Summary of iCub Software Development Guidelines

Overview

Our goal in this document is to summarize all of the the different guidelines on standards for iCub software to make it easier for developers to find out what they need to do so that their code conforms to best practice (e.g. configuration management, parameter usage, port naming and renaming, use of RFModule helper class, use of threads, graceful shut-down, run-time interaction, documentation, coding standards, application description, etc). A full list of all the resources that were used in putting together this summary is provided at the end.

To help bring all this material together, we will develop a simple example module to highlight the key issues. The code for the complete module is also provided at the end.

If you want to jump in the deep end, start by reading

- Module Standards [1] to find out the minimal responsibilities of your module with respect to handling parameters and ports, as well configuration and shut down. Then read the following tutorials:
- Resource finder overview
- How to organize the command line parameters of your modules [2]
- Organizing Parameters: Advanced Tutorial [3]
- Using the module helper class to write a program [4]

The fourth of these tutorials is essential reading and introduces you to the RFModule [5] class, the mandatory starting point when developing iCub modules. Note that we are moving away from using Module [6] class as RFModule offers similar functionality to Module, but it adds support for the ResourceFinder class.

Alternatively, read through this page and then refer to the documents above for more detail and clarification, if necessary.

The `RFModule` Class

The starting point in writing a module for the iCub repository is to develop a sub-class of the yarp::os::RFModule [5] class.

First, we define a sub-class, or derived class, of the `yarp::os::RFModule` class. The module's variables - *and specifically the module's parameters and ports* - go in the private data members part and you need to override three methods:

- `bool configure();`
- `bool interruptModule();`
- `bool close();`

We will see later that there are three other methods which can be useful to override:

- `bool respond();`
- `double getPeriod();`
- `bool updateModule();`

In the following, we assume we are writing a module named `myModule`. This module will be implemented as a sub-class yarp::os::RFModule [5] called `MyModule` (capital M because we are going to create a sub-class).

```
#include <iostream>

#include <string>
```

```
#include <yarp/os/RFModule.h>
#include <yarp/os/Network.h>

using namespace std;
using namespace yarp::os;

class MyModule:public RFModule
{
   /* module parameters */

   /* class variables */

public:

   bool configure(yarp::os::ResourceFinder &rf); // configure all the module parameters and return true if successful
   bool interruptModule();                       // interrupt, e.g., the ports
   bool close();                                 // close and shut down the module
   bool respond();
   double getPeriod();
   bool updateModule();
}
```

We will deal with the various issues of implementing the module under several headings, addressing configuration, doing some work, run-time interaction, graceful shut-down, standards, and application description.

Configuration

By configuration, we mean the ability to specify the way that a given module is used. There are two aspects to this:

1. how the module is presented (i.e. which particular interfaces are used: the names of ports used, the name of the configuration file, the path to the configuration file, the name of the module, and the name of the robot) and
2. the module parameters that govern its behaviour (e.g. thresholds, set-points, and data files)

We refer to these collectively as resources. Typically, the configuration file name, the configuration file path (called the context), the module name, and the robot name are specified as command-line parameters, while all the remaining resources, including the names of the ports, are typically specified in the configuration file.

What's important to realize, however, is that *all* resources are handled the same way using the ResourceFinder which not only greatly simplifies the process of finding the resources but also simplifies the process of parsing them. There is more detail on handling resources in Configuration and resource files.

It's worth noting that parameters that are specified in the configuration file can also be specified in the command-line if you wish. The reverse is also true, with some restrictions (e.g. it only makes sense to specify the configuration file and the configuration file path on the command-line). Finally, modules should be written so that default values are provided for all resources so that the module can be launched without any parameters. Again, the ResourceFinder makes it easy to arrange this.

Right now, what's important to grasp is that all these configuration issues are implemented by

- preparing the Resource Finder in the `main` function by setting the default configuration file and its path,
- overriding the `yarp::os::RFModule::configure()` method to parse all the parameters from the command-line and the configuration file.

The following sections explain the implementation details of each aspect of this configuration.

Essential Command-line Parameters

Configuration File

Configuration can be changed by changing configuration files. The configuration file which the module reads can be specified as a command line option.

```
--from myModule.ini
```

The module should set a default configuration file using `yarp::os::ResourceFinder::setDefaultConfigFile("myModule.ini")`. This should be done in the `main()` function before running the module.

