

RTMaps
Software Development Kit
Version 3.0

Developer's manual

by Intempora S.A.

DISCLAIMER

INTEMPORA S.A. MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. Further, Intempora S.A. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Intempora S.A. to notify any person of such revision or changes.

Copyright © 1999-2005 Intempora S.A.

All rights reserved. Any transfer to anyone or reproduction of any part of this document by any means without prior written authorization from Intempora S.A. is strictly forbidden. The authorization to use any part of this document does not imply an unlimited public access to it.

^{RT}Maps (Real Time, Multisensor, Advanced Prototyping Software) is a registered trademark of Intempora S.A.
All other company, product, or service names mentioned in this book may be trademarks or service marks of their respective owners.

Intempora S.A.
2, place Jules Gévelot
92 130 Issy les Moulineaux
FRANCE
Tel: +33 1 41 90 03 59
Fax: +33 1 41 90 08 39
<http://www.intempora.com>

Contents

1	Introduction	5
1.1	RTMaps SDK content and purpose	5
1.2	Limitations of this document	7
1.3	Installing RTMaps Developer	7
1.4	Installation content	8
1.4.1	Under Windows 2000/XP	8
1.4.2	Under Linux	9
2	Your first RTMaps application	11
2.1	The RTMaps component model	11
2.2	Your first component	12
2.2.1	Running RTMaps SDK on Microsoft Visual C++ .NET 2003	14
2.2.2	The RTMaps SDK Wizard	15
2.2.3	The RTMaps component skeleton	19
2.2.4	The header file: <code>maps_MyCounter.h</code>	19
2.2.5	The implementation file: <code>maps_MyCounter.cpp</code>	20
2.2.6	Component structure definition	22
2.2.7	Initialization	27
2.2.8	Component Core	28
2.2.9	Conclusion	30
2.3	Your first Record/Replay Method (RRM)	33
2.3.1	The Record/Replay method skeleton	33
2.3.2	The RRM definitions part	34
2.3.3	The record-specific part	35
2.3.4	The replay-specific part	37
2.3.5	MyRRM implementation	39
2.3.6	The Header file	39

2.3.7	Implementation of the Record part	40
2.3.8	Implementation of the Replay part	43
2.3.9	Conclusion	44
2.4	Coding conventions	46
3	In the heart of ^{RT}Maps	47
3.1	^{RT} Maps components in detail	47
3.1.1	Data flow and data links	47
3.1.2	The MAPSIOEl _t class	48
3.1.3	Outputs or how to send data	50
3.1.4	Inputs or how to receive data	53
3.1.5	Properties	62
3.1.6	Actions	67
3.1.7	The component itself	69
3.1.8	Dynamically creating inputs/outputs/properties/ac- tions	71
3.1.9	The components threading model	72
3.1.10	Constructor/destructor of a component	76
3.2	^{RT} Maps record/replay methods in detail	77
3.2.1	RRM definition macros	77
3.2.2	The RRM methods	80
3.3	Time handling functions	81
3.4	Controlling the ^{RT} Maps clock	82
3.4.1	Applying corrections to the standard clock	82
3.4.2	Implementing your own custom clock	83
3.5	Debugging an ^{RT} Maps module	84
3.5.1	With Microsoft Visual Studio	84
3.5.2	Under Linux, using <code>ddd</code>	85

