Mapping between the Model Definition Files and C++ source code

Release 1.8.0

Software and Controls Group

Jul 27, 2020

Contents

CHAPTER

ONE

C++ SOURCE CODE FILE TREE STRUCTURE

The C++ generated source files are located under the `<MODULE_DIR>/src/cpp` directory, which has the following structure:

```
<MODULE_DIR>/
   |-- ...
   |-- src/
         |-- ...
         |-- cpp/
               |-- build/
               |-- include/
               |-- <pk_1_pkg>/
               |-- ...
               |-- <pk_n_pkg>/
               |
               |-- Makefile
```

Makefile The `` `Makefile` `` file contains the directives to include the system wide make rules that are defined in `$GMT_GLOBAL`. Users can modify it to add the specific Makefile definitions that are needed for the module, but the recommended way is to use the `` `module.mk` `` file of each package.

build/ The `` `build/` `` directory is the place where the compilation products will be generated

include/ The `` `include/` `` directory contains the generated include files which must be part of the module external interface. In general, the contents of the this directory are the include files for the module-defined data types. In addition, in the `` `include/`` `` directory there is one convenience header file (`` `<module>_port_types.h` ``) that contains the includes to all of the module headers.

<pk_i_pkg>/ One directory is created for each package defined in the DCS definition file of the model. The name of the directory is exactly the same name that was given to the package *(note: the suffix of the name should be '_pkg'; see the Model Specification Guide document for more details)*.

The structure inside the package directories is as follows:

```
<pk_i_pkg>/
    |
    |-- module.mk
    |
    |-- <comp_1>/
    |-- ...
    |-- <comp_n>/
    |
    |-- <a_1>_app.cpp
    |-- ...
    |-- <a_n>_app.cpp
```

<comp_i>/ For each component there is a directory where all its source files are placed. The name of the directory is the same as the component. See Section 3 for more details.

<a_i>_app.cpp For each application defined in the model for the current package there is a C++ file which contains the source of the application.

module.mk The `module.mk` file contains all the directives that are needed to compile and link the current package. See Section 1.1 for more details.

1.1 Makefiles

As seen above, there is one `Makefile` in the `src/cpp/` directory of each module, which only the system level Makefile rules. This Makefile should not be modified by the user, unless necessary.

In each package directory there is a `module.mk` file. This file is included by *make* when the package is built, and it is the place where the user must add the needed compiler and linker directives to build the module.

In general, the set of Makefile rules defined globally in the SDK are sufficient to build any package, so the user must not add any rule. However, the libraries used by each package are not known by the make system, and therefore the user must specify them. In the auto-generated version of the `module.mk` files there is an example of such directives.

The user must specify the compiler and linker directives in the `MOD_BUILD_CFLAGS`, `MOD_BUILD_CXXFLAGS`, `MOD_BUILD_LDFLAGS`, `MOD_SHRLIBS_CFLAGS`, `MOD_SHRLIBS_CXXFLAGS` and `MOD_SHRLIBS_LDFLAGS` macros, in the `module.mk` file of each package.

CHAPTER

TWO

DATA TYPES MAPPING

2.1 Primitive types

The mapping from the model primitive types to the corresponding C++ types can be found in the following table

Model Type	C++ type
Integer	int
String	std::string
Boolean	bool
Date	struct tm
TimeStamp	struct timeval
TimeInterval	struct timeval
void	void
bool	bool
byte	uint8_t
char	char
uchar	unsigned char
int	int
int8	int8_t
int16	int16_t
int32	int32_t
int64	int64_t
uint	unsigned int
uint8	uint8_t
uint16	uint16_t
uint32	uint32_t
uint64	uint64_t
float	float
float32	float
float64	double
float128	long double
string	std::string
complex	std::complex<float>
complex64	std::complex<double>
complex128	std::complex<long double>
TimeValue_ns	struct timespec
TimeValue_us	struct timeval
TimeValue_Date	struct tm

2.2 Struct types

The data types defined in the model files as `StructType` are mapped to C++ struct types. The convention in the model files syntax is to define the types in snake_case, but in C++ they are mapped in CamelCase. In addition, as the user-defined types are shared in the gmt namespace (because types are part of the interface between modules), the convention is to prefix the name of the module to the type name.

