CommonAPI C++ User Guide

Contents

1 Introduction

1.1 Aim of this document

This user guide has the following content:

- installation instructions for CommonAPI in general including the code generator (CommonAPI Tools)
- a step by step tutorial on how you can write your first Hello World program
- examples for a deeper insight into the usage of CommonAPI in conjunction with Franca IDL
- additional information to CommonAPI which is not part of the CommonAPI specification

1.2 CommonAPI C++

CommonAPI C++ is a standardized C++ API specification for the development of distributed applications which communicate via a middleware for interprocess communication. The main intention is to make the C++ interface for applications independent from the underlying IPC stack. The basic principle can be seen in the following picture.

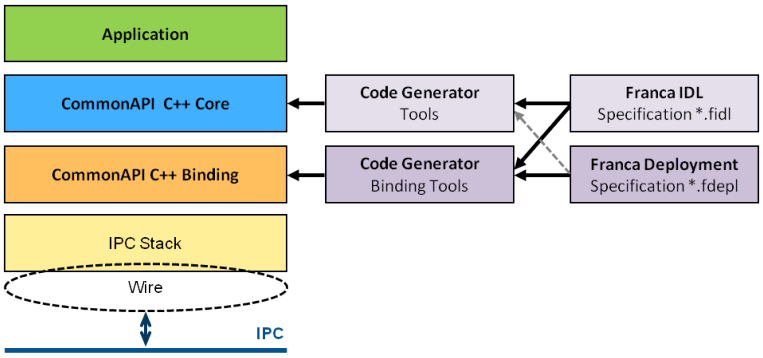


Figure 1: CommonAPI C++ Overview 1

- CommonAPI C++ is divided up in a middleware-independent part (CommonAPI Core) and in a middleware-specific part (CommonAPI Binding).
- CommonAPI uses the interface description language FrancaIDL for the specification of interfaces (logical interface specification). Code generation from FrancaIDL is an integrated part of CommonAPI.
- The code generator for CommonAPI C++ bindings needs middleware-specific parameters (deployment parameters). These parameters are defined in Franca deployment files (*.fdepl).

Note
CommonAPI C++ Core has no obligatory deployment parameters. But it turned out that it makes sense to add also additional deployment parameter to CommonAPI C++ Core itself.

The user API of CommonAPI is divided up into two parts (see picture below):

- A FrancaIDL based, generated part which contains API functions that are related to the types, attributes and methods of the FrancaIDL files.
- A "common" part (Runtime API) which contains API functions for loading the runtime environment, creating proxies and so on.

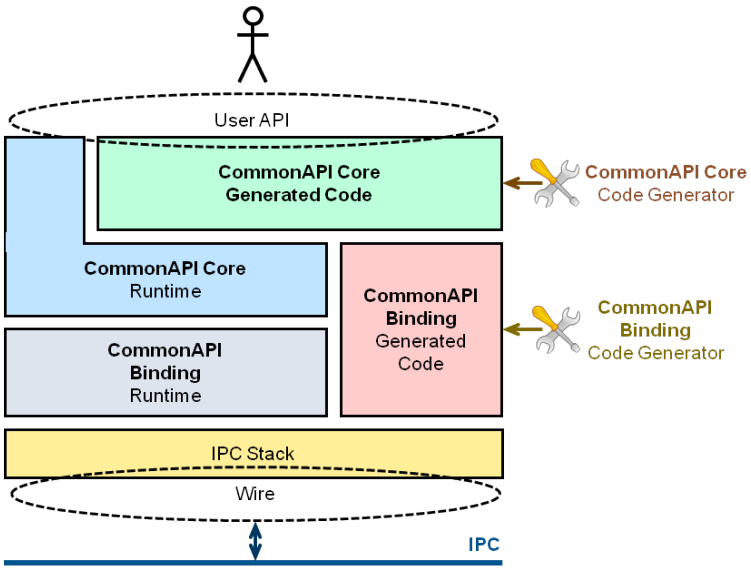


Figure 2: CommonAPI C++ Overview 2

This picture shows in more detail how the elements of CommonAPI C++ fit together. Note that:

- the vast majority of the user API is the generated part of CommonAPI.
- there is no direct relation between CommonAPI Core and the IPC stack.
- the generated code of the CommonAPI Binding has interfaces to all other parts of CommonAPI.

The workflow for developers of applications is as follows:

- Create a FrancaIDL file with the specification of an interface with methods and attributes.
- Generate code for client and service by starting the CommonAPI code generator.
- Implement the service by implementing the methods in the generated skeleton.
- Implement the client by creating proxies and calling these methods by using the proxy.

1.3 Links

CommonAPI is a GENIVI project. Source code and latest news can be found at http://projects.genivi.org/commonapi/. Source code can be found in the Git repository (http://git.projects.genivi.org/). For documentation please visit the GENIVI document page http://docs.projects.genivi.org/.

Note
At http://git.projects.genivi.org/ you will find only source code even for the code generator. It might be cumbersome to build the code generator of your own; for your convenience you will find executables and update-sites at http://docs.projects.genivi.org/-yamaica-update-site/.

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

The official FrancaIDL site is at the moment: https://code.google.com/a/eclipselabs.org/p/franca/

2 Integration Guide for CommonAPI users

The following descriptions assume that host and target platform are Linux platforms. However CommonAPI supports also Windows as host and target platform. All you need to know for Windows concerning CommonAPI you find in the separate Windows paragraph below at the end of this Integration Guide.

2.1 Requirements

CommonAPI was developed for GENIVI and will run on most Linux platforms. Additionally it is possible to run it under Windows for test and development purposes. Please note:

- CommonAPI uses a lot of C++11 features, as variadic templates, std::bind, std::function and so on. Make sure that the compiler of your target platform is able to compile it (e.g. gcc 4.8).
- The build system of CommonAPI is CMake; please make sure that it is installed on your host. CommonAPI requires a CMake version > 2.8.12.
- This user guide describes only the "common" CommonAPI part; there are no binding specific explanations. Please refer to the binding specific user guide for further information.
- Do not use earlier versions of Eclipse as Luna; it could work but there is no warranty.
- The build tool chain for the code generators is Maven; make sure that at least Maven 3 is available. If you use eclipse make sure that the maven plug-in is installed.

2.2 Dependencies

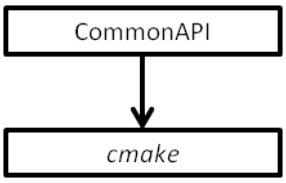


Figure 3: CommonAPI-Dependencies

2.3 Compile Runtime

2.3.1 Command-line

For building CommonAPI from the command-line download the Common API runtime via Git from the GENIVI Git repository, switch to the desired tag (e.g. 3.0.0) and compile it using CMake:

```
$ git clone git://git.projects.genivi.org/ipc/common-api-runtime.git
$ cd common-api-runtime
$ git checkout tags/3.0.0
$ mkdir build
$ cd build
$ cmake ..
$ make
```

This is the standard procedure and will hopefully create the shared CommonAPI runtime library libCommonAPI.so in *build/src/CommonAPI*. Note that CMake checks if doxygen and asciidoc are installed. These tools are only necessary if you want to generate the documentation of your own. The unit tests of CommonAPI are implemented by using the Google C++ Testing Framework. If you want to build and run the unit tests the environment variable `GTEST_ROOT` must point to the correct directory (see the contributor's guide below).

