FAW HS7

IVI System Design

CommonAPI C++

Released on the: 2016.02.15

Version Number: 0.100

Neusoft Automotives CoC

**Version History**

| No. | Version | Section | Brief Description | Date | Responsible |
| --- | --- | --- | --- | --- | --- |
| 1 | 0.100 | All | Initial Create | 2016-02-15 | - |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

Content

[1 CommonAPI C++ 4](#_Toc445364897)

[1.1 Software Framework 4](#_Toc445364898)

[1.2 Communication Framework 6](#_Toc445364899)

[1.3 User Manual 7](#_Toc445364900)

[1.3.1 Configuration 7](#_Toc445364901)

[1.3.2 Interface Definition 7](#_Toc445364902)

[1.3.3 Server 10](#_Toc445364903)

[1.3.4 Client 10](#_Toc445364904)

[1.3.5 Example 10](#_Toc445364905)

[1.4 Performance 12](#_Toc445364906)

[1.5 System Limition 12](#_Toc445364907)

[2 Refrences 14](#_Toc445364908)

# CommonAPI C++

本节用于描述FAW HS7 IVI模型中的进程间通讯机制——CommonAPI C++。

## Software Framework

FAW HS7 IVI模型的进程间通讯将会通过GENIVI的CommonAPI实现，因此，首先需要介绍一下GENIVI IPC CommonAPI C++的软件模型。

Application

CommonAPI Core

*Generated Codes*

Code

Generator

Core

Franca IDL

Specification \*.fidl

Common

API

Binding

*Generated Codes*

Code

Generator

Binding

Franca Deployment

Specification \*.fdepl

IPC Stack

CommonAPI

Core

*Runtime*

CommonAPI Binding

*Runtime*

图 5-3-1 CommonAPI C++ Framework

如上图所示，CommonAPI C++在通讯层分为Core和Binding两个部分。其中Core部分负责和Application交互而Binding部分负责和系统底层的IPC服务栈交互。

其中的Binding即IPC的具体实现方式。目前GENIVI支持的Binding分为两种，分别是SOME/IP和DBus。

在本系统中，使用DBus作为底层Binding。

如果需要使用CommonAPI C++接口实现进程间通讯，则用户需要完成如下工作：

* 编写Core层和Binding层的接口说明文件（\*.fidl用于Core和DBus；\*.fdepl用于SOME/IP）。
* 使用Code Generator生成客户端和服务端的C++代码。命令行格式类似如下：
  + $(COMMONAPI\_CORE\_GEN\_PATH)/ commonapi-generator-linux-x86\_64 source/fidl/Source.fidl -d src-gen/core –sk
  + $(COMMONAPI\_DBUS\_GEN\_PATH) /commonapi-dbus-generator-linux-x86\_64 source/fidl/Source.fidl -d src-gen/dbus
* 参照骨架文件（skeleton）实现服务端的接口。
* 在客户端创建代理（Proxy）并借助代理调用各个接口。

以GENIVI提供的演示程序（[E01HelloWorld](http://git.projects.genivi.org/?p=ipc/common-api-tools.git;a=tree;f=CommonAPI-Examples/E01HelloWorld)）为例，下面给出了Generated Codes：

|  |
| --- |
| **src-gen/**  |-- **core**  | `-- **v0\_1**  | `-- **commonapi**  | `-- **examples**  | |-- E01HelloWorld.hpp  | |-- E01HelloWorldProxyBase.hpp  | |-- E01HelloWorldProxy.hpp  | |-- E01HelloWorldStubDefault.cpp  | |-- E01HelloWorldStubDefault.hpp  | `-- E01HelloWorldStub.hpp  |-- **dbus**  | `-- **v0\_1**  | `-- **commonapi**  | `-- **examples**  | |-- E01HelloWorldDBusDeployment.cpp  | |-- E01HelloWorldDBusDeployment.hpp  | |-- E01HelloWorldDBusProxy.cpp  | |-- E01HelloWorldDBusProxy.hpp  | |-- E01HelloWorldDBusStubAdapter.cpp  | `-- E01HelloWorldDBusStubAdapter.hpp  `-- **someip**  `-- **v0\_1**  `-- **commonapi**  `-- **examples**  |-- E01HelloWorldSomeIPDeployment.cpp  |-- E01HelloWorldSomeIPDeployment.hpp  |-- E01HelloWorldSomeIPProxy.cpp  |-- E01HelloWorldSomeIPProxy.hpp  |-- E01HelloWorldSomeIPStubAdapter.cpp  `-- E01HelloWorldSomeIPStubAdapter.hpp |

src-gen/core目录下的代码（即Core Codes）用于和Application交互，而src-gen/dbus和src-gen/someip目录下的代码（即Binding Codes）用于和IPC Stack交互。

其中，“E01HelloWorldStubDefault.cpp”和“E01HelloWorldStubDefault.hpp”即前文所提到的骨架文件（skeleton）。

