ISample example

Release 1.6.2

Software and Controls Group

Dec 10, 2019

Contents

ISample DCS is an instrument control system example that provides a template that instrument developers can use as a model.

Note: The following instructions assume that the SDK has been installed and the Development Environment configured correctly according to the instructions in installation or upgrade.

CHAPTER

ONE

CLONE THE ISAMPLE_DCS REPOSITORY

On the development machine, clone the repository in the development folder:

```
$ cd $GMT_LOCAL/modules
$ gds clone isample_dcs -d gmto
```

where the `-d option` defines the git repository owner. The output from the command will be:

```
[WRN] [hdk_dcs] Module lib [hdk_dcs] not found, trying to load from source
[ERR] [hdk_dcs] model definition for module hdk_dcs not found
[ERR] [gds] module isample_dcs not defined in bundle ocs_sdk_bundle
[INF] [gds] clone module isample_dcs
[INF] [isample_dcs] Cloning module: isample_dcs
```

These warning are normal, the first two are because the hdk module is not installed yet, and the other one is generated because the isample_dcs is not defined in the SDK bundle. As it is included in the local bundle, the module is cloned successfully, as the two latest messages inform us.

CHAPTER

TWO

MODEL FILES

The model files can be found in the **$GMT_LOCAL/modules/ocs_isample_dcs/model/** folder.

isample_core_if.coffee Lists the connectors between the isample and GMT core systems

isample_dcs.coffee Lists the connectors between the supervisor layer and the component layer. For this example, these are limited to monitoring the heartbeat of each component.

isample_dcs_def.coffee High-level definition file, representing the WBS for the submodule. It lists the components and how many instances of each are required.

isample_dcs_types.coffee Definitions of structs and data types used by the isample components.

isample_ctrl_pkg/isample_ctrl_fb.coffee Fieldbus definitions for the isample control package.

isample_ctrl_pkg/isample_ctrl_pkg.coffee Lists the connectors between components.

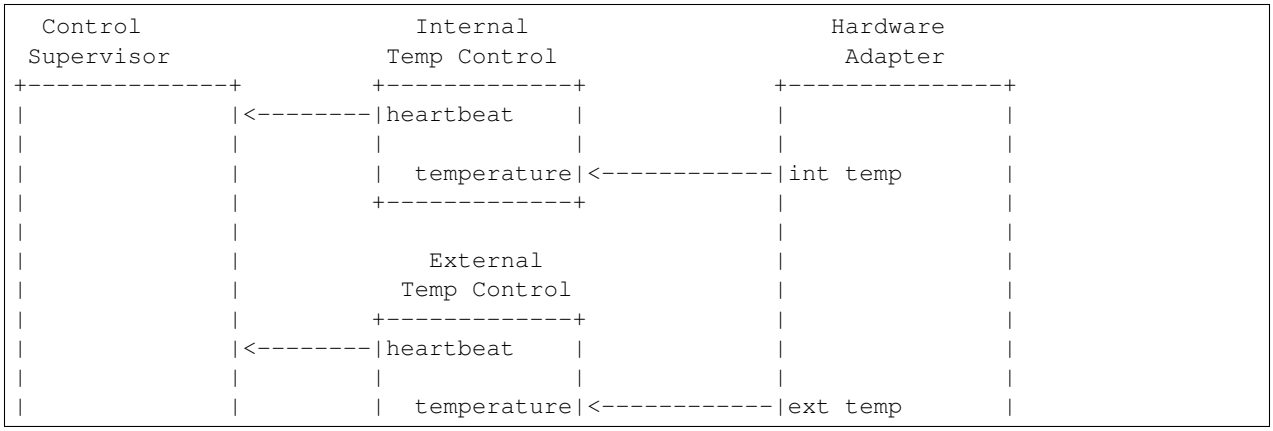
isample_ctrl_pkg/isample_ctrl_super.coffee Definition of the *Control Supervisor* component. State variables, input and output ports are specified here. A single instance called **isample_ctrl_super** will be created.