Note
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/

There are several options for calling CMake and make targets.

Generate makefile for building a static CommonAPI library (default is a shared library). The library will be in */build/src/CommonAPI*.

```
$ cmake -DBUILD_SHARED_LIBS=OFF ..
```

Generate makefile for building the release version of CommonAPI (default is debug).

```
$ cmake -DCMAKE_BUILD_TYPE=Release ..
```

Without any further settings `make install` will copy CommonAPI libraries and header files to */usr/local*. You can change this destination directory by changing the installation prefix (e.g. to test).

```
$ cmake -DCMAKE_INSTALL_PREFIX=/test  ..
```

Additional cmake parameters:

`-DUSE_INSTALLED_COMMONAPI`	OFF, ON	use uninstalled / installed CommonAPI core library
`-DMAX_LOG_LEVEL`	ERROR, WARNING, INFO, DEBUG, VERBOSE	log messages with lower log level are ignored

Make targets:

`make all`	Same as make. Will compile and link CommonAPI.
`make clean`	Deletes binaries, but not the files which has been generated by CMake.
`make maintainer-clean`	Deletes everything in the build directory.
`make install`	Copies libraries to */user/local/lib/commonapiX.X.X* and header files to */user/local/include/commonapiX.X.X/CommonAPI*.
`make DESTDIR=< install_dir > install`	The destination directory for the installation can be influenced by `DESTDIR`.

Further make targets will be described in the contributor's guide below.

2.3.2 Eclipse

Start with importing your project by *File→New→Makefile Project with Existing Code*. Select your project directory and *Linux GCC*.

If not yet available, create a `Make Target` (e.g. with the name `Run cmake`) and edit it. Set the "`Build command:`" to:

```
cmake -E chdir build/ cmake -G "Unix Makefiles" ../
```

and delete the "`Make target:`" field and let it empty.

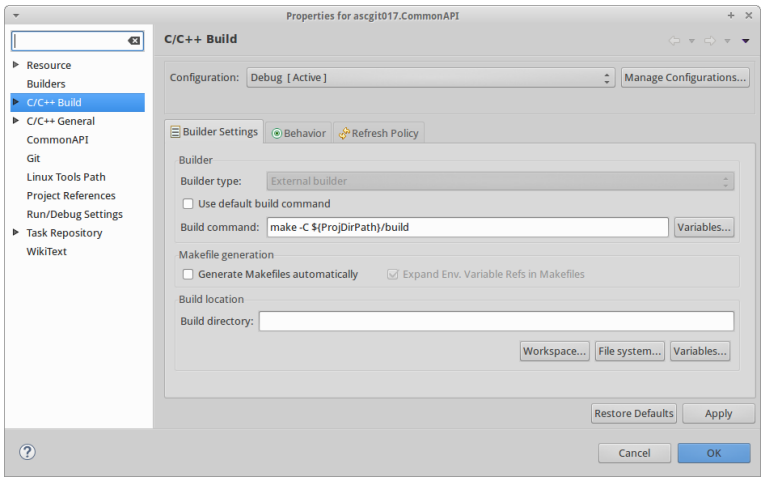


Figure 4: CommonAPI C++ Eclipse Settings 01

Create the `build` directory directly in your project directory.

Then you can start `Build target` in the context menu of your `Make target` and then build the project by *Project→Build Project*.

2.4 Compile tools

An executable version of the command-line version of the CommonAPI code generator is available for Linux (32 bit) as zip-file. An update-site for installing the code generators in eclipse is also available (see below). The following instructions are for the case that you have to build the code generator yourself.

2.4.1 Command-line

You can build all code generators by calling maven from the command-line. Open a console and change in the directory org.genivi.commonapi.core.releng of your CommonAPI-Tools directory. Then call:

```
mvn -Dtarget.id=org.genivi.commonapi.core.target clean verify
```

After the successful build you will find the commond-line generators archived in org.genivi.commonapi.core.cli.product/target/products/c
generator.zip and the update-sites in org.genivi.commonapi.core.updatesite/target.

2.4.2 Eclipse

Make sure that you have installed the *m2e - Maven Integration for Eclipse* plug-in. Then create a *Run Configuration* in Eclipse. Open the *Run configuration* settings. On the left side of you should find a launch configuration category *Maven Build*. Create a new launch configuration and add the parameter `target.id` with the value `org.genivi.commonapi.core.target`. Set the *Goals* to *clean verify*.

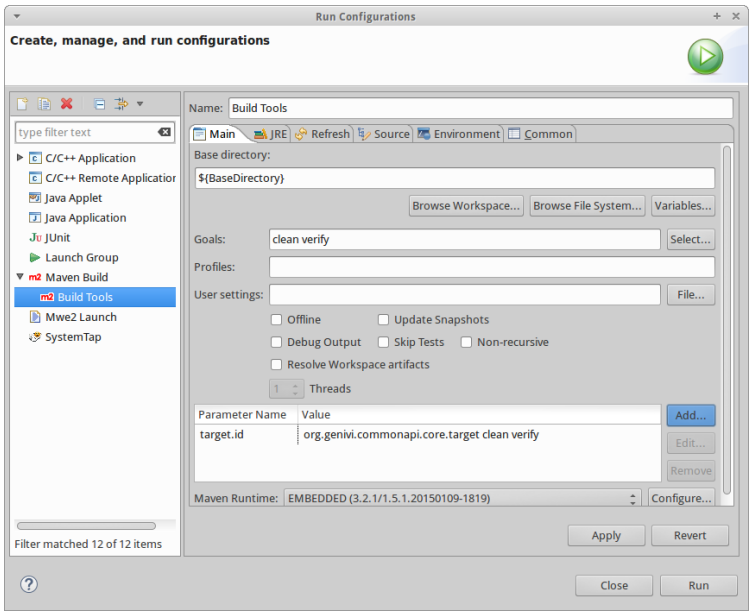


Figure 5: CommonAPI C++ Eclipse Build Tools Settings

2.5 Write Applications

CommonAPI requires a basic workflow for creating executable applications.

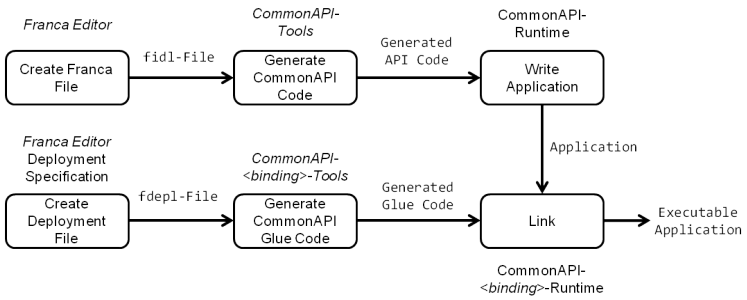


Figure 6: CommonAPI C++ Workflow

2.5.1 Generating Code

No matter which development environment is used, the API for the applications is created by the CommonAPI code generator which is available as command-line version and as Eclipse update-site.

