection window, select the entire "GENIVI CommonAPI Generators" Tree. Ignore all the other entries in the selection part of the "Available Software" window.

After the software has been installed in Eclipse you can right-click on any .fidl file and generate C++ code for CommonAPI D-Bus by selecting the "CommonAPI→Generate Common API Code" option.

- The CommonAPI Validator must not necessarily be installed. But he recognizes Franca language constructs that Franca IDL permits, but result in non-executable or non-compilable code. An example is the use of C++ keywords in the interface specification.
- From Franca IDL, you will only need to install the sub-category "Franca Feature" for the Common API and Common API D-Bus generators to work.

If you want to use the command line version of the code generator, you can get the actual version at:

http://docs.projects.genivi.org/yamaica-update-site/CommonAPI/generator/

where should be a link to the git repository:

http://git.projects.genivi.org/yamaica-update-site.git/

For the usage of the command line version please consider the README file of the CommonAPI-Tools project.

2.2.2 CommonAPI Runtime

Download the Common API runtime via git from the download site of http://projects.genivi.org/commonapi/, then compile and install the library on your computer:

```
$ git clone git://git.projects.genivi.org/ipc/common-api-runtime.git
$ cd common-api-runtime
$ autoreconf -i
$ ./configure
$ make
$ sudo make install (or alternative install process, eg. checkinstall on debian-based ↩
    distributions, such as Ubuntu)
```

With this, the Common API runtime library will be installed in `/usr/local/lib`. The package is accessible for your application e.g. via pkgconfig. The pkgconfig data is located at /usr/local/lib/pkgconfig.

To build Common API D-Bus, the Common API runtime and libdbus version 1.4.16 patched with the marshaling patch must be available through PkgConfig. The marshalling patch is provided within the Common API D-Bus package.

Download the Common API D-Bus library via git from the download site of http://projects.genivi.org/commonapi/:

```
$ git clone git://git.projects.genivi.org/ipc/common-api-dbus-runtime.git
```

Download, patch and install version 1.4.16 of libdbus (**WARNING**: *Not* following these instructions may result in corruption of the preinstalled libdbus library of your computer, thereby rendering your system unusable):

```
$ wget http://dbus.freedesktop.org/releases/dbus/dbus-1.4.16.tar.gz
$ tar -xzf dbus-1.4.16.tar.gz
$ cd dbus-1.4.16
$ patch -p1 < </your/commonapi/path>/common-api-dbus-runtime/dbus-DBusMessage-add-support- ↩
    for-custom-marshaling.patch
$ ./configure --prefix=/usr/local
$ make -C dbus
$ sudo make -C dbus install
$ sudo make install-pkgconfigDATA
```

The path to CommonAPI and patched libdbus pkgconfig files must be added to the `PKG_CONFIG_PATH` for the rest of the entire build process. If you followed the instructions above, both will be located in `/usr/local/lib/pkgconfig`, so you can just type:

```
$ export PKG_CONFIG_PATH="/usr/local/lib/pkgconfig:$PKG_CONFIG_PATH"
```

Note
If you want to make sure that you do not destroy your linux installation then it is also possible not to install the patched lidbus. In this case you just have to set your package config path correctly.

Now, compile and install the Common API D-Bus library on your computer

```
$ cd </your/download/path>/common-api-dbus-runtime
$ autoreconf -i
$ ./configure
$ make
$ sudo make install (or alternative install process, eg. checkinstall on debian-based  ↩
    distributions, such as Ubuntu)
```

With this, the libraries for Common API and Common API D-Bus are installed and ready for use.

- If you prefer to install CommonAPI from a tar file you can get the actual tar file from: http://docs.projects.genivi.org/yamaica-update-site/CommonAPI/runtime/
- In Linux please don't forget to add the installation path of your CommonAPI libraries to your LD_LIBRARY_PATH environment variable.

2.3 Set up your environment as CommonAPI contributor

2.3.1 Preliminary remarks

First get the code from the git:

```
$ git clone
```

Get an overview of all branches:

```
$ git branch
```

Switch to the branch you want to work on (master is the feature branch) and verify that it has switched (* changed)

```
$ git checkout <your branch>
$ git branch
```

Best practice is to create a local branch based on the current branch:

```
$ git branch working_branch
```

Start working, best practice is to commit smaller, compilable pieces during the development process that makes it easier to handle later on. If you want to commit you changes, send them to the author, you can create a patch like this:

```
$ git format-patch working_branch <your branch>
```

This creates a set of patches that are published via the mailing list.The patches will be discussed and then merged & uploaded on the git by the maintainer.

Patches can be accepted either under GENIVI Cla or MPL 2.0 (see section License). Please be sure that the signed-off-by is set correctly. For more, check out http://gerrit.googlecode.com/svn/documentation/2.0/user-signedoffby.html

2.3.2 CommonAPI Tools

For the build instructions of the CommonAPI tools with maven see again the README file.

3 Hello World

The examples of the use of Franca IDL in conjunction with CommonAPI can be found in the folder CommonAPI-Examples in the subproject CommonAPI tools. The first example substantially contains the code for the Hello World example, which will be described below. But even if the code already exists, it is recommended to build the sample from scratch on for a better understanding of the individual steps.

It is assumed that you have created a C++ project in your Eclipse in which all further development will happen.

3.1 Creating the RMI interface definition

The first step in developing a Common API application likely will be the definition of the RMI interface the client will use to communicate with the server. In the context of CommonAPI, the definition of this interface always happens via the Franca IDL, regardless of which communication mechanism you intend to use in the end. For this tutorial, create an arbitrarily namend file ending in *.fidl* in your Eclipse project. It is not relevant where in your project you have placed this file, as the the code generated from this file will always be put in the automatically created src-gen folder at the top level of the project hierarchy.

Open your newly created *.fidl*-file, and type the following lines:

```
package commonapi.examples

interface e01HelloWorld {
    version { major 1 minor 0 }

    method sayHello {
        in {
            String name
        }
        out {
            String message
        }
    }
}
```

Note
The *version* parameter in every interface is mandatory! No code will be generated if it is malformed or not present!

Now, save the *.fidl* file and right click it. As you have installed the Common API and Common API D-Bus generators, you will see a menu item saying *"Common API"*, with sub menu items for generating either the Common API level code only ("*Generate C++ Code*") or for generating both the Common API level code and the glue code required to run applications with using Common API D-Bus ("*Generate D-Bus C++ Code*").

3.2 Generating code

We do want to use D-Bus as middleware, so we will need the D-Bus specific glue code as well as the Common API level code which we will program agains. Therefore, you might want to chose the latter of the two options provided by the generator plugin ("*Generate D-Bus C++ Code*"). After having done so, you will see the newly created src-gen folder and it's contents. The files will be created according to their fully qualified names relative to src-gen as the top level folder, as defined in the *.fidl*-file:

```
E01HelloWorld.h
E01HelloWorldProxy.h
E01HelloWorldProxyBase.h
E01HelloWorldStub.h
E01HelloWorldStubDefault.cpp
E01HelloWorldStubDefault.h

E01HelloWorldDBusProxy.cpp
E01HelloWorldDBusProxy.h
E01HelloWorldDBusStubAdapter.cpp
E01HelloWorldDBusStubAdapter.h
```

All files that have a "DBus" in their name are glue code required by the D-Bus binding and are not relevant while developing your application, they only need to be compiled with your application (there are ways to NOT compile these sources with your applications and include them at runtime instead; see the README of Common API D-Bus for details).

All other files that have a *Proxy* in their name are relevant for you if you develop a client, all other files that have a *Stub* in their name are relevant for you if you develop a service. A proxy is a class that provides method calls that will result in remote method invocations on the service, plus registration methods for events that can be broadcasted by the service.