This is overridden by the `--from` parameter.

The .ini file should usually be placed in the `$ICUB_ROOT/app/myModule/conf` sub-directory.

Context

The relative path from `$ICUB_ROOT/app/` to the directory containing the configuration file is specified from the command line by

```
--context  myModule/conf
```

The module should set a default context using `yarp::os::ResourceFinder::setDefaultContext("myModule/conf")`. This should be done in the `main()` function before running the module.

This is overridden by the `--context` parameter.

Module Name and Port Names

Warning: the naming convention for the --name, --robot, and port name arguments in key-value pairs has changed. The arguments of --name and --robot **do not** having a leading "/" prefix and port name arguments **always** have a leading "/" prefix ... exactly the opposite of what was considered acceptable practice in the past.

It should be possible to specify the names of any ports created by a module via configuration. There are two aspects to this: the stem of the port name and the port name itself.

A command-line option of

```
--name altName
```

sets the name of the module and will cause the module to use "/altName" as a stem for all port names provided the port names are generated using the `yarp::os::RFModule::getName()` method. Note that he leading "/" prefix has to be added explicitly to the module name to create the port name.

The module should set a default name (and, hence, a default stem) using `yarp::os::RFModule::setName("myModule")`.

This is overridden by the `--name` parameter but you must check for this parameter and call `setName()` accordingly, e.g.

```
string moduleName;

moduleName = rf.check("name",
             Value("myModule"),
             "module name (string)").asString();

setName(moduleName.c_str());  // do this before processing any port name parameters
```

The port names should be specified as parameters, typically as key-value pairs in the `.ini` configuration file, e.g.

```
myInputPort    /image:i
myOutputPort   /image:o
```

These key-value pairs can also be specified as command-line parameters, viz: `--myInputPort /image:i --myOutputPort /image:o`

The module should set a default port name using the ResourceFinder, e.g using `yarp::os::ResourceFinder::check();`

For example

```
string inputPortName  = "/";
       inputPortName += getName(
                        rf.check("myInputPort",
                        Value("/image:i"),
                        "Input image port (string)").asString()
                        );
```

will assign to `inputPortName` the value `/altName/image:i` if `--name altName` is specified as a parameter. Otherwise, it would assign `/myModule/image:i` On the other hand, it would assign `/myModule/altImage:i` if the key-value pair `myInputPort /altImage:i` was specified (either in the `.ini` file or as a command-line parameter) but not the `--name altName`

When providing the names of ports as parameter values (whether as a default value in `ResourceFinder::check`, as a value in the key-value list in the `.ini` configuration file, or as a command line argument), you always include the leading `/`.

All this code goes in the `configure()` method.

Robot Name

If you connect automatically to the robot, make sure the name of the ports to which you connect to can be changed from the command line. This will make it possible to switch from using the simulator (whose ports are prefixed with `icubSim`) to the real robot (whose ports are prefixed with `icub`).

Usually this is achieved with a `--robot` parameter. For example, to access the left camera on the simulator (`/iCubSim/cam/left`) use `--robot icubSim` and to access the left camera on the robot (`/icub/cam/left`) use `––robot icub`

Which Parameters Are Parsed Automatically?

Parsing the `--from` and `--context` parameters is handled automatically by the `RFModule` but `--name` and `--robot` must be handled explicitly.

As noted above, you would handle the `--name` parameter by using `ResourcFinder::check()` to parse it and get the parameter value, then user `setName()` to set it. You should do this before proceeding to process any port name parameters, otherwise the wrong stem will be used when constructing the port names from the parameter values.

Typically, you would handle the `--robot` parameter by using ResourceFinder to parse the `--robot` or `--name` parameter to get the root of the port name and then construct the full port name by appending the specific part of the robot required. An example is provided below.

Configuration File Parameters

The configuration file, typically named `myModule.ini` and located in the `$ICUB_ROOT/app/myModule/conf` directory, contains a key-value list: a list of pairs of keywords (configuration parameter names) and values (configuration parameter values), e.g.

```
myInputPort  /altImage:i
myOutputPort /altImage:o
threshold 9
...
```

These parameters are parsed using the `ResourceFinder` [7] within an `RFModule` object (i.e. by methods inherited by your module such as `yarp::os::Searchable::check()` [8]).

