

ERC CISST Software

Quick start for cisstMultiTask

Anton Deguet

Date: 2008/09/04 20:08:53

Contents

1	Introduction	2
2	Main concepts	2
2.1	Tasks and devices	2
2.2	Interfaces	2
2.3	Commands	3
2.4	State data and table	3
2.5	Connecting tasks interfaces	3
3	Two tasks communication	4
3.1	Sine wave generator task	4
3.2	Display task	5
3.3	Main program	7
4	Adding events and a device	9
4.1	Sine wave generator task	9
4.2	Clock device	11
4.3	Display task	12
4.4	Main program	13
5	Adding multiple interfaces, complex data types and qualified read commands	14
5.1	Multiple provided and required interfaces	14
5.2	Complex data types	15
5.3	Using the state table index	17
5.4	Main program	18
6	Using a task or device without creating a new task	18
6.1	User interface	19
6.2	Main program	21
7	Lists of figures and listings	21

1 Introduction

These examples provide a quick introduction to the features of `cisstMultiTask`. The code is part of the CVS module `cisst/examples/multiTaskTutorial`. To compile your own code, remember to include `cisstMultiTask.h`. Per convention, all the symbols starting with `mts` are defined in `cisstMultiTask`. Symbols prefixed with `cmn` and `osa` come from `cisstCommon` and `cisstOSAbstraction`, respectively.

To compile the examples provided in this document, you will need the [cisst package](#) as well as the FLTK libraries and its GUI builder (`fluid`) which can be download from www.fltk.org.

2 Main concepts

The `cisstMultiTask` library is intended to help users develop multi-threaded applications and provide a flexible interface between tasks. The base components of the `cisstMultiTask` library are:

- `mtsDevice`, `mtsTask` and `mtsTaskPeriodic`
- `mtsDeviceInterface` and `mtsTaskInterface`
- `mtsCommandVoid`, `mtsCommandRead`, `mtsCommandWrite...`
- `mtsStateData` and `mtsStateTable`
- `mtsTaskManager`

2.1 Tasks and devices

A `cisstMultiTask` task corresponds to a thread with a periodic user defined function. For most applications, the programmer will have to write classes derived from this class, `mtsTaskPeriodic`, which will handle all the timing issues as well as the means to establish the communication with other tasks in a thread safe manner. Communication with other tasks is performed using the task interface class which will be introduced shortly.

The class `mtsTask` is a base class of `mtsTaskPeriodic` and has the same features except the periodic thread. This class assumes that an external library provides a mechanism to call a main `Run` method periodically. This is useful when a device comes with a driver or interrupt more precise than a software based periodic call.

For some specific cases, there might not be any need for a periodic computation (either because it is performed by an external device or because it is performed by another library). Even so, it can be very useful to interface to these devices as if they were `mtsTaskPeriodics`. The class `mtsDevice`, used as a base class, allows to create a wrapper which can be interfaced as a `mtsTaskPeriodic`.

2.2 Interfaces

To provide a flexible interface, the `cisstMultiTask` library uses the “command pattern”. In this design, an object interface is not defined by its public methods but rather by a list of pointers on methods. The list of methods pointers (commands) can be defined and queried at run-time which provides much more flexibility than compilation time binding.

During initialization, a given task has to populate its interface object with pointers to existing methods. This task, the *resource task*, can then be used by another task and by convention, the interface objects containing providing the commands is called *provided interface*. The *user task* will have to query the commands by name and group them in its *required interface* object.

Populating and querying the provided interfaces is performed using the `mtsDevice`, `mtsTask` and `mtsTaskPeriodic` functionalities. Internally these maintain a list of `mtsDeviceInterface` and `mtsTaskInterface`, respectively.

2.3 Commands

Each *provided interface* contains a list of commands. As internally the commands are method pointers, a limited number of signatures is supported. The simplest command doesn't take any parameters and is called `mtsCommandVoid`. To send a data object, one can use a write command (`mtsCommandWrite`) and to receive a data object one can use a read command (`mtsCommandRead`). Finally, *cisstMultiTask* provides a qualified read command (`mtsCommandQualifiedRead`) which allows to send one data object and retrieve one.

Since the signatures of the methods used for the commands must all match, all the parameters must be derived from a base class. This base type, `cmnGenericObject`, is defined in *cisstCommon*. Convenience types are also defined such as `cmnDouble`, `cmnInt...`. These classes are mere proxies to the actual data which can be access using the `Data` data member.

Internally, *cisstMultiTask* will use buffers to ensure thread-safety. For a write or void command sent to a task, a mailbox will be used. For a read command, one should use a special buffer also built in `mtsTask`, i.e. the state table.

2.4 State data and table

Each task owns a state table (`mtsStateTable`) which can be used to store the state of the task (the data member is `mtsTask::StateTable`). The table is a matrix indexed by time. At each iteration, one or more data objects used to define the state (`mtsStateData`) are saved in the table. At any given time, the task can write in the last row while anyone can safely read the previous states (including from other threads/tasks).

The state table length is fixed to avoid dynamic re-allocation. Its size is defined by a `mtsTask` constructor parameter. The table will not overflow because it is implemented as a circular buffer. The class `mtsStateData` provides easy ways to create commands to access the state table. These can be used in the task interface.

2.5 Connecting tasks interfaces

Once all the tasks are defined, it is necessary to connect them. Each task or device can have multiple provided interfaces. Reciprocally, a task can connect to multiple required interfaces, i.e. a task “user” can connect to multiple provided interfaces provided by one or more “resource” tasks. For each interface provided by a resource, a user task must define a *required interface*. To manage the devices, tasks and their connections, use the *cisstMultiTask* class `mtsTaskManager`.

3 Two tasks communication

In this example we will create two tasks, a sine wave generator and a data display. The communication works both ways. The display task will be able to read the last value created by the sine wave generator. The display task will also be able to change the sine wave amplitude by sending/writing the new amplitude. The sine wave generator interface is designed so that it could be replaced by any other signal generator. Hence the commands are called “GetData” and “SetAmplitude”.