Chapter 1

Introduction

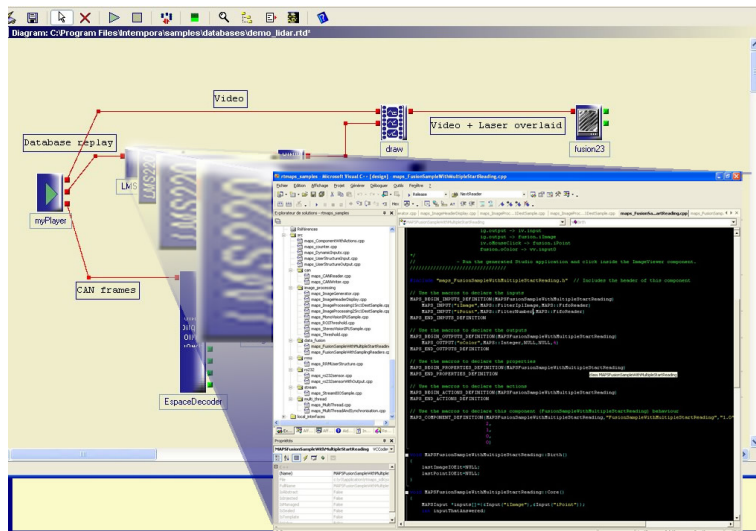
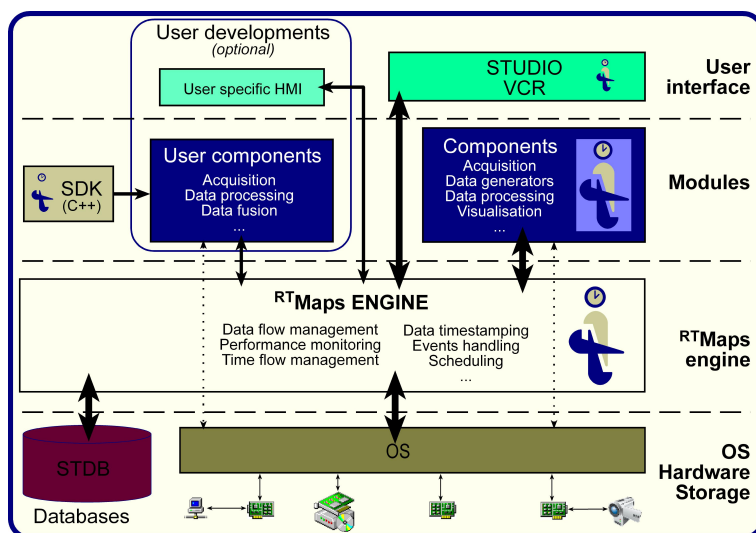
This manual is intended for developers. If you are new to ^{RT}Maps, first read the ^{RT}Maps User's manual. The ^{RT}Maps Software Development Kit (SDK), included in the ^{RT}Maps Developer Edition, enables C++ programmers to develop their own ^{RT}Maps components and RRM (Record-Replay-Methods), compile them into binary packages (‘.pck’ files) and load them dynamically within the ^{RT}Maps applications for real-time execution.

Such ^{RT}Maps components can then be shared between different development teams, either as the source code or as binaries according to your needs and constraints. This greatly facilitates cooperation on advanced real-time applications. Figure 1.2 shows the different features of the ^{RT}Maps product and the way they work together.

1.1 ^{RT}Maps SDK content and purpose

The ^{RT}Maps SDK is part of the ‘^{RT}Maps Developer Edition’. It contains:

- A set of C++ header files and compiled libraries that will be used to create packages.
- The source code of several simple components, which you may modify or use as samples.
- This linear documentation for ^{RT}Maps SDK beginners which goes through step by step information about how to program and compile ^{RT}Maps components.

Figure 1.1: The $RTMaps$ Software Development Kit.Figure 1.2: The $RTMaps$ architecture.

- An online *HTML Reference* which details the ^{RT}Maps API classes, methods and data structures. *It is strongly advised for ^{RT}Maps beginners to print the online documentation and read it entirely at least once in order to have an exhaustive overview of the possibilities offered by the ^{RT}Maps API.*
- A *Wizard* which helps you to create compilation projects and source code skeletons (available for Microsoft VC .NET 2003 and Microsoft VC 6 on Windows or gcc on Linux).

The purpose of the SDK is to expand the capabilities of ^{RT}Maps Standard by the creation of new components and RRM.

Components are modules used to acquire, process or visualize data.

Record Replay Methods (RRM) are modules used to store data in an ^{RT}Maps database or to retrieve data from it.

1.2 Limitations of this document

This document is *not*:

- A C++ or Object Oriented Programming guide. The reader is supposed to be familiar with C++ and OOP concepts.
- A documentation about any IDE or compiler (Microsoft Visual C++, gcc for example).

1.3 Installing ^{RT}Maps Developer

The installation of ^{RT}Maps Developer Edition is a standard and simple setup procedure (see the ^{RT}Maps User's manual).

Requirements under Windows: To use ^{RT}Maps SDK, you will need Visual C 6 SP5 or later (Visual C .NET 2002, Visual C .NET 2003, ...). Other environments may be used but are not supported by Intempora.

Important note: ^{RT}Maps SDK is a software development framework you are developing in. You will add some or most of your code inside this framework, and this will change its content. Thus, do not forget that the entire framework and your own code will become precious data you will need to keep in a safe path, and probably to back up periodically in

order to secure your work.