Typically, key-value pairs specify the parameters and their values that govern the behaviour of the module, as well as the names of the module ports, should you wish to rename them.

Other Configuration Files

Apart from processing the parameters in the configuration file `myModule.ini`, it's often necessary to access configuration data in other files. For example, you might want to read the intrinsic camera parameters from a camera calibration file. Let's assume this configuration file is called `icubEyes.ini` and we wish to extract the principal points of the left and right cameras. The coordinates of the principle points, and other intrinsic camera parameters, are generated by the camera calibration module [9] and are stored as a sequence of key-value pairs:

```
cx 157.858
cy 113.51
```

Matters are somewhat complicated by the fact that we need to read two sets of coordinates, one for the left camera and one for the right. *Both* sets have the same key associated with them so the left and right camera parameters, including the principal point coordinates, are typically listed under different *group* headings, viz.

```
[CAMERA_CALIBRATION_RIGHT]
...
cx 157.858
cy 113.51
...

[CAMERA_CALIBRATION_LEFT]
...
cx 174.222
cy 141.757
...
```

So, to read these two pairs of coordinates, we need to

- find the name of the file (e.g. `icubEyes.ini`)
- locate the file (i.e. get its full path)
- open the file and read its content
- find the `CAMERA_CALIBRATION_RIGHT` and `CAMERA_CALIBRATION_LEFT` groups
- read the respective `cx` and `cy` key values.

All of this is accomplished straightforwardly with the `ResourceFinder` [7] and `Property` [10] classes.

The first step is to get the name of the configuration file. This will typically be one of the key-value pairs in the module configuration file `myModule.ini`, e.g.

```
cameraConfig icubEyes.ini
```

so that it can be read in exactly the same way as the other parameters in the previous section, e.g. using `yarp::os::Searchable::check()` [8].

The full path can then be determined by the `yarp::os::ResourceFinder::findFile()` [11] method.

The contents of this file can then be read into a `Property` [10] object using the `yarp:os:Property::fromConfigFile()` [12] method.

Locating the required group (e.g. `CAMERA_CALIBRATION_LEFT`) is accomplished with the `yarp:os:Property::findGroup()` [13] method.

This method returns a `Bottle` [14] with the full key-value list under this group. This list can then be searched for the required key and value using the `yarp::os::Searchable::check()` [8] method, as before.

Please refer to the `myModule` example for further details.