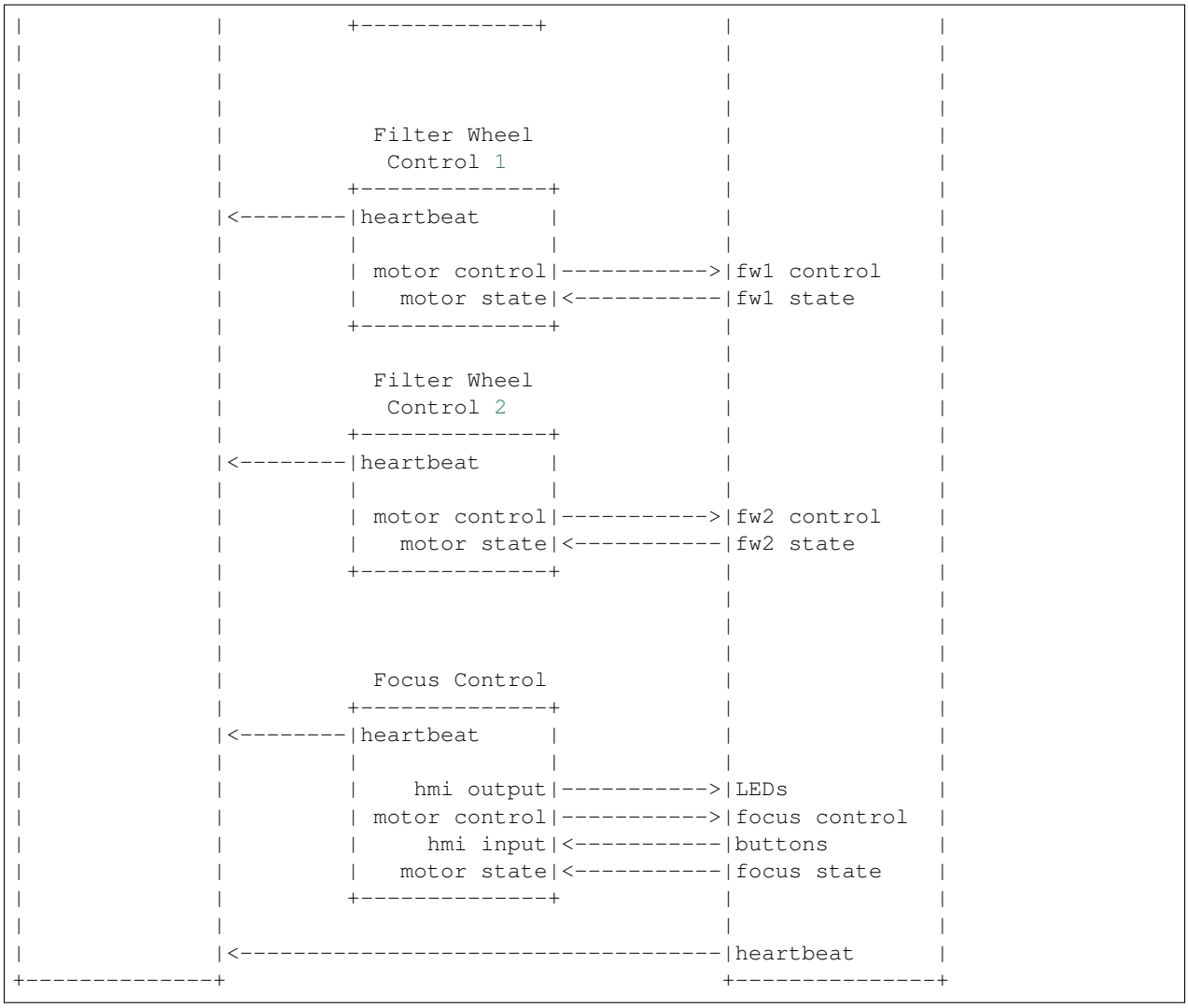
isample_ctrl_pkg/isample_filter_wheel_ctrl.coffee Definition of the *Filter Wheel Controller* component. State variables, input and output ports are specified here. Two instances, called **isample_fw1_ctrl** and **isample_fw2_ctrl** will be created.

isample_ctrl_pkg/isample_focus_ctrl.coffee Definition of the *Focus Controller* component. State variables, input and output ports are specified here. A single instance called **isample_focus_ctrl** will be created.

isample_ctrl_pkg/isample_hw_adapter.coffee Definition of the *Hardware Adapter* component, used to interface with the isample Actuators and Sensors. State variables, input and output ports are specified here. A single instance called **isample_hw_ctrl** will be created.

isample_ctrl_pkg/isample_temp_ctrl.coffee Definition of the *Temperature Controller* component. State variables, input and output ports are specified here. Two instances, called **isample_cryo_internal_temp_ctrl** and **isample_cryo_external_temp_ctrl** will be created.