Since ^{RT}Maps SDK is frequently used as a prototyping framework, your source code may often change. The use of a source code versioning system (such as `CVS` or `Subversion`) is strongly recommended to keep the track of your modifications.

In this document, the installation destination path will be referred to as `%RTMAPS_PATH%` from now on.

1.4 Installation content

1.4.1 Under Windows 2000/XP

In the `%RTMAPS_PATH%` directory, you will find the following directory structure:

- **bin:** This folder contains the `rtmaps.exe` main executable file and some associated DLLs.
- **documentation:** contains all the ^{RT}Maps SDK documentation files (including this documentation).
 - **studio_basics:** contains the ^{RT}Maps user's manual.
 - **sdk_basics:** contains this documentation.
 - **sdk_reference:** contains the ^{RT}Maps SDK API online reference documentation.
 - **xmlcomponentsdefinitions:** contains the ^{RT}Maps components xml files for online help and some design features (such as file browsers in the properties dialog boxes, ...).
- **include:** contains ^{RT}Maps SDK C++ header files.
- **java:** contains some ^{RT}Maps java files (for ^{RT}Maps Studio and the VCR) and the Java Runtime Environment needed for them.
- **lib:** contains the ^{RT}Maps SDK static libraries you will need to link to your own ^{RT}Maps components.
- **license:** contains ^{RT}Maps license and hardware key drivers.

- **packages:** contains the packages files distributed by Intempora along with ^{RT}Maps.
- **rtmaps_sdk:** contains your own ^{RT}Maps projects (projects and source code and packages) but also sample components.
 - **samples.u:** contains the ^{RT}Maps SDK samples.
 - **user_modules.u:** is a typical user project.
 - * **local_interfaces:** contains the project specific header files.
 - * **src:** contains the project source files (**.cpp** files).
 - **templates.u** those files are used by the ^{RT}Maps SDK Wizard. Do not modify them!
 - **packages:** contains your own compiled packages. Debug versions are compiled in the **debug** subfolder.
- **samples:** contains ^{RT}Maps samples (sample databases and diagrams).
- **tools:** some tools such as the ^{RT}Maps wizard for Visual Studio 6.

1.4.2 Under Linux

The directory structure for ^{RT}Maps SDK respects the common Linux tree:

- **/usr/local/**
 - **bin:** contains the ^{RT}Maps application binaries.
 - **doc/RTMaps:** contains all ^{RT}Maps user and developer documentation files.
 - **include/RTMaps:** contains the ^{RT}Maps SDK C++ headers.
 - **lib/RTMaps:** contains the static libraries of ^{RT}Maps SDK (**' .a '** files).
 - **share/RTMaps**
 - * **packages:** contains the distributed package files (**' .pck '** files).
 - * **samples:** contains the ^{RT}Maps samples (databases and diagrams).

- * `rtmaps_sdk`: contains ^{RT}Maps SDK sample files and projects (source code and common makefiles).
- `~` (your home directory)¹.
 - `~rtmaps_sdk/samples` a project containing all ^{RT}Maps SDK samples.
 - `~rtmaps_sdk/packages` contains your own compiled packages. Debug versions are compiled in the `debug` subfolder.

¹Note that this folders tree is not created during installation but at the first use of the ^{RT}Maps Wizard. See [2.2.2](#)

Chapter 2

Your first ^{RT}Maps application

So as to give you a quick overview of the whole process covered by the ^{RT}Maps SDK software, this chapter teaches you how to create two simple modules: first a data generator component, then a RRM (Record/Replay Method) module.

2.1 The ^{RT}Maps component model

The ^{RT}Maps Component Model is a *data flow-based* component model. In addition to the classical *properties* and *methods* traditionally associated with a component, the ^{RT}Maps components have *inputs* and *outputs*.

Beyond this *data flow* orientation, the ^{RT}Maps component model was designed with *dynamism* in mind. *Dynamism* means that new inputs, outputs, properties or actions can be dynamically added to or removed from a component, even at run time; components can be started and stopped at run time. “Dynamism” also means that you can attach and detach components at will. An application is thus not statically defined but can evolve with time.