A stub is the part of the service that will be called when a remote method invocation from a client arrives. It also contains methods to fire events (broadcasts) to several or all clients. The Stub comes in two flavors: One default stub that contains empty implementations of all methods, thereby allowing you to implement only the ones you are interested in, and a Stub skeleton where you have to implement everything yourself before you can use it. A service will have to implement a subclass of either of the two in order to make itself available to the outside world (or just use the default stub if your service should not be able to do anything except firing events).

In this tutorial, we will create both a client and a service in order to be able to see some communication going on.

3.3 Implement the Client

Start by creating a new .cpp source file in your project (e.g. e01HelloWorldClient.cpp). Make sure you have a main method in order to start the client application.

Here, you will need two includes in order to access the Common API client functionality:

```
#include <iostream>

#include <CommonAPI/CommonAPI.h> //Defined in the Common API Runtime library
#include <commonapi/examples/E01HelloWorldProxy.h> //Part of the code we just generated
```

The first thing each and every Common API application will do is to load a runtime:

```
std::shared_ptr<CommonAPI::Runtime> runtime = CommonAPI::Runtime::load();
```

If you link the Common API DBus library to and compile the generated DBus specific code with your executable, this runtime "magically" will be a runtime that provides access to the DBus communication infrastructure via a strictly CommonAPI level

interface. If you link the library and add the generated code of another Common API middleware binding instead, this runtime will provide access to this other communication infrastructure.

In order to be able to communicate with a specific service, we need a proxy. We can create a proxy by using a factory, which in turn we can get from the runtime we just created:

```
std::shared_ptr<CommonAPI::Factory> factory = runtime->createFactory();
const std::string& serviceAddress = "local:commonapi.examples.HelloWorld:commonapi.examples ↩
    .HelloWorld";
std::shared_ptr<E01HelloWorldProxy<>> myProxy = factory->buildProxy<E01HelloWorldProxy>( ↩
    serviceAddress);
```

The parameter *serviceAddress* is the address at which the service that shall be accessed will be available. This address will be translated internally to an actual DBus-Address - or whatever format fits the communication infrastructure you use. Semantically, this address consists of three parts, separated by colons:

Domain	The first part, defines in which domain the service is located. For DBus use cases, only "local" makes any sense, as no services that are more remote than "on the same operating system" are accessible.
ServiceID	The second part. This defines the name or type of the service that shall be accessed.
InstanceID	The third part. This defines the specific instance of this service that shall be accessed.

There are ways to influence the translation of the Common API address to the specific address (of course once again without the need to change your code). Please have a look at the README of Common API DBus if you want to know more about this possibility in the context of DBus, or the corresponding documentation of the other middleware binding you are using.

With this, the client is set up and ready to use. We should wait for the service to be available, then we can start issuing calls:

```
while (!myProxy->isAvailable()) { usleep(10); }

const std::string name = "World";
CommonAPI::CallStatus callStatus;
std::string returnMessage;

myProxy->sayHello(name, callStatus, returnMessage);
if (callStatus != CommonAPI::CallStatus::SUCCESS) {
    std::cerr << "Remote call failed!\n";
    return -1;
}
std::cout << "Got message: '" << returnMessage << "'\n";
```

3.4 Implement the Service

Works about the same way as implementing the client. The includes that are required are the following:

```
#include <iostream>
#include <thread>

#include <CommonAPI/CommonAPI.h>
#include "E01HelloWorldStubImpl.h"
```

And we also need a stub that actually does something when the method we call in the client gets called:

```
void E01HelloWorldStubImpl::sayHello(const std::shared_ptr<CommonAPI::ClientId> clientId,
        std::string name, std::string& message) {

    std::stringstream messageStream;
```

```
    messageStream << "Hello " << name << "!";
    message = messageStream.str();
    std::cout << "sayHello('" << name << "'): '" << message << "'\n";
};
```

The rest looks quite similar to the client side, with the difference that we do not issue calls via a proxy, but instead register a service that then will be provided to the outside world. The service is registered using the same Common API address, which allows the proxy to actually find the service. Afterwards, we just wait for calls:

```
std::shared_ptr<CommonAPI::Runtime> runtime = CommonAPI::Runtime::load();
std::shared_ptr<CommonAPI::Factory> factory = runtime->createFactory();
std::shared_ptr<CommonAPI::ServicePublisher> servicePublisher =
        runtime->getServicePublisher();

const std::string& serviceAddress =
        "local:commonapi.examples.HelloWorld:commonapi.examples.HelloWorld";

std::shared_ptr<E01HelloWorldStubImpl> myService =
        std::make_shared<E01HelloWorldStubImpl>();

servicePublisher->registerService(myService, serviceAddress, factory);

while(true) {
    std::cout << "Waiting for calls... (Abort with CTRL+C)" << std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(60));
}
```

3.5 Running the Demo

Build the two applications using your favourite build system. If all worked well, you should see communication ongoing via DBus (e.g. via dbus-monitor), and you should get output from your client once, saying

```
"Got Message: 'Hello World'".
```

4 Advanced Topics

4.1 Special Franca Features

4.1.1 Selective

Selective broadcasts, indicated with the selective keword after the name of the broadcast in Franca can be sent to individual clients rather than all participants. This is accomplished with the use of a ClientID parameter which can serve to identify the targets. Note if you wish to identify clients absolutely you must do this on the application level.

We can add a block to our interface for the selective broadcast:

```
broadcast saySomething selective {
    out {
        String message
    }
}
```

We can the add some code to our stub implementation to handle this:

```
virtual void onSaySomethingSelectiveSubscriptionChanged(
        const std::shared_ptr<CommonAPI::ClientId> clientId,
        const CommonAPI::SelectiveBroadcastSubscriptionEvent event) {

    if (event == CommonAPI::SelectiveBroadcastSubscriptionEvent::SUBSCRIBED) {
        lastId = clientId;
    }
}
```

and to the sayHello method:

```
std::shared_ptr<CommonAPI::ClientIdList> receivers =
        std::make_shared<CommonAPI::ClientIdList>();

if (lastId) {
    receivers->insert(lastId);
}

this->fireSaySomethingSelective("Broadcast to last ID", receivers);
```

and finally a member to the stub:

```
private:
  std::shared_ptr<CommonAPI::ClientId> lastId;
```

The onSaySomethingSelectiveSubscriptionChanged method is called when a subscription for this boroadcast changes. Is added, we set the memeber lastId to this value. Whenever sayHello is called we now also send a broadcast only to the last client which registered. The ClientIdList contains all the intended recipients, and the middleware will send the message to all memeber of this list who are registered for the broadcast. If NULL is passed instead the broadcast is sent to all clients.

On the client side we can add this before the const std::string name = "World"; line:

```
helloWorldProxy->getSaySomethingSelectiveEvent().subscribe(
        [&](const std::string& message) {
            std::cout << "Received broadcast message: " << message << "\n";
});
```

This casues a subscription to the broadcast, and the lambda function will be called whenever we receive a message.

4.1.2 Managed Stubs

In Franca a relationship between two interfaces can be declared in the form "x manages y". This indicates that x has a number of y as childeren whose lifecycle is tied to and managed by x. This allows a an application to activare y stubs on an instance of x stubs. Similarily on the proxy side a manager for y interfaces is available on the x proxy, where we can be informed of appearing and disappearing instances, and interrogate the network about the state of instances.

