

---

# Extending the Real-Time Toolkit

Copyright © 2006 Peter Soetens, FMTC

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation, with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of this license can be found at <http://www.fsf.org/copyleft/fdl.html>.

Revision History  
Revision 1.0.1                      24 Nov 2006                      ps  
Separated from the Developer's Manual.

## Abstract

This document is an introduction to making user defined types (classes) visible within Orocos. You need to read this document when you want to see the value of an object you defined yourself, for example in the TaskBrowser component or in an Orocos script. Other uses are reading and writing objects to and from XML and generally, anything a built-in Orocos type can do, so can yours.

## Table of Contents

1. The Orocos Type System : Toolkits .....	1
1.1. The Real-Time Toolkit .....	1
1.2. Telling Orocos about your data .....	2
1.3. Building your own Toolkit .....	5

# 1. The Orocos Type System : Toolkits

Most applications define their own classes or structs to exchange data between objects. It is possible to tell Orocos about these user defined types such that they can be displayed, stored to XML, used in the scripting engine or even transferred over a network connection.

## 1.1. The Real-Time Toolkit

Orocos uses the 'Toolkit' principle to make it aware of user types. Orocos' Real-Time Toolkit already provides support for the C++ types int, unsigned int, double, char, bool, float, vector<double> and string.

A toolkit can be imported into the application by writing:

```
#include <rtt/RealTimeToolkit.hpp>
// ...
RTT::Toolkit::Import( RTT::RealTimeToolkit );
```

This is however done automatically, unless you disabled that option in the configuration system. After this line is executed, Orocos is able to display, transfer over a network or recognise these types in scripts.

## 1.2. Telling Orocos about your data

Say that you have an application which transfers data in a struct `ControlData` :

```
struct ControlData {
    double x, y, z;
    int sample_nbr;
}
```

How can you tell Orocos how to handle this type ? A helper class is provided which you can extend to fit your needs, `TemplateTypeInfo`.

```
#include <rtt/TemplateTypeInfo.hpp>
// ...
struct ControlDataTypeInfo
: public RTT::TemplateTypeInfo<ControlData>
{
    ControlDataTypeInfo()
    : RTT::TemplateTypeInfo<ControlData>("ControlData")
    {}
};

// Tell Orocos the name and type of this struct:
RTT::TypeInfoRepository::Instance()->addType( new ControlDataTypeInfo() );
```

From now on, Orocos knows the 'ControlData' type name and allows you to create a scripting variable of that type. It does however not know yet how to display it or write it to an XML file.



### Note

The type is now usable as a 'var' in a script, however, in order to initialise a variable, you need to add a constructor as well. See Section 1.3.2, "Loading Constructors".

### 1.2.1. Displaying

In order to tell Orocos how to display your type, you may overload the `TemplateTypeInfo::write` function or define operator `<<()` for your type:

```
#include <rtt/TemplateTypeInfo.hpp>
#include <ostream>

std::ostream& operator<<(std::ostream& os, const ControlData& cd) {
```

```

return os << '(' << cd.x << cd.y << cd.z << ')': ' << cd.sample_nbr;
}
// ...
// 'true' argument: it has operator<<
struct ControlDataTypeInfo
: public RTT::TemplateTypeInfo<ControlData,true>
{
ControlDataTypeInfo()
: RTT::TemplateTypeInfo<ControlData,true>("ControlData")
{}
};

// Tell Orocos the name and type of this struct
RTT::TypeInfoRepository::Instance()->addType( new ControlDataTypeInfo() );

```

If you use the above line of code to add the type, Orocos will be able to display it as well, for example in the TaskBrowser or in the ReportingComponent.

## 1.2.2. Writing to XML

In order to inform Orocos of the structure of your data type, it must be given a 'decompose' function: Of which primitive types does the struct consists ? Representing structured data is what Orocos Property objects do. Here is how to tell Orocos how the "ControlData" is structured:

```

// ...
struct ControlDataTypeInfo
: public TemplateTypeInfo<ControlData,true>
{
// ... other functions omitted

virtual bool decomposeTypeImpl(const ControlData& in, PropertyBag& targetbag )
const {
targetbag.setType("ControlData");
targetbag.add( new Property<double>("X", "X value of my Data", in.x ) );
targetbag.add( new Property<double>("Y", "Y value of my Data", in.y ) );
targetbag.add( new Property<double>("Z", "Z value of my Data", in.z ) );
targetbag.add( new Property<int>("Sample", "The sample number of the Data",
in.sample_nbr ) );
return true;
}
}

```

That was easy ! For each member of your struct, add a Property of the correct type to the targetbag and you're done ! setType() can be used lateron to determine the version or type of your XML representation. Next, if Orocos tries to write an XML file with ControlData in it, it will look like:

```

<struct name="MyData" type="ControlData">
<simple name="X" type="double">
<description>X value of my Data</description>

```

```

    <value>0.12</value>
  </simple>
  <simple name="Y" type="double">
    <description>Y value of my Data</description>
    <value>1.23</value>
  </simple>
  <simple name="Z" type="double">
    <description>Z value of my Data</description>
    <value>3.21</value>
  </simple>
  <simple name="Sample" type="short">
    <description>The sample number of the Data</description>
    <value>3123</value>
  </simple>
</struct>