3.1 Sine wave generator task

As mentioned earlier, the sine wave generator is derived from `mtsTaskPeriodic`:

Listing 1: `sineTask.h`

```

11 class sineTask: public mtsTaskPeriodic {
    // used to control the log level, 5 by default
13     CMN_DECLARE_SERVICES(CMN_NO_DYNAMIC_CREATION, 5);
    protected:
15     // data generated by the sine wave generator
        mtsStateData<cmnDouble> SineData;
17     // this is the amplitude of the sine wave
        mtsStateData<cmnDouble> SineAmplitude;
19
    public:
21     // provide a name for the task and define the frequency (time
        // interval between calls to the periodic Run). Also used to
23     // populate the interface(s)
        sineTask(const std::string & taskName, double period);
25     ~sineTask() {};
        // all four methods are pure virtual in mtsTask
27     void Configure(const std::string & CMN_UNUSED(filename)) {};
        void Startup(void); // set some initial values
29     void Run(void); // performed periodically
        void Cleanup(void) {}; // user defined cleanup
31 };

```

Besides the declaration of the constructor and the methods `Configure`, `Startup`, `Run` and `Cleanup`, the class declaration must define all the state data we intend to use. To declare the state data objects, we used the templated class `mtsStateData`. The template parameter is the actual type of the data. As we plan to pass these objects as parameters for the interface commands, we used `cmnDouble` which is derived from `cmnGenericObject`.

The two most significant methods are the constructor and the `Run` method.

Listing 2: `sineTask.cpp: constructor`

```

11 sineTask::sineTask(const std::string & taskName, double period):
    // base constructor, same task name and period. Set the length of
13     // state table to 5000
        mtsTaskPeriodic(taskName, period, false, 5000)
15 {
    // add SineData to the StateTable defined in mtsTask
17     SineData.AddToStateTable(StateTable, "SineData");
    // add one interface, this will create an mtsTaskInterface
19     AddProvidedInterface("MainInterface");

```

```

21 // add command to access state table values to the interface
    SineData.AddReadCommandToTask(this, "MainInterface", "GetData");
23 // add command to modify the sine amplitude
    SineAmplitude.AddWriteCommandToTask(this, "MainInterface",
                                         "SetAmplitude");
25 }

```

In the constructor, we added the state data “SineData” to the state table and then created the provided interface “MainInterface” (for lack of better name). We then used the state data objects which provide methods that can be bound to commands. At run-time, the instantiated task will have a “MainInterface” with the read command “GetData” and the write command “SetAmplitude”. Both of these commands will operate directly on the state table in a thread safe manner.

Listing 3: sineTask.cpp: Run method

```

31 void sineTask::Run(void) {
    // the state table provides an index
33     const mtsStateIndex now = StateTable.GetIndexWriter();
    // process the commands received, i.e. possible SetSineAmplitude
35     ProcessQueuedCommands();
    // compute the new values based on the current time and amplitude
37     SineData = SineAmplitude.Data
        * sin(2 * cmnPI * static_cast<double>(now.Ticks()) * Period / 10.0);
39 }

```

The second significant method is `Run`. We first use the state table indexing to retrieve a time tick. This time tick is used to represent time and compute the sine wave data. Before this computation, the call to `ProcessQueuedCommands` ensures that all posted commands are processed (in this case, the only possible queued command is the write command “SetAmplitude”).

3.2 Display task

The display task for this example is implemented using FLTK. The code for the GUI itself has been generated using the FLTK GUI builder, `fluid`. The user interface event loop has been replaced by our own display task periodic method `Run`. As for the sine wave generator task, the display task is derived from `mtsTaskPeriodic`.

Listing 4: displayTask.h

```

11 class displayTask: public mtsTaskPeriodic {
    // set log level to 5
13     CMN_DECLARE_SERVICES(CMN_NO_DYNAMIC_CREATION, 5);
    volatile bool ExitFlag;
15     double StartValue;

17 protected:
    // local copy of data used in commands
19     cmnDouble Data;
    cmnDouble AmplitudeData;
21     // functions which will be bound to commands
    mtsFunctionRead GetData;
23     mtsFunctionWrite SetAmplitude;
    // user interface generated by FTLK/fluid
25     displayUI UI;

```

```

27 public:
    // see sineTask.h documentation
29 displayTask(const std::string & taskName, double period);
    ~displayTask() {};
31 void Configure(const std::string & CMN_UNUSED(filename) = "");
    void Startup(void);
33 void Run(void);
    void Cleanup(void) {};
35 bool GetExitFlag (void) { return ExitFlag;}
};

```

The class declaration is very similar to the sine wave generator except that we don't need any state data and we declared two `mtsFunctions`. These objects are helpers to provide a more familiar syntax than commands. They nevertheless rely on the actual commands as provided by the resource task. The implementation is:

Listing 5: displayTask.cpp: constructor

```

11 displayTask::displayTask(const std::string & taskName, double period):
12     mtsTaskPeriodic(taskName, period, false, 5000),
    ExitFlag(false)
14 {
    // to communicate with the interface of the resource
16     AddRequiredInterface("DataGenerator");
}

```

In the constructor, we did not define any provided interface as this task is not intended to be used as a resource. But we had to add a required interface to be able to connect to a sine wave generator.

