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

Release 1.5.0

Software and Controls Group

Jul 02, 2019

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

For example, one structured type defined in the model as

```
StructType "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: "my_other_struct", desc: "This is field 3"}
        data_field4: {type: "float[4]", desc: "This is field 4"}
```

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

```
#pragma once
#include <msgpack.hpp>
#include <string>
#include <array>
#include <vector>

struct my_struct {
    std::string          data_field1;    // This is field 1
    int                  data_field2;    // This is field 2
    my_other_struct      data_field3;    // This is field 3
    std::array<float,4>  data_field4;    // This is field 4

    MSGPACK_DEFINE_MAP(data_field1, data_field2, data_field3)
};
```

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 "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:

```
#pragma once

enum class my_enum_type : std::uint8_t { THE_FIRST_LABEL=0,
                                          THE_SECOND_LABEL=1,
                                          THE_THIRD_LABEL=2};
MSGPACK_ADD_ENUM(my_enum_type)
```

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

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>
         |-- <component_name>.h
         |-- <component_name>.cpp
         |-- <component_name>_step.cpp
         |-- <component_name>_setup.cpp
         |-- <component_name>_msgpack.h
         |-- <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 header file

The C++ class definition of the component is located in the `` `<component_name>.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 be mapped as a class named `` `MyComponent` ``.

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
            blocking_mode:          'async'
            is_controllable:        true

    input_ports:
        my_input_port1:
            desc:               'One input port'
            type:               'Integer'
            protocol:           'pull'
            max_rate:           1000
            blocking_mode:      'async'

    output_ports:
        my_output_port1:
            desc:               'One output port'
            type:               'float64'
            protocol:           'push'
            max_rate:           1000
            blocking_mode:      'async'

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

then the generated C++ class would be:

```
#ifndef _my_component_h_
#define _my_component_h_

#include "ocs_core_fwk.h"
#include "ocs_ctrl_fwk.h"
#include "../../include/my_subsystem_port_types.h"

class MyComponentSetup;

namespace gmt
{

class MyComponent : public BaseComponent
{
    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,
            float comp_scan_rate);

        virtual ~MyComponent();
```

```
        void setup() override;
        void step(bool setup_ok) override;

    protected:

        typedef MyComponentSetup Setup;

        virtual void create_config() override;

        // Create state variables
        StateVar<my_custom_type> my_state1_sv;

        // Input port declaration
        int    my_input_port1;      // One input port

        // Output port declaration
        double my_output_port1;    // One output port

        // Configuration properties
        float   my_prop1;           // One property
};

} // namespace gmt

#endif   // _my_component_h_
```

As we can see, the contents of the class definition are: the overridden methods from the base class, the *State Variables* definition, the *Input Ports* definition, the *Output Ports* definition and the *Properties* definition.

In the class definition there will be only the State Variables, Properties and Ports 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
ops_state	state_vars	Component
heartbeat_out	output_ports	Component
uri	properties	Component
name	properties	Component
host	properties	Component
port	properties	Component
acl	properties	Component
scan_rate	properties	Component
ecat_bus	input_ports	EthercatAdapter
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. The definition is in the `` `my_component.cpp` `` source file.
- **step() method:** The `step()` method is one of the most important methods of any component. This is the place where the developer must insert the code for the component step function, which implements the main functionality of the component. A basic version of this method is generated in the `` `my_component_step.cpp` `` source file, which must be updated by the module developer to add the component specific code.
- **setup() method:** Contains the code that handles the component configuration, and that creates the links between class member variables and the corresponding ports, state variables or properties. The definition of the `setup()` method is generated in the `` `my_component_setup.cpp` `` source file.
- **create_config():** This is an auxiliary method called by `my_component::setup()` in order to polymorphically create the configuration member variable.

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<my_type> my_statevar_sv;` (note that the suffix `` `_sv` `` has been added to the variable name). The type of the State Variable is mapped to the C++ equivalent one, if needed.

The `` `StateVar<my_type>` `` template is a struct that contains the fields

```
std::string  name;
bool         is_controlable;
my_type      value;
my_type      goal;
my_type      max;
my_type      min;
```

Therefore, the goal and the current value of the `my_statevar_sv` State Variable are accessible by means of `my_statevar_sv.goal` and `my_statevar_sv.value`.

Input Ports

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

```
input_ports:
    my_input_port:
        desc:            'One input port'
        type:            'my_type'
        protocol:        'pull'
        max_rate:        1000
        blocking_mode:   'async'
```

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

```
my_type   my_input_port;
```

The type of the port declared in the model file is mapped to its C++ equivalent, if needed.

