

ERC CISST Software

Quick start for cisstMultiTask

Anton Deguet

Date: 2009/01/10 05:02:51

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	4
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	8
4.1	Sine wave generator task	9
4.2	Clock device	11
4.3	Display task	11
4.4	Main program	12
5	Adding multiple interfaces, complex data types and qualified read commands	13
5.1	Multiple provided and required interfaces	14
5.2	Complex data types	14
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	20
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`, `osa` and `vct` come from `cisstCommon`, `cisstOSAbstraction` and `cisstVector`, 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`, `mtsTaskPeriodic...`
- `mtsDeviceInterface` and `mtsTaskInterface` (provided interfaces)
- `mtsRequiredInterface`
- `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 compile-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 containing the commands is called the *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 does not 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.

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. To summarize, all void and write commands are queued by default, i.e. the execution of the actual method or function will occur in the thread space of the task who dequeues the command. All read and qualified read commands are not queued, i.e. they are executed in the thread space of the caller. Programmers will have to be careful when creating read commands not based on the state table (see next section).

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, `mtsGenericObject`, is defined in *cisstMultiTask*. Convenience types are also defined for native types such as `mtsDouble`, `mtsInt...`. These classes are mere proxies to the actual data which can be accessed using the `Data` data member. For non native types, *cisstMultiTask* provides `mtsVector` (derived from `vctDynamicVector`), `mtsMatrix` (derived from `vctDynamicMatrix`) and many transformation types (e.g. `mtsDoubleQuatRot3`, `mtsDoubleMatFrm3`, ...). The later types use multiple inheritance so they can be used as their *cisstVector* counterparts.

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 may have 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, "Run Error" by default
13     CMN_DECLARE_SERVICES(CMN_NO_DYNAMIC_CREATION, CMN_LOG_LOD_RUN_ERROR);
    protected:
15     // data generated by the sine wave generator
        double SineData;
17     // this is the amplitude of the sine wave
        double SineAmplitude;
19
21     void SetAmplitude(const mtsDouble & amp) { SineAmplitude = amp.Data; }
22
23 public:
    // provide a name for the task and define the frequency (time
    // interval between calls to the periodic Run). Also used to
25     // populate the interface(s)
        sineTask(const std::string & taskName, double period);
27     ~sineTask() {};
    // all four methods are pure virtual in mtsTask
29     void Configure(const std::string & CMN_UNUSED(filename)) {};
        void Startup(void); // set some initial values
31     void Run(void); // performed periodically
        void Cleanup(void) {}; // user defined cleanup
33 };

```

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 `mtsDouble` which is derived from `mtsGenericObject`.

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    StateTable.AddData(SineData, "SineData");
    // add one interface, this will create an mtsInterfaceProvided
19    mtsInterfaceProvided * provided = AddInterfaceProvided("MainInterface");
    if (provided) {
21        // add command to access state table values to the interface
        provided->AddCommandReadState(StateTable, SineData, "GetData");
23        // add command to modify the sine amplitude
        provided->AddCommandWrite(&sineTask::SetAmplitude, this, "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

```

32 void sineTask::Run(void) {
33    // process the commands received, i.e. possible SetSineAmplitude
    ProcessQueuedCommands();
35    // compute the new values based on the current time and amplitude
    SineData = SineAmplitude
37        * sin(2 * cmnPI * static_cast<double>(this->GetTick()) * Period / 10.0);
    //SineData.SetTimestamp(StateTable.GetTic());
39    //SineData.SetValid(true);
    std::cout << SineData << std::endl;
}

```

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 {
12    // set log level to "Run Error"
    CMN_DECLARE_SERVICES(CMN_NO_DYNAMIC_CREATION, CMN_LOG_LOD_RUN_ERROR);
}