In our Franca file change the defenition of the HelloWorldInterface to:

```
interface HelloWorldInterface manages HelloWorldLeaf {
```

and after this interface block add:

```
interface HelloWorldLeaf {
    version { major 1 minor 0 }

    method sayHelloLeaf {
        in {
            String name
        }
        out {
```

```
            String message
        }
    }
}
```

This causes HelloWorldInterface to be able to manage HelloWorldLeaf instances. Note that HelloWorldLeaf instances can still be registered and accessed in the normal way via the service publisher and factory. To the stub application add to the top as a new class defenition:

```
class MyHelloWorldLeafStub: public commonapi::examples::HelloWorldLeafStubDefault {
 public:
    virtual void sayHelloLeaf(std::string name, std::string& message) {
        std::stringstream messageStream;

        messageStream << "Hello Leaf " << name << "!";
        message = messageStream.str();

        std::cout << "sayHelloLeaf('" << name << "'): '" << message << "'\n";
    }
};
```

To the bottom of the main function before the while loop add:

```
auto helloWorldLeafStub = std::make_shared<MyHelloWorldLeafStub>();

const std::string leafInstance = "commonapi.examples.HelloWorld.Leaf";

const bool leafOk = helloWorldStub->registerManagedStubHelloWorldLeaf(
        helloWorldLeafStub, leafInstance);

if (!leafOk) {
    std::cerr << "Error: Unable to register leaf service!\n";
    return -1;
}
```

To the client programm add to the bottom of the main function just before return 0:

```
const std::string leafInstance = "commonapi.examples.HelloWorld.Leaf";
CommonAPI::CallStatus callStatusAv;
CommonAPI::AvailabilityStatus availabilityStatus;
helloWorldProxy->getProxyManagerHelloWorldLeaf().getInstanceAvailabilityStatus(
        leafInstance, callStatusAv, availabilityStatus);

if (callStatusAv == CommonAPI::CallStatus::SUCCESS &&
        availabilityStatus == CommonAPI::AvailabilityStatus::AVAILABLE) {

    auto helloWorldLeafProxy =
        helloWorldProxy->getProxyManagerHelloWorldLeaf().
            buildProxy<commonapi::examples::HelloWorldLeafProxy>(leafInstance);

    const std::string nameLeaf = "WorldLeaf";
    CommonAPI::CallStatus callStatusLeaf;
    std::string helloWorldLeafMessage;

    std::cout << "Sending name: '" << nameLeaf << "'\n";
    helloWorldLeafProxy->sayHelloLeaf(nameLeaf, callStatusLeaf, helloWorldLeafMessage);
    if (callStatusLeaf != CommonAPI::CallStatus::SUCCESS) {
        std::cerr << "Remote call failed!\n";
        return -1;
    }

    std::cout << "Got message: '" << helloWorldLeafMessage << "'\n";
```

```
} else {
    std::cout << "Leaf Proxy not available\n";
    sleep(5);
    return -1;
}
```

This instantiates and calls a managed leaf stub and proxy respectively.

4.2 Attribute Extensions

As described in the CommonAPI specification there is a general scheme to include individual extensions in order to provide any additional features for attributes. In principle Attribute Extensions work as follows:

- Decide whether you want to implement a common attribute extension for all attributes of an interface or a specific extension for one attribute.
- Define a so-called templated extension class which inherits from `CommonAPI::AttributeExtension` within a hpp-file. The example below defined in `AttributeCacheExtension.hpp` shows an extension class for the caching of attributes on proxy side.
- Generate the proxy for your interface with the API method `buildProxyWithDefaultAttributeExtension` if you want a common extension. For the specific extension call build proxy with the attribute extension as template parameter.
- Now you can call the implemented functions of your extension via a `getNameAttributeExtension()` call (where `Name` is the name of your attribute).

Example of an extension class for caching attributes (file `AttributeCacheExtension.hpp`, see example e02Attributes):

```
#include <CommonAPI/CommonAPI.h>

template<typename _AttributeType>
class AttributeCacheExtension: public CommonAPI::AttributeExtension<_AttributeType> {
    typedef CommonAPI::AttributeExtension<_AttributeType> __baseClass_t;

 public:
    typedef typename _AttributeType::ValueType value_t;
    typedef typename _AttributeType::AttributeAsyncCallback AttributeAsyncCallback;

    AttributeCacheExtension(_AttributeType& baseAttribute) :
        CommonAPI::AttributeExtension<_AttributeType>(baseAttribute),
        isCacheValid_(false) {
    }

    ~AttributeCacheExtension() {}

    bool getCachedValue(value_t& value) {

        if (isCacheValid_) {

            value = cachedValue_;
        } else {

            __baseClass_t::getBaseAttribute().getChangedEvent().subscribe(std::bind(
                        &AttributeCacheExtension<_AttributeType>::onValueUpdate,
                        this,
                        std::placeholders::_1));

            CommonAPI::CallStatus callStatus;
            __baseClass_t::getBaseAttribute().getValue(callStatus, value);
```

```
        }

        return isCacheValid_;
    }

 private:

    void onValueUpdate(const value_t& t) {
        isCacheValid_ = true;
        cachedValue_ = t;
    }

    mutable bool isCacheValid_;
    mutable value_t cachedValue_;
};
```

In your main function you can call now:

```
#include "AttributeCacheExtension.hpp"
using your::namespace; // eg commonapi::examples

... // Other code here

const std::string& serviceAddress = "...";

// your proxy class is ProxyName
std::shared_ptr<CommonAPI::DefaultAttributeProxyFactoryHelper<ProxyName,
                                    AttributeCacheExtension>::class_t> myProxy =
    factory->buildProxyWithDefaultAttributeExtension<ProxyName,
                                        AttributeCacheExtension>(serviceAddress);

// and then if your attribute is x of type Int32:
int32_t valueCached = 0;
myProxy->getXAttributeExtension().getCachedValue(valueCached);
```

5 Examples

5.1 Preliminary remarks

The examples describe how some standard problems of interface design and implementation can be solved with Franca IDL and CommonAPI. Before you start make sure that your environment is properly installed. That means:

- You can start the code generator in your Eclipse installation or from command line.
- Your PKG_CONFIG_PATH contains the paths to the package config files of CommnAPI, CommonAPI-DBus and your patched D-Bus.
- You have patched the DBus library with the patch in CommonAPI-D-Bus and you are sure that your LD_LIBRARY_PATH contains the path to the patched DBus library.
- Make sure dbus session bus is available ($ env |grep DBUS_SESSION_BUS_ADDRESS, output must not be empty!).
- If your platform is Windows, please read the instructions for Windows first. The following instructions assume that your are working on a Linux platform.

The examples provide a more or less generic CMake file for building the executables. Two executables are needed: a service and a client program. The standard procedure to build one of these example programs is:

```
$ mkdir build; cd build
$ cmake -D DBUS_PATH_LOCAL=<your_local_path_of_the_patched_dbus_library> -D PRJ_NAME=< ↩
    name_of_the_example> ..
$ make
```

name_of_the_example is `e01HelloWorld` for the first example.

Here are some hints if you want to configure your eclipse project for the CMake build:

- Install the CMakeEd plug-in which provides an editor for CMake files (http://cmakeed.sourceforge.net/eclipse/).
- Create a build directory (eg /build) directly in the project directory.
- Select the projects properties, then C/C++ build and set the Build Command to your make call:

```
make -C ${ProjDirPath}/build VERBOSE=1
```

- The Build Directory must be empty.
- Create a make target (eg via Window→Show View→Make Target). Set the build command to:

```
cmake -E chdir build/ cmake -D DBUS_PATH_LOCAL=
    <your_local_path_of_the_patched_dbus_library> -D PRJ_NAME=<name_of_the_example> ..
```

and leave the Make Target field empty. Finally, create in "Run Configurations" two configurations for client and service. Note that it might be necessary to start the executables with a local setting of the `LD_LIBRARY_PATH` (eg to `/usr/local/lib`) since CommonAPI applications use a patched version of libdbus.

5.2 Example 1: Hello World

This example contains the code for the step by step introduction to CommonAPI.

5.3 Example 2: Attributes

Consider the Franca IDL specification of example 2:

```
package commonapi.examples

interface E02Attributes {
    version { major 1 minor 0 }

    attribute Int32 x

    attribute CommonTypes.a1Struct a1
}

typeCollection CommonTypes {

    struct a1Struct {
        String s
        a2Struct a2
    }

    struct a2Struct {
        Boolean b
        Double d
    }
}
```

Modelling attributes in interfaces means in general that the service that implements this interface has an internal state which shall be visible for external clients like a HMI (Human Machine Interface). A developer of a client would normally expect that he can set and get the attribute and that he can notify or subscribe to changes of the value of the attribute. We will see below in the implementation how exactly this is realized by CommonAPI. Franca offers two key words that indicate exactly how the attribute can be accessed: `readonly` and `noSubscriptions`. The default setting is that everything is allowed; with these two additional key words these possibilies can be limited (eg if someone tries to call a set method and the attribute is readonly he will get an error at compile time).