The C++ definition is generated to a header file inside the `include/` directory, with its name equal to the user-defined type name, in CamelCase.

For example, one structured type defined in the model as

```
StructType "mymod_my_struct",
    desc: "Struct type example"
    elements:
        data_field1: {type: "String", desc: "This is field 1"}
        data_field2: {type: "Integer", desc: "This is field 2"}
        data_field3: {type: "mymod_my_other_struct", desc: "This is field 3"}
        data_field4: {type: "float[4]", desc: "This is field 4"}
        data_field5: {type: "uint16[]", desc: "This is field 5"}
```

will have its C++ representation in the `include/my_struct.h` file, with the following contents:

```
#ifndef _mymod_my_struct_h_
#define _mymod_my_struct_h_

#include <msgpack.hpp>
#include <string>
#include <array>
#include <vector>

struct MyModMyStruct
{
    std::string           data_field1;    // This is field 1
    int                   data_field2;    // This is field 2
    MyModMyOtherStruct    data_field3;    // This is field 3
    std::array<float,4>   data_field4;    // This is field 4
    std::vector<unit16_t> data_field5;    // This is field 5

    MSGPACK_DEFINE_MAP(data_field1, data_field2, data_field3, data_field4, data_
↪field5)
};

#endif // _mymod_my_struct_h_
```

As it can be seen in the example, the types of the fields can either be primitive types, user-defined types and arrays or sequences.

The `MSGPACK_DEFINE_MAP` directive allows the C++ instances of this type to be serialized automatically by the msgpack library.

2.3 Enumerated types

The `Enum` types defined in the DCS model files are mapped to C++ as enum classes. The labels of the C++ enum class will be the literals defined in the Enum type in the model, and in the same order.

A header file will be generated for each `Enum` type. These headers will be named after the type name, and they will be placed in the `include/` directory.

As an example, if we have the following enumerated type in the `<module_name>_dcs_types.coffee` model file,

```
Enum "mymod_my_enum_type",
desc: "An enumerated type"
literals:
    THE_FIRST_LABEL:     {desc: "First label of the enumerate"}
    THE_SECOND_LABEL:    {desc: "First label of the enumerate"}
    THE_THIRD_LABEL:     {desc: "Third label of the enumerate"}
```

then the generated C++ file will be `include/my_enum_type.h`, and its contents will be:

```
#ifndef _mymod_my_enum_type_
#define _mymod_my_enum_type_

#include <msgpack.hpp>
#include <ocs_core_fwk/GmtMsgpackAdaptors.h>

enum class MyModMyEnumType : std::uint8_t { THE_FIRST_LABEL=0,
                                            THE_SECOND_LABEL=1,
                                            THE_THIRD_LABEL=2};

GMT_ADD_ENUM(MyModMyEnumType, "THE_FIRST_LABEL", "THE_SECOND_LABEL", "THE_THIRD_LABEL
↪")

#endif   // _mymod_my_enum_type_
```

As in the Struct Type mapping, the `GMT_ADD_ENUM` directive allows the variables of this type to be automatically serialized.

CHAPTER

THREE

COMPONENT MAPPING

Each Component has its own directory in the file tree of the package it belongs to, as shown in Section 1. The Component directory will have the following contents:

```
<pk_i_pkg>/
   |-- ...
   |-- <component_name>
         |-- <ComponentName>.h
         |-- <ComponentName>.cpp
         |-- <ComponentName>Base.h
         |-- <ComponentName>Base.cpp
         |-- <ComponentName>Setup.cpp
         |-- <component_name>_app.cpp
```

The name of the Component directory and the prefix of all the generated source files inside it is equal to the Component name.