An Example of how to Configure the Module

The following simple module shows how to handle the foregoing configuration issues.

```
#include <yarp/os/all.h>

#include <yarp/os/RFModule.h>

#include <yarp/os/Network.h>

#include <yarp/os/Thread.h>

#include <yarp/sig/all.h>


using namespace std;

using namespace yarp::os;

using namespace yarp::sig;


class MyModule:public RFModule

{

   /* module parameters */


   string moduleName;

   string robotName;

   string robotPortName;

   string inputPortName;

   string outputPortName;

   string cameraConfigFilename;

   float  fxLeft,  fyLeft;          // focal length

   float  fxRight, fyRight;         // focal length

   float  cxLeft,  cyLeft;          // coordinates of the principal point

   float  cxRight, cyRight;         // coordinates of the principal point

   int thresholdValue;


   /* class variables */


   BufferedPort<ImageOf<PixelRgb> > imageIn;      //example input port
```

```
   BufferedPort<ImageOf<PixelRgb> > imageOut;     //example output port

public:

   bool configure(yarp::os::ResourceFinder &rf); // configure all the module parameters and return true if successful
   bool interruptModule();                       // interrupt, e.g., the ports
   bool close();                                 // close and shut down the module
   bool respond(const Bottle& command, Bottle& reply)
   double getPeriod();
   bool updateModule();
};

/*
 * Configure method. Receive a previously initialized
 * resource finder object. Use it to configure your module.
 * If you are migrating from the old Module, this is the
 *  equivalent of the "open" method.
 */

bool MyModule::configure(yarp::os::ResourceFinder &rf)
{
   /* Process all parameters from both command-line and .ini file */

   /* get the module name which will form the stem of all module port names */

   moduleName            = rf.check("name",
                           Value("myModule"),
                           "module name (string)").asString();

   /*
    * before continuing, set the module name before getting any other parameters,
    * specifically the port names which are dependent on the module name
    */

   setName(moduleName.c_str());

   /* now, get the rest of the parameters */

   /*
    * get the robot name which will form the stem of the robot ports names
    * and append the specific part and device required
    */

   robotName             = rf.check("robot",
                           Value("icub"),
                           "Robot name (string)").asString();
```

```
robotPortName         = "/" + robotName + "/head";

/*
 * get the cameraConfig file and read the required parameter values cx, cy
 * in both the groups [CAMERA_CALIBRATION_LEFT] and [CAMERA_CALIBRATION_RIGHT]
 */

cameraConfigFilename  = rf.check("cameraConfig",
                        Value("icubEyes.ini"),
                        "camera configuration filename (string)").asString();

cameraConfigFilename = (rf.findFile(cameraConfigFilename.c_str())).c_str();

Property cameraProperties;

if (cameraProperties.fromConfigFile(cameraConfigFilename.c_str()) == false) {
   cout << "myModule: unable to read camera configuration file" << cameraConfigFilename;
   return 0;
}
else {
   cxLeft  = (float) cameraProperties.findGroup("CAMERA_CALIBRATION_LEFT").check("cx", Value(160.0), "cx left").asDouble();
   cyLeft  = (float) cameraProperties.findGroup("CAMERA_CALIBRATION_LEFT").check("cy", Value(120.0), "cy left").asDouble();
   cxRight = (float) cameraProperties.findGroup("CAMERA_CALIBRATION_RIGHT").check("cx", Value(160.0), "cx right").asDouble();
   cyRight = (float) cameraProperties.findGroup("CAMERA_CALIBRATION_RIGHT").check("cy", Value(120.0), "cy right").asDouble();
}

/* get the name of the input and output ports, automatically prefixing the module name by using getName() */

inputPortName         = "/";
inputPortName        += getName(
                        rf.check("myInputPort",
                        Value("/image:i"),
                        "Input image port (string)").asString()
                        );

outputPortName        = "/";
outputPortName       += getName(
                        rf.check("myOutputPort",
                        Value("/image:o"),
                        "Output image port (string)").asString()
                        );

/* get the threshold value */

thresholdValue        = rf.check("threshold",
                        Value(8),
                        "Key value (int)").asInt();
```

```
   /* do all initialization here */

   /* open ports  */

   if (!imageIn.open(inputPortName.c_str())) {
      cout << getName() << ": unable to open port " << inputPortName << endl;
      return false;  // unable to open; let RFModule know so that it won't run
   }

   if (!imageOut.open(outputPortName.c_str())) {
      cout << getName() << ": unable to open port " << outputPortName << endl;
      return false;  // unable to open; let RFModule know so that it won't run
   }

   return true ;   // let the RFModule know everything went well
                   // so that it will then run the module
}
```

```
int main(int argc, char * argv[])
{
   /* initialize yarp network */

   Network yarp;

   /* create your module */

   MyModule myModule;

   /* prepare and configure the resource finder */

   ResourceFinder rf;
   rf.setVerbose(true);
   rf.setDefaultConfigFile("myModule.ini"); //overridden by --from parameter
   rf.setDefaultContext("myModule/conf");   //overridden by --context parameter
   rf.configure("ICUB_ROOT", argc, argv);

   /* run the module: runModule() calls configure first and, if successful, it then runs */

   myModule.runModule(rf);

   return 0;
}
```

Graceful Shut-down

To achieve clean shutdown, two methods `yarp::os::RFModule::interruptModule()` and `yarp::os::RFModule::close()` should be overridden. The `interruptModule()` method will be called when it is desired that `updateModule()` finish up. When it has indeed finished, `close()` will be called. For example:

```
bool MyModule::interruptModule()
{
   imageIn.interrupt();
   imageOut.interrupt();
   return true;
}
```

```
bool MyModule::close()
{
   imageIn.close();
   imageOut.close();
   return true;
}
```

For `yarp::os::RFModule`, the method `yarp::os::RFModule::updateModule()` will be called from the main control thread until it returns false. After that a clean shutdown will be initiated. The period with which it is called is determined by the method `yarp::os::RFModule::getPeriod()`. Neither method need necessarily be overridden. The default methods provide the required functionality.

```
/* Called periodically every getPeriod() seconds */

bool MyModule::updateModule()
{
   return true;
}
```

```
double MyModule::getPeriod()
{
   /* module periodicity (seconds), called implicitly by myModule */

   return 0.1;
}
```

Note that the `updateModule()` method is not meant to run code that that implements the algorithm encapsulated in the module. Instead `updateModule()` is meant to be used as a periodic mechanism to check in on the operation of the thread that implements the module (e.g. gather interim statistics, change parameter settings, etc.). The `updateModule()` is called periodically by the RFModule object, with the period being determined by the `getPeriod()` method. Both `updateModule()` and `getPeriod()` can be overridden in your implementation of `myModule`.