```

```

14 protected:
15     // local copy of data used in commands
16     double Data;
17     mtsDouble AmplitudeData;
18
19     struct {
20         // functions which will be bound to commands
21         mtsFunctionRead GetData;
22         mtsFunctionWrite SetAmplitude;
23     } Generator;
24
25     // user interface generated by FTLK/fluid
26     displayUI UI;
27
28 public:
29     // see sineTask.h documentation
30     displayTask(const std::string & taskName, double period);
31     ~displayTask() {};
32     void Configure(const std::string & CMN_UNUSED(filename) = "");
33     void Startup(void);
34     void Run(void);
35     void Cleanup(void) {};
36 };

```

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)
13 {
14     // to communicate with the interface of the resource
15     mtsInterfaceRequired * required = AddInterfaceRequired("DataGenerator");
16     if (required) {
17         required->AddFunction("GetData", Generator.GetData);
18         required->AddFunction("SetAmplitude", Generator.SetAmplitude);
19     }
20 }

```

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

```

37 void displayTask::Startup(void)
38 {
39     // make the UI visible
40     UI.show(0, NULL);
41 }

```

The `Startup` method is called just before the task begins periodic execution. In this case, it initializes the `AmplitudeData` and displays the GUI.

Listing 7: displayTask.cpp: Run method

```

43 void displayTask::Run(void)
44 {
45     // get the data from the sine wave generator task
46     Generator.GetData(Data);
47     UI.Data->value(Data);
48     // check if the user has entered a new amplitude in UI
49     if (UI.AmplitudeChanged) {
50         // retrieve the new amplitude and send it to the sine task
51         AmplitudeData = UI.Amplitude->value();
52         AmplitudeData.SetTimestamp(mtsTaskManager::GetInstance()
53                                 ->GetTimeServer().GetRelativeTime());
54         AmplitudeData.SetValid(true);
55         // send it
56         Generator.SetAmplitude(AmplitudeData);
57         UI.AmplitudeChanged = false;
58         CMN_LOG_CLASS_RUN_VERBOSE << "Run:_" << this->GetTick()
59                                 << "_-_" << AmplitudeData << std::endl;
60     }
61     // log some extra information
62     CMN_LOG_CLASS_RUN_VERBOSE << "Run:_" << this->GetTick()
63                             << "_-_" << Data << std::endl;
64     // update the UI, process UI events
65     if (Fl::check() == 0) {
66         Kill();
67     }
68 }

```

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
30 const double PeriodSine = 1 * cmn_ms; // in milliseconds
31 const double PeriodDisplay = 50 * cmn_ms; // in milliseconds
32 mtsTaskManager * taskManager = mtsTaskManager::GetInstance();
33 sineTask * sineTaskObject = new sineTask("SIN", PeriodSine);
34 displayTask * displayTaskObject = new displayTask("DISP", PeriodDisplay);
35 displayTaskObject->Configure();
36
37 // add the tasks to the task manager
38 taskManager->AddComponent(sineTaskObject);
39 taskManager->AddComponent(displayTaskObject);
40
41 // connect the tasks, task.RequiresInterface -> task.ProvidesInterface
42 taskManager->Connect("DISP", "DataGenerator", "SIN", "MainInterface");
43
44 // generate a nice tasks diagram

```

```

46     std::ofstream dotFile("example1.dot");
        taskManager->ToStreamDot(dotFile);
        dotFile.close();
48
        // create the tasks, i.e. find the commands
50     taskManager->CreateAll();
        // start the periodic Run
52     taskManager->StartAll();
54
        // wait until the close button of the UI is pressed
        int cnt = 0;
56     while (!displayTaskObject->IsTerminated()) {
            if (cnt++ == 3) {
58         std::cout << "#####_SUSPEND" << std::endl;
                sineTaskObject->Suspend();
60         } else if (cnt == 8) {
                std::cout << "#####_RESUME" << std::endl;
62         sineTaskObject->Start();

```

The first part of the main function (not shown) 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.

4 Adding events and a device

For the second example we are adding events and a plain device. An event allows communication 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 provided 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. When the tasks are connected, the system will automatically bind the user event handler to the event.