Furthermore, the ^{RT}Maps component model was designed with some keywords in mind:

- **Speed**, since ^{RT}Maps was designed to target real-time applications involving vision processing and control. This inclination for speed is reflected by the *multi-threaded* architecture, the ‘*copyless*’ data links between components, and the choice of the C++ language for both the ^{RT}Maps engine and the main components.
- **Time**, since one of ^{RT}Maps tasks is the timestamping of all incoming and processed data. One of the unique features of ^{RT}Maps is its *double timestamping* feature.
- **Recording/Replaying**, since ^{RT}Maps was designed to build multi-sensor databases. The idea is that any data flow can be recorded at will, be replayed later and reproduced with a high accuracy. In order to provide powerful recording and replaying facilities, the ^{RT}Maps Component Model supports some specific modules, called Record/Replay methods, or `MAPSRecordReplayMethod`. These components have only one input (during recording) or one output (during replay). They provide methods to record the data (the `Store` function), to replay the data (the `Replay` function) and to copy the data (the `Copy` function), the latter being necessary for database editing. `MAPSRecordReplayMethods` are the only components that are authorized to interact with files through the ^{RT}Maps engine. This ensures that data recording is made in an ^{RT}Maps compatible way.
- **Ease of use**, since ^{RT}Maps was designed to simplify the development of complex applications. Component programming remains very simple and only requires C++ knowledge. Have a look at COM/DCOM or CORBA programming and you will see the difference.

Furthermore, as the ^{RT}Maps component model is directly built upon the C++ object model, it can take advantage of C++ facilities such as inheritance or polymorphism. Figure 2.1 shows a brief overview of the class hierarchy for `MAPSComponent` and `MAPSRecordReplayMethod` classes. Refer to the HMTL online reference for more information.

2.2 Your first component

As a first blink on ^{RT}Maps module conception, we build a component from scratch that periodically increments its counter and outputs the

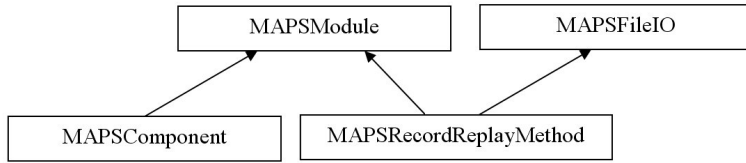


Figure 2.1: The class hierarchy for the `MAPSComponent` and `MAPSRecordReplayMethod` classes.

counter value to other component inputs. Although very simple, this component covers some fundamental and helpful features of `RTMaps` SDK:

1. Output declarations and management.
2. Property definitions and access.
3. Use of some standard `RTMaps` API functions (i.e. the `MAPS::Wait` method).

The following figure shows the schematic view of the component we want to realize here:

- The component name is `Counter`.
- It has one output named `outputInteger`, that allows it to send its counter value to other modules or that could be recorded in a file.
- It has two properties to configure its behavior at run-time:
 - `startTime`: contains the time from which the component will start to count and send data (in microseconds).
 - `interval`: the time interval between two increments of the counter value (in microseconds).

Figure 2.2 shows the structure of such a component.

In `RTMaps`, names are case sensitive (input, outputs, properties, components, ...).

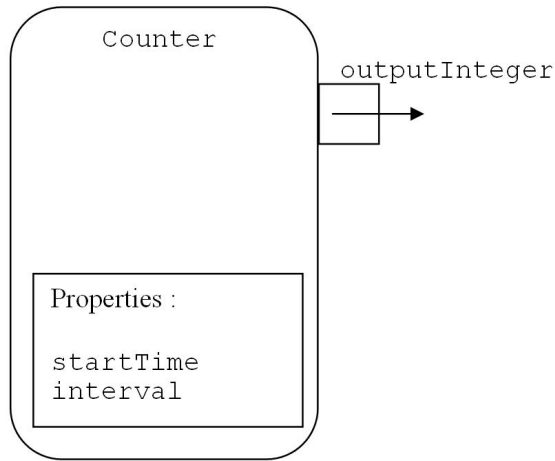


Figure 2.2: The `Counter` sample component.

2.2.1 Running ^{RT}Maps SDK on Microsoft Visual C++ .NET 2003

The paragraph arbitrarily describes the procedure within the VC .NET 2003 environment. Note that the procedure with VC 6.0 is nearly the same concerning these operations. Under Linux, the compilation and code edition procedures are the usual ones.

First of all, if not already done, install the development environment you will use with ^{RT}Maps SDK: after a successful installation of both Visual C++ .NET 2003¹ and ^{RT}Maps Developer Edition (see 1.3), launch ^{RT}Maps SDK using the Start menu of Windows. Visual C++ .NET 2003 will open the solution and display a window similar to the one shown on figure 2.3.