Thread-based Implementation

Using Threads to Implement Your Algorithm

For the module to actually do anything, it should start or stop threads using the YARP Thread [15] and RateThread [16] classes. Typically, these threads are started and stopped in the `configure` and `close` methods of the RFModule class. If you are writing a control loop or an algorithm that requires precise scheduling we strongly advise that you use the RateThread [16] class.

Just as the starting point in writing a module for the iCub repository is to develop a sub-class of the yarp::os::RFModule [5] class, **the starting point for implementing the algorithm within that module is to develop a sub-class of either Thread [15] or RateThread [16]**.

In the following, we will explain how to do it with Thread [15]; it's straightforward to extend this to RateThread [16] (effectively, you provide an argument with the `RateThread` instantiation specifying the period with which the thread should be spawned, the thread just runs once so that you don't have to check `isStopping()` to see if the thread should end).

Perhaps one of the best ways of thinking about this is to view it as a two levels of encapsulation, one with `RFModule`, and another with `Thread`; the former deals with the configuration of the module and the latter dealing with the execution of the algorithm. The only tricky part is that somehow these two objects have to communicate with one another.

You need to know three things:

1. The thread is instantiated and started in the `configure()` method.
2. The thread is stopped in the `close()` method.
3. When the thread is instantiated, you pass the module parameters to it as a set of arguments (for the constructor).

Let's begin with the definition of a thread `MyThread` (capital M because we are going to create a sub-class) and then turn our attention to how it is used by `MyModule`.

An Example of how to use the `Thread` Class

First, we define a sub-class, or derived class, of the `yarp::os::Thread` class. The algorithm's variables - *and specifically the thread's parameters and ports* - go in the private data members part and you need to override four methods:

1. `MyThread::MyThread(); // the constructor`
2. `bool threadInit(); // initialize variables and return true if successful`
3. `void run(); // do the work`
4. `void threadRelease(); // close and shut down the thread`

There are a number of important points to note.

First, the variables in the `myThread` class which represent the thread's parameters and port should be pointer types and the constructor parameters should initialize them. In turn, the arguments of the `myThread` object instantiation in the `configure()` should be the addresses of (pointers to) the module parameters and ports in the `myModule` object. In this way, the thread's parameter and port variables are just references to the original module parameters and ports that were initialized in the `configure` method of the `myModule` object.

Second, `threadInit()` returns `true` if the initialization was successful, otherwise it should return `false`. This is significant because if it returns `false` the thread will not subsequently be run.

Third, the `run()` method is where the algorithm is implemented. Typically, it will run continuously until some stopping condition is met. This stopping condition should include the return value of a call to the `yarp::os::Thread::isStopping()` method which flags whether or not the thread is to terminate. In turn, the value of `yarp::os::Thread::isStopping()` is determined by the `yarp::os::Thread::stop()` method which, as we will see, is called in `myModule.close()`