The simplest way to use the CommonAPI Tools is to add the update-site available on the GENIVI project servers to your Eclipse. Add the update site in Eclipse by calling *Help→Install New Software→Add*. Enter the URL: http://docs.projects.genivi.org/-yamaica-update-site/CommonAPI/updatesite/ and confirm.

Then select the newly added site in the site selection dropdown box, and in the Software selection window, select the entire "GENIVI Common API" Tree.

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

If you have built the update-site of your own, you can add the site as well; just add it as archive.

An executable version of the command-line version of the CommonAPI code generator is available only for Linux (32 bit) as zip-file at http://docs.projects.genivi.org/yamaica-update-site/CommonAPI/generator. Download the zip-file and unzip it to an appropriate directory. Then it is ready for use.

Call the executable `commonapi_generator` as follows:

```
commonapi_generator [options] file [file...]
```

Valid Options are:

`-dest < path/to/output/folder >`	The generated files will be saved at this Location.
`-pref < path/to/header/file >`	Here you can set the text which will be placed as a comment on each generated file (for example your license).

Note
If your CommonAPI binding requires deployment files the input for the code generator is the appropriate deployment file which imports the Franca file. For some bindings (e.g. D-Bus) and deployment file is not obligatory; in this case it is also possible to start the code generator with fidl-files as input.

2.5.2 Build Applications

Your application should compile and link only with the generated CommonAPI code and the CommonAPI runtime library. For a fast setup please consider the provided examples in CommonAPI-Tools/CommonAPI-Examples.

2.6 Project Setup

2.6.1 Structuring CommonAPI project libraries

CommonAPI executables typically consist of 6 parts:

1. The application code itself which is written manually by the developer.
2. The generated CommonAPI (binding independent) code. In clients this code contains proxy functions which are called by the application; in services it contains generated functions which must be manually implemented by the developer (optionally it is possible to generate default implementations).
3. The CommonAPI runtime library.
4. The generated, binding specific code (so-called glue code).
5. The runtime library of the binding.
6. Generic libraries of the used middleware (e.g. `libdbus`).

Even if there are several possibilities to divide these 6 parts up into shared or static libraries and to integrate them on a target platform, there is a standard way which is intended for the integration of CommonAPI applications (see picture below).

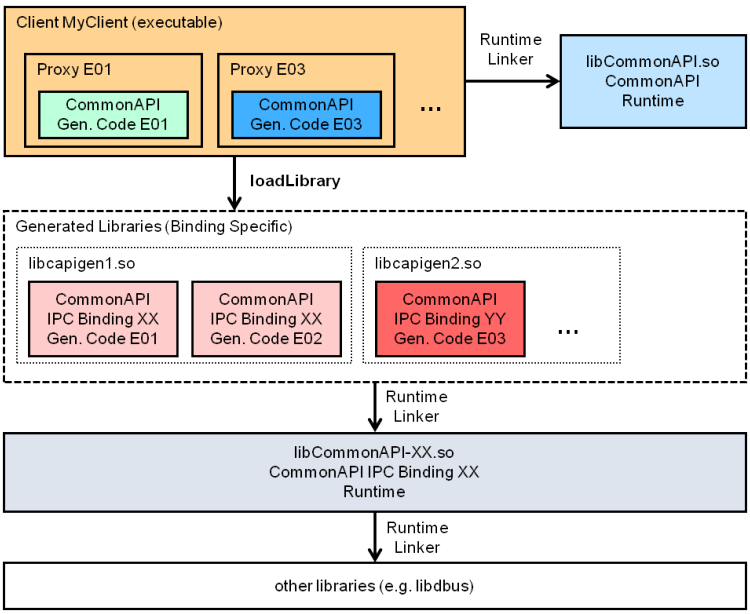


Figure 7: CommonAPI C++ Structuring Libraries

The standard way assumes that there is a platform software which provides necessary standard libraries in a suitable version. For CommonAPI these are the runtime libraries of CommonAPI itself, bindings and the associated middleware. Furthermore the platform should provide binding specific libraries which contain generated binding specific code (glue code). The glue code can be divided up into several shared libraries depending on used bindings, number of interfaces and other platform and project specific requirements. The application can now be delivered together with the generated binding independent code which can be statically or dynamically linked.

The glue code library will now be loaded at the exact moment in which a proxy is created. The right library will be found by the evaluation of the CommonAPI configuration file that contains the association between CommonAPI address and the glue code library. In the case that there are no entries in the configuration file, default settings are used.

The glue code library is binding specific; that means that the required runtime libraries are loaded automatically by the runtime linker. In the case that a certain instantiated interface in a service is offered via another middleware as before, it is sufficient just to change the entries in the configuration file, provided that a generated glue code library is available.

2.6.2 Write CommonAPI Configuration Files

CommonAPI and available bindings can be configured by ini-files (see e.g. http://en.wikipedia.org/wiki/INI_file).

The CommonAPI configuration file is `commonapi.ini`. There are three places where CommonAPI Runtime tries to find this file (in the following order):

1. in the directory of the current executable. If there is a `commonapi.ini` file, it has the highest priority.
2. in the directory which is specified by the environment variable `COMMONAPI_CONFIG`.
3. in the global default directory `/etc`.

The configuration file has 4 possible sections; all sections are optional.

`logging`

CommonAPI has an internal logging mechanism which can be parameterized by the settings of this section:

- `console=true/false`

- `file=` <*file name*>
- `dlt=true/false`
- `level=fatal/error/warning/info/debug/verbose`

Example:

```
[logging]
console=true
file=./mylog.log
dlt=true
level=verbose
```

Note
If the configured log level is higher than the maximum log level that was defined at compile time, the maximum log level will be used.

`default`

Section for setting the default value for the used binding.

- `binding=dbus/someip`

Example:

```
[default]
binding=dbus
```

`proxy`

This section defines for each required CommonAPI address the shared library with the binding specific, generated glue code which has to be loaded when the proxy is created.

- <*CommonAPI address*>=<*library name*>

If no library is defined, CommonAPI uses default settings for library names and paths; for further information see chapter *Creating Proxies And Stubs*.

Example:

```
[proxy]
local:commonapi.examples.Test:commonapi.examples.Test=libTest-DBus.so
```

`stub`

Analogous to the proxy section.