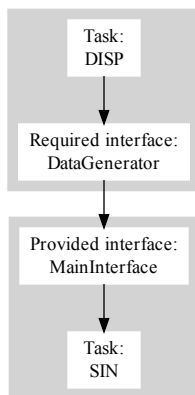


Figure 1: Example 1: tasks graph

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.

Finally this example shows how to save or log some of the data stored in the state table using the `mtsCollectorState`. This class allows to select which “signal” to save (i.e. column of the state table) and define a subsampling (save all or every so often sample). The data is stored in a file in binary or text mode (comma or space separated) when everytime the state stable fills up. As this is performed in a separate thread, the data collection has a limited impact on the application.

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

```

24 // method used to reset the trigger, we will add a command to the
25 // interface for this method
void ResetTrigger(void);
27 // function bound to the command used to send the event, one could
// use a command instead but this is somewhat more convenient
29 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):
  
```

```

    mtsTaskPeriodic(taskName, period, false, 5000)
12 {
    StateTable.AddData(SineData, "SineData");
14    mtsInterfaceProvided * mainInterface = AddInterfaceProvided("MainInterface");
    if (mainInterface) {
16        // add commands to access state table values
        mainInterface->AddCommandReadState(StateTable, SineData, "GetData");
18        // add (queued) commands to set member data
        mainInterface->AddCommandWrite(&sineTask::SetAmplitude, this, "SetAmplitude");
20        mainInterface->AddCommandWrite(&sineTask::SetTrigger, this, "SetTriggerValue");
        // add a command bound to a user defined method
22        mainInterface->AddCommandVoid(&sineTask::ResetTrigger, this, "ResetTrigger");
        // define an event and setup our event sending function
24        mtsDouble eventData; // data type used for the event payload
        mainInterface->AddEventWrite(TriggerEvent, "TriggerEvent", eventData);
26    }
}

```

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

```

41 void sineTask::Run(void) {
    ProcessQueuedCommands();
43    SineData = SineAmplitude
        * sin(2 * cmnPI * static_cast<double>(this->GetTick()) * Period / 10.0);
45    // check if the trigger is enabled and if the conditions are right
    // to send an event
47    if (TriggerEnabled) {
        if (SineData >= TriggerValue) {
49            // use the mtsFunctionWrite to send the event along with
            // the current data
51            TriggerEvent(SineData);
            TriggerEnabled = false;
53        }
    }
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 mtsComponent {
    CMN_DECLARE_SERVICES(CMN_NO_DYNAMIC_CREATION, CMN_LOG_LOD_RUN_ERROR);
13
    protected:
15     osaStopwatch Timer; // this is the actual device (wrapped)
        void GetTime(mtsDouble & time) const; // used by the command "GetTime"
17
    public:
19     // constructor doesn't need a period!
        clockDevice(const std::string & deviceName);
21     ~clockDevice() {};
        void Configure(const std::string & CMN_UNUSED(filename) = "") {};
23     // no Startup, Run, Cleanup required
};

```

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     mtsComponent(deviceName) {
    mtsInterfaceProvided * mainInterface = AddInterfaceProvided("MainInterface");
13     mainInterface->AddCommandRead(&clockDevice::GetTime, this, "GetTime");
    Timer.Reset(); // reset the clock
15     Timer.Start(); // start the clock
}
17
void clockDevice::GetTime(mtsDouble & time) const
19 {
    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

```

22 // functions which will be bound to commands

```

```

    mtsFunctionRead GetData;
24    mtsFunctionWrite SetAmplitude;
    mtsFunctionWrite SetTriggerValue;
26    mtsFunctionVoid ResetTrigger;
    mtsEventReceiverWrite TriggerEvent;
28    } Generator;

30    struct ClockStruct {
        mtsFunctionRead GetClockData;
32    } Clock;

34    // event handler
    void HandleTrigger(const mtsDouble & 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