Listing 6: displayTask.cpp: Startup method

```

34 void displayTask::Startup(void)
{
36     // set the initial amplitude based on the configuration
    AmplitudeData.Data = StartValue;
38     // find the interface which has been connected to our resource port
    mtsDeviceInterface * interface = GetProvidedInterfaceFor("DataGenerator");
40     // make sure an interface has been connected
    if (interface) {
42         // bound the mtsFunction to the command provided by the interface
        GetData.Bind(interface, "GetData");
44         // the output stream insertion operator << is overloaded for mtsFunction
        CMN_LOG_CLASS(3) << "Startup:␣GetData␣function:␣"
46             << GetData << std::endl;

        // bound the second function
48         SetAmplitude.Bind(interface, "SetAmplitude");
        CMN_LOG_CLASS(3) << "Startup:␣SetAmplitude␣function:␣"
50             << SetAmplitude << std::endl;

        // and now we can use the function and its underlying command!
52         SetAmplitude(AmplitudeData); // nice!
    } else {
54         CMN_LOG_CLASS(1) << "Startup:␣can␣not␣find␣interface␣for␣resource␣port␣DataGenerator"
            << std::endl;
56         exit(-1);
    }
}

```

```

58     // make the UI visible
    UI.show(0, NULL);
60 }

```

The `Startup` method is *important to understand*. When `Startup` runs, it is assumed that a provided interface has been connected to the display task required interface. This must have been performed using the task manager (`mtsTaskManager::Connect`). If a provided interface has been connected, the display task will look for two commands: “GetData” and “SetAmplitude”. For this example, we create the sine wave generator task with these two commands, but *any task with a similar interface or interface with these two commands* can be used instead. This makes the whole system very flexible. In general, the programmer should also test the result of `GetSineData.Bind` and `SetSineAmplitude.Bind` as they would return `false` if the commands were not provided by the interface.

Listing 7: displayTask.cpp: Run method

```

62 void displayTask::Run(void)
63 {
    // get the current time index to display it in the UI
65     const mtsStateIndex now = StateTable.GetIndexWriter();
    // get the data from the sine wave generator task
67     GetData(Data);
    UI.Data->value(Data.Data);
69     // check if the user has entered a new amplitude in UI
    if (UI.AmplitudeChanged) {
71         // retrieve the new amplitude and send it to the sine task
        AmplitudeData.Data = UI.Amplitude->value();
73         SetAmplitude(AmplitudeData);
        UI.AmplitudeChanged = false;
75         CMN_LOG_CLASS(7) << "Run:_" << now.Ticks()
            << "_Amplitude:_" << AmplitudeData.Data << std::endl;
77     }
    // log some extra information
79     CMN_LOG_CLASS(7) << "Run:_" << now.Ticks()
        << "_Data:_" << Data << std::endl;
81     // update the UI, process UI events
    if (Fl::check() == 0) {
83         ExitFlag = true;
    }
85 }

```

The `Run` method gets the data from the signal source and displays it on the UI. It then checks if the amplitude has been modified by the user and if so sends a command to the signal source. The important thing to note is that all actions on other tasks can be performed using the `mtsFunction` objects which have the “look and feel” of regular C/C++ functions.

3.3 Main program

The goal of the main function is to declare the tasks, connect them and then start them:

Listing 8: main.cpp: first example

```

29     // create our two tasks
    const long PeriodSine = 1; // in milliseconds

```

```

31  const long PeriodDisplay = 50; // in milliseconds
    mtsTaskManager * taskManager = mtsTaskManager::GetInstance();
33  sineTask * sineTaskObject =
        new sineTask("SIN", PeriodSine * cmn_ms);
35  displayTask * displayTaskObject =
        new displayTask("DISP", PeriodDisplay * cmn_ms);
37  displayTaskObject->Configure();

39  // add the tasks to the task manager
    taskManager->AddTask(sineTaskObject);
41  taskManager->AddTask(displayTaskObject);
    // connect the tasks, task.RequiresInterface -> task.ProvidesInterface
43  taskManager->Connect("DISP", "DataGenerator", "SIN", "MainInterface");
    // generate a nice tasks diagram
45  std::ofstream dotFile("example1.dot");
    taskManager->ToStreamDot(dotFile);
47  dotFile.close();

49  // create the tasks, i.e. find the commands
    taskManager->CreateAll();
51  // start the periodic Run
    taskManager->StartAll();

53

55  // wait until the close button of the UI is pressed
    while (1) {
        if (displayTaskObject->GetExitFlag()) {
57             break;
        }
59     }
    // cleanup
61  taskManager->KillAll();

63  osaSleep(PeriodDisplay * 2);
    while (!sineTaskObject->IsTerminated()) osaSleep(PeriodDisplay);
65  while (!displayTaskObject->IsTerminated()) osaSleep(PeriodDisplay);

```

The first part of the main function is dedicated to the log control, i.e. add `std::cout` as an output for the log and add a log per thread using `osaThreadedLogFile`. Log files named “example1-0,1,2.txt” will be created, each one containing the log for a given thread (one for the main, one for each task).

The second part is the creation of the tasks and their insertion in the task manager. The important call is:

```
taskManager->Connect("DISP", "DataGenerator", "SIN", "MainInterface");
```

This tells the task manager to connect the required interface “DataGenerator” of the task “DISP” to the provided interface “MainInterface” of the task “SIN”.

The method `ToStreamDot` generates a “dot” file. “dot” is a nice program that generates graphs from a text description (see fig 1). For more information, visit www.graphviz.org.

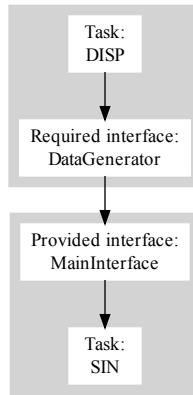


Figure 1: Example 1: tasks graph

4 Adding events and a device

For the second example we are adding events and a plain device. An event allows to communicate from the resource back to the user. The `cisstMultiTask` implementation of events is based on commands, i.e. the observer needs to provide a command to be called when the event occurs (this command corresponds to a callback method). The required steps are:

1. Declare the event on the resource side and associate it to a provide interface. This step creates a command which must be used by the resource programmer when he or she desires to throw that event.
2. Create a callback method on the user side and add it to a given required interface as an event handler command.
3. Once the tasks are connected, specify which user event handler is used for the resource generated event.

Also added in this example is a device (`mtsDevice`), i.e. a wrapper for an existing resource which will have an interface similar to a `mtsTask`. For this example we used the system clock as our resource. It is important to understand that since we don't create a thread for this device, there is no thread safety mechanism added by `cisstMultiTask`. This example remains safe as long as the newly created device is used by one and only one task.