In the **Solutions explorer** window, you can see a tree of folders, corresponding to the source code files of a standard ^{RT}Maps SDK. As this is the first time you open this solution, you can build the ^{RT}Maps sample components with Visual C++: in the 'Build' menu, click on

¹Under Windows, Visual C++ 6.0 is still supported.

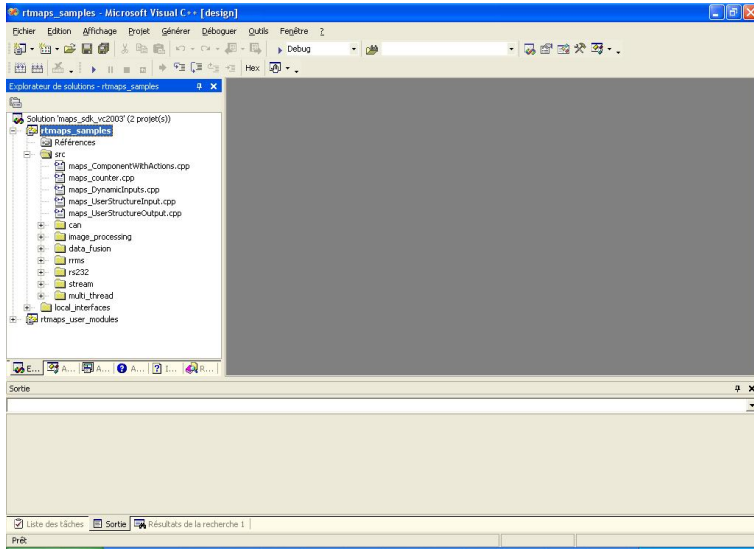


Figure 2.3: The standard ^{RT}Maps SDK solution and sample projects.

‘Build `rtmaps_samples`’ item (or press ‘F7’). Visual C++ will build a binary file leading to the package ‘`rtmaps_samples.pck`’.

The **Output** window (generally, at the bottom of the Visual Studio window) will indicate a successful build: the default ^{RT}Maps SDK package is generated! You can find the package file in the directory `%RTMAPS_PATH%/packages/rtmaps_samples.pck`.

This created `pck` file is a dynamically linked library. The sample components contained in this file will not be included by default in the ^{RT}Maps application but it can be loaded dynamically while running ^{RT}Maps. The components contained in the file will be automatically inserted in the list of available components (see the related paragraphs in the ^{RT}Maps User’s manual).

2.2.2 The ^{RT}Maps SDK Wizard

The aim is now to write our first ^{RT}Maps component as described above, the `Counter` component. The best way to create a new module

(component or RRM) is most of the time to start from automatically generated project and code skeletons. This is the job of the ^{RT}Maps SDK Wizard.

Using Windows and MSVC .NET 2003

The ^{RT}Maps SDK is integrated within Microsoft Visual C++ .NET 2003. You can find it in the 'Tools | RTMaps' menu.

To add a new component to an existing project (a *package* project), first make sure that the project is currently selected in the Project Explorer window of VC .NET. Then choose 'Tools | RTMaps | RTMaps new module wizard'. As shown on figure 2.5, enter 'MyCounter' in the Module name section. Then click on OK.

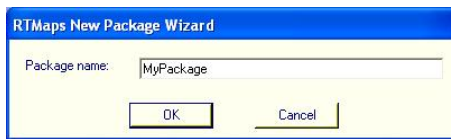


Figure 2.4: The ^{RT}Maps SDK Wizard form for a new package (VC .NET 2003).

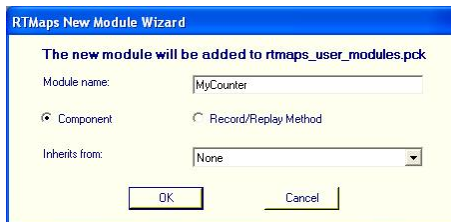


Figure 2.5: The ^{RT}Maps SDK Wizard form for a new component (VC .NET 2003).

Two new files called `maps.MyCounter.cpp` and `maps.MyCounter.h` have been generated (respectively in `%RTMAPS_PATH%/rtmaps_sdk/user_modules.u/src/` and in `%RTMAPS_PATH%/rtmaps_sdk/user_modules.u/local_interfaces`) and included in the `user_modules` project. The new component is now created. At this state, the new component can be

compiled (you can generate the `user_modules` project), even if it does not do much in the ^{RT}Maps Studio.