用户需要实现一个“E01HelloWorldStubDefault”的子类（如：E01HelloWorldStubImpl），并在其中封装实际的接口功能。

通常的使用中，CommonAPI工程编译之后会生成三个组件：

* 客户端可执行文件
* 服务端可执行文件
* Binding库文件。这个文件不与上述两个可执行文件进行Link，而是在运行时动态导入。

## Communication Framework

CommonAPI C++的IPC接口使用的是通用的“服务端、客户端”模式。

对于客户端主动发起的通讯，采取的是方法（Methods）调用的方式来实现。

Service

Client

**Methods**

Method分为两种类型：

* 标准的带有返回值的Method（Basic Method）
* 不需要返回值的Method（Fire-and-forgot Method）

服务端主动发起的通讯通过广播（Broadcasts）的方式实现。

Service

Client1

Client2

**Broadcasts**

需要注意的是，CommonAPI C++提供的广播模式也有两种：

* 标准的广播方式（Basic Broadcasts），即广播内容发送给所有的客户端
* 选择性广播方式（Selective Broadcasts），即广播内容可以发送给某一个或一组客户端。此种情况下，通常是由服务端提供类似Registor的Method用于客户端注册。然后服务端根据注册情况将广播发送给不同的客户端。（参照 [E04PhoneBook](http://git.projects.genivi.org/?p=ipc/common-api-tools.git;a=tree;f=CommonAPI-Examples/E04PhoneBook) ）

## User Manual

### Configuration

CommonAPI在运行时，默认会寻找一个配置文件。配置文件的查找顺寻如下：

* 环境变量COMMONAPI\_CONFIG指向的位置
* /etc/commonapi.ini
* 当前工作路径下的commonapi.ini

配置文件内会设置一些标准参数：

|  |
| --- |
| **[default]**  binding=dbus  **[logging]**  console = true  file = ./mylog.log  dlt = true  level = verbose |

其中，binding类型是关键参数。而“Logging”下的参数由于Genivi的编码错误，并不生效。

注意：如果配置文件未找到，则binding的默认值就是dbus。

### Interface Definition

接口说明文件基于标准的Franca协议。下面给出一个例子说明一下常见的表达方法：

|  |
| --- |
| package commonapi.examples  interface HelloWorld {  version { major 0 minor 1 }  <\*\* @description : 这是一个标准的函数接口，拥有一个输入参数，一个输出参数 \*\*>  method sayHello {  in {  String name  }  out {  String message  }  }  <\*\* @description : fireAndForget的函数，只有一个输入参数 \*\*>  method tell fireAndForget {  in {  String name  }  }  <\*\* @description : selective的广播，只有一个输出参数。注意，这里的输出是针对服务端来说的，也就是说，从服务端输出到客户端。从通常意义来讲，这是个输入参数。下同 \*\*>  broadcast reply selective{  out {  String name  }  }  <\*\* @description : 一个标准的广播接口，带有一个参数 \*\*>  broadcast yell {  out {  String name  }  }  <\*\* @description : 定义一个私有成员变量和一组对应的Get（客户端）/ Set（服务端）函数\*\*>  attribute Int32 x  <\*\* @description : 可以自定义数据类型 \*\*>  typedef MyTypedef is Int32  <\*\* @description : 枚举类型 \*\*>  enumeration someEnum {  SOME\_ENUM\_A  SOME\_ENUM\_B  }  <\*\* @description : 结构体 \*\*>  struct someStruct {  String name  someEnum some  someMap aMap  }  <\*\* @description : 联合体 \*\*>  union Union1 {  UInt16 e1  Boolean e2  String e3  }  <\*\* @description : MAP \*\*>  map someMap {  someEnum to MyTypedef  }  <\*\* @description : 一维数组 \*\*>  array ExampleArrayRow of UInt8  <\*\* @description : 二维数组 \*\*>  array ExampleArray of ExampleArrayRow  } |

常用数据类型（没有列出全部的数据类型）：

|  |  |
| --- | --- |
| Type name | Description |
| UInt8 | unsigned 8-bit integer (range 0..255) |
| Int8 | signed 8-bit integer (range -128..127) |
| UInt16 | unsigned 16-bit integer (range 0..65535) |
| Int16 | signed 16-bit integer (range -32768..32767) |
| UInt32 | unsigned 32-bit integer (range 0..4294967295) |
| Int32 | signed 32-bit integer (range -2147483648..2147483647) |
| UInt64 | unsigned 64-bit integer |
| Int64 | signed 64-bit integer |
| Boolean | boolean value, which can take one of two values: false or true. |
| Float | foating point number (4 bytes, range +/- 3.4e +/- 38, ~7 digits) |
| Double | double precision foating point number (8 bytes, range +/- 1.7e +/- 308,~15 digits) |
| String | character string |
| ByteBuffer | buffer of bytes (aka BLOB) |