4.1 Sine wave generator task

The class declaration is the same as in the first example except for the declaration of some data members:

Listing 9: `sineTask.h`: event data members

```

18 // trigger value, i.e. when the data generated reaches this value
19 // an event is generated.
    mtsStateData<cmnDouble> TriggerValue;
21 // method used to reset the trigger, we will add a command to the
    // interface for this method
23 void ResetTrigger(void);
    // function bound to the command used to send the event, one could
25 // use a command instead but this is somewhat more convenient
    mtsFunctionWrite TriggerEvent;

```

The trigger value will be set by the user and the mtsFunction TriggerEvent will be used to send the event.

Listing 10: sineTask.cpp: constructor with event

```

10 sineTask::sineTask(const std::string & taskName, double period):
11     mtsTaskPeriodic(taskName, period, false, 5000)
12 {
13     SineData.AddToStateTable(StateTable, "SineData");
    AddProvidedInterface("MainInterface");
15     // add commands to access state table values
    SineData.AddReadCommandToTask(this, "MainInterface", "GetData");
17     SineAmplitude.AddWriteCommandToTask(this, "MainInterface", "SetAmplitude");
    TriggerValue.AddWriteCommandToTask(this, "MainInterface", "SetTriggerValue");
19     // add a command bound to a user defined method
    this->AddCommandVoid(&sineTask::ResetTrigger, this,
21                         "MainInterface", "ResetTrigger");
    // define an event and setup our event sending function
23     cmnDouble eventData; // data type used for the event payload
    TriggerEvent.Bind(AddEventWrite("MainInterface", "TriggerEvent",
25                                 eventData));
}

```

In the constructor, we used a nested call of Bind and AddEventWrite. The AddEventWrite add the event name to the interface and creates a multi-cast command (internally, an object of type mtsMulticastCommandWrite). It is important to note that the user needs to provide the name of the command as well as the type of object carried with the event (also known as payload). In *cisstMultiTask*, the type is defined using an object of the desired type. So far we didn't need to provide an object to define the parameters type as all the commands were based on state data (which is already typed).

A user task willing to receive the event will have to register its callback command (event handler). Doing so, the event handler will be added to the list of commands to “multi-cast” the event to. The method AddEventWrite returns a pointer on a write command. As manipulating commands requires to call their Execute method, an mtsFunction can be used to get a more C/C++ looking code.

Listing 11: sineTask.cpp: Run method with event

```

39
40 void sineTask::Run(void) {
    const mtsStateIndex now = StateTable.GetIndexWriter();
42     ProcessQueuedCommands();
    SineData = SineAmplitude.Data
44     * sin(2 * cmnPI * static_cast<double>(now.Ticks()) * Period / 10.0);

```

```

46 // check if the trigger is enabled and if the conditions are right
47 // to send an event
48 if (TriggerEnabled) {
49     if (SineData.Data >= TriggerValue.Data) {
50         // use the mtsFunctionWrite to send the event along with
51         // the current data
52         TriggerEvent(SineData);
53         TriggerEnabled = false;
54     }
55 }

```

In the Run method, we can use our mtsFunction TriggerEvent to generate an event and send the current data along.

4.2 Clock device

A cisstMultiTask device is a wrapper meant to give the appearance of a cisstMultiTask to an existing task or device. For this example we use a resource available on most systems, the clock. Compared to mtsTaskPeriodic, mtsDevice doesn't have state data nor state table nor the methods Run, Startup and Cleanup.

Listing 12: clockDevice.h

```

11 class clockDevice: public mtsDevice {
12     CMN_DECLARE_SERVICES(CMN_NO_DYNAMIC_CREATION, 5);
13
14 protected:
15     osaStopwatch Timer; // this is the actual device (wrapped)
16     void GetTime(cmnDouble & time) const; // used by the command "GetTime"
17
18 public:
19     // constructor doesn't need a period!
20     clockDevice(const std::string & deviceName);
21     ~clockDevice() {};
22     void Configure(const std::string & CMN_UNUSED(filename) = "") {};
23     // no Startup, Run, Cleanup required
24 };

```

Internally, we used the class osaStopwatch to provide an operating system independent clock.

Listing 13: clockDevice.cpp

```

10 clockDevice::clockDevice(const std::string & deviceName):
11     mtsDevice(deviceName) {
12     AddProvidedInterface("MainInterface"); // interface name for lack of better name
13     AddCommandRead(&clockDevice::GetTime, this, "MainInterface", "GetTime");
14     Timer.Reset(); // reset the clock
15     Timer.Start(); // start the clock
16 }
17
18 void clockDevice::GetTime(cmnDouble & time) const
19 {
20     time = Timer.GetElapsedTime(); // get the time since started
21 }

```

To be used like a `cisstMultiTask` task, our device simply needs to create a provide interface (“MainInterface”) and add commands to it (e.g. “GetTime”) relying on existing methods (e.g. `clockDevice::GetTime`).

4.3 Display task

The display task for this example is very similar to the previous one. For the GUI, used FTLK/fluid to add a text widget to display the time. In the header file we need to add an `mtsFunction` to bind to the clock “GetTime” command and declare an event handler (trigger event) associated to the interface used for the data source (e.g. sine wave generator).

Listing 14: `displayTask.h`: with clock device

```

23 // functions which will be bound to commands
    mtsFunctionRead GetData;
25 mtsFunctionRead GetClockData;
    mtsFunctionWrite SetAmplitude;
27 mtsFunctionWrite SetTriggerValue;
    mtsFunctionVoid ResetTrigger;
29
    // event handler
31 void HandleTrigger(const cmnDouble & value);

```

In the implementation, the constructor has been updated to register the event handler and add one required interface for the clock:

Listing 15: `displayTask.cpp`: constructor with clock device

```

15 AddRequiredInterface("DataGenerator");
    // create an event handler associated to the output port. false
17 // means not queued.
    AddEventHandlerWrite(&displayTask::HandleTrigger, this,
19                      "DataGenerator", "HandleTrigger", this->Data, false);
    AddRequiredInterface("Clock");

```

The last parameter used for `AddEventHandlerWrite` determines if the event command is queued or not. If the event is queued, the associated callback will have to be dequeued by the user task itself and it will run in the user thread space. If the event command is not queued, it will be executed immediately, in the resource thread. This means that the callback implementation (e.g. `displayTask::HandleTrigger`) must be thread safe and not use any data member of the user task without some kind of safety mechanism (mutex, semaphore, ...). In this example, we use a non queued event handler to wake up the display task thread along with a semaphore for thread safety. Example 3 will show how to create and manage queued event handlers (both void and write events).