3.1 Component classes

There are two classes per each Component, plus a struct that holds the configuration:

- **<ComponentName>Base**: This class contains the communications infrastructure for the Component, as well as the declarations of all the member variables related to the Component features (properties, inputs, outputs, etc.). It is fully autogenerated by the `gds gen` command and it is not supposed to be modified by the user. Any modification will be overwritten by `gds`.
- **<ComponentName>**: This is a class derived from `<ComponentName>Base` that will have the behavior for the Component, specifically the step function and the custom initialization. The `gds gen` command will provide an initial skeleton of the class, but then the user must modify it to add the Component functionality. Further executions of `gds gen` will never overwrite its contents.
- **<ComponentName>Setup**: This is a struct that will be used to read the configuration from the config files.

3.1.1 Component base class files

The C++ class definition of the base class for the component is located in the `` `<ComponentName>Base.h` `` file. The name of the class is set as the name defined in the model file, but in CamelCase and with the Base suffix. For example, a component named `` `my_component` `` in the model files would have a base class named `` `MyComponentBase` ``.

The generated class will inherit from the C++ version of the superclasses listed in the `` `extends` `` list of the component model. In the following table there is the list of the mapping between the most common model superclasses and the corresponding C++ base classes:

Model class	extends item	C++ class
Component	BaseComponent	Component
Controller	BaseController	BaseController
Supervisor	BaseController	BaseController
Adapter	HwAdapter	HwAdapter
Adapter	EthercatAdapter	EthercatAdapter

As an example, if we have the following component in the model definition:

```
Component          'my_component',
    info:           'My Component'
    desc:           'This is an example component'
    extends:        ['BaseComponent']
    abstract:       false
    uses:           ["ocs_core_fwk", "ocs_ctrl_fwk"]

    state_vars:
        my_state1:
            desc:               'One State Var'
            type:               'my_custom_type'
            max_rate:           1000
            is_controllable:    true

    inputs:
        my_input_port1:
            desc:           'One input port'
            type:           'Integer'
            max_rate:       1000

    outputs:
        my_output_port1:
            desc:           'One output port'
            type:           'float64'
            max_rate:       1000

    properties:
        my_prop1:
            desc:     'One property'
            type:     'float32'
            default:  30.0

    faults:
        f0:
            default: 'NOT_ACTIVE'
            type: "FaultState"
            parent: ''
            kind: 'or'
        f1:
            default: 'NOT_ACTIVE'
            type: "FaultState"
            parent: 'f0'
            kind: 'primary'

    alarms:
        a0:
            default: 'NORM'
            type: "AlarmState"
```

```
            parent: ''
            kind: 'or'
        a1:
            default: 'NORM'
            type: "AlarmState"
            parent: 'a0'
            kind: 'primary'
```

then the generated C++ class would be:

```
#ifndef _MyComponentBase_h_
#define _MyComponentBase_h_

// GMT AUTO GENERATED CODE

#include <ocs_core_fwk.h>
#include <ocs_ctrl_fwk.h>
#include "../../include/isample_dcs_port_types.h"

class MyComponentSetup;

namespace gmt
{

class MyComponentBase : public BaseComponent
{
    public:
        MyComponentBase(
                        const  std::string& comp_uri,
                        const  std::string& comp_name,
                        const  std::string& comp_host,
                        int    comp_port,
                        const  std::string& comp_acl,
                        double comp_scan_rate = 1.0,
                        int comp_prio = GMT_THREAD_DEFAULT_PRIO,
                        int comp_stack_size = GMT_DEFAULT_STACK_SIZE);

        virtual ~MyComponentBase();

    protected:

        typedef MyComponentSetup Setup;
        typedef BaseComponent Base;

        /**
        * Creates the state of the Component, i.e., state variables,
        * inputs, outputs, properties, alarms and faults
        * Overriden from the Component class
        */
        void create_state() override;

        /**
        * Uses the given Component::Setup parameter to configure all the
        * Component interface features (state vars, inputs, outputs, properties, ...)
        * Overriden from the Component class
        */
        void setup_state (const ComponentSetup& conf) override;
```

```
        /**
        * Configure the object from a file. The classes that derive from Component
        * must reimplement this method, in order to unpack the binary contents of
        * the configuration file with their Setup structure.
        * Overriden from the Component class
        */
        void configure_from_file (const std::string& fname) override;

    protected:

        // State variables declaration
        StateVar<MyCustomType>          my_state1;  ///< One State Var

        // Inputs declaration
        DataIO<int>                     my_input_port1;  ///< One input port

        // Outputs declaration
        DataIO<double>                  my_output_port1;  ///< One output port

        // Configuration properties declaration
        Property<float>                 my_prop1;  ///< One property

        // Faults declaration
        Fault                           f0;
        Fault                           f1;

        // Alarms declaration
        Alarm                           a0;
        Alarm                           a1;
};

} // namespace gmt

#endif   // _MyComponentBase_h_
```