Example:

```
[stub]
local:commonapi.examples.Test:commonapi.examples.Test=libTest-DBus.so
```

2.6.3 CommonAPI Deployment

The CommonAPI code generator supports nearly the full feature set of Franca IDL and works without any deployment file. However it is possible to write deployment files for interface specifications not only for bindings but also for CommonAPI itself based on the following deployment specification.

```
specification org.genivi.commonapi.core.deployment {

    for interfaces {
        /*
         * define the enumeration backing type on CommonAPI C++ level for whole interface.
         */
        DefaultEnumBackingType : {UInt8, UInt16, UInt32, UInt64, Int8, Int16, Int32, Int64} ↩
             (default: UInt32);
    }

    for providers {
        /*
         * Enumerate all service instances this provider depends on (if any).
         */
        ClientInstanceReferences : Instance[] (optional);
    }

    for instances {
        /*
         * The CommonAPI address string has the format "domain:interfaceid:instanceid"
         * according to CommonAPI specification.
         * To avoid inconsistencies only domain and instance id can be specified during  ↩
            deployment,
         * while the interface id is fixed by the interface the instance realizes.
         */
        Domain :     String (default: "local"); // the domain part of the CommonAPI address ↩
            .
        InstanceId : String;                    // the instance id of the CommonAPI  ↩
            address.

        /*
         * Define default timeout for all methods awaiting results of an instance in  ↩
            seconds.
         * 0s means no timeout/waiting forever.
         * if the timeout elapsed without arrival of a valid result an error will be  ↩
            delivered to the application.
         */
        DefaultMethodTimeout : Integer (default:0);

        /*
         * provide properties to register for instance. use "Name=Value" as format.
         * This deployment property is currently not supported.
         */
        PreregisteredProperties : String [] (optional);
    }

    for methods {
        /*
         * timeout for method calls in ns.
         * If timeout is defined with value > 0, then a method call will return with a  ↩
            timeout error
         * in case no result arrived before the timeout elapsed.
         * 0 means no timeout.
         */
        Timeout : Integer (default: 0);
    }
```

```
    for enumerations {
        /*
         * define the enumeration backing type on CommonAPI C++ level for a specific  ↩
            enumeration.
         * If not specified use "ApiDefaultEnumBackingType".
         */
        EnumBackingType : {UInt8, UInt16, UInt32, UInt64, Int8, Int16, Int32, Int64} ( ↩
           optional);
    }
}
```

The enumeration backing type can be set on CommonAPI C++ level not only specific for single enumerations but also generally for the whole interface. Please note that bindings might also have deployment settings which concern the backing type of enumerations.

It is also possible to define timeouts for function calls. Another possibility to set this timeout is to define it in the optional `CallInfo` parameter of the method call.

The settings for instances and providers are not evaluated by the code generator.

See the following example for the usage of the deployment parameters. The Franca specification is:

```
package commonapi.examples

interface Example {

    version { major 0  minor 1 }

    method test {
        in {
            EN x
        }
    }

    enumeration EN {
        DEFAULT
        NEW
    }
}
```

Possible deployment settings could look like:

```
import "CommonAPI_deployment_spec.fdepl"
import "Example.fidl"

define org.genivi.commonapi.core.deployment for interface commonapi.examples.Example {

    DefaultEnumBackingType = UInt8

    method test {
        Timeout = 1
    }

    enumeration EN {
        EnumBackingType = UInt64
    }
}
```

2.6.4 CommonAPI Logging

CommonAPI Runtime informs about internal errors, warnings and other events by means of the CommonAPI Logger. Please refer to the chapter *Write CommonAPI Configuration Files* if you want to know in general how it is possible to get CommonAPI log messages.

CommonAPI supports the DLT standard. To get DLT log messages make sure that DLT is installed when running *CMake*. If *CMake* finds the DLT library the compiler switch `-DUSE_DLT` is set. Additionally set the key `dlt=true` in the logging section of your configuration file.

The DLT application ID is always set to `CAPI`; the context ID can be set by the application (see e.g. example 01):

```
CommonAPI::Runtime::setProperty("LogContext", "E01C");
```

The default value for the *LogContext* is `CAPI`.

2.6.5 CMake

Since version 3 the build system of CommonAPI has been converted from Autotools to CMake (http://www.cmake.org/). A good starting point for writing CMake files for your application are the CMake files which are delivered with the CommonAPI examples in the CommonAPI Tools project.

Here are some additional hints how specific problems related to Franca IDL and CMake can be solved.

Get a CMake variable with all generated files

Let's assume that the code generator has generated all source fules into thedirectory src-gen. The CMake command:

```
FILE(GLOB GEN_SRCS src-gen/*cpp)
```

creates the CMake variable `GEN_SRC` which can be used e.g. like

```
add_executable(${GEN_SRCS} _<other sources>_)
```

for building the execuatble.

Generate code within your CMake file

As described before one step in building your application is the generation of source files from your Franca interface specification. If you do not want to start the code generator manually you have several possibilities to automate that step. The following extract of a CMake file shows how it could be done within a CMake file.

We assume that your fidl files are located in the subdirecory *fidl* of your project directory and the generated output in *src-gen*.

```
set(FIDL_DIR ${PRJ_SRC_DIR}/fidl)
set(GEN_SRC_DIR ${PRJ_SRC_DIR}/src-gen)

set(GEN_COMMAND _<path-to-generator>_/commonapi_generator -dest ${GEN_SRC_DIR} ${FIDL_DIR}/ ↩
    MyFidl.fidl)

add_custom_command(OUTPUT ${SRC_GEN_DIR}/_<generated-source-files>_.cpp
    COMMAND ${GEN_COMMAND}
    DEPENDS ${FIDL_DIR}/MyFidl.fidl
    WORKING_DIRECTORY ${PRJ_SRC_DIR}
    COMMENT "Call CommonAPI code generator."
)
```

Set the `OUTPUT` in the custom command to all generated src files.

Note
For CommonAPI itself it might be possible that there are no cpp files in the output. This is the case if you don't generate the skeletons for the stub and you don't have any complex datatypes in yor fidl file.

2.7 Windows

CommonAPI can also run on Windows platforms. In order to get a running system please refer to the binding specific user guides. To compile CommonAPI for Windows, do the following steps:

- Open the file `CommonAPI.sln` with Visual Studio 2013 (at least Update 4).
- Build the project `CommonAPI`.

3 Basic Features

3.1 New CommonAPI 3 Features

The new CommonAPI version 3 contains a number of bug fixes, some internal modifications and new features. For the user of CommonAPI there are not as many changes as it may seem at first glance. The following sections provide an overview of the most important changes.

3.1.1 Franca And Deployment

CommonAPI 3 supports Franca IDL 0.9.1 and additionally some new deployment parameters (see chapter *CommonAPI Deployment*).

3.1.2 Build System CMake

The build system of CommonAPI has been converted from Autotools to CMake. Please refer to the build instructions of this user guide. For an improved window support also Windows project files are delivered.

3.1.3 Changed Configuration Files

The configuration files have now the ini file format and some additional settings. Please refer to the chapter *Write CommonAPI Configuration Files*.

3.1.4 CommonAPI Logging

CommonAPI 3 provides logging messages, e.g. in DLT format. Refer to chapter *CommonAPI Logging*.