```

14    mtsInterfaceRequired * required = AddInterfaceRequired("DataGenerator");
    if (required) {
16        required->AddFunction("GetData", Generator.GetData);
        required->AddFunction("SetAmplitude", Generator.SetAmplitude);
18        required->AddFunction("SetTriggerValue", Generator.SetTriggerValue);
        required->AddFunction("ResetTrigger", Generator.ResetTrigger);
20        required->AddEventReceiver("TriggerEvent", Generator.TriggerEvent);
    }
22    required = AddInterfaceRequired("Clock");
    if (required) {
24        required->AddFunction("GetTime", Clock.GetClockData);
    }
26 }

28 displayTask::~displayTask()

```

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).

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 16: main.cpp: second example

```

35    // create the task manager and the tasks/devices
36    mtsTaskManager * taskManager = mtsTaskManager::GetInstance();

```

```

sineTask * sineTaskObject = new sineTask("SIN", PeriodSine);
38 clockDevice * clockDeviceObject = new clockDevice("CLOC");
displayTask * displayTaskObject = new displayTask("DISP", PeriodDisplay);
40 displayTaskObject->Configure();
// add the tasks to the task manager and connect them
42 taskManager->AddComponent(sineTaskObject);
taskManager->AddComponent(clockDeviceObject);
44 taskManager->AddComponent(displayTaskObject);
taskManager->Connect("DISP", "DataGenerator", "SIN", "MainInterface");
46 taskManager->Connect("DISP", "Clock", "CLOC", "MainInterface");

```

Before all the tasks get started, it is possible to add a state collector to collect all or part of the data stored in the state table. This requires to first create an object of type `mtsCollectorState` and then specify which state data to collect (in our example, the data “SineData”).

Listing 17: main.cpp: second example, setting up data collection

```

51 // add data collection for sineTask state table
52 mtsCollectorState * collector = 0;
if (choice == 'b') {
54     collector =
        new mtsCollectorState("SIN",
56                             sineTaskObject->GetDefaultStateTableName(),
                             mtsCollectorBase::COLLECTOR_FILE_FORMAT_BINARY);
58 } else if (choice == 't') {
    collector =
60     new mtsCollectorState("SIN",
                             sineTaskObject->GetDefaultStateTableName(),
62                             mtsCollectorBase::COLLECTOR_FILE_FORMAT_PLAIN_TEXT);
} else if (choice == 'c') {
64     collector =
        new mtsCollectorState("SIN",
66                             sineTaskObject->GetDefaultStateTableName(),
                             mtsCollectorBase::COLLECTOR_FILE_FORMAT_CSV);
68 }
// specify which signal (aka state data) to collect

```

The task collaboration graph is shown in figure 2.

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

This example introduces:

- Tasks or devices with multiple provided and required interfaces.
- Multiple 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 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.

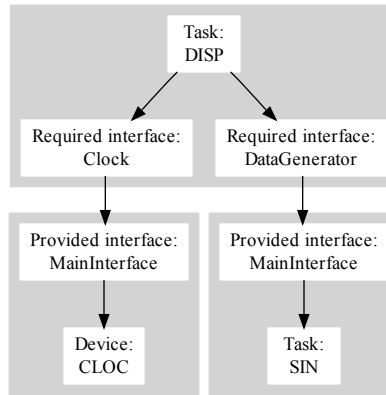


Figure 2: Example 2: tasks graph

- 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 instantiations 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.

5.2 Complex data types

So far the data types we used were simple types as defined by `mtsDouble`, `mtsULong`, ... 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 `mtsGenericObject` and the copy constructor is correct (i.e. performs a deep copy if needed).

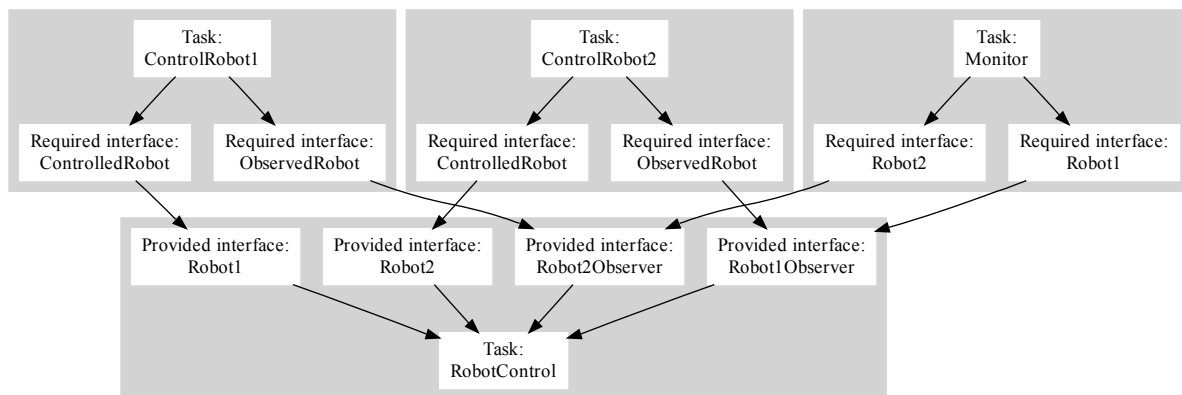


Figure 3: Example 3: tasks graph

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, CMN_LOG_LOD_VERY_VERBOSE);
21 public:
    enum {NB_JOINTS = 2};
23 typedef mtsDoubleVec PositionJointType;
protected:
25 PositionJointType GoalJointRobot1;
    PositionJointType GoalJointRobot2;
27 PositionJointType PositionJointRobot1;
    PositionJointType PositionJointRobot2;