The following is an example declaration and definition of the `MyThread` class.

```
#include <yarp/os/Thread.h>


using namespace std;

using namespace yarp::os;


class MyThread : public Thread

{

private:


   /* class variables */


   int      x, y;

   PixelRgb rgbPixel;

   ImageOf<PixelRgb> *image;


   /* thread parameters: they are pointers so that they refer to the original variables in myModule */


   BufferedPort<ImageOf<PixelRgb>> *imagePortIn;

   BufferedPort<ImageOf<PixelRgb>> *imagePortOut;

   int *thresholdValue;


public:


   /* class methods */


   MyThread(BufferedPort<ImageOf<PixelRgb>> *imageIn,  BufferedPort<ImageOf<PixelRgb>> *imageOut, int *threshold );

   bool threadInit();

   void threadRelease();

   void run();

};
```

```
MyThread::MyThread(BufferedPort<ImageOf<PixelRgb>> *imageIn, BufferedPort<ImageOf<PixelRgb>> *imageOut, int *threshold)

{

   imagePortIn    = imageIn;

   imagePortOut   = imageOut;

   thresholdValue = threshold;

}


bool MyThread::threadInit()

{

   /* initialize variables and create data-structures if needed */


   return true;

}
```

```
void MyThread::run(){

   /*
    * do some work ....
    * for example, convert the input image to a binary image using the threshold provided
    */

   unsigned char value;

   while (isStopping() != true) { // the thread continues to run until isStopping() returns true

      cout << "myThread: threshold value is " << *thresholdValue << endl;

      do {
         image = imagePortIn->read(true);
      } while (image == NULL);

      ImageOf<PixelRgb> &binary_image = imagePortOut->prepare();
      binary_image.resize(image->width(),image->height());

      for (x=0; x<image->width(); x++) {
         for (y=0; y<image->height(); y++) {

             rgbPixel = image->safePixel(x,y);

             if (((rgbPixel.r + rgbPixel.g + rgbPixel.b)/3) > *thresholdValue) {
                value = (unsigned char) 255;
             }
             else {
                value = (unsigned char) 0;
             }

             rgbPixel.r = value;
             rgbPixel.g = value;
             rgbPixel.b = value;

             binary_image(x,y) = rgbPixel;
          }
       }

       imagePortOut->write();
   }
}

void MyThread::threadRelease()
{
```

```
   /* for example, delete dynamically created data-structures */
}
```

Creating, Starting, and Stopping the Thread

As we said already, the thread is instantiated and started in the `configure()` method in `myModule`, the thread is stopped in the `close()` method, and when the thread is instantiated, you pass the pointers to the module parameters to it as a set of arguments. First, however, we add a new variable to the `MyModule` class.

```
/* pointer to a new thread to be created and started in configure() and stopped in close() */

MyThread *myThread;
```

The following code would then go in the `configure()` method.

```
/* create the thread and pass pointers to the module parameters */

myThread = new MyThread(&imageIn, &imageOut, &thresholdValue);

/* now start the thread to do the work */

myThread->start(); // this calls threadInit() and it if returns true, it then calls run()
```

The following code would go in the `close()` method.

```
/* stop the thread */

myThread->stop();
```

Run-time Interaction

The `respond()` Method

Often, it is very useful for a user or another module to send commands to control the behaviour of the module, e.g. interactively changing parameter values. The controlGaze2 [17] module is a good example of this type of usage (see also VVV09_Control_Gazers_Group).

We accomplish this functionality for the `yarp::os::RFModule` by overridding the `yarp::os::RFModule::respond()` method which can then be configured to receive messages from either a port (typically named `/myModule`) or the terminal. This is effected by the `yarp::os::RFModule::attach(port)` and `yarp::os::RFModule::attachTerminal()` methods, respectively. Attaching both the port and the terminal means that commands from both sources are then handled in the same way.

An Example of how to change module parameters at run-time

In the following example, we handle three commands:

- help
- quit
- set
  - set thr <n> ... set the threshold

    (where <n> is an integer number)

Apart from the way that the commands are parsed and the form of the reply, the key thing to note here is the fact that the value of `MyModule::thresholdValue` is updated. Since `myThread` references this variable, it too is updated and the updated value is used in the thread.

```
bool MyModule::respond(const Bottle& command, Bottle& reply)
{
  string helpMessage =  string(getName().c_str()) +
                        " commands are: \n" +
                        "quit \n" +
                        "set thr <n> ... set the threshold \n" +
                        "(where <n> is an integer number) \n";

  reply.clear();

  if (command.get(0).asString()=="quit") {
       reply.addString("quitting");
       return false;
   }
   else if (command.get(0).asString()=="help") {
      cout << helpMessage;
      reply.addString("ok");
   }
   else if (command.get(0).asString()=="set") {
      if (command.get(1).asString()=="thr") {
         thresholdValue = command.get(2).asInt(); // set parameter value
         reply.addString("ok");
      }
   }
   return true;
}
```