In addition, the `` `Commponent` `` base class provides an `` `inputs` `` member variable, which has the collection of all the inputs ports. This collection can be indexed by the port name, for example `inputs["my_input_port"]`. The object returned contains the actual structure that supports the port functionality. In particular, the port parameters (nominal rate, etc) are stored in the `config` field (e.g.: inputs["my_input_port"].config), of type PortConfig:

```
struct Port_config
{
    std::string name;
    std::string protocol;
    std::string url;
    std::string blocking_mode;
    float max_rate;
    float nom_rate;
    Transport_config_ifce& transport_config;
};
```

Output Ports

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

```
output_ports:
    my_output_port:
        desc:             'One output port'
        type:             'my_type'
        protocol:         'push'
        max_rate:         1000
        blocking_mode:    'async'
```

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

```
my_type   my_output_port;   // One output port
```

The type of the port declared in the model file is mapped to its C++ equivalent, if needed.

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

Configuration Properties

The Configuration Properties section is marked with the comment // Configuration properties. A class member variable will be generated for each Property defined in the model. As the previous cases, the type of the properties member variables will be the C++ mapping of the Property model type.

As an example, if the component model file contains

then the C++ class will have:

```
float   my_prop1;   // One property
```

The type of the port declared in the model file is mapped to its C++ equivalent, if needed.

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

3.2 Component setup class header file

The configuration of any component (values of the properties and setup information for the state variables and ports) 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 `` `<component_name>_msgpack.h` ``. The generated code for the example component of Section 3.1 would be in the file `` `my_component_msgpack.h` ``, with the following content:

```
#ifndef _my_component_msgpack_h_
#define _my_component_msgpack_h_

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

struct MyComponentSetup : public  BaseComponentSetup {

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

    struct StateVarConf : public  BaseComponentSetup::StateVarConf {
        StateVarDef<my_custom_type>    my_state1;
        MSGPACK_DEFINE_MAP(my_state1, ops_state)
    };

    struct InputPortConf : public BaseComponentSetup::InputPortConf {
        PortDef<int>                    my_input_port1;
        PortDef<my_custom_type>         my_state1_goal;
        MSGPACK_DEFINE_MAP(my_input_port1, my_state1_goal, ops_state_goal)
    };

    struct OutputPortConf : public BaseComponentSetup::OutputPortConf {
        PortDef<double>                 my_output_port1;
        PortDef<my_custom_type>         my_state1_value;
        MSGPACK_DEFINE_MAP(my_output_port1, heartbeat_out, my_state1_value, ops_state_
↪value)
    };

    PropertyConf     properties;
    StateVarConf     state_vars;
    InputPortConf    input_ports;
    OutputPortConf   output_ports;

    MSGPACK_DEFINE_MAP(properties, state_vars, input_ports, output_ports)
};

#endif // _my_component_msgpack_h_
```

Here we can see 5 main blocks:

`struct PropertyConf` definition: This is the definition for the inner struct where all the configuration properties will be stored. There is one entry `PropertyDef<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 StateVarConf definition: This is the definition for the inner struct where all the state variables meta-information will be stored. There is one entry `StateVarDef<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 InputPortConf definition: This is the definition for the inner struct where all the input ports meta-information will be stored. There is one entry `PortDef<type> port` for each input port defined in the component model, and also one entry for the goal of each state variable. The suffix `_goal` is added automatically to the state variable names. In addition, there is the MSGPACK clause that allows the struct to be serialized automatically by msgpack. Note that although the input ports defined in the base class are inherited from `BaseComponentSetup::InputPortConf` and, therefore, they are not re-defined here, they are explicitly listed in the MSGPACK directive.

struct OutputPortConf definition: This is the definition for the inner struct where all the output ports meta-information will be stored. There is one entry `PortDef<type> port` for each output port defined in the component model, and also one entry for the value of each state variable. The suffix `_value` is added automatically to the state variable names. In addition, there is the MSGPACK clause that allows the struct to be serialized automatically by msgpack. Note that although the output ports defined in the base class are inherited from `BaseComponentSetup::OutputPortConf` 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 `properties` member variable, of type `PropertyConf`
- The `state_vars` member variable, of type `StateVarConf`
- The `input_ports` member variable, of type `InputPortConf`
- The `output_ports` member variable, of type `OutputPortConf`

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

3.3 Component setup method definition file

The code for the `<component_name>::setup()` and `<component_name>::create_config()` methods is automatically generated from the model to the file `<component_name>_setup.cpp`.

The code generated for the `create_config()` method creates polymorphically the `config` pointer.

The code generated for the `setup()` method creates the structures for the input ports, output ports and properties, initializes them and creates the links between the ports and properties to the actual member variables of the component class.

3.4 Component step method definition file

The `<component_name>::step()` method is the one that defines the behavior of the component. Hence, it is one of the most important methods of any component, and the only one from the auto-generated methods that the user should change.

The code of the *step* method is in the `` `<component_name>_step.cpp` `` file. When the component code is auto-generated from the model, the code generator places a skeleton of the *step* method in this file. The developer must overwrite it with the actual implementation for the current component.