As we can see, the contents of the class definition are: the overridden methods from the base class, the *State Variables* definition, the *Inputs* definition, the *Outputs* definition and the *Properties* definition. The types that are custom defined in the model have been translated to CamelCase.

In the class definition there will be only the State Variables, Properties and Data IO (Inputs or Outputs) from the model that are owned (first defined by) by this class. Please note that there is a set of other class members that will be inherited from the base classes. The list of class members inherited from the most common superclasses are listed in the following table:

Class member	Kind	Inherited from
op_state	state_vars	Component
uri	properties	Component
name	properties	Component
host	properties	Component
port	properties	Component
acl	properties	Component
scan_rate	properties	Component
sim_mode	state_vars	BaseController
control_mode	state_vars	BaseController

Includes

The first section of the component header file is a set of *#include* directives. This list is composed by:

- The include to the DCS types header, `` `../../include/my_subsystem_port_types.h` ``
- The includes to the header of each of the frameworks listed in the `` `uses` `` element of the model

Typedefs

The class definition always contains a *typedef* directive for the component setup class. Therefore, one can always refer to the component configuration class as `` `MyComponent::Setup` ``.

Methods

The class definition contains the declarations of the constructor and the overridden methods from the base class:

- **Constructor and destructor:** Constructor and virtual destructor for the class
- **create_state() method:** Creates the internal data structures for the component features, and initializes them with a default value.
- **setup_state() method:** Contains the code that handles the component configuration.
- **configure_from_file():** This polymorphic method will read the configuration file with the appropriate Setup type.

Inputs

The Inputs definition section is marked with the comment `// Inputs declaration`. A class member variable will be generated for each Input defined in the model. For example, if the component model file contains

```
inputs:
    my_input:
        desc:           'One input'
        type:           'my_type'
        max_rate:       1000
```

then the C++ counterpart will be a member variable defined as:

```
DataIO<MyType>   my_input;
```

Here we can see that the type of the `` `my_input` `` member variable is a specialization of the `` `DataIO` `` templated struct. The template parameter is the input type, converted to the appropriate representation for C++.

The `` `DataIO<T>` `` struct contains the following fields, that correspond one to one to the fields of an Input in the model files specification:

Field	Type	Description
name	string	Name of the DataIO
value	T	The actual value of the DataIO
info	string	Brief description
desc	string	Detailed Description
type	string	Type of the DataIO
units	string	The units that must be used to interpret the value
min	T	The minimum value that can be assigned to the DataIO
max	T	The maximum value that can be assigned to the DataIO
default_value	T	The value read from the configuration
regexp	string	Regular expression that has to be matched by the value
max_rate	double	Maximum rate allowed for this DataIO
sampling_rate	double	Sampling rate inherent to the current magnitude
storage	uint32	Decimation factor (against the sampling rate) for telemetry

In addition, the `Component` base class provides an `inputs` member variable, which has the collection of all the inputs. This collection can be indexed by the input name, for example `inputs["my_input"]`.