However, for any of this to work, we have to set up a port in the first place. We put port declaration in the private data member part of `MyModule` class

```
string handlerPortName;
Port handlerPort;      //a port to handle messages
```

and open it in the `configure()` method, viz.

```
/*
 * attach a port of the same name as the module (prefixed with a /) to the module
 * so that messages received from the port are redirected to the respond method
```

```
 */

handlerPortName =  "/";
handlerPortName += getName();          // use getName() rather than a literal

if (!handlerPort.open(handlerPortName.c_str())) {
   cout << getName() << ": Unable to open port " << handlerPortName << endl;
   return false;
}

attach(handlerPort);                   // attach to port

attachTerminal();                      // attach to terminal
```

Interrupt it in the `interrupt()` method, viz.

```
handlerPort.interrupt();
```

Close it in the `close()` method, viz.

```
handlerPort.close();
```

Remote Connection

Note that the `handlerport` can be used not only by other modules but also interactively by a user through the `yarp rpc` directive, viz.:

```
yarp rpc /myModule
```

This opens a connection from a terminal to the port and allows the user to then type in commands and receive replies from the `respond()` method.

Documentation and Coding Guidelines

Note: This paragraph has been copied in Section 11 of the manual (Guidelines).

RobotCub code follows some fairly strict documentation and coding standards defined in Section III of RobotCub Deliverable 8.2 [18].

For convenience, here are the

- iCub File Organization Guidelines
- iCub Documentation Guidelines
- iCub Coding Guidelines

Please take the time to read through the three documents.

As we move towards the creation of a release version of the iCub software, we will begin to enforce a sub-set of these guidelines as mandatory standards. The current set of standards is set out in iCub Software Standards. **Ultimately, all modules to be included in the standard iCub release version will have to comply with these standards.**

The principal documentation for `myModule` is provided in the full example at the end of this page.

Application Description

iCub applications, i.e. collections of inter-connected YARP modules, are described in XML and launched using an automatically-generated GUI. Refer to Managing Applications for more details on how to write these application descriptions.

An application description containing an example invocation of the `myModule` with some command-line parameters is shown below.

```
<application>

<name>Test myModule</name>

<dependencies>
   <port>/icub/cam/left</port>
</dependencies>

<module>
   <name>myModule</name>
   <parameters>--threshold 128</parameters>
   <node>icub1</node>
   <tag>myModule</tag>
</module>

<module>
   <name>yarpview</name>
   <parameters>--name /rgbImage --x 000 --y 0 --synch</parameters>
   <node>icub1</node>
   <tag>left_image</tag>
</module>

<module>
   <name>yarpview</name>
   <parameters>--name /binaryImage --x 350 --y 0 --synch</parameters>
   <node>icub1</node>
   <tag>right_image</tag>
</module>

<connection>
  <from>/icub/cam/left</from>
  <to>/myModule/image:i</to>
  <protocol>tcp</protocol>
</connection>

<connection>
  <from>/icub/cam/left</from>
  <to>/rgbImage</to>
  <protocol>tcp</protocol>
</connection>
```

```
<connection>
  <from>/myModule/image:o</from>
  <to>/binaryImage</to>
  <protocol>tcp</protocol>
</connection>

</application>
```

To run the application, you simple need to run the XML application description shown in the previous section. You can execute the xml script using the yarpmanager tool from YARP (http://wiki.icub.org/yarpdoc/yarpmanager.html).

Do an update on your iCub repository to make sure you have the `icubapp` pseudo-command. Alternatively, you can launch the python application manager directly (see below).

Once you have done all this, you are *almost* ready to run your application. There's just one more thing to be aware of.

You need to start an instance of `yarprun --server` on the local machine (for a complete explanation see Cluster management). This `yarprun` is what the node in an XML application description gets mapped to. At present, the standard for creating these `yarpruns` is for the `yarprun` argument to be the name of the node identifier in the XML `<node></node>` field but prefixed by a / to make it explicit that the argument is a port.

So, if you have used, for example, `<node>icub1</node>` in your `<module>` description in the XML file, then you would do

```
PC> yarprun --server /icub1
```

In general, at present (this may change in the future), you need to do a

```
PC> yarprun --server /<mc_n>
```

for each <mc_n> node values specified in the xml file.

These `yarprun` commands are run on the machine to which that node is mapped. An XML `<node>` is a logical machine and the `yarprun` associates it with the physical machine on which it to be instantiated.