The `Startup` method has been updated to add the different commands to set and reset the trigger value. To associate the event handler to the event, we need to call `AddObserverToRequiredInterface` with the required interface name, the name of the event and the name of the event handler. We also need to check that a provided interface has been connected to the resource clock port and if so retrieve the “GetTime” command.

Listing 16: `displayTask.cpp`: `Startup` method with clock device

```

40     mtsDeviceInterface * dataInterface = GetProvidedInterfaceFor("DataGenerator");
41     if (dataInterface) {
42         GetData.Bind(dataInterface, "GetData");
43         CMN_LOG_CLASS(3) << "Startup:␣GetData␣function:␣"
44             << GetData << std::endl;
45         SetAmplitude.Bind(dataInterface, "SetAmplitude");
46         CMN_LOG_CLASS(3) << "Startup:␣SetAmplitude␣function:␣"
47             << SetAmplitude << std::endl;
48         SetTriggerValue.Bind(dataInterface, "SetTriggerValue");
49         CMN_LOG_CLASS(3) << "Startup:␣SetTriggerValue␣function:␣"
50             << SetTriggerValue << std::endl;
51         ResetTrigger.Bind(dataInterface, "ResetTrigger");
52         CMN_LOG_CLASS(3) << "Startup:␣ResetTrigger␣function:␣"
53             << ResetTrigger << std::endl;
54         SetAmplitude(Amplitude);
55         SetTriggerValue(TriggerValue);
56         AddObserverToRequiredInterface("DataGenerator", "TriggerEvent", "HandleTrigger");
57     } else {
58         CMN_LOG_CLASS(1) << "Startup:␣can␣not␣find␣interface␣for␣required␣interface␣DataGenerato
59             << std::endl;
60         exit(-1);
61     }
62     mtsDeviceInterface * clockInterface = GetProvidedInterfaceFor("Clock");
63     if (clockInterface) {
64         GetClockData.Bind(clockInterface, "GetTime");
65         CMN_LOG_CLASS(3) << "GetClockData␣function:␣"
66             << GetClockData << std::endl;
67     } else {
68         CMN_LOG_CLASS(1) << "Startup:␣can␣not␣find␣interface␣for␣resource␣port␣Clock"
69             << std::endl;
70         exit(-1);
71     }

```

4.4 Main program

The main program now needs to create the clock device, add it to the task manager and make sure the device is connected to the display task.

Listing 17: main.cpp: second example

```

29     // create the task manager and the tasks/devices
30     mtsTaskManager * taskManager = mtsTaskManager::GetInstance();
31     sineTask * sineTaskObject =
32         new sineTask("SIN", PeriodSine * cmn_ms);
33     clockDevice * clockDeviceObject =
34         new clockDevice("CLOC");
35     displayTask * displayTaskObject =
36         new displayTask("DISP", PeriodDisplay * cmn_ms);
37     displayTaskObject->Configure();
38     // add the tasks to the task manager and connect them
39     taskManager->AddTask(sineTaskObject);
40     taskManager->AddDevice(clockDeviceObject);
41     taskManager->AddTask(displayTaskObject);
42     taskManager->Connect("DISP", "DataGenerator", "SIN", "MainInterface");
43     taskManager->Connect("DISP", "Clock", "CLOC", "MainInterface");

```

The task collaboration graph is shown in figure 2.

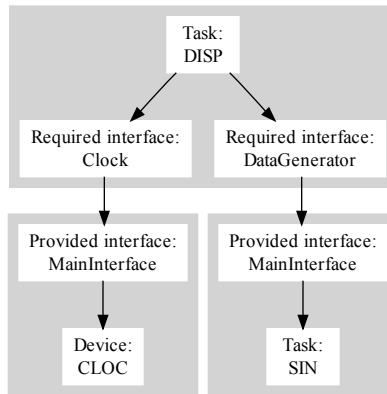


Figure 2: Example 2: tasks graph

5 Adding multiple interfaces, complex data types and qualified read commands

This example introduces:

- Tasks or devices with multiple provided and required interfaces.
- Multiples tasks connected to a single *task* provided interface. It is important to note that connecting multiple tasks to a single *device* interface is not thread safe unless the device itself is thread safe.
- Using argument’s prototypes to work with data types requiring deep copy. In this example, we will use the `mtsVector` type which requires a dynamic memory allocation during the configuration phase.
- Queued void and write events.
- Using the state table index to retrieve state data from the history.

5.1 Multiple provided and required interfaces

The class diagram for this example is shown in figure 3. The low level task, “RobotControl” emulates a robot controller with two arms (“Robot1” and “Robot2”). Each arm has two provided interfaces, a control interface which allows to move the robot as well as read the current position and stop it (sort of a read-write interface). The observer provided interface allows to read the robot

position and stop it in case of emergency but it doesn't provide a command to move the robot (sort of a read-only interface).

On top of the robot controller, we are using two instantiation of a display class. The display task has two required interfaces, "ControlledRobot" and "ObservedRobot". When the two display objects are connected to the robot controller, we simply swap which required interface is connected to which provided one.

The last task, the safety "Monitor" shows that multiple tasks can use the same provided interface without losing thread safety.