The nested structure `a1Struct` is defined in a type collection `CommonTypes`. Structures can be defined just like other type definitions within an interface definition or outside in a type collection. Since Franca 0.8.9 type collections can also be anonymous (without name). A type collection is transferred by the CommonAPI code generator in an additional namespace.

The Franca interface specification of attributes does not contain any information about whether the access from client side is synchronous or asynchronous or whether the attribute is cached by the proxy. CommonAPI provides always methods for synchronous and asynchronous setter and getter methods; caching can be realized via an API extension.

Now let's have a look to the CommonAPI code on the service side. The default implementation of the stub which is generated by the CommonAPI codegenerator defines the attribute as private attribute of the stub class. This attribute can be accessed from the stub implementation via getter and setter functions. Additionaly the API for the stub implementation provides some callbacks (the following code snippet shows parts of the generated stub class which refer to the attribute x):

```
class E02AttributesStubDefault : public virtual E02AttributesStub {
 public:
    E02AttributesStubDefault();

    virtual const int32_t& getXAttribute(
        const std::shared_ptr<CommonAPI::ClientId> clientId);

    virtual void setXAttribute(
        const std::shared_ptr<CommonAPI::ClientId> clientId, int32_t value);

 protected:
    virtual bool trySetXAttribute(int32_t value);
    virtual bool validateXAttributeRequestedValue(const int32_t& value);
    virtual void onRemoteXAttributeChanged();

 private:
    int32_t xAttributeValue_;
};
```

If the implementation of the stub has to change the value of the attribute `x`, let's say in a class `E02AttributesStubImpl` that is derived from `E02AttributesStubDefault`, then it can call `setXAttribute` (analog the usage of `getXAttribute`). The callback `onRemoteXAttributeChanged` informs that a change of the attribute `x` has been completed. The other callbacks can prevent the set of the attribute (`validateXAttributeRequestedValue`) or change the given value from the client (`trySetXAttribute`).

In the example the service increments a counter every 2 seconds and publishes the counter value via the interface attribute `x`.

Now see the implementation of the client. The simplest case is to get the current value of `x`. The following extract shows one part of the main function:

```
#include <iostream>

#include <CommonAPI/CommonAPI.h>
#include <commonapi/examples/E02AttributesProxy.h>

using namespace commonapi::examples;

int main() {

    std::shared_ptr < CommonAPI::Runtime > runtime = CommonAPI::Runtime::load();

    std::shared_ptr < CommonAPI::Factory > factory = runtime->createFactory();
```

```
    const std::string& serviceAddress =
        "local:commonapi.examples.Attributes:commonapi.examples.Attributes";
    std::shared_ptr < E02AttributesProxyDefault > myProxy =
        factory->buildProxy < E02AttributesProxy > (serviceAddress);

    while (!myProxy->isAvailable()) { usleep(10); }

    CommonAPI::CallStatus callStatus;
    int32_t value = 0;

    // Get actual attribute value from service
    myProxy->getXAttribute().getValue(callStatus, value);
    if (callStatus != CommonAPI::CallStatus::SUCCESS) {
        std::cerr << "Remote call A failed!\n";
        return -1;
    }
    std::cout << "Got attribute value: " << value << std::endl;

}
```

The `getXAttribute` method will deliver the type `XAttribute` which has to be used for the access to `x`. Every access returns a flag named callStatus (please see the CommonAPI specification). Subscription requires in general the definition of a callback function which is called in case of an attribute change. The subscribe method of CommonAPI requires a function object; for a compact notation this function object can be defined as lambda function:

```
myProxy->getXAttribute().getChangedEvent().subscribe([&](const int32_t& val) {
    std::cout << "Received change message: " << val << std::endl;
});
```

Of course it is also possible to define a separate callback function with an user-defined name (here recv_cb):

```
void recv_cb(const CommonAPI::CallStatus& callStatus, const int32_t& val) {
    std::cout << "Receive callback: " << val << std::endl;
}

.... // main method

// Subscribe for receiving values, alternative implementation 1
std::function<void (int32_t)> f = recv_msg;
myProxy->getXAttribute().getChangedEvent().subscribe(f);

// Subscribe for receiving values, alternative implementation 2
myProxy->getXAttribute().getChangedEvent().subscribe(
    std::bind(recv_msg, std::placeholders::_1));
```

Asynchronous setting of attributes via setValueAsync works analog as shown in the following code extract where the more complex attribute *a1* is set from the client:

```
void recv_cb_s(const CommonAPI::CallStatus& callStatus,
        const CommonTypes::a1Struct& valStruct) {

    std::cout << "Receive callback for structure: a1.s = " <<
        valStruct.s << ", valStruct.a2.d = " << valStruct.a2.d << std::endl;
}
.... // main method

CommonTypes::a1Struct valueStruct;

valueStruct.s = "abc";
valueStruct.a2.b = true;
valueStruct.a2.d = 1234;
```

```
std::function<void (const CommonAPI::CallStatus&, CommonTypes::a1Struct)> fcb_s =
        recv_cb_s;

myProxy->getA1Attribute().setValueAsync(valueStruct, fcb_s);
```

As described above, in the chapter "Attribute Extensions" of this tutorial, it is possible to extend the standard CommonAPI for attributes by defining Attribute Extensions. In this example you have to:

- include the template definition of the extension (AttributeCacheExtension.hpp)
- to call a different factory method for creating the proxy (e.g. buildProxyWithDefaultAttributeExtension)
- and then it is possible to call the new defined methods, e.g.

```
int32_t valueCached = 0;
bool r;
r = myProxy->getXAttributeExtension().getCachedValue(valueCached);
std::cout << "Got cached attribute value[" << (int)r << "]: " << valueCached << std::endl;
```

See the source code of the example for a deeper insight.

5.4 Example 3: Methods

Franca attributes represent status variables or data sets of services which shall be accessible by the clients of the service. In contrast, methods can be used for example to start a process in the service or to query for certain information (eg, from a database). See the following example 3:

```
package commonapi.examples

interface E03Methods {

    version { major 1 minor 0 }

    method foo {
        in {
            Int32 x1
            String x2
        }
        out {
            Int32 y1
            String y2
        }
        error {
            stdErrorTypeEnum
        }
    }

    broadcast myStatus {
        out {
            Int32 myCurrentValue
        }
    }

    enumeration stdErrorTypeEnum {
        NO_FAULT
        MY_FAULT
    }
}
```

Basically Franca methods have input parameters and output parameters and can return an optional application error which reports for example if the started process in the sevice could be finished successfully or not. Input and output parameters can have arbitrarily complex types, a separate definition of so-called InOut arguments of functions was not considered necessary.

A special case are broadcasts. They can be used like readonly attributes. But there are several output parameters allowed (and no input parameters). Another difference is the additional optional keyword selective, which indicates that only selected clients can register on the broadcast see example 4).

Another optional keyword (only for methods) is fireAndForget which indicates that neither return values nor a call status is expected.