```

The data type `mtsDoubleVec` is equivalent to `mtsVector<double>` which is derived from both `mtsGenericObject` 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, 500)

```

```

14  // set all vectors to the right size
    GoalJointRobot1.SetSize(NB_JOINTS);
16  GoalJointRobot2.SetSize(NB_JOINTS);
    PositionJointRobot1.SetSize(NB_JOINTS);
18  PositionJointRobot2.SetSize(NB_JOINTS);
    DeltaJointRobot1.SetSize(NB_JOINTS);
20  DeltaJointRobot2.SetSize(NB_JOINTS);

22  // create 4 interfaces, two for each robot
    mtsInterfaceProvided * robot1Interface =
24      AddInterfaceProvided("Robot1");
    mtsInterfaceProvided * robot1ObserverInterface =
26      AddInterfaceProvided("Robot1Observer");
    mtsInterfaceProvided * robot2Interface =
28      AddInterfaceProvided("Robot2");
    mtsInterfaceProvided * robot2ObserverInterface =
30      AddInterfaceProvided("Robot2Observer");
    // add the state data to the table
32  StateTable.AddData(PositionJointRobot1, "PositionJointRobot1");
    StateTable.AddData(PositionJointRobot2, "PositionJointRobot2");
34  // add a method to read the current state index
    robot1ObserverInterface->AddCommandRead(&mtsStateTable::GetIndexReader,
36      &StateTable,
        "GetStateIndex");
38  robot2ObserverInterface->AddCommandRead(&mtsStateTable::GetIndexReader,
        &StateTable,
40      "GetStateIndex");
    // provide read method to all 4 interfaces
42  robot1Interface->AddCommandReadState(StateTable, PositionJointRobot1,
        "GetPositionJoint");
44  robot1ObserverInterface->AddCommandReadState(StateTable, PositionJointRobot1,
        "GetPositionJoint");
46  robot2Interface->AddCommandReadState(StateTable, PositionJointRobot2,
        "GetPositionJoint");
48  robot2ObserverInterface->AddCommandReadState(StateTable, PositionJointRobot2,
        "GetPositionJoint");
50  // provide write methods to the controlling interfaces
    // requires: method, object carrying the method, interface name, command name
52  // and argument prototype
    robot1Interface->AddCommandWrite(&robotLowLevel::MovePositionJointRobot1, this,
54      "MovePositionJoint", PositionJointRobot1);
    robot2Interface->AddCommandWrite(&robotLowLevel::MovePositionJointRobot2, this,
56      "MovePositionJoint", PositionJointRobot2);
    robot1Interface->AddCommandVoid(&robotLowLevel::StopRobot1, this,
58      "StopRobot");
    robot1ObserverInterface->AddCommandVoid(&robotLowLevel::StopRobot1, this,
60      "StopRobot");
    robot2Interface->AddCommandVoid(&robotLowLevel::StopRobot2, this,
62      "StopRobot");
    robot2ObserverInterface->AddCommandVoid(&robotLowLevel::StopRobot2, this,
64      "StopRobot");
    // define events, provide argument prototype for write events
66  robot1Interface->AddEventWrite(MotionFinishedRobot1, "MotionFinished",
        PositionJointRobot1);

```