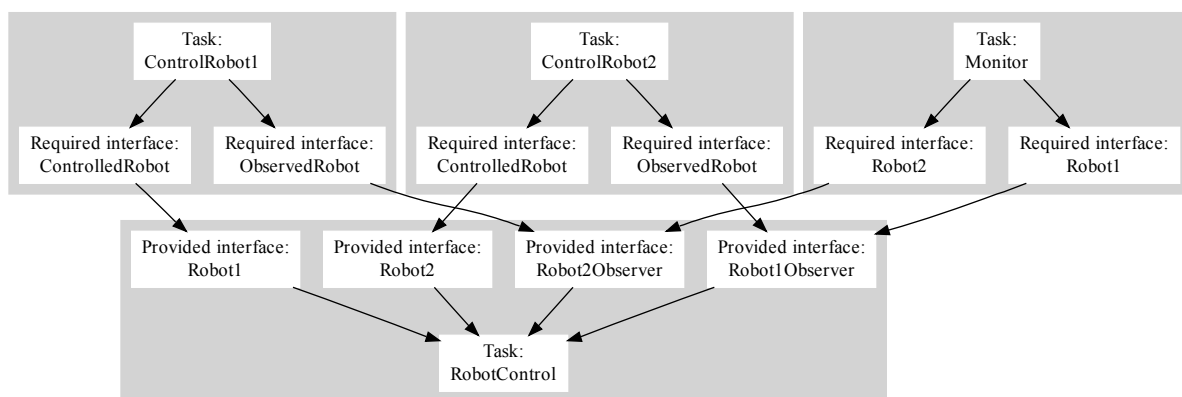


Figure 3: Example 3: tasks graph

5.2 Complex data types

So far the data types we used were simple types as defined by `cmnDouble`, `cmnULong`, ... In general one will need more complex types such as classes possibly containing pointers corresponding to dynamic memory allocation. For these types, a shallow copy (C++ default copy constructor) might not work as memory needs to be allocated for each newly created object. When a new type is to be added, the programmer must make sure that it is derived from `cmnGenericObject` and the copy constructor is correct (i.e. performs a deep copy if needed).

Internally, `cisstMultiTask` makes copies of objects for the state table or the queues of parameters (void and write commands). When using a complex type, the `cisstMultiTask` methods allow to provide a sample (also called argument prototype) which will be used to create all internal copies (for the curious programmers, we use the "in-place" copy constructor). In this example, we first declare the state data object in the header file:

Listing 18: `robotLowLevel.h`

```

19 class robotLowLevel: public mtsTaskPeriodic {
    CMN_DECLARE_SERVICES(CMN_NO_DYNAMIC_CREATION, 10);
21 public:

```

```

23     enum {NB_JOINTS = 2};
24     typedef mtsDoubleVec PositionJointType;
protected:
25     mtsStateData<PositionJointType> GoalJointRobot1;
26     mtsStateData<PositionJointType> GoalJointRobot2;
27     mtsStateData<PositionJointType> PositionJointRobot1;
28     mtsStateData<PositionJointType> PositionJointRobot2;

```

The data type `mtsDoubleVec` is equivalent to `mtsVector<double>` which is derived from both `cmnGenericObject` and `vctDynamicVector`. For more details on `vctDynamicVector`, consult the [cisstVector quickstart](#).

In the constructor, we first need to “configure” the argument prototypes. In this case we need to set the size correctly using the `SetSize` method inherited from `vctDynamicVector`.

We can then add the multiple provided interfaces and commands using our robot joint types with the correct size.

Listing 19: `robotLowLevel.cpp`: constructor

```

11 robotLowLevel::robotLowLevel(const std::string & taskName, double period):
12     mtsTaskPeriodic(taskName, period, false, 5000)
13 {
14     // set all vectors to the right size
15     GoalJointRobot1.Data.SetSize(NB_JOINTS);
16     GoalJointRobot2.Data.SetSize(NB_JOINTS);
17     PositionJointRobot1.Data.SetSize(NB_JOINTS);
18     PositionJointRobot2.Data.SetSize(NB_JOINTS);
19     DeltaJointRobot1.SetSize(NB_JOINTS);
20     DeltaJointRobot2.SetSize(NB_JOINTS);
21
22     // create 4 interfaces, two for each robot
23     AddProvidedInterface("Robot1");
24     AddProvidedInterface("Robot1Observer");
25     AddProvidedInterface("Robot2");
26     AddProvidedInterface("Robot2Observer");
27     // add the state data to the table
28     PositionJointRobot1.AddToStateTable(StateTable, "PositionJointRobot1");
29     PositionJointRobot2.AddToStateTable(StateTable, "PositionJointRobot2");
30     // add a method to read the current state index
31     AddCommandRead(&mtsStateTable::GetIndexReader, &StateTable,
32                   "Robot1Observer", "GetStateIndex");
33     AddCommandRead(&mtsStateTable::GetIndexReader, &StateTable,
34                   "Robot2Observer", "GetStateIndex");
35     // provide read method to all 4 interfaces
36     PositionJointRobot1.AddReadCommandToTask(this,
37                                               "Robot1", "GetPositionJoint");
38     PositionJointRobot1.AddReadCommandToTask(this,
39                                               "Robot1Observer", "GetPositionJoint");
40     PositionJointRobot2.AddReadCommandToTask(this,
41                                               "Robot2", "GetPositionJoint");
42     PositionJointRobot2.AddReadCommandToTask(this,
43                                               "Robot2Observer", "GetPositionJoint");
44     // provide write methods to the controlling interfaces
45     // requires: method, object carrying the method, interface name, command name
46     // and argument prototype
47     AddCommandWrite(&robotLowLevel::MovePositionJointRobot1, this,

```

```

48         "Robot1", "MovePositionJoint", PositionJointRobot1.Data);
AddCommandWrite(&robotLowLevel::MovePositionJointRobot2, this,
50         "Robot2", "MovePositionJoint", PositionJointRobot2.Data);
AddCommandVoid(&robotLowLevel::StopRobot1, this,
52         "Robot1", "StopRobot");
AddCommandVoid(&robotLowLevel::StopRobot1, this,
54         "Robot1Observer", "StopRobot");
AddCommandVoid(&robotLowLevel::StopRobot2, this,
56         "Robot2", "StopRobot");
AddCommandVoid(&robotLowLevel::StopRobot2, this,
58         "Robot2Observer", "StopRobot");
// define events, provide argument prototype for write events
60 MotionFinishedRobot1.Bind(AddEventWrite("Robot1", "MotionFinished",
PositionJointRobot1.Data));
62 MotionFinishedRobot2.Bind(AddEventWrite("Robot2", "MotionFinished",
PositionJointRobot2.Data));
64 MotionStartedRobot1.Bind(AddEventVoid("Robot1", "MotionStarted"));
MotionStartedRobot2.Bind(AddEventVoid("Robot2", "MotionStarted"));
66 }