The implementation of the service class is straight:

```
#include "E03MethodsStubImpl.h"

using namespace commonapi::examples;

... // Additional code

void E03MethodsStubImpl::foo(int32_t x1, std::string x2,
        E03Methods::fooError& methodError, int32_t& y1, std::string& y2) {

    std::cout << "foo called, setting new values." << std::endl;

    methodError = (E03Methods::fooError)E03Methods::stdErrorTypeEnum::MY_FAULT;
    y1 = 42;
    y2 = "xyz";
}
```

The input parameters are available as values, the output parameter as references. The standard name for the application error is methodError. In the example there is another function incCounter implemented which sends the broadcast myStatus via the generated method fireMyStatusEvent:

```
void E03MethodsStubImpl::incCounter() {
    cnt++;
    fireMyStatusEvent((int32_t)cnt);
    std::cout <<  "New counter value = " << cnt << "!" << std::endl;
};
```

The subscription to the broadcast is nearly identical to the subscription to the change of the value of an attribute. The example shows further an asynchronous and a synchronous call of the function foo; in the asynchronous case the callback function `recv_cb` is defined.

```
#include <iostream>

#include <CommonAPI/CommonAPI.h>
#include <commonapi/examples/E03MethodsProxy.h>

using namespace commonapi::examples;

void recv_cb(const CommonAPI::CallStatus& callStatus, const E03Methods::fooError&  ↩
    methodError,
        const int32_t& y1, const std::string& y2) {

... // your code
}

int main() {

    // Subscribe to broadcast
    myProxy->getMyStatusEvent().subscribe([&](const int32_t& val) {
        std::cout << "Received status event: " << val << std::endl;
    });
```

```
    while(true) {

        int32_t inX1 = 5;
        std::string inX2 = "abc";
        CommonAPI::CallStatus callStatus;
        E03Methods::fooError methodError;
        int32_t outY1;
        std::string outY2;

        // Synchronous call
        std::cout << "Call foo with synchronous semantics ..." << std::endl ;
        myProxy->foo(inX1, inX2, callStatus, methodError, outY1, outY2);

        // Asynchronous call
        std::cout << "Call foo with asynchronous semantics ..." << std::endl;

        std::function<void (const CommonAPI::CallStatus&,
            const E03Methods::fooError&, const int32_t&, const std::string&)> fcb = recv_cb ↩
                ;

        myProxy->fooAsync(inX1, inX2, recv_cb);

        std::this_thread::sleep_for(std::chrono::seconds(5));
    }
    return 0;
}
```

A frequently asked question is what happens if the service does not answer. In this case, the callback function `recv_cb` is called anyway, but the `callStatus` has the value `REMOTE_ERROR`. It is however the responsibility of the specific middleware to implement this behavior. The CommonAPI specification says:

- **NOT** considered to be a remote error is an application level error that is defined in the corresponding Franca interface, because from the point of view of the transport layer the service still returned a valid answer.
- It **IS** considered to be a remote error if no answer for a sent remote method call is returned within a defined time. It is discouraged to allow the sending of any method calls without a defined timeout. This timeout may be middleware specific. This timeout may also be configurable by means of a Franca Deployment Model. It is **NOT** configurable at runtime by means of the Common API.

The actual version of the D-Bus binding has a non configurable timeout of about 5 seconds.

5.5 Example 4: PhoneBook

This slightly more complex example illustrates the application of some Franca features in combination with CommonAPI:

- explicit named arrays and inline arrays
- selective broadcasts
- polymorphic structs

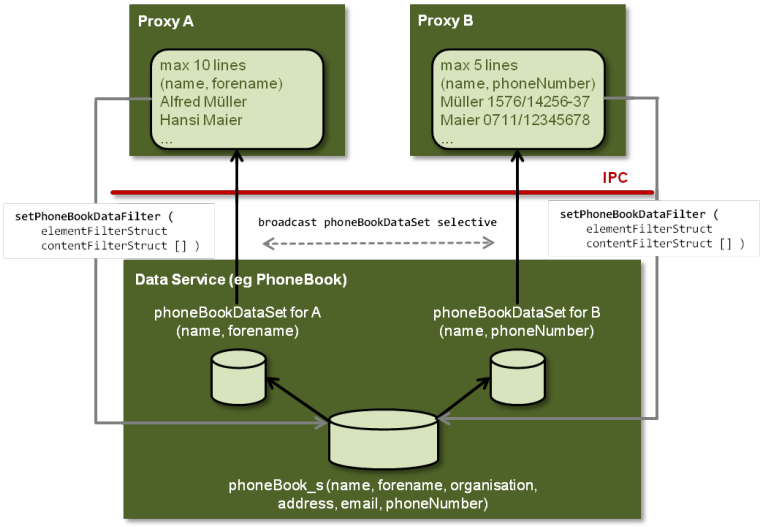
Concerning arrays please note the following points:

- In Franca there are two ways to define arrays: explicitly named (array myArray of UInt8) or implicit without defining a new name for the array (UInt8 []).
- The implicit definition of multidimensional arrays is not possible at the moment (like UInt8 [][]), but multidimensional arrays can be defined with explicit names.

- In CommonAPI arrays are implemented and generated as std::vector.

A common problem in the specification of interfaces between user frontends and services which contain large data sets is, that the clients usually need only extracts from the database. That means that only a filtered excerpt from the database has to be transmitted via IPC to the client, but probably every client needs a different excerpt. The filter can affect the selection of the elements (element filter), the contents of the elements (content filter) or the number of elements (array window).

The following example shows how different extracts of a central data array can be accessed by the several clients via a data filter mechansim and selective broadcasts. As example of a central data array a phonebook is selected; the following picture shows the basic content of the example.



The Franca IDL specification is:

```
package commonapi.examples

interface E04PhoneBook {

    version { major 1 minor 0 }

    <** @description : the phone book itself **>
    attribute phoneBookStruct [] phoneBook readonly

    <** @description : filter operations **>
    method setPhoneBookDataFilter {
        in {
            elementFilterStruct elementFilter
            contentFilterStruct [] contentFilter
        }
    }

    <** @description : filter result **>
    broadcast phoneBookDataSet selective {
        out {
            phoneBookDataElementMap [] phoneBookDataSet
        }
    }

    <** @description : Data types of the phone book itself **>
    enumeration phoneNumberEnum {
        WORK
        HOME
        MOBILE1
```

```
        MOBILE2
    }

    map phoneNumberMap {
        phoneNumberEnum to String
    }

    struct phoneBookStruct {
        String name
        String forename
        String organisation
        String address
        String email
        phoneNumberMap phoneNumber
    }

    <** @description : Data types for the filter operations **>

    struct elementFilterStruct {
        Boolean addName
        Boolean addForename
        Boolean addOrganisation
        Boolean addAddress
        Boolean addEmail
        Boolean addPhoneNumber
    }

    struct contentFilterStruct {
        phoneBookDataElementEnum element
        String expression
    }

    <** @description : Data types for the result of the phone book filter **>
    enumeration phoneBookDataElementEnum {
        NAME
        FORENAME
        ORGANISATION
        ADDRESS
        EMAIL
        PHONENUMBER
    }

    struct phoneBookDataElement polymorphic {
    }

    struct phoneBookDataElementString extends phoneBookDataElement {
        String content
    }

    struct phoneBookDataElementPhoneNumber extends phoneBookDataElement {
        phoneNumberMap content
    }

    map phoneBookDataElementMap {
        phoneBookDataElementEnum to phoneBookDataElement
    }
}
```

The phone book itself is modeled as an attribute which is an array of the structure `phoneBookStruct`. Here the phone book is readonly, that means that the whole content can be accessed only via subscription and the getter function. A special difficulty is the phone number, because there are several kinds of phone numbers allowed (home, mobile, ...). Therefore the element

`phoneNumber` in `phoneBookStruct` is a map with an enumeration key and a value of type string for the number. The client can set a filter to the phone book data (in the example only content filter and element filter, but other filters are conceivable) via the method `setPhoneBookDataFilter` and gets the data back via the selective broadcast `phoneBookDataSet`. Since the content of the data set depends on the filter, the elements of the client specific data set are specified as maps where the key is the type of the element (name, forename, ...) and the value is the content of the element. The content can be of the type String or of the user defined type phoneNumberMap. Therefore the value is defined as polymorphic struct which can be a String or a phoneNumberMap.

In the following we consider only some interesting implementation details, for the complete implementation please see the source code.

The interesting part of the service is the implementation of the set function for the data filter. At the moment only the element filter is implemented, but the implementation of the other filters can be added analogously.