Outputs

The Outputs definition section is marked with the comment `// Outputs declaration`. A class member variable will be generated for each Output defined in the model. Analogously as the Inputs, the type of the generated member variables for the Outputs will be `DataIO<T>`, where `T` is the specific type of the output.

```
output_ports:
    my_output:
        desc:           'One output'
        type:           'my_type'
        max_rate:       1000
```

then the C++ counterpart will be a member variable defined as:

```
DataIO<MyType>   my_output;  // One output port
```

The type of the parameter of the DataIO template is mapped to its C++ equivalent, if needed.

Similarly to the Inputs case, the Outputs can be navigated using the `outputs` member variable, which is inherited from the `Component` base class.

State Variables

The start of the State Variables section is marked by the comment `// Create state variables`. For each component State Variable `my_statevar` of `my_type`, a class member variable will be created, with the form `StateVar<MyType> my_statevar;`.

The `StateVar<MyType>` template inherits from `DataIO`, and it adds the following fields:

Field	Type	Description
goal	T	The setpoint for the StateVar
is_controllable	bool	True if a goal can be defined for this StateVar
control_rate	double	Control rate

The State Variables can be navigated using the `state_vars` member variable, which is inherited from the `Component` base class.

Configuration Properties

The Configuration Properties section is marked with the comment `// Configuration properties declaration`. A class member variable will be generated for each Property defined in the model. The type of the generated member variables will be `Property<T>`, where `T` is the C++ mapping of the type declared in the model.

The `Property<T>` struct contains the following fields, that correspond one to one to the fields of an Property in the model files specification:

Field	Type	Description
name	string	Name of the Property
value	T	The actual value of the Property
info	string	Brief description
desc	string	Detailed Description
type	string	Type of the Property
units	string	The units that must be used to interpret the value
min	T	The minimum value that can be assigned to the Property
max	T	The maximum value that can be assigned to the Property
default_value	T	The value read from the configuration
regexp	string	Regular expression that has to be matched by the value

As an example, if the component model file contains

then the C++ class will have:

```
Property<float>                 my_prop1;   ///< One property
```

The Properties of a component are also navigable, using the `properties` member variable, which is inherited from the base class.

3.1.2 Component setup class header file

The configuration of any component (values of the properties and setup information for the state variables and DataIO) is stored in a class named `<ComponentName>Setup`, and as stated above, the component class has an alias for this setup class which always has the name <ComponentName>::Setup.

The `<ComponentName>Setup` inherits from the base class Setup class. The root of the setup classes hierarchy is the BaseComponent::Setup class.

The Setup class definition is generated in the file `<ComponentName>Setup.h`. The generated code for the example component of Section 3.1 would be in the file `MyComponentSetup.h`, with the following content:

```
#ifndef _MyComponentSetup_h_
#define _MyComponentSetup_h_

#include <ocs_core_fwk.h>

#include <msgpack.hpp>
#include <ocs_core_fwk.h>
#include <ocs_ctrl_fwk.h>
#include "../../include/isample_dcs_port_types.h"

namespace gmt
{

struct MyComponentSetup : public BaseComponentSetup
{

    struct StateVarConf : public  BaseComponentSetup::StateVarConf
    {
        StateVar<MyCustomType>          my_state1;
        MSGPACK_DEFINE_MAP(my_state1, op_state)
    };
```

```
    struct InputConf : public BaseComponentSetup::InputConf
    {
        DataIO<int>                     my_input_port1;
        MSGPACK_DEFINE_MAP(my_input_port1)
    };

    struct OutputConf : public BaseComponentSetup::OutputConf
    {
        DataIO<double>                  my_output_port1;
        MSGPACK_DEFINE_MAP(my_output_port1)
    };

    struct PropertyConf : public  BaseComponentSetup::PropertyConf
    {
        Property<float>                 my_prop1;
        MSGPACK_DEFINE_MAP(my_prop1, uri, name, host, port, acl, scan_rate, priority)
    };

    struct FaultConf : public BaseComponentSetup::FaultConf
    {
        Fault f0;
        Fault f1;
        MSGPACK_DEFINE_MAP(f0, f1)
    };

    struct AlarmConf : public BaseComponentSetup::AlarmConf
    {
        Alarm a0;
        Alarm a1;
        MSGPACK_DEFINE_MAP(a0, a1)
    };

    StateVarConf    state_vars;
    InputConf       inputs;
    OutputConf      outputs;
    PropertyConf    properties;
    FaultConf       faults;
    AlarmConf       alarms;

    MSGPACK_DEFINE_MAP(properties, state_vars, inputs, outputs, faults, alarms,␣
↪connectors)
};

} //namespace gmt

#endif // _MyComponentSetup_h_
```