```

5.3 Using the state table index

In the previous listing, we also added a read command named “GetStateIndex” based on the state table method `GetIndexReader`. So far we used the `AddReadCommandToTask` method to make state data readable via an interface. This provides a convenient and easy way to read the latest available data in the table, but this will not suffice for all users. Instead, one might want to retrieve older data or make sure the data is all synchronized. In this case, sequential collection of the latest available data might not work as the underlying task could have incremented its counter. In this example, we decided to retrieve the latest data as well as the prior data using the state index.

Listing 20: `monitorTask.h`

```

14 class monitorTask: public mtsTaskPeriodic {
15     CMN_DECLARE_SERVICES(CMN_NO_DYNAMIC_CREATION, 5);
public:
17     enum {NB_JOINTS = 2};
    typedef mtsDoubleVec PositionJointType;
19 protected:
    mtsFunctionRead GetStateIndex1, GetStateIndex2;
21     mtsFunctionQualifiedRead GetPositionJoint1, GetPositionJoint2;
    mtsFunctionVoid StopRobot1, StopRobot2;
23     PositionJointType CurrentPosition1, CurrentPosition2;
    PositionJointType PreviousPosition1, PreviousPosition2;
25     mtsStateIndex StateIndex;

```

In the class declaration, note that we are using the same type for the robot joints and declared a `mtsFunctionQualifiedRead` which will allow us to specify the state index when retrieving the data from the state table.

The `Run` method can now retrieve the state index and use it to get the data from the state table by index:

Listing 21: `monitorTask.cpp: Run method`

```

47 // check the positions of both robots
   GetStateIndex1(StateIndex); // current time index of robot 1 state table
49 if (StateIndex.Index() > 1) { // make sure we have two elements in the table
       GetPositionJoint1(StateIndex, CurrentPosition1); // current data
51       GetPositionJoint1(StateIndex - 1, PreviousPosition1); // current - 1 data
       if (CurrentPosition1 != PreviousPosition1) { // use the vctDynamicVector !=
53           if ((!CurrentPosition1.Greater(lowerBound))
               || (!CurrentPosition1.Lesser(upperBound))) {
55               CMN_LOG_CLASS(1) << "Run:robot1outofbounds" << std::endl;
               StopRobot1();
57           }
       }
59 }

```

5.4 Main program

When connecting the tasks, note that the required interfaces are swapped when connected to the provided interfaces of the robot controller:

Listing 22: main.cpp: third example

```

33 // add all tasks
   taskManager->AddTask(robotTask);
35   taskManager->AddTask(monitor);
   taskManager->AddTask(appTaskControl1);
37   taskManager->AddTask(appTaskControl2);
   // connect: name of user, resource port, name of resource, resource interface
39   taskManager->Connect("Monitor", "Robot1",
                       "RobotControl", "Robot1Observer");
41   taskManager->Connect("Monitor", "Robot2",
                       "RobotControl", "Robot2Observer");
43   taskManager->Connect("ControlRobot1", "ControlledRobot",
                       "RobotControl", "Robot1");
45   taskManager->Connect("ControlRobot1", "ObservedRobot",
                       "RobotControl", "Robot2Observer");
47   taskManager->Connect("ControlRobot2", "ControlledRobot",
                       "RobotControl", "Robot2");
49   taskManager->Connect("ControlRobot2", "ObservedRobot",
                       "RobotControl", "Robot1Observer");

```

6 Using a task or device without creating a new task

In the previous examples we always created a task for the user interface. This is not a requirement and it is possible to use a task from a plain function (i.e. main). We can list a few reasons to not use a task for the user interface (or any other “top task”):

- The top task is not used by anyone as a resource and therefore doesn’t need any provided interface or command.
- A user interface is unlikely to need a state table.

- Some user interface toolkits or application frameworks have their own event loop and this is somewhat redundant or even incompatible with the periodic `Run` method of `mtsTaskPeriodic`.

On the other hand, the class `mtsTaskPeriodic` provides a required interface and can be connected easily using the task manager. If one decides to not use a task, these features have to be implemented by hand. It is *very important* to note that some built-in thread safety mechanisms are being bypassed (e.g. mailboxes for queued events) and the programmer will have to pay *special attention to critical sections*.

In this example we are going to use the exact same robot task as before. The user interface will be simpler in that we are only going to use one of the robot's arms.

6.1 User interface

Using the task manager to connect tasks allows to retrieve commands from the resource provided interface and subscribe to events. Since we are not using the task manager, we need to re-implement these steps. We first need to declare some function objects and create the callbacks used for the events:

Listing 23: `userInterface.h`: some data members

```

29 // Use cisstMultiTask function objects
30 mtsFunctionRead GetPositionJoint;
31 mtsFunctionWrite MovePositionJoint;
32
33 // mts events callbacks, in this example started event is void,
34 // end event is write
35 void CallbackStarted(void);
36 mtsCommandVoidBase * CallbackStartedCommand;
37 void CallbackFinished(const PositionJointType &);
38 mtsCommandWriteBase * CallbackFinishedCommand;

```

In the constructor, we assume that we already have a pointer on the correct provided interface. We then need to retrieve the commands by name, create commands for our event callbacks and then add them as observers.