- Each client is identified via its clientId; the implementation of client ID class allows the usage of cientId objects as key in a map (see the specification).
- The data sets of the filtered data for the clients are stored in a map with the clientId as key; in this example the filtered data are sent back to the client directly in the filter set function. Please note, that firePhoneBookDataSetSelective sends the data to only one receiver.
- The value of the key has to be the right type (`phoneNumberMap` for phoneNumbers and Strings for the rest).

```
void E04PhoneBookStubImpl::setPhoneBookDataFilter (
        const std::shared_ptr<CommonAPI::ClientId> clientId,
        E04PhoneBook::elementFilterStruct elementFilter,
        std::vector<E04PhoneBook::contentFilterStruct> contentFilter ) {

    std::shared_ptr<CommonAPI::ClientIdList> clientIdList =
            getSubscribersForPhoneBookDataSetSelective();

    std::vector<E04PhoneBook::phoneBookDataElementMap> lPhoneBookDataSet;
    phoneBookClientData.erase (clientId);

    std::vector<E04PhoneBook::phoneBookStruct>::const_iterator it0;
    for (it0 = getPhoneBookAttribute().begin();
            it0 != getPhoneBookAttribute().end(); it0++ ) {

        E04PhoneBook::phoneBookDataElementMap lPhoneBookDataElement;

        if ( elementFilter.addName ) {
            std::shared_ptr<E04PhoneBook::phoneBookDataElementString> name =
                std::make_shared<E04PhoneBook::phoneBookDataElementString>();

            name->content = it0->name;

            lPhoneBookDataElement[E04PhoneBook::phoneBookDataElementEnum::NAME] =
                name;
        }

        ... // analogue for the other elements

        if ( elementFilter.addPhoneNumber ) {
            std::shared_ptr<E04PhoneBook::phoneBookDataElementPhoneNumber> phoneNumber =
                std::make_shared<E04PhoneBook::phoneBookDataElementPhoneNumber>();

            phoneNumber->content = it0->phoneNumber;

            lPhoneBookDataElement[E04PhoneBook::phoneBookDataElementEnum::PHONENUMBER] =
                phoneNumber;
        }
```

```
        lPhoneBookDataSet.push_back(lPhoneBookDataElement);

    }

    phoneBookClientData[clientId] = lPhoneBookDataSet;

    // Send client data
    const std::shared_ptr<CommonAPI::ClientIdList> receivers(new CommonAPI::ClientIdList);
    receivers->insert(clientId);

    firePhoneBookDataSetSelective(lPhoneBookDataSet, receivers);
    receivers->erase(clientId);
}
```

On client side we create two proxies which shall set different filters and get different data sets. For these two proxies we need different factories! Otherwise CommonAPI cannot keep the proxies apart. Each proxy has to subscribe to phoneBookDataSet, but gets different contents depending on the filter. The whole phoneBookData can be obtained via the standard get function.

```
int main() {

    std::shared_ptr<CommonAPI::Runtime> runtime = CommonAPI::Runtime::load();

    std::shared_ptr<CommonAPI::Factory> factoryA = runtime->createFactory();
    std::shared_ptr<CommonAPI::Factory> factoryB = runtime->createFactory();

    const std::string& serviceAddress =
        "local:commonapi.examples.PhoneBook:commonapi.examples.PhoneBook";

    std::shared_ptr<E04PhoneBookProxy<>> myProxyA =
        factoryA->buildProxy<E04PhoneBookProxy>(serviceAddress);
    while (!myProxyA->isAvailable()) { usleep(10); }

    std::shared_ptr<E04PhoneBookProxy<>> myProxyB =
        factoryB->buildProxy<E04PhoneBookProxy>(serviceAddress);
    while (!myProxyB->isAvailable()) { usleep(10); }

    // Subscribe A to broadcast
    myProxyA->getPhoneBookDataSetSelectiveEvent().subscribe(
        [&](const std::vector<E04PhoneBook::phoneBookDataElementMap>& phoneBookDataSet) {
        printFilterResult(phoneBookDataSet, "A");
    });

    // Subscribe B to broadcast
    myProxyB->getPhoneBookDataSetSelectiveEvent().subscribe(
        [&](const std::vector<E04PhoneBook::phoneBookDataElementMap>& phoneBookDataSet) {
        printFilterResult(phoneBookDataSet, "B");
    });

    // Get actual phoneBook from service
    CommonAPI::CallStatus myCallStatus;
    std::vector<E04PhoneBook::phoneBookStruct> myValue;
    myProxyA->getPhoneBookAttribute().getValue(myCallStatus, myValue);

    // Synchronous call setPhoneBookDataFilter
    E04PhoneBook::elementFilterStruct lElementFilterA =
        { true, true, false, false, false, false};

    std::vector<E04PhoneBook::contentFilterStruct> lContentFilterA =
        { {E04PhoneBook::phoneBookDataElementEnum::NAME, "*"} };
    myProxyA->setPhoneBookDataFilter(lElementFilterA, lContentFilterA, myCallStatus);

    E04PhoneBook::elementFilterStruct lElementFilterB =
```

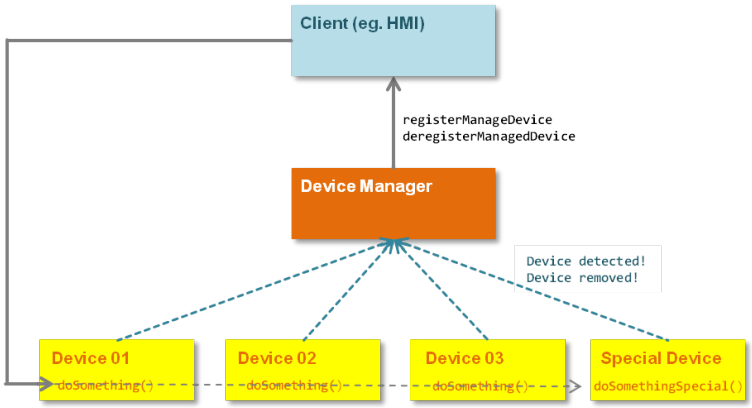
```
        { true, false, false, false, false, true };

    std::vector<E04PhoneBook::contentFilterStruct> lContentFilterB =
        { {E04PhoneBook::phoneBookDataElementEnum::NAME, "*"} };
    myProxyB->setPhoneBookDataFilter(lElementFilterB, lContentFilterB, myCallStatus);

    ... // further code
}
```

5.6 Example 5: Managed

So far we have looked at software systems, which consisted of services and users of these servces, the clients. However, in some systems there is a slightly different kind of relationship between the software components: a central manager manages other services (let's call them slaves or leaves). This central manager is a service itself but acts as client for the managed services. One example for such a system is, for example, a device manager that manages several devices, which can be availabe for usage in the system or not. The central manager handles all adminitrative tasks related to the slaves and provides a central, common interface to the user frontend.



Franca IDL supports this kind of software structure by the keyword *manages*, which can be added to the keyword interface. The following example illustrates the application of this keyword.

```
package commonapi.examples

interface E05Manager manages E05Device, E05SpecialDevice {
    version { major 1 minor 0 }

    attribute String [] myDevices
}

interface E05Device {
    version { major 1 minor 0 }

    method doSomething {
    }
}

interface E05SpecialDevice extends E05Device {
    version { major 1 minor 0 }

    method doSomethingSpecial {
    }
}
```

The device manager has the service interface E05Manager and it manages devices with the interfaces E05Device and E05SpecialDevice. It is important to understand, that the exact meaning of the keyword `manages` cannot be defined by the IDL; the keyword just indicates that there is a relationship between software components which implement manager and managed interfaces. It can be used in bindings as CommonAPI to provide API functions for a more convenient implementation of this certain kind of relationship.