Filter Wheel
Control 1
heartbeat
motor control
motor state
fw1 control
fw1 state
Filter Wheel
Control 2
heartbeat
motor control
motor state
fw2 control
fw2 state
Focus Control
heartbeat
hmi output
motor control
hmi input
motor state
LEDs
focus control
buttons
focus state
heartbeat

CHAPTER

THREE

CODE GENERATION

To generate the code skeleton from the model files, execute:

```
$ cd $GMT_LOCAL/modules/ocs_isample_dcs/model
$ webpack
$ gds gen isample_dcs
```

This will generate the basic framework of source code and configuration files for each component. The generated source files will be located in the *src/* folder. It is possible that gds outputs some warning because there are missing modules which are defined in the local bundle (in particular, the hdk). This is not a problem, and the code will be successfully generated.

To see the generated folders and files, navigate to:

```
$ cd $GMT_LOCAL/modules/ocs_isample_dcs/src/
$ ls -la
```

CHAPTER

FOUR

COMPONENT ATTRIBUTES

Components are defined by their state variables, input ports, output ports and step function.

The Filter Wheel component has the following attributes:

4.1 State Variables

Type	Name	Range	Default
float	position	min: 5, max: 40	20
OperationalState	op_state	OFF, STARTING, ON, INITIALIZING, RUN, HALTING, SHUTTING_DOWN, FAULT, RESETTING, DISABLED	OFF
SimulationMode	sim_mode	SIMULATION, ON_LINE	ON_LINE
ControlMode	control_mode	STANDALONE, INTEGRATED	STANDALONE

OperationalState, SimulationMode and ControlMode are enums with their respective values shown in the "Range" column above.

4.2 Inputs

Type	Name	Internal variable
isample_motor_status	motor_state	motor_state
float	position_goal	position.goal
OperationalState	ops_state_goal	ops_state.goal
SimulationMode	sim_mode_goal	sim_mode.goal
ControlMode	control_mode_goal	control_mode.goal

where the struct *isample_motor_status* is defined as:

```
struct isample_motor_status {
    bool            ready;                  // Axis Ready
    bool            enabled;                // Axis Enabled
```

```
    bool            warning;                // Axis Warning
    bool            error;                  // Axis Error
    bool            moving_positive;        // Axis Moving +
    bool            moving_negative;        // Axis Moving -
    MSGPACK_DEFINE_MAP(ready, enabled, warning, error, moving_positive,␣
↪moving_negative)
};
```

4.3 Outputs

Type	Name	Internal Variable
isample_motor_control	motor_control	motor_control
float	position_value	position.value
OperationalState	ops_state_value	ops_state.value
SimulationMode	sim_mode_value	sim_mode.value
ControlMode	control_mode_value	control_mode.value

where the struct *isample_motor_control* is defined as:

```
struct isample_motor_control {
    bool            enable;                 // Axis Enable
    bool            reset;                  // Axis Reset
    int16_t         velocity;               // Velocity
    MSGPACK_DEFINE_MAP(enable, reset, velocity)
};
```

CHAPTER

FIVE

(OPTIONAL) DEFINING COMPONENT BEHAVIOR

The core component behavior is specified in the component cpp file. The component has a periodic thread that reads input from the input ports, runs the step function and then writes output to the output ports. Initially, the generated step function will check whether the component is correctly configured and if so, will log the current step counter value.

In the following examples we will replace the basic step functionality with simulated controller behavior.