Listing 24: `userInterface.cpp`: constructor

```

27 // tell task or device that this thread will use it, will create a
28 // mailbox if needed. if this method is not called, call to
29 // GetCommandXyz will likely fail
30 interfacePointer->AllocateResourcesForCurrentThread();
31 // now get a handle on the different commands we want to use
32 if (!GetPositionJoint.Bind(
33     interfacePointer->GetCommandRead("GetPositionJoint")
34 )) {
35     CMN_LOG(1) << "Interface provided does have the method \"GetPositionJoint\"
36     << std::endl;
37 }
38 if (!MovePositionJoint.Bind(
39     interfacePointer->GetCommandWrite("MovePositionJoint")
40 )) {
41     CMN_LOG(1) << "Interface provided does have the method \"MovePositionJoint\"

```

```

42         << std::endl;
    }
44     // create the command to handle start event and add it as observer
    CallbackStartedCommand =
46         new mtsCommandVoidMethod<userInterface>(&userInterface::CallBackStarted,
                                                    this,
48                                                    "MyStartedCallBack");
    if (!interfacePointer->AddObserver("MotionStarted", CallbackStartedCommand)) {
50         CMN_LOG(1) << "Interface provided does have the event \"MotionStarted\"
        << std::endl;
52     }
    // create the command to handle finished event and add it as observer
54     CallbackFinishedCommand =
        new mtsCommandWrite<userInterface,
56                 PositionJointType>(&userInterface::CallBackFinished,
                                     this,
58                                     "MyFinishedCallBack",
                                     Position);
60     if (!interfacePointer->AddObserver("MotionFinished", CallbackFinishedCommand)) {
        CMN_LOG(1) << "Interface provided does have the event \"MotionFinished\"
62         << std::endl;
    }
}

```

Note that it is possible to get a print-out of the current configuration of a task or interface using the stream out operator (<<). This includes the list of registered observers for each event as well as all the available commands.

One important thing to note in this example is that the callbacks associated to the underlying task events are not queued. This means that when the event will occur at the task level, the callback will be called by the task itself in its own thread. As our callbacks modify the user interface which is also manipulated by the main thread, it is important to use a mutex for all the UI calls to ensure thread safety.

Listing 25: userInterface.cpp: Event callback methods

```

115 void userInterface::CallBackStarted(void)
    {
117     // mutex is important as the callback method will be called from
    // the low level task thread
119     Mutex.Lock();
    Moving->label("@>");
121     Mutex.Unlock();
    }
123
124 void userInterface::CallBackFinished(const PositionJointType & finalPosition)
125 {
127     // mutex is important as the callback method will be called from
    // the low level task thread
129     Mutex.Lock();
    Moving->label("@||");
    Position1Window->value(finalPosition[0]);
131     Position2Window->value(finalPosition[1]);
    Mutex.Unlock();
133 }

```

The update method looks a lot like the Run method of the previous examples except that we don't process commands nor events:

Listing 26: userInterface.cpp: Update method

```

135 void userInterface::Update(void)
136 {
137     // mutex is used as the interface (as well as any data member of
138     // this class) can be used by the current thread as well as the
139     // low level task thread
140     Mutex.Lock();
141     GetPositionJoint(Position);
142     Position1Window->value(Position[0]);
143     Position2Window->value(Position[1]);
144     TicksWindow->value(Ticks++);
145     Fl::check(); // all the FTLK events
146     Mutex.Unlock();
147 }

```

6.2 Main program

The main program creates the robot task, look for a robot provided interface named “Robot1” and then creates the user interface. Once the user interface is created, it loops as long as the close button has not been pressed. In the loop, we periodically call the user interface Update method.

Listing 27: main.cpp: fourth example

```

25 // create a single task
26 mtsTaskManager * taskManager = mtsTaskManager::GetInstance();
27 robotLowLevel * robotTask = new robotLowLevel("RobotControl", 100 * cmn_ms);
28 // display a human readable description of the task, its
29 // interfaces with commands provided and possible events
30 std::cout << *robotTask << std::endl;
31
32 // add single task, do not connect anything
33 taskManager->AddTask(robotTask);
34 // create the thread for the task
35 taskManager->CreateAll();
36
37 // look for the interface we are going to use
38 mtsDeviceInterface * robotInterface = robotTask->GetProvidedInterface("Robot1");
39 userInterface * UI = 0;
40 if (robotInterface) {
41     // instantiate the UI in the current thread
42     UI = new userInterface("Robot1", robotInterface);
43     // display again to show registered callbacks
44     std::cout << *robotTask << std::endl;
45 } else {
46     CMN_LOG(1) << "It looks like there is no \"Robot1\" interface" << std::endl;
47 }
48
49 // start the task
50 taskManager->StartAll();
51
52 // loop while the UI did not get a close request

```

```

53  while (!UI->CloseRequested) {
      // the user interface method Update is equivalent to the task
55  // Run method
      UI->Update();
57  osaSleep(10 * cmn_ms); // ask for an update every 10 ms
  }

```

7 Lists of figures and listings

List of Figures

1	Example 1: tasks graph	9
2	Example 2: tasks graph	14
3	Example 3: tasks graph	15

Listings

1	sineTask.h	4
2	sineTask.cpp: constructor	4
3	sineTask.cpp: Run method	5
4	displayTask.h	5
5	displayTask.cpp: constructor	6
6	displayTask.cpp: Startup method	6
7	displayTask.cpp: Run method	7
8	main.cpp: first example	7
9	sineTask.h: event data members	9
10	sineTask.cpp: constructor with event	10
11	sineTask.cpp: Run method with event	10
12	clockDevice.h	11
13	clockDevice.cpp	11
14	displayTask.h: with clock device	12
15	displayTask.cpp: constructor with clock device	12
16	displayTask.cpp: Startup method with clock device	12
17	main.cpp: second example	13
18	robotLowLevel.h	15
19	robotLowLevel.cpp: constructor	16
20	monitorTask.h	17
21	monitorTask.cpp: Run method	17
22	main.cpp: third example	18
23	userInterface.h: some data members	19
24	userInterface.cpp: constructor	19
25	userInterface.cpp: Event callback methods	20
26	userInterface.cpp: Update method	20
27	main.cpp: fourth example	21