You can now launch an application. Simply navigate to the directory where the XML file resides (typically `$ICUB_ROOT/app/myModule/scripts`) and do

```
PC> icubapp myModule.xml
```

Alternatively, if you prefer, you can launch the Python application manager directly:

```
PC> manager.py myModule.xml
```

assuming that `$ICUB_ROOT\app\default\scripts\` is defined in you path and assuming `.py` files are associated with Python.

In either case, doing this will launch a GUI with which you can then "Run Modules" and "Connect" the ports by clicking on the appropriate buttons.

NB: turn off your firewall before launching the application.

Resources

- Module Standards [19]
- Resource finder overview
- How to organize the command line parameters of your modules [20]
- Organizing Parameters: Advanced Tutorial [21]
- Using the module helper class to write a program [22]
- RFModule Class Reference [5]
- Module Class Reference [6]
- Coding and Documentation Standards [23]
- exampleModule [24]
- Cluster management
- exampleApplication [25]
- iCub tutorials [26]

The Complete `myModule` Example

The complete code for `myModule` is here.

References

[1] http://wiki.icub.org/iCub/main/dox/html/module_standards.html
[2] http://wiki.icub.org/iCub/main/dox/html/icub_resource_finder_basic.html
[3] http://wiki.icub.org/iCub/main/dox/html/icub_resource_finder_advanced.html
[4] http://wiki.icub.org/iCub/main/dox/html/icub_tutorial_module.html
[5] http://wiki.icub.org/yarpdoc/d9/d26/classyarp_1_1os_1_1RFModule.html
[6] http://wiki.icub.org/yarpdoc/d1/d03/classyarp_1_1os_1_1Module.html
[7] http://wiki.icub.org/yarpdoc/d9/ddf/classyarp_1_1os_1_1ResourceFinder.html
[8] http://wiki.icub.org/yarpdoc/d2/d0c/classyarp_1_1os_1_1Searchable.html#818029558a2d8772db43a5a3c8b61125
[9] http://wiki.icub.org/iCub/dox/html/group__icub__camcalibconf.html
[10] http://wiki.icub.org/yarpdoc/da/d1f/classyarp_1_1os_1_1Property.html
[11] http://wiki.icub.org/yarpdoc/d9/ddf/classyarp_1_1os_1_1ResourceFinder.html#355586da9ad41565a2a0daa36e7ec2e1
[12] http://wiki.icub.org/yarpdoc/da/d1f/classyarp_1_1os_1_1Property.html#06c34c056e399f1cad1ad74b3a147a76
[13] http://wiki.icub.org/yarpdoc/da/d1f/classyarp_1_1os_1_1Property.html#ed956fea82f3b54bc846946c1f836ccb
[14] http://wiki.icub.org/yarpdoc/d3/d3e/classyarp_1_1os_1_1Bottle.html
[15] http://wiki.icub.org/yarpdoc/d2/d2d/classyarp_1_1os_1_1Thread.html
[16] http://wiki.icub.org/yarpdoc/d9/d9c/classyarp_1_1os_1_1RateThread.html
[17] http://wiki.icub.org/iCub/dox/html/group__icub__controlGaze2.html
[18] http://www.robotcub.org/index.php/robotcub/more_information/deliverables/deliverable_8_2_pdf
[19] http://wiki.icub.org/iCub/dox/html/module_standards.html
[20] http://wiki.icub.org/iCub/dox/html/icub_resource_finder_basic.html
[21] http://wiki.icub.org/iCub/dox/html/icub_resource_finder_advanced.html
[22] http://wiki.icub.org/iCub/dox/html/icub_tutorial_module.html
[23] http://wiki.icub.org/iCub/dox/html/coding_standards.html
[24] http://wiki.icub.org/iCub/dox/html/group__icub__exampleModule.html
[25] http://wiki.icub.org/iCub/dox/html/group__icub__exampleApplication.html
[26] http://wiki.icub.org/iCub/dox/html/icub_tutorials.html

Article Sources and Contributors

Summary of iCub Software Development Guidelines *Source*: http://wiki.icub.org/index.php?oldid=20966 *Contributors*: Daniele.Domenichelli@iit.it, Dvernon, Gsaponaro, Lorenzo, Paulfitz, Robotics

License