```

68     robot2Interface->AddEventWrite(MotionFinishedRobot2, "MotionFinished",
                                   PositionJointRobot2);
70     robot1Interface->AddEventVoid(MotionStartedRobot1, "MotionStarted");
71     robot2Interface->AddEventVoid(MotionStartedRobot2, "MotionStarted");
72 }

```

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, CMN_LOG_LOD_RUN_ERROR);
16 public:
17     typedef mtsDoubleVec PositionJointType;
18 protected:
19     // Robot[0] and Robot[1] are required interfaces
20     struct RobotRequired {
21         mtsFunctionRead GetStateIndex;
22         mtsFunctionQualifiedRead GetPositionJoint;
23         mtsFunctionVoid StopRobot;
24     } Robot[2];
25
26     PositionJointType CurrentPosition[2];
27     PositionJointType PreviousPosition[2];
28     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

```

36 void monitorTask::Run(void)
37 {
38     mtsExecutionResult result;
39     // check the positions of both robots
40     for (unsigned int i = 0; i < 2; i++) {
41         Robot[i].GetStateIndex(StateIndex); // time index of robot state table
42         Robot[i].GetPositionJoint(StateIndex, CurrentPosition[i]); // current data
43         result = Robot[i].GetPositionJoint(StateIndex - 1, PreviousPosition[i]);
44         if ((result == mtsExecutionResult::DEV_OK) &&
45             (CurrentPosition[i] != PreviousPosition[i])) {

```

```

48         if ((!CurrentPosition[i].Greater(lowerBound))
50             || (!CurrentPosition[i].Lesser(upperBound))) {
52             CMN_LOG_CLASS_RUN_WARNING << "Run:robot" << i+1 << "out of bounds"
54             << std::endl;
           Robot[i].StopRobot();
         }
     }
 }

```

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
34 // create and add Component Viewer
mtsComponentViewer * componentViewer = new mtsComponentViewer("ComponentViewer", 1.0*cmn_s);
36 if (!taskManager->AddComponent(componentViewer)) {
38     CMN_LOG_INIT_ERROR << "Failed to add ComponentViewer" << std::endl;
39     exit(1);
40 }
41
42 // add all tasks
43 taskManager->AddComponent(robotTask);
44 taskManager->AddComponent(monitor);
45 taskManager->AddComponent(appTaskControl1);
46 taskManager->AddComponent(appTaskControl2);
47 // connect: name of user, resource port, name of resource, resource interface
48 taskManager->Connect("Monitor", "Robot1",
49                     "RobotControl", "Robot1Observer");
50 taskManager->Connect("Monitor", "Robot2",
51                     "RobotControl", "Robot2Observer");

```

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` or `mtsTaskContinuous` 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 by-passed (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

```

28 // Use cisstMultiTask function objects
29 mtsFunctionRead GetPositionJoint;
    mtsFunctionWrite MovePositionJoint;
31
    // mts events callbacks, in this example started event is void,
33 // end event is write
    void CallbackStarted(void);
35 mtsCommandVoid * CallbackStartedCommand;
    void CallbackFinished(const PositionJointType &);
37 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