Therefore let's have a look at the generated and implemented source code. Since we have three interfaces the code generator generates for every interface proxy and stub classes. In our example we just want to illuminate one aspect of the managed interfaces: the registration of the managed interfaces via the manager. We assume that we have only one service (the manager) which gets informed about a detected or a removed device via the public function deviceDetected and deviceRemoved. The devices are distinguished by a number; in principle there is an arbitrary number of devices possible. The registration of the devices at CommonAPI does the manager in his stub implementation.

```
... // includes, namespaces, constructors as usual

E05ManagerStubImpl::E05ManagerStubImpl (const std::string instanceName) {

    managerInstanceName = instanceName;
}

void E05ManagerStubImpl::deviceDetected (unsigned int n) {

    std::string deviceInstanceName = getDeviceName (n);
    myDevices[deviceInstanceName] = DevicePtr (new E05DeviceStubImpl);
    const bool deviceRegistered = this->registerManagedStubE05Device(
        myDevices[deviceInstanceName], deviceInstanceName);
}

void E05ManagerStubImpl::deviceRemoved (unsigned int n) {

    std::string deviceInstanceName = getDeviceName (n);
    const bool deviceDeregistered = this->deregisterManagedStubE05Device(
        deviceInstanceName);
    if ( deviceDeregistered ) { myDevices.erase (deviceInstanceName); }
}

std::string E05ManagerStubImpl::getDeviceName (unsigned int n) {

    std::stringstream ss;
    ss <<  managerInstanceName <<
        ".device" << std::setw(2) << std::hex << std::setfill('0') << n;
    return ss.str();
}

... // implementation of special device analogously
```

The functions for the special device has been omitted for clarity.The implementations of the devices themselves are not important here as well. See now the implementation of the client. The client gets informed about new or removed devices including the addresses of these devices via the callback function newDeviceAvailable. This function can be subscribed as function object at an status event that is triggered when a device is added or removed.

```
#include <iostream>

#include <CommonAPI/CommonAPI.h>
#include <commonapi/examples/E05ManagerProxy.h>

using namespace commonapi::examples;

void newDeviceAvailable (const std::string address,
        const CommonAPI::AvailabilityStatus status) {

    if ( status == CommonAPI::AvailabilityStatus::AVAILABLE ) {
```

```
        std::cout << "New device available: " << address << std::endl;
    }

    if ( status == CommonAPI::AvailabilityStatus::NOT_AVAILABLE ) {

        std::cout << "Device removed: " << address << std::endl;
    }
}

int main() {

    std::shared_ptr<CommonAPI::Runtime> runtime = CommonAPI::Runtime::load();

    std::shared_ptr<CommonAPI::Factory> factory = runtime->createFactory();

    const std::string& serviceAddress =
        "local:commonapi.examples.Manager:commonapi.examples.Manager";

    std::shared_ptr<E05ManagerProxy<>> myProxy =
        factory->buildProxy<E05ManagerProxy>(serviceAddress);

    while (!myProxy->isAvailable()) { usleep(10); }

    std::cout << "Proxy available." << std::endl;

    CommonAPI::ProxyManager::InstanceAvailabilityStatusChangedEvent& deviceEvent =
            myProxy->getProxyManagerE05Device().
                getInstanceAvailabilityStatusChangedEvent();

    CommonAPI::ProxyManager::InstanceAvailabilityStatusChangedEvent& specialDeviceEvent =
            myProxy->getProxyManagerE05SpecialDevice().
                getInstanceAvailabilityStatusChangedEvent();

    std::function<void (const std::string, const CommonAPI::AvailabilityStatus)>
        newDeviceAvailableFunc = newDeviceAvailable;

    deviceEvent.subscribe(newDeviceAvailableFunc);
    specialDeviceEvent.subscribe(newDeviceAvailableFunc);

    while(true) {
        std::this_thread::sleep_for(std::chrono::seconds(5));
    }
    return 0;
}
```

5.7 Example 6: Unions

Until now, some simple and complex data types in the examples already occurred. This example intends to describe the use of unions closer and to compare it with the usage of polymorphic structs. Consider the following Franca IDL example:

```
package commonapi.examples

interface E06Unions {
    version { major 1 minor 0 }

    attribute CommonTypes.SettingsUnion u
    attribute CommonTypes.SettingsStruct x
}
```

```
typeCollection CommonTypes {

    typedef MyTypedef is Int32

    enumeration MyEnum {
        DEFAULT
        ON
        OFF
    }

    union SettingsUnion {
        MyTypedef id
        MyEnum status
        UInt8 channel
        String name
    }

    struct SettingsStruct polymorphic {
    }

    struct SettingsStructMyTypedef extends SettingsStruct {
        MyTypedef id
    }

    struct SettingsStructMyEnum extends SettingsStruct {
        MyEnum status
    }

    struct SettingsStructUInt8 extends SettingsStruct {
        UInt8 channel
    }

    struct SettingsStructString extends SettingsStruct {
        String name
    }
}
```

We first want to leave the question aside whether this example makes sense from an application point of view or not; it is just an example for demonstration purposes. With unions we can transmit data of different types in one attribute. These different types are enumerated in one structure with the keyword **union**. D-Bus knows a similar data type which is called variant. Variants are used in the D-Bus binding for the implementation of unions. The interesting point is here not the definition of the union, but the realization in CommonAPI. I just want to point out here that it can lead to problems with the compiler or generally to problems with your toolchain if you define unions with an significant number of members (eg. >10), because each of these members appears in the generated C++ code as template argument in the template declaration.

On the other hand we see the definition of a `polymorphic struct` which can lead to a similar but not the same behavior. The difference is that the types of the `polymorphic struct` definitions are extensions of a base type (here `SettingsStruct`), that means that they are inherited from this base type. The base type might contain some base elements which are then be inherited by the children. Another difference is, that the C++ API allows real polymorphic behavior. With Unions that is not possible, since there is no base type as we will see below.

The implementation of the set function for the attribute u in the stub implementation could be as follows:

```
void E06UnionsStubImpl::setMyValue(int n) {

    if ( n >= 0 && n < 4 ) {

        CommonTypes::MyTypedef t0 = -5;
        CommonTypes::MyEnum t1 = CommonTypes::MyEnum::OFF;
        uint8_t t2 = 42;
        std::string t3 = "abc";
```

```
        if ( n == 0 ) {
            CommonTypes::SettingsUnion v(t0);
            setUAttribute(v);
            setXAttribute(std::make_shared<CommonTypes::SettingsStructMyTypedef>(t0));
        } else if ( n == 1 ) {
            CommonTypes::SettingsUnion v(t1);
            setUAttribute(v);
            setXAttribute(std::make_shared<CommonTypes::SettingsStructMyEnum>(t1));
        } else if ( n == 2 ) {
            CommonTypes::SettingsUnion v(t2);
            setUAttribute(v);
            setXAttribute(std::make_shared<CommonTypes::SettingsStructUInt8>(t2));
        } else if ( n == 3 ) {
            CommonTypes::SettingsUnion v(t3);
            setUAttribute(v);
            setXAttribute(std::make_shared<CommonTypes::SettingsStructString>(t3));
        }
    }
}
```

Depending on a condition (here the value of n) the attributes u and x are filled with data of different types. Please note that the argument of `setUAttribute` has the type `CommonAPI::Variant<MyTypedef, MyEnum, uint8_t, std::string>`, whereas the argument of `setXAttribute` is a pointer to the base type `SettingsStruct`.