```

### 1.2.3. Reading from XML

Orocos does not know yet how to convert an XML format back to the ControlData object. This operation is called 'composition' and is fairly simple as well: Here is how to tell Orocos how the "ControlData" is read:

```

// ...
struct ControlDataTypeInfo
: public TemplateTypeInfo<ControlData,true>
{

  // ... other functions omitted

  virtual bool composeTypeImpl(const PropertyBag& bag, ControlData& out ) const
  {
    if ( bag.getType() == std::string("ControlData") ) // check the type
    {
      Property<double>* x = targetbag.getProperty<double>("X");
      Property<double>* y = targetbag.getProperty<double>("Y");
      Property<double>* z = targetbag.getProperty<double>("Z");
      Property<int>* t = targetbag.getProperty<int>("Sample");

      if ( !x || !y || !z || !t )
        return false;

      out.x = x->get();
      out.y = y->get();
      out.z = z->get();
      out.sample_nbr = t->get();
      return true;
    }
    return false; // unknown type !
  }
}

```

First the properties are located in the bag, it should look just like we stored them. If

not, return false, otherwise, read the values and store them in the out variable.

### 1.2.4. Network transfer (CORBA)

In order to transfer your data between components over a network, Orocos requires that you provide the conversion from your type to a CORBA::Any type and back, quite similar to the 'composition' and 'decomposition' of your data. Look at the TemplateTypeInfo interface for the functions you need to implement.

The first step is describing your struct in IDL and generate the 'client' headers with 'Any' support. Next you create such a struct, fill it with your data type's data and next 'stream' it to an Any. The other way around is required as well.

In addition, you will need the CORBA support of Orocos enabled in your build configuration.

### 1.2.5. Advanced types

In order to add more complex types, take a look at the code of the RealTimeToolkit and the KDL Toolkit Plugin of Orocos.

## 1.3. Building your own Toolkit

The number of types may grow in your application to such a number or diversity that it may be convenient to build your own toolkit and import them when appropriate. Non-Orocos libraries benefit from this system as well because they can introduce their data types into Orocos.

Each toolkit must inherit from the ToolkitPlugin class and implement four functions: loadTypes(), loadConstructors, loadOperators() and getName().

The name of a toolkit must be unique. Each toolkit will be loaded no more than once. The loadTypes function contains all 'TemplateTypeInfo' constructs to tell Orocos about the types of your toolkit. The loadOperators function contains all operations that can be performed on your data such as addition ('+'), indexing ('[i]'), comparison ('==') etc. Finally, type constructors are added in the loadConstructors function. They allow a newly created script variable to be initialised with a (set of) values.

Mimick the code of the RealTimeToolkitPlugin and KDLToolkitPlugin to build your own.

### 1.3.1. Loading Operators

Operator are stored in the class OperatorRepository in Operators.hpp. The list of supported operators is set by the toolkit and added to the OperatorRepository. It looks something like this:

```
bool loadOperators() {
    OperatorRepository::shared_ptr or = OperatorRepository::Instance();
```

```
// boolean stuff:
or->add( newUnaryOperator( "!", std::logical_not<bool>() ) );
or->add( newBinaryOperator( "&&", std::logical_and<bool>() ) );
or->add( newBinaryOperator( "||", std::logical_or<bool>() ) );
or->add( newBinaryOperator( "==", std::equal_to<bool>() ) );
or->add( newBinaryOperator( "!=", std::not_equal_to<bool>() ) );
return true;
}
```

Adding your own should not be terribly hard. The hardest part is that as the second argument to `newUnaryOperator`, `newBinaryOperator` or `newTernaryOperator`, you need to specify a STL Adaptable Functor, and even though the STL provides many predefined one's, it does not provide all possible combinations, and you might end up having to write your own. The STL does not at all provide any "ternary operators", so if you need one of those, you'll definitely have to write it yourself.

Note that this section is only about adding overloads for existing operators, if you want to add new operators to the scripting engine, the parsers need to be extended as well.

### 1.3.2. Loading Constructors

Constructors can only be added *after* a type has been loaded using `addType`. Say that the `ControlData` struct has a constructor:

```
struct ControlData {
    ControlData(double a, double b, double c)
        : x(a), y(b), z(c), sample_nbr(0)
    {}
    double x, y, z;
    int sample_nbr;
}
```

This constructor is not automatically known to the type system. You need to write a constructor function and add that to the type info:

```
ControlData createCD(double a, double b, double c) {
    return ControlData(a,b,c);
}

// Tell Orocos a constructor is available:
// Attention: "ControlData" must have been added before with 'addType' !
RTT::TypeInfoRepository::Instance()->type("ControlData")->addConstructor(
&createCD );
```

From now on, one can write in a script:

```
var ControlData cd = ControlData(3.4, 5.0, 1.7);
```

Multiple constructors can be added for the same type. The first one that matches

with the given arguments is then taken.