Here we can see 7 main blocks:

`struct StateVarConf` definition: This is the definition for the inner struct where all the state variables meta-information will be stored. There is one entry `StateVar<type> state_var` for each state variable defined in the component model. In addition, there is the `MSGPACK` clause that allows the struct to be serialized automatically by msgpack. Note that although the state variables defined in the base class are inherited from `BaseComponentSetup::StateVarConf` and, therefore, they are not re-defined here, they are explicitly listed in the `MSGPACK` directive.

`struct InputConf` definition: This is the definition for the inner struct where all the inputs meta-information will be stored. There is one entry `DataIO<type> inp` for each input defined in the component model. In

addition, there is the `MSGPACK` clause that allows the struct to be serialized automatically by msgpack. Note that although the inputs defined in the base class are inherited from `BaseComponentSetup::InputConf` and, therefore, they are not re-defined here, they are explicitly listed in the `MSGPACK` directive.

`struct OutputConf` definition: This is the definition for the inner struct where all the outputs meta-information will be stored. There is one entry `DataIO<type> outp` for each output defined in the component model. In addition, there is the `MSGPACK` clause that allows the struct to be serialized automatically by msgpack. Note that although the outputs defined in the base class are inherited from `BaseComponentSetup::OutputConf` and, therefore, they are not re-defined here, they are explicitly listed in the `MSGPACK` directive.

`struct PropertyConf` definition: This is the definition for the inner struct where all the configuration properties will be stored. There is one entry `Property<type> prop` for each configuration property defined in the component model. In addition, there is the `MSGPACK` clause that allows the struct to be serialized automatically by msgpack. Note that although the properties defined in the base class are inherited from `BaseComponentSetup::PropertyConf` and, therefore, they are not re-defined here, they are explicitly listed in the `MSGPACK` directive.

`struct FaultConf` definition: This is the definition for the inner struct where all the faults meta-information will be stored. There is one entry `Fault f` for each fault defined in the component model. In addition, there is the `MSGPACK` clause that allows the struct to be serialized automatically by msgpack. Note that although the faults defined in the base class (if any) are inherited from `BaseComponentSetup::FaultConf` and, therefore, they are not re-defined here, they are explicitly listed in the `MSGPACK` directive.

`struct AlarmConf` definition: This is the definition for the inner struct where all the alarms meta-information will be stored. There is one entry `Alarm a` for each alarm defined in the component model. In addition, there is the `MSGPACK` clause that allows the struct to be serialized automatically by msgpack. Note that although the alarm defined in the base class (if any) are inherited from `BaseComponentSetup::AlarmConf` and, therefore, they are not re-defined here, they are explicitly listed in the `MSGPACK` directive.