The standard implementation on client side to get the value of the attribute uses the API call `isType` in case of the union attribute. First we have to subscribe; in the callback function it is possible to get the value of our attribute by checking the type which leads to an if / then / else cascade:

```
#include <iostream>

#include <CommonAPI/CommonAPI.h>
#include <commonapi/examples/E06UnionsProxy.h>
#include <commonapi/examples/CommonTypes.h>

using namespace commonapi::examples;

void evalA (const CommonTypes::SettingsUnion& v) {

    if ( v.isType<CommonTypes::MyTypedef>() ) {
        std::cout << "Received (A) MyTypedef with value " <<
            v.get<CommonTypes::MyTypedef>() << " at index " <<
            (int)v.getValueType() << std::endl;
    } else if ( v.isType<CommonTypes::MyEnum>() ) {
        std::cout << "Received (A) MyEnum with value " <<
            (int) (v.get<CommonTypes::MyEnum>()) << " at index " <<
            (int)v.getValueType() << std::endl;
    } else if ( v.isType<uint8_t>() ) {
        std::cout << "Received (A) uint8_t with value " <<
            (int) (v.get<uint8_t>()) << " at index " <<
            (int)v.getValueType() << std::endl;
    } else if ( v.isType<std::string>() ) {
        std::cout << "Received (A) string " << v.get<std::string>() <<
            " at index " << (int)v.getValueType() << std::endl;
    } else {
        std::cout << "Received (A) change message with unknown type." << std::endl;
    }
}

void recv_msg(const CommonTypes::SettingsUnion& v) {
    evalA(v);
}
```

```
int main() {
    std::shared_ptr<CommonAPI::Runtime> runtime = CommonAPI::Runtime::load();
    std::shared_ptr<CommonAPI::Factory> factory = runtime->createFactory();
    const std::string& serviceAddress =
        "local:commonapi.examples.Unions:commonapi.examples.Unions";
    std::shared_ptr<E06UnionsProxyDefault> myProxy =
        factory->buildProxy<E06UnionsProxy> (serviceAddress);
    while (!myProxy->isAvailable()) { usleep(10); }

    std::function<void (CommonTypes::SettingsUnion)> f = recv_msg;
    myProxy->getUAttribute().getChangedEvent().subscribe(f);

    while (true) { usleep(10); }
    return 0;
}
```

The example shows, how it is possible to detect the type and the value of the received attribute. However, the if / then / else cascade is not the only, perhaps not the best way to get the value of the union on client side [1]. One alternative implementation is based on an `typeIdOf` function, which is at the moment not part of CommonAPI but can be additionally implemented (see `typeUtils.hpp` of this example):

```
#include <type_traits>

template <typename SearchT, typename... T>
struct typeIdOf;

template <typename SearchT, typename T>
struct typeIdOf<SearchT, T> {

    static const int value = std::is_same<SearchT, T>::value ? 1 : -1;
};

template <typename SearchT, typename T1, typename... T>
struct typeIdOf<SearchT, T1, T...> {
    static const int value = std::is_same<SearchT, T1>::value ?
        sizeof...(T)+1 : typeIdOf<SearchT, T...>::value;
};
```

The evaluation method (corresponding to `evalA` above) looks like:

```
template <typename T1, typename... T>
void evalB (const CommonAPI::Variant<T1, T...>& v) {

    switch (v.getValueType()) {

    case typeIdOf<CommonTypes::MyTypedef, T1, T...>::value:
        std::cout << "Received (B) MyTypedef with value " <<
            (int)(v.template get<CommonTypes::MyTypedef>()) << std::endl;
        break;
    case typeIdOf<CommonTypes::MyEnum, T1, T...>::value:
        std::cout << "Received (B) MyEnum with value " <<
            (int)(v.template get<CommonTypes::MyEnum>()) << std::endl;
        break;
    case typeIdOf<uint8_t, T1, T...>::value:
        std::cout << "Received (B) uint8_t with value " <<
            (int)(v.template get<uint8_t>()) << std::endl;
        break;
    case typeIdOf<std::string, T1, T...>::value:
        std::cout << "Received (B) string " <<
            v.template get<std::string>() << std::endl;
```

[1] These two very good alternative implementations come from Martin Häfner from Harman. Thank you!

```
        break;
    default:
        std::cout << "Received (B) change message with unknown type." << std::endl;
        break;
    }
}
```

One advantage here is that instead of the if / then / else statement a switch / case statement can be used.

The second alternative implementation uses the function overloading. The overloaded functions are defined within a structure (`MyVisitor` in the example) that is used as a visitor in the evaluation function:

```
struct MyVisitor {

    explicit inline MyVisitor() {}

    template<typename... T>
    inline void eval(const CommonAPI::Variant<T...>& v) {
        CommonAPI::ApplyVoidVisitor<MyVisitor,
                CommonAPI::Variant<T...>, T...>::visit(*this, v);
    }

    void operator()(CommonTypes::MyTypedef val) {
        std::cout << "Received (C) MyTypedef with value " << (int)val << std::endl;
    }

    void operator()(CommonTypes::MyEnum val) {
        std::cout << "Received (C) MyEnum with value " << (int)val << std::endl;
    }

    void operator()(uint8_t val) {
        std::cout << "Received (C) uint8_t with value " << (int)val << std::endl;
    }

    void operator()(std::string val) {
        std::cout << "Received (C) string " << val << std::endl;
    }

    template<typename T>
    void operator()(const T&) {
        std::cout << "Received (C) change message with unknown type." << std::endl;
    }

    void operator()() {
        std::cout << "NOOP." << std::endl;
    }
};

void evalC(const CommonTypes::SettingsUnion& v) {
    MyVisitor visitor;
    visitor.eval(v);
}
```

Finally, it should given here for comparison the implementation on the client side for the `polymorphic struct`. The subscription for a message receive function is identical to the previous implementation; the message receive function now looks as follows:

```
void recv_msg(std::shared_ptr<CommonTypes::SettingsStruct> x) {

    if ( std::shared_ptr<CommonTypes::SettingsStructMyTypedef> sp =
        std::dynamic_pointer_cast<CommonTypes::SettingsStructMyTypedef>(x) ) {
        std::cout << "Received (D) MyTypedef with value " <<
```

```
            (int)sp->id << std::endl;
    } else if ( std::shared_ptr<CommonTypes::SettingsStructMyEnum> sp =
        std::dynamic_pointer_cast<CommonTypes::SettingsStructMyEnum>(x) ) {
        std::cout << "Received (D) MyEnum with value " <<
            (int)sp->status << std::endl;
    } else if ( std::shared_ptr<CommonTypes::SettingsStructUInt8> sp =
        std::dynamic_pointer_cast<CommonTypes::SettingsStructUInt8>(x) ) {
        std::cout << "Received (D) uint8_t with value " <<
            (int)sp->channel << std::endl;
    } else if ( std::shared_ptr<CommonTypes::SettingsStructString> sp =
        std::dynamic_pointer_cast<CommonTypes::SettingsStructString>(x) ) {
        std::cout << "Received (D) string " << sp->name << std::endl;
    } else {
        std::cout << "Received (D) change message with unknown type." << std::endl;
    }
}
```

The result you get now by dynamic cast of the base type.

6 FAQ

6.1 The middleware loading mechanism of Common API

6.1.1 CommonAPI::Runtime::load() returns no runtime object, why?

As it was mentioned before, when you call `CommonAPI::Runtime::load()` you "magically" will have access to a specific middleware library. In a very basic case, the library and thereby communication mechanism you will have access to will be the only Common API middleware library you linked to your executable during compilation.

However, this call to `load()` most likely will **FAIL** if you have no generated middleware specific code that is compiled with your application. Why that?

The reason is simple, once understood: Most linkers are lazy. They do not link libraries that seem to be unused. Due to the fact that there is no reference whatsoever from Common API (and therefore your application) to any of the middleware libraries, the linker considers any and all middleware libraries as unused if not referenced by middleware specific generated code, and therefore will not add them to the executable.

You can disable this behavior by passing the linker flag `whole-archive` during the build process. Note however that this behavior normally is a good optimization without repercussions - except probably in the context of CommonAPI.

6.1.2 How can I use Using more than one middleware binding?

CommonAPI provides the possibility to use more than one middleware binding at once. In this case, you should no longer use `CommonAPI::Runtime::load()`, but instead `CommonAPI::Runtime::load("NameOfSomeMiddleware")`.

The "NameOfSomeMiddleware" is the well known name of the middleware you want to load. It is defined and made public by each of the middlewares that support Common API. For DBus, this name is simply "DBus".

6.1.3 Fully dynamic loading and additional information

This topic is handled in-depth in the specification of Common API. Please refer to this file for any further information.