3.1.5 New Code Generator Command-line Parameters

The parameters of the code generator are now significantly extended. The usage of the code generator is now (Linux 32 bit):

```
commonapi-generator-linux-x86 <options> <fidl files>
```

Options	Description
`-h, --help`	Print out options of the code generator
`-d, --dest` <*arg*>	The default output directory
`-dc, --dest-common` <*arg*>	The directory for the common code (relevant for proxy and stub, e.g. type definitions)
`-dp, --dest-proxy` <*arg*>	The directory for proxy code
`-ds, --dest-stub` <*arg*>	The directory for stub code
`-dsk, --dest-skel` <*arg*>	The directory for the skeleton code
`-l, --license` <*arg*>	The file path to the license text that will be added to each generated file
`-ll, --loglevel` <*arg*>	The log level (quiet or verbose)

Options	Description
`-np, --no-proxy` <*arg*>	Switch off generation of proxy code
`-ns, --no-stub` <*arg*>	Switch off generation of stub code
`-pre, --prefix-enum-literal` <*arg*>	The prefix added to all generated enumeration literals
`-sk, --skel` <*arg*>	Generate skeleton code. The optional argument specifies the postfix. Without argument, the postfix is *default*

Please note:

- It is now possible to define different destination directories for proxy and stub code and to switch off the generation of proxy or stub code.
- The CommonAPI 2.x code generator always generated default implementations for the stubs (skeleton). This is not the default behavior anymore. Call the code generator with the option `-sk` without an argument to get the same result as before.
- It is possible to influence the literals of enumerations by setting the option `-pre`. With this option you can add a custom prefix to all generated enumeration literals.

3.1.6 Version In Namespace

Since CommonAPI version 3, the version of the Franca interface becomes part of the C++ namespace. The version is added at the beginning of the namespace path, e.g. for an interface with version 1.0 the modified namespace begins with `v1_0`. Please refer to chapter *Namespaces* for further information.

3.1.7 New Runtime Loading Concept

With CommonAPI 2.x you had to follow three steps to create proxies (or stubs):

- Get a runtime object for your required middleware.
- Create a factory from this runtime object (optionally with a mainloop context).
- Build your proxy using this factory; one argument is the CommonAPI address.

This procedure has been simplified and improved for the dynamic loading of CommonAPI bindings. For the relevant details see chapter *Creating Proxies And Stubs*.

3.1.8 Asynchronous Stubs

For client implementations CommonAPI offers the possibility to call functions (Franca methods or setter/getter functions for attributes) synchronously or asynchronously. In the asynchronous case the reply for the call will come as callback-function call. On stub side CommonAPI required to calculate the return values immediatly (see picture).

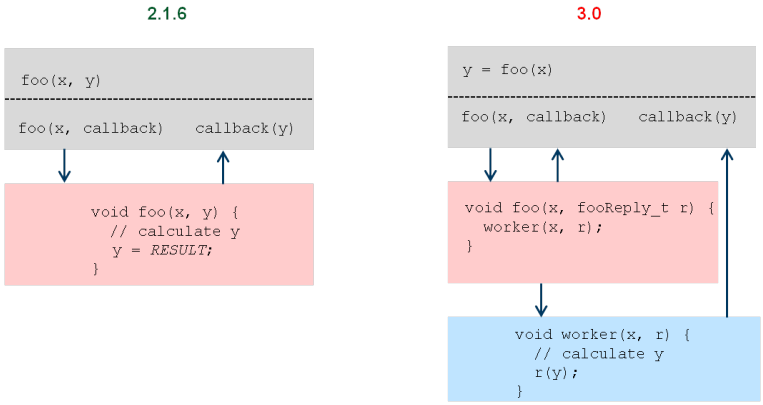


Figure 8: Asynchronous Stubs

But some applications work as kind of intermediate layer and simply pass on function calls to a next application. In this case it was only under relatively high cost possible to pass the function calls to another application and to delegate the calculation of the return values. This is now possible because the return values are now wrapped into a function object which can be passed to another execution path.

3.1.9 CallInfo For Method Calls

Method calls now have an additional, optional parameter which is called `CallInfo`. See e.g. the generated call `sayHello` on proxy side:

```
virtual void sayHello(
    const std::string &_name, CommonAPI::CallStatus &_status,
    std::string &_message,
    const CommonAPI::CallInfo *_info = nullptr
);
```

`CallInfo` is defined as:

```
struct COMMONAPI_EXPORT CallInfo {
    CallInfo()
        : timeout_(DEFAULT_SEND_TIMEOUT_MS), sender_(0) {
    }
    CallInfo(Timeout_t _timeout)
        : timeout_(_timeout), sender_(0) {
    }

    Timeout_t timeout_;
    Sender_t sender_;
};
```

The `timeout_` defines how long the client will wait for the response from the service before it returns with CallStatus `REMOTE_ERROR`.

The attribute `sender_` can be used for passing an application specific sender handle to the method call which can be used in bindings to log the correlation between the sender handle and middleware specific information like serial numbers.

3.2 Creating Proxies And Stubs

The CommonAPI runtime creates proxies/registers stubs at the binding which is either configured for the given proxy/stub instance or is the default binding if no configured binding could be found. The configuration of a binding for a specific instance is done by providing an address mapping for the instance. An address mapping can either be specified using a deployment file to generate the interface specific binding code or by an entry in the binding configuration file. Such entries look like:

```
[_<commonapi address>_]
_<ipc specific address parameter>_ = ...
_<ipc specific address parameter>_ = ...
...
```

Note
CommonAPI does not check for multiple definitions for the same CommonAPI address. In this case, the binding that was registered first, will be used to create the proxy/register the stub.

The default default binding is "dbus". This can be changed by setting the "binding" variable in the "default"-section of the CommonAPI configuration file or by setting the environment variable "COMMONAPI_DEFAULT_BINDING". The environment variable overwrites the setting provided by the configuration file.

Note
A binding that is used as default binding should be able to automatically (without further configuration) translate CommonAPI addresses to the addresses used by the communication mechanism it binds to. Otherwise all used instances must be defined in the deployment.

Bindings are implicitly registered by the binding specific generated code for an interface. This code can either be statically linked or dynamically loaded at runtime. In the first case, the bindings an application uses are statically bound to the application while in the second case the application and their used bindings are independent from each other. The name of a library that needs to be loaded before being able to create a proxy or register a stub for a specific interface instance is determined as follows:

1. It is checked whether the requested action can be done be an already registered factory.
2. If 1. fails, the name of the library that contains the needed code is determined:
   a. If the requested proxy/stub instance is configured, the configured name will be used. Such configurations are done in the "proxy" and "stub" sections of the CommonAPI configuration file and look like: *<commonapi address>* = *<library name>*
   b. If no configuration exists it is checked whether the property "LibraryBase" is set. If yes, the name is set to *lib<LibraryBase>*.
   c. If neither a configuration exists nor the property "LibraryBase" is set, the name will be built from the CommonAPI address as *lib<domain>_<interface>_<instance>*.
3. The runtime tries to load the library with the determined name. If it succeeds, it tries to create the proxy/register the stub again.

3.3 Namespaces

See the example `E01HelloWorld`:

```
package commonapi.examples


interface E01HelloWorld {
    version { major 1 minor 0 }

    // Interface definition here
}
```

The generated code looks as follows:

```
namespace v1_0 {
namespace commonapi {
namespace examples {

class E01HelloWorld {
public:

// Code here

CommonAPI::Version E01HelloWorld::getInterfaceVersion() {
    return CommonAPI::Version(1, 0);
}

} // namespace examples
} // namespace commonapi
} // namespace v1_0
```

The generated code will be generated into the directory `src-gen/v1_0/commonapi/examples`.