Setup class fields definition: The previous sections were only type definitions. After these sections, the following setup class member variables are defined:

- The `` `state_vars` `` member variable, of type `` `StateVarConf` ``
- The `` `inputs` `` member variable, of type `` `InputConf` ``
- The `` `outputs` `` member variable, of type `` `OutputConf` ``
- The `` `properties` `` member variable, of type `` `PropertyConf` ``
- The `` `faults` `` member variable, of type `` `FaultConf` ``
- The `` `alarms` `` member variable, of type `` `AlarmConf` ``

In addition, there is an aditional field, inherited from the base class:

- The `` `connectors` `` member variable, of type `` `ConnectorConf` ``, that contains the list of connectors for the current Component

Analogously to the previous sections, the `` `MSGPACK` `` directive allows the component Setup class to be serialized automatically.

3.1.3 Component implementation class

The C++ class definition of the class for the component is located in the `` `<ComponentName>.h` `` file. The name of the class is set as the name defined in the model file, but in CamelCase. For example, a component named `` `my_component` `` in the model files would have a base class named `` `MyComponent` ``. The implementation of the class is in the `` `<ComponentName>.cpp` `` file. These two files will be initially generated with `gds gen`, but then the user must add its own implementation details. When the file already exist, `` `gds gen` `` will not overwrite them.

The contents of the generated header file will be:

```
#ifndef _MyComponent_h_
#define _MyComponent_h_


#include "MyComponentBase.h"

namespace gmt
{

class MyComponent : public MyComponentCtrlBase
{
public:
    MyComponent(
                    const  std::string& comp_uri,
                    const  std::string& comp_name,
                    const  std::string& comp_host,
                    int    comp_port,
                    const  std::string& comp_acl,
                    double comp_scan_rate = 1.0,
                    int comp_prio = GMT_THREAD_DEFAULT_PRIO,
                    int comp_stack_size = GMT_DEFAULT_STACK_SIZE);

    virtual ~MyComponent();

    //XXX add your public methods here

protected:

    void step() override;
    void setup() override;

    //XXX add your protected class members here

private:

    //XXX add your private class members here
};

} // namespace gmt

#endif   // _MyComponent_h_
```

If needed, the user can add more member variables or methods to this class.

The contents of the cpp file will be:

```
#include <ocs_core_fwk.h>

#include "MyComponent.h"

using namespace std;
using namespace gmt;

MyComponent::MyComponent(
                        const string& comp_uri,
                        const string& comp_name,
                        const string& comp_host,
```

```
                        int comp_port,
                        const string& comp_acl,
                        double comp_scan_rate,
                        int comp_prio,
                        int comp_stack_size)
    : MyComponentBase(comp_uri, comp_name, comp_host, comp_port, comp_acl, comp_scan_
↪rate, comp_prio, comp_stack_size)
{
}

MyComponent::~MyComponent()
{
}

void MyComponent::step()
{
    //XXX add your code here

    /*if (is_step_rate(100))
    {
        // this will be executed every 100 steps
        log_info("step  = " + std::to_string(step_counter));
    }*/
}

void MyComponent::setup()
{
    //setup the base class
    Base::setup();

    //setup async input handlers

    //ex: new_async_input_handler ("my_input_name", this, &MyComponent::my_input_
↪handler);

    //add behaviors to features

    //setup fault evaluation functions
    set_fault_eval_func ("f0", [this](FaultFSM& fsm){return true;}); //TODO change␣
↪the "return true" to the specific evaluation of the fault
    set_fault_eval_func ("f1", [this](FaultFSM& fsm){return true;}); //TODO change␣
↪the "return true" to the specific evaluation of the fault

    //setup alarm evaluation functions
    set_alarm_eval_func ("a0", [this](AlarmFSM& fsm){return true;}); //TODO change␣
↪the "return true" to the specific evaluation of the fault
    set_alarm_eval_func ("a1", [this](AlarmFSM& fsm){return true;}); //TODO change␣
↪the "return true" to the specific evaluation of the fault

    //other initializations

}
```

The user must add the specific step function implementation to `MyComponent::step()` and, of course, the implementation of any other method that has been added to the class.

Note: the call to `` `Base::setup()` `` in the `` `setup()` `` method is important for the initialization of the base classes structures, and must not be removed.