```

24 // -1- Initialize the required interface. In this implementation, we are not
    // using a mailbox for events, so the events aren't queued. As a result, we
26 // need to use a mutex in the event handlers (CallbackStarted and CallbackFinished).
    mtsInterfaceRequired Robot("Robot", 0);
    Robot.AddFunction("GetPositionJoint", GetPositionJoint);
    Robot.AddFunction("MovePositionJoint", MovePositionJoint);
30 // false --> Event handlers are not queued
    Robot.AddEventHandlerVoid(&userInterface::CallbackStarted, this,
32                             "MotionStarted");
    Robot.AddEventHandlerWrite(&userInterface::CallbackFinished, this,
34                             "MotionFinished");

36 // -2- Connect to the device/task that provides the required resources
    Robot.ConnectTo(interfacePointer);
38

    // -3- Setup the user interface
40 Window = new Fl_Double_Window(500, 400, Name.c_str());
    enum {row1 = 20, row2 = 60, row3 = 100, row4 = 140, row5 = 180, row6 = 220};
42 enum {col1 = 20, col2 = 100, col3 = 200, col4 = 300, col5 = 400};
    // create labels to make the interface readable

```

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

```

95     Mutex.Unlock();
96 }
97
98 void userInterface::CallBackFinished(const PositionJointType & finalPosition)
99 {
100     // mutex is important as the callback method will be called from
101     // the low level task thread
102     Mutex.Lock();
103     Moving->label("@||");
104     Position1Window->value(finalPosition[0]);
105     Position2Window->value(finalPosition[1]);
106     Mutex.Unlock();
107 }
108
109 void userInterface::Update(void)
110 {
111     // mutex is used as the interface (as well as any data member of
112     // this class) can be used by the current thread as well as the
113     // low level task thread

```

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

```

115     GetPositionJoint(Position);
116     Position1Window->value(Position[0]);
117     Position2Window->value(Position[1]);
118     TicksWindow->value(Ticks++);
119     Fl::check(); // all the FTLK events
120     Mutex.Unlock();
121 }
122
123 void userInterface::CallBackClose(Fl_Widget * CMN_UNUSED(widget),
124                                 userInterface * object)
125 {
126     // do not use mutex here, this is called by Fl::check which is
127     // already mutex protected

```

6.2 Main program

The main program creates the robot task, looks 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
mtsTaskManager * taskManager = mtsTaskManager::GetInstance();
27 robotLowLevel * robotTask = new robotLowLevel("RobotControl", 100 * cmn_ms);

29 // add single task, do not connect anything
taskManager->AddComponent(robotTask);
31 // create the thread for the task
taskManager->CreateAll();
33

// look for the interface we are going to use
35 mtsInterfaceProvided * robotInterface = robotTask->GetInterfaceProvided("Robot1");
userInterface * UI = 0;
37 if (robotInterface) {
    // instantiate the UI in the current thread
39     UI = new userInterface("Robot1", robotInterface);
} else {
41     std::cerr << "It looks like there is no \"Robot1\" interface" << std::endl;
}
43

// start the task
45 taskManager->StartAll();

47 // loop while the UI did not get a close request
while (!UI->CloseRequested) {
49     // the user interface method Update is equivalent to the task
    // Run method
51     UI->Update();
    osaSleep(10 * cmn_ms); // ask for an update every 10 ms
53 }

```

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	5
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	9
11	sineTask.cpp: Run method with event	10
12	clockDevice.h	11
13	clockDevice.cpp	11
14	displayTask.h: with clock device	11
15	displayTask.cpp: constructor with clock device	12
16	main.cpp: second example	12
17	main.cpp: second example, setting up data collection	13
18	robotLowLevel.h	15
19	robotLowLevel.cpp: constructor	15
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