The main reason for adding the version at the beginning of the namespace is that we consider the name of the interfaces together with the package path (fully qualified name) as a unit, as it is specified in the Franca file. This name should not be destroyed by an intervening add on.

It is however possible to create project specific namespaces if needed, as the following example shows:

```
#include <iostream>

namespace v1_0 {
namespace commonapi {
namespace examples {

class MySample {
public:
    void print() const { std::cout << "commonapi.examples.v1_0.MySample" << std::endl; }
};

} // namespace examples
} // namespace commonapi
} // namespace v1_0

namespace v1_1 {
namespace commonapi {
namespace examples {

class MySample {
public:
        void print() const { std::cout << "commonapi.examples.v1_1.MySample" << std::endl;  ↩
            }
};

} // namespace examples
} // namespace commonapi
} // namespace v1_1

namespace commonapi {
namespace examples {

namespace V1_0 = v1_0::commonapi::examples;
namespace V1_1 = v1_1::commonapi::examples;

} // namespace examples
} // namespace commonapi
```

```
using namespace commonapi::examples;

int
main(int argc, char **argv) {
    V1_0::MySample aSample;
    V1_1::MySample bSample;

    aSample.print();
    bSample.print();

    return 0;
}
```

3.4 Multithreading and Mainloops

Please refer to example 07 for an example of the CommonAPI mainloop integration with the `glib` mainloop.

4 Examples

4.1 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 as described in the integration guide. Please do not consider only the integration guide in this document but also in the binding specific tutorial.

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:

```
$ cd build
$ cmake -DUSE_INSTALLED_COMMONAPI=ON/OFF ..
$ make
```

Set `USE_INSTALLED_COMMONAPI` to `ON` if you use an installed version of CommonAPI; set it to `OFF` if you use the working copy.

In case you are using the D-Bus binding and you have not installed the patched D-Bus library (*libdbus*) it might be necessary to set a D-Bus specific option:

```
cmake -DUSE_INSTALLED_COMMONAPI=OFF -DUSE_INSTALLED_DBUS=OFF ..
```

Please read also the chapters in the user guide about the configuration files. One possibility to specify the path to the CommonAPI configuration file is to set the environment variable:

```
export COMMONAPI_CONFIG=<path to CommonAPI-Examples>/E01HelloWorld/commonapi4dbus.ini
```

If do not specify any configuration file it is assumed that you want to use the D-Bus binding. Other environment variables you might need are:

- `LD_LIBRARY_PATH` should contain the path to the *libdbus* if you use the D-Bus binding with an uninstalled *libdbus* and should contain the path to your gluecode libraries, e.g. `LD_LIBRARY_PATH=<path to libdbus>/dbus-1.8.20/dbus/.libs:<path to CommonAPI-Tools>/CommonAPI-Examples/E01HelloWorld/build`
- `VSOMEIP_CONFIGURATION_FILE` should contain the path to your vsomeip configuration file if you are using the SOME/IP binding, e.g. `VSOMEIP_CONFIGURATION_FILE=<path to CommonAPI-Tools>/CommonAPI-Examples/E01HelloWorld/vsomeip-local.json`

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 (e.g. /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 (e.g. via *Window→Show View→Make Target*). Set the build command to:

```
cmake -E chdir build/ cmake <options> ..
```

The above described environment variables you could specify in the *Run Configuration* of your executable files (cleint and service).

4.2 Example 01: Hello World

This example contains step-by-step instructions how to create the code for the `Hello World` introductory example of CommonAPI C++. Even this is a CommonAPI tutorial with no references to any special bindings, we give here some hints concerning existing bindings. For a deeper insight please refer to the binding specific tutorials. We assume that you use the Eclipse tool chain and that everything is properly installed as described in the Integration Guide of this tutorial.

The first step in developing a CommonAPI 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 named file ending in *.fidl* in your Eclipse project (in this case *E01HelloWorld.fidl*). It is not relevant where in your project you have placed this file, as 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 *E01HelloWorld.fidl*-file, and type the following lines:

```
package commonapi.examples

interface E01HelloWorld {
    version { major 0 minor 1 }

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

Now, save the *.fidl* file and right click it. As you have installed the CommonAPI code generator and further generators for the bindings, you will see a menu item saying *Common API*, with sub menu items for generating the Common API level code only (*Generate C++ Code*) and for generating the required glue code (e.g. for D-Bus you find *Generate D-Bus C++ Code*).

Now start the code generators. You will find the generated code in the sub-directory `src-gen`.

All files that have a the name of your binding (e.g. *DBus*) in their name are glue code required by the binding and are not relevant while developing your application, they only need to be compiled with your application.

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.

Start by implementing the client 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 <string>
#include <unistd.h>

#include <CommonAPI/CommonAPI.hpp>
#include <v0_1/commonapi/examples/E01HelloWorldProxy.hpp>
```

Please note that you always have to include `CommonAPI.hpp` for accessing the runtime part of CommonAPI and the generated proxy. If your defined interface has a version number then you will find the version in the namespace of your interface class and in the directory structure.

One of the first things each and every CommonAPI application will do is to get a pointer to the runtime object:

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

Please ignore for this introduction the `CommonAPI::Runtime::setProperty` calls (refer to the Integration Guide of this tutorial).

In order to be able to communicate with a specific service, we need a proxy:

```
std::string domain = "local";
std::string instance = "commonapi.examples.HelloWorld";
std::string connection = "client-sample";

std::shared_ptr<E01HelloWorldProxyDefault> myProxy = runtime->buildProxy<E01HelloWorldProxy ↩
    >(domain, instance, connection);
```

The domain and the instance name determine explicitly together with the generated proxy class as template parameter which stub will be addressed by this proxy. The *connection* is an optional argument. This argument allows to group several proxies in a so-called connection. Internally a connection corresponds to one receiver thread if there is no mainloop integration.

With the instantiation of the proxy, the client is set up and ready to use. In this example we wait for the service to be available, then we start issuing calls:

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


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

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

The implementation of 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.hpp>
#include "E01HelloWorldStubImpl.hpp"
```

In the main function of the service one of the first things to do is to get the runtime object. After that we have to instantiate our stub implementation (here `E01HelloWorldStubImpl`) and then to register it:

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

std::string domain = "local";
std::string instance = "commonapi.examples.HelloWorld";
std::string connection = "service-sample";

std::shared_ptr<E01HelloWorldStubImpl> myService = std::make_shared<E01HelloWorldStubImpl ↩
    >();
runtime->registerService(domain, instance, myService, connection);

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

The implementation of the generated stub method `sayHello` looks like:

```
void E01HelloWorldStubImpl::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());
};
```

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

4.3 Example 02: 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 {
    version { major 1 minor 0 }

    struct a1Struct {
```

```
        String s
        a2Struct a2
    }

    struct a2Struct {
        Int32 a
        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 possibilities can be limited (e.g. 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 code generator defines the attribute as private attribute of the stub class. This attribute can be accessed from the stub implementation via getter and setter functions. Additionally the API for the stub implementation provides some callbacks (the following code snippet shows parts of the generated header of the stub class which refer to the attribute x):