更多的信息可以参考《附录：FrancaUserGuide-0.9.2.0.pdf》。这里不做过多的描述。

### Server

* 设置运行参数。目前版本的DLT只存在两个有效参数：
  + LogContext：如果使用DLT作为LOG输出手段，则本参数生效。本参数用于设置DLT的LOG输出时使用的Context值，是一个最大4字节的字符串。如果不进行设置，默认值是“CAPI”。
  + LibraryBase：用于设置binding-library的名称。例如设置本参数为Abc并指定Binding类型为“dbus”，则CommonAPI自动寻找一个叫做“libAbc-dbus.so”的库文件进行载入。
* 获取CommonAPI的运行实体
* 注册服务器

### Client

* 设置运行参数。目前版本的DLT只存在两个有效参数：
  + LogContext：如果使用DLT作为LOG输出手段，则本参数生效。本参数用于设置DLT的LOG输出时使用的Context值，是一个最大4字节的字符串。如果不进行设置，默认值是“CAPI”。
  + LibraryBase：用于设置binding-library的名称。例如设置本参数为Abc并指定Binding类型为“dbus”，则CommonAPI自动寻找一个叫做“libAbc-dbus.so”的库文件进行载入。
* 获取CommonAPI的运行实体
* 构建代理（Build Proxy）。后续的所有操作（如调用Method）都通过代理进行。

### Example

服务端示例代码：

|  |
| --- |
| int server\_main()  {  CommonAPI::Runtime::setProperty("LogContext", "E01S");  CommonAPI::Runtime::setProperty("LibraryBase", "E01HelloWorld");  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>();  bool successfullyRegistered  = runtime->registerService(domain, instance, myService, connection);  **return** 0;  } |

其中：

* damain和instance：用于配合配置文件进行库文件查找。如果不需要配置文件的方案，则可以直接使用空字符串（"”）。
* connection：一个名字，任意字符串皆可。

客户端示例代码：

|  |
| --- |
| int client\_main()  {  CommonAPI::Runtime::setProperty("LogContext", "E01C");  CommonAPI::Runtime::setProperty("LibraryBase", "E01HelloWorld");  std::shared\_ptr < CommonAPI::Runtime > runtime = CommonAPI::Runtime::get();  std::string domain = "local";  std::string instance = "commonapi.examples.HelloWorld";  std::string connection = "client-sample";  std::shared\_ptr<E01HelloWorldProxy<>> myProxy  = runtime->buildProxy<E01HelloWorldProxy>(domain,instance,connection);  **return** 0;  } |

配置文件示例：

|  |
| --- |
| **[default]**  binding=dbus  **[logging]**  console = true  file = ./mylog.log  dlt = true  level = verbose  **[proxy]**  local:commonapi.examples.E01HelloWorld:commonapi.examples.HelloWorld=/usr/lib/libE01HelloWorld-dbus.so  **[stub]**  local:commonapi.examples.E01HelloWorld:commonapi.examples.HelloWorld=/usr/lib/libE01HelloWorld-dbus.so |

注意：

* proxy（即客户端）和stub（即服务端）的内容是用来查找库文件的，作用和“LibraryBase”参数相同。
* 等号左边的字符串格式为“<domain>:<interface>:< instance >”。其中domain和instance对应代码中registerService和buildProxy的参数，而interface对应fidl文件。
* CommonAPI优先查找配置文件，如果配置文件内没有配置proxy或stub项目，则“LibraryBase”生效。
* 在寻找库文件时，基于配置文件的方案要求提前做成配置文件，灵活性不足。本次模型不使用。

## Performance

（T.B.D）

|  |  |  |  |
| --- | --- | --- | --- |
| 数据量 | 方式 | 传输耗时 | 备注 |
| 1KBytes | Methos |  |  |
| 1KBytes | Braodcast |  |  |
| 5KBytes | Methos |  |  |
| 5KBytes | Braodcast |  |  |
|  |  |  |  |

## System Limition

* 惊群效应

DBUS广播存在惊群效应。当多个客户端连接到DBUS上时，任意一个服务端发送广播，都会导致所有客户端（无论是否与这个服务有关）都被系统激活。如果服务端连续快速多次发送广播，就会导致系统资源被大量占用。

虽然Common API引入Selective Broadcast的概念，但也仅仅是能够保证客户端的Callback不会被调用，而不能保证客户端不被唤醒。

# Refrences

CommonAPI Tutorial

-- <http://docs.projects.genivi.org/ipc.common-api-tools/2.1.4/Tutorial.html>

CommonAPI C++ User Guide

-- <http://docs.projects.genivi.org/ipc.common-api-tools/3.1.3/html/CommonAPICppUserGuide.html>

# Review Record

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| NO | Ver | Question | Staff | Answer | Staff | Status | Comments |
| 1 | 0.100 | 无 |  |  |  |  |  |
| 2 |  |  |  |  |  |  |  |