To edit the *Filter Wheel Controller* step function:

```
$ cd $GMT_LOCAL/modules/ocs_isample_dcs/src/cpp/
$ cd isample_ctrl_pkg/isample_filter_wheel_ctrl
$ vi IsampleFilterWheelCtrl.cpp
```

The following example step function for the filter wheel controller validates positional input and immediately sets the position value to the new goal, if possible.

```
void IsampleFilterWheelCtrl::step()
{
    if (is_step_rate(1000))
    {
        if (position.goal != position.value)
        {
            // check range
            if (position.goal >= position.max)
            {
                log_warning("Position is at or exceeding maximum value: " +␣
↪std::to_string(position.max));
                // prevent further movement
                position.value = position.max;
            }
            else if (position.goal <= position.min)
            {
                log_warning("Position is at or exceeding minimum value: " +␣
↪std::to_string(position.min));
                // prevent further movement
                position.value = position.min;
            }
            else
            {
                // achieve target position immediately
                position.value = position.goal;
            }
            // report value
            log_info(position.name + " = " + std::to_string(position.value));
        }
```

```
    }
}
```

CHAPTER

SIX

COMPILATION

To compile the C++ Control Package code, edit the module.mk file to contain the correct library definitions:

```
$ vi $GMT_LOCAL/modules/ocs_isample_dcs/src/cpp/isample_ctrl_pkg/module.mk
```

Ensure that the following lines are defined:

```
# Add in this file the compile flags for the package, eg:
MOD_BUILD_LDFLAGS += -lcore_core_pkg -lio_core_pkg -lctrl_core_pkg
```

Run **make** to compile the code:

```
$ cd $GMT_LOCAL/modules/ocs_isample_dcs/src/cpp
$ make
```

CHAPTER

SEVEN

INSTALLING THE CONFIGURATION

The configuration files are autogenerated in the *$GMT_LOCAL/modules/ocs_isample_dcs/src/etc/conf* directory, but they need to be installed to *$GMT_LOCAL/etc/conf* in order to be used by the application.

To install the configuration files, execute the following commands:

```
$ gds install isample_dcs
$ grs compile -i isample_cryo_external_temp_ctrl
$ grs compile -i isample_cryo_internal_temp_ctrl
$ grs compile -i isample_ctrl_super
$ grs compile -i isample_focus1_ctrl
$ grs compile -i isample_fw1_ctrl
$ grs compile -i isample_fw2_ctrl
$ grs compile -i isample_hw1_adapter
```

CHAPTER

EIGHT

RUNNING THE EXAMPLE

Start the logging and telemetry services:

```
$ log_server &
$ tele_server &
```

Start the ISample Control Package application in the background

```
$ isample_ctrl_app &
```

The application is running in the background and will not provide any console output. All output will be directed to the logging service after the components have been successfully set up.

8.1 Log Service

In a separate terminal (for example, *tty2*), **start the logging service client**.

```
$ log_client listen
```

8.2 Telemetry Service

In a separate terminal (for example *tty3*), **start the telemetry service client**.

```
$ tele_client listen
```

In this example, we don't filter, to show data for all monitors. The output can be filtered on substrings of the monitor name by specifying the topic to be a specific component type (`filter_wheel_ctrl`) or an output port name, such as `position` or `heartbeat`. For example,

```
$ tele_client listen --topic=isample_focus1_ctrl/hmi_outputs
```

will show only the values of the `hmi_outputs` monitor from `isample_focus1_ctrl`.

8.3 Interacting with a component

The *grs* command line application can be used to interact with Components. Some of the functionalities provided by this application are querying the current value of a given Component feature (property, state variable, input or output), setting a value or inspecting the whole Component state.

The *grs get* subcommand allows to query the current value of a feature. The syntax is

```
$ grs get -i <instance> -f <feature>
```

For example, to read the value of the *position* state variable of the *isample_fw1_ctrl* instance, execute:

```
$ grs get -i isample_fw1_ctrl -f state_vars/position/value
```

To set a value, the *grs set* subcommand can be used:

```
$ grs set -i <instance> -f <feature> -v <value>
```

For example, to set the goal of the *position* state variable of the *isample_fw1_ctrl* instance, execute:

```
$ grs set -i isample_fw1_ctrl -f state_vars/position/goal -v 2.0
```

Finally, to inspect the whole state, use the *grs inspect* command:

```
$ grs inspect -i <instance>
```

As before, to inspect the state of the *isample_fw1_ctrl* instance, execute:

```
$ grs inspect -i isample_fw1_ctrl
```

[back to top]