```
class E02AttributesStubDefault : public virtual E02AttributesStub {
public:
    /* some other code here */
    virtual const int32_t& getXAttribute();
    virtual const int32_t& getXAttribute(const std::shared_ptr<CommonAPI::ClientId> _client ↩
        );
    virtual void setXAttribute(int32_t _value);
    virtual void setXAttribute(const std::shared_ptr<CommonAPI::ClientId> _client, int32_t  ↩
        _value);

protected:
    /* some other code here */
    virtual bool trySetXAttribute(int32_t _value);
    virtual bool validateXAttributeRequestedValue(const int32_t &_value);
    virtual void onRemoteXAttributeChanged();

private:
    /* some other code here */
    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 <unistd.h>

#include <CommonAPI/CommonAPI.hpp>
#include <v1_0/commonapi/examples/E02AttributesProxy.hpp>

#include "AttributeCacheExtension.hpp"

using namespace v1_0::commonapi::examples;

int main() {

    std::shared_ptr < CommonAPI::Runtime > runtime = CommonAPI::Runtime::get();
    std::string domain = "local";
    std::string instance = "commonapi.examples.Attributes";

    std::shared_ptr<CommonAPI::DefaultAttributeProxyHelper<E02AttributesProxy,
                                                AttributeCacheExtension>::class_t> myProxy =
        runtime->buildProxyWithDefaultAttributeExtension<E02AttributesProxy,
                                                AttributeCacheExtension>(domain, instance);

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

    CommonAPI::CallStatus callStatus;

    int32_t value = 0;
    myProxy->getXAttribute().getValue(callStatus, value);
    std::cout << "Got attribute value: " << value << std::endl;

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

    value = 100;
    std::function<void(const CommonAPI::CallStatus&, int32_t)> fcb = recv_cb;
    myProxy->getXAttribute().setValueAsync(value, fcb);

    while (true) { usleep(1000000);  }
}
```

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.

Of course it is also possible to define a separate callback function with an user-defined name (here recv_cb) as can be seen at the asynchronous set call for the attribute `x`:

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

.... // main method

value = 100;
std::function<void(const CommonAPI::CallStatus&, int32_t)> fcb = recv_cb;
myProxy->getXAttribute().setValueAsync(value, fcb);
```

This example uses a special feature of CommonAPI which is called *Attribute Extension*. This feature is described separately. At this point we do not want to go into more detail. Please see the source code of the example for a deeper insight.

4.4 Example 03: 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 (e.g., from a database). See the following example 3:

```
package commonapi.examples

interface E03Methods {

    version { major 1 minor 2 }

    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 service 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:

```
void E03MethodsStubImpl::foo(const std::shared_ptr<CommonAPI::ClientId> _client,
                int32_t _x1,
                std::string _x2,
                fooReply_t _reply) {


    E03Methods::fooError methodError = (E03Methods::fooError)
```

```
    E03Methods::stdErrorTypeEnum::MY_FAULT;
    int32_t y1 = 42;
    std::string y2 = "xyz";
    _reply(methodError, y1, y2);
}
```

The input parameters are available as values, the output parameter are wrapped in a generated reply object with the type fooReply_t. This is slightly different to earlier versions of CommonAPI where the return values were passed as references.

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);
}
```

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 <unistd.h>

#include <CommonAPI/CommonAPI.hpp>
#include <v1_2/commonapi/examples/E03MethodsProxy.hpp>

using namespace v1_2::commonapi::examples;

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

    std::cout << "Result of asynchronous call of foo: " << std::endl;

    std::cout << "   callStatus: "
         << ((callStatus == CommonAPI::CallStatus::SUCCESS) ? "SUCCESS" : "NO_SUCCESS")
         << std::endl;

    std::cout << "   error: "
        << ((methodError.stdErrorTypeEnum ==
            E03Methods::stdErrorTypeEnum::NO_FAULT) ? "NO_FAULT" : "MY_FAULT")
        << std::endl;

    std::cout << "   Output values: y1 = " << y1 << ", y2 = " << y2 << std::endl;
}

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 timeout of function calls can be set by the optional `CallInfo` argument on client side.

4.5 Example 04: PhoneBook

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

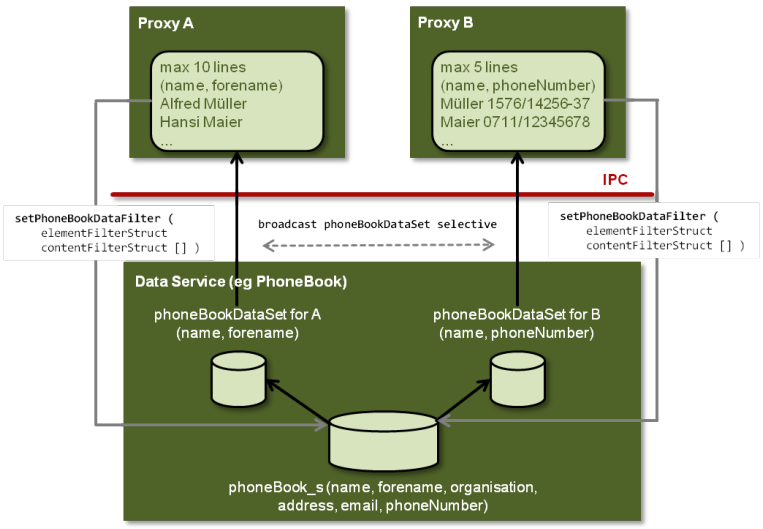
- explicit named arrays and inline arrays
- selective broadcasts
- polymorphic structs

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 several clients via a data filter mechanism and selective broadcasts. As example of a central data array a phone book is selected; the following picture shows the basic content of the example.



The Franca IDL specification is:

```
package commonapi.examples

interface E04PhoneBook {
    version { major 0 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 client ID (`ClientId`); the implementation of client ID class allows the usage of client ID 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 client ID 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).

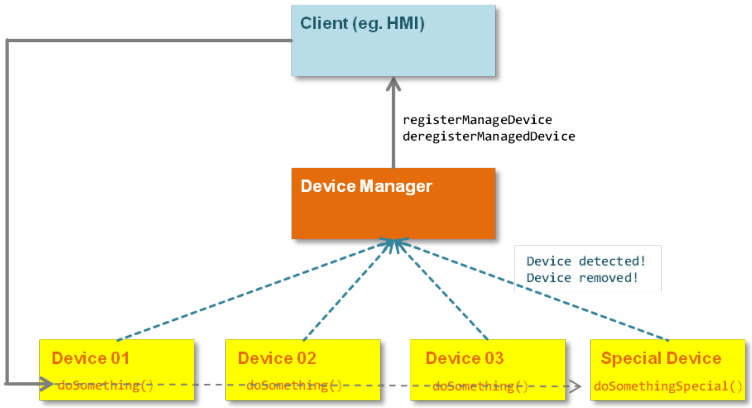
```
void E04PhoneBookStubImpl::setPhoneBookDataFilter(
            const std::shared_ptr<CommonAPI::ClientId> _client,
            E04PhoneBook::elementFilterStruct _elementFilter,
            std::vector<E04PhoneBook::contentFilterStruct> _contentFilter,
            setPhoneBookDataFilterReply_t _reply) {

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

    std::vector < E04PhoneBook::phoneBookDataElementMap > lPhoneBookDataSet;

    phoneBookClientData.erase(_client);

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

        E04PhoneBook::phoneBookDataElementMap lPhoneBookDataElement;

        if (_elementFilter.getAddName()) {
            std::shared_ptr<E04PhoneBook::phoneBookDataElementString> name =
                std::make_shared<E04PhoneBook::phoneBookDataElementString>();
            name->setContent(it0->getName());
            lPhoneBookDataElement[E04PhoneBook::phoneBookDataElementEnum::NAME] = name;
        }

        /*  ... Similar for all other elements */

        lPhoneBookDataSet.push_back(lPhoneBookDataElement);
    }

    phoneBookClientData[_client] = lPhoneBookDataSet;

    const std::shared_ptr<CommonAPI::ClientIdList> receivers(new CommonAPI::ClientIdList);
    receivers->insert(_client);

    firePhoneBookDataSetSelective(lPhoneBookDataSet, receivers);

    receivers->erase(_client);

    _reply();
}
```

On client side we create two proxies which shall set different filters and get different data sets. With CommonAPI 2.x we needed two different factories for these two proxies; this can be achieved by now by creating a new Connection ID for the second proxy. 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::get();

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

    std::shared_ptr < E04PhoneBookProxyDefault > myProxyA =
        runtime->buildProxy < E04PhoneBookProxy > (domain, instance);

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

    const CommonAPI::ConnectionId_t otherConnectionId = "42";

    std::shared_ptr < E04PhoneBookProxyDefault > myProxyB =
        runtime->buildProxy < E04PhoneBookProxy > (domain, instance, otherConnectionId);

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

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

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

    CommonAPI::CallStatus myCallStatus;
    std::vector<E04PhoneBook::phoneBookStruct> myValue;

    myProxyA->getPhoneBookAttribute().getValue(myCallStatus, myValue);
    printPhoneBook (myValue);

    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);

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

4.6 Example 05: Managed

So far we have looked at software systems, which consisted of services and users of these services, 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 available for usage in the system or not. The central manager handles all administrative 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 <unistd.h>
#include <iostream>

#include <CommonAPI/CommonAPI.hpp>
#include <v1_0/commonapi/examples/E05ManagerProxy.hpp>

using namespace v1_0::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::get();

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

    std::shared_ptr<E05ManagerProxyDefault> myProxy =
        runtime->buildProxy<E05ManagerProxy>(domain, instance);

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

    CommonAPI::ProxyManager::InstanceAvailabilityStatusChangedEvent& deviceEvent =
```

```
        myProxy->getProxyManagerE05Device().getInstanceAvailabilityStatusChangedEvent();

    /* special device analogously */

    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;
}
```

4.7 Example 06: 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 0 minor 0 }

    attribute CommonTypes.SettingsUnion u
    attribute CommonTypes.SettingsStruct x
}

typeCollection CommonTypes {
    version { major 0 minor 0 }

    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 = "∃y ∀x \ensuremath{\lnot}(x ≺ y)";

        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));
        }

    } else {
        std::cout << "Type number " << n << " not possible." << std::endl;
    }
}
```

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 <unistd.h>
#include <iostream>

#include <CommonAPI/CommonAPI.hpp>

#include "../src-gen/commonapi/examples/CommonTypes.hpp"
#include "../src-gen/commonapi/examples/E06UnionsProxy.hpp"
#include "typeUtils.hpp"

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::get();

    const std::string &domain = "local";
    const std::string &instance = "commonapi.examples.Unions";
    std::shared_ptr<E06UnionsProxyDefault> myProxy =
        runtime->buildProxy<E06UnionsProxy>(domain, instance);

    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):

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

```
#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;
        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.

4.8 Example 07: Mainloop

This example shows how CommonAPI can be integrated with an external main loop. For demonstration purposes the `glib` main loop is used.

CommonAPI does not contain a main loop implementation itself; just a demo implementation is provided for test purposes (please have a look to the verification tests in the CommonAPI tools project to find out how the demo main loop is used).

5 Contributor's Guide

5.1 Preliminary Remarks

You can contribute as little or as much as you like: bug reports, patches or documentation are all equally appreciated:

- Create bug reports for CommonAPI and bindings at http://bugs.genivi.org, Product *Common API for IPC*
- For the creation of patches: see chapter *Contribution of Code* below
- Provide changes in the existing documentation or new documentations as patch or *asciidoc* document

5.2 Build Tests and Documentation

There are additional make targets for contributors and developers:

`make dist`	Generates a packed tarball (*.tar.bz) from your current branch in your git repository.
`make doxygen-doc`	Generates a packed tarball (*.tar.bz) from your current branch in your git repository.

The CommonAPI documentation is written with *asciidoc* (http://www.methods.co.nz/asciidoc/). You will find the source of this user guide and the specification in the *docx* directory of the CommonAPI-Tools project. Please make sure that *asciidoc* is installed before you try to create the pdf- or html-files yourself. The documentation can be created by starting the Makefile in the *docx* directory. The doxygen output can be obtained by the make target *doxygen-doc*.

Please note that CommonAPI itself does not contain any unit tests (only the source code of CommonAPI bindings might be delivered with unit tests). CommonAPI provides so-called verification tests which can be found in the ComonAPI Tools repository and there in the project *org.genivi.commonapi.core.verification*.

The verification tests can be used to test features and the correct behavior of the CommonAPI framework together with the binding. These tests shall guarantee that CommonAPI and binding implement the CommonAPI specification and are stable.

The verification tests are implemented by using *googletest* (http://code.google.com/p/googletest). For compiling and running the tests *googletest* must be built and the configuration script location must be available in the environment variable `GTEST_CONFIG`.

The first step is to create the glue code library. The term "glue code" refers to a binary that contains the binding specific generated code from the provided Franca files which you can find in *org.genivi.commonapi.core.verification/fidl*. The glue code library must contain the generated binding specific proxy and stub files. Please follow the instructions in your binding code for creating the glue code library.

Now create a *build* directory in the verification project and start CMake. Call CMake with the following parameters:

`-DUSE_INSTALLED_COMMONAPI`	OFF/ON
`-DBINDING_NAME`	DBus/SomeIP/...
`-DCMAKE_BINDING_NAME`	CommonAPI-DBus/CommonAPI-SomeIP/...
`-DCMAKE_BINDING_PATH`	Path to the binding library
`-DCMAKE_GLUECODE_NAME`	Name of the glue code library

DCMAKE_GLUECODE_PATH	Path to the glue code library
-DBINDING_EXTRA	Path to additional libraries which are needed by the linker (e.g. *libdbus*)
-DCOMMONAPI_CMAKE_INSTALL_PATH	Path to the CommonAPI library
-DCOMMONAPI_DBUS_TOOL_GENERATOR	Code generator executable with path

After that start *make check* to build and run the verification tests.

5.3 Formatting Code

Use the Eclipse internal formatter *K&R [built-in]* for formatting the code.

5.4 Contribution of Code

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