Using Windows and MSVC 6.0 SP5

The wizard for Microsoft Visual C++ 6.0 is not integrated in the IDE. It is a separate program called `rtmaps_sdk_wizard.exe` and it can be found in the directory `%RTMAPS_PATH%/tools`.

In the wizard API, ask for a new component creation. Then choose `user_modules` in the **Package name** text box, and type `MyCounter` in the **Component Name** text box. Then click on the button ‘**Make**’. (Leave the center field as ‘`- None -`’ for the moment) as shown on figure 2.7.



Figure 2.6: The ^{RT}Maps SDK Wizard for VC6.

Two new files called `maps_Counter.cpp` and `maps_Counter.h` have been generated (respectively in `%RTMAPS_PATH%/rtmaps_sdk/user_modules.u/src/` and in `%RTMAPS_PATH%/rtmaps_sdk/user_modules.u/local_interfaces`) and included in the `user_modules` project (`rtmaps_user_modules.dsp` has then been modified). The new component is now

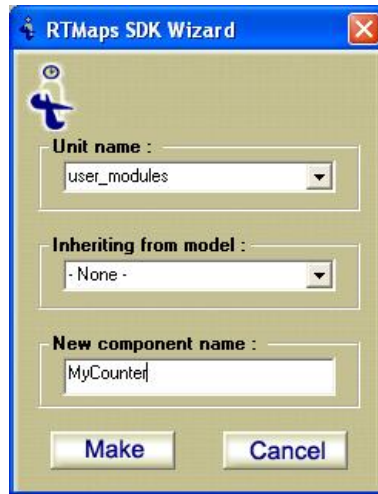


Figure 2.7: The ^{RT}Maps SDK Wizard form for a new component (VC6).

created. You can switch back to Visual C++. Since a project of the current workspace has been modified, Visual C++ should display a dialog box asking if you want to reload the `rtmaps.user_modules.dsp` project. Choose 'yes'. At this state, the new component can be compiled (you can generate the `user_modules` project), even if it does not do much in the ^{RT}Maps Studio.

Using Linux

On Linux, the wizard is called `RTMapsSDKWizard`. It can be found in `/usr/local/bin`. If you have not already added `/usr/local/bin` to your path, it is a good idea to do it now.

To create your first component `MyCounter` in the package `user_modules`, simply type in a terminal:

```
RTMapsSDKWizard -c user_modules MyCounter
```

The wizard will ask you first if you want to create the sdk folder tree in your home directory. You have to say yes (type 'y') to use the ^{RT}Maps

SDK. Then the wizard will create the directory `~/rtmaps_sdk` and all of its subdirectories (`samples.u`, `user_modules.u` and `templates.u`). If the directory `user_modules.u` does not exist in `~/rtmaps_sdk`, you are asked if you want to create it.

Then you can choose the type of module (component or RRM) you want to create. Choose a component (type '1'). Two new files called `maps_MyCounter.cpp` and `maps_MyCounter.h` have been generated (respectively in `~/rtmaps_sdk/user_modules.u/src/` and in `~/rtmaps_sdk/user_modules.u/local_interfaces`) and included in the `user_modules` package (`~/rtmaps_sdk/user_modules.u/makefile` has then been modified). The new component is now created.

Important note: you may have to edit the file `templates.u/makefile.inc` to fit your Linux installation (especially if you do not use `g++`).

2.2.3 The ^{RT}Maps component skeleton

The template files for ^{RT}Maps component creation are included in the `templates.u` directory; they are called `maps_ComponentTemplate.cpp` (in `src` directory) and `maps_ComponentTemplate.h` (in `local_interfaces` directory). They consist in a complete definition of the component called `ComponentTemplate` (that makes nothing else but being a component!). They can be considered an ^{RT}Maps component skeleton.

The wizard created a file called `maps_MyCounter.h` in the `local_interfaces` directory of the `user_modules` package. This file is the same as the file `maps_ComponentTemplate.h`, but the wizard has replaced the string `“ComponentTemplate”` by the string `“MyCounter”` everywhere. The wizard also created a file called `maps_MyCounter.cpp` in the `src` directory of the `user_modules` package. Understanding that, we can step into these new files.

2.2.4 The header file: `maps_MyCounter.h`

The header file (`maps_MyCounter.h`) contains the declaration of the `MAPSMYCounter` class. This class is a public child of the `MAPSComponent` class (which is itself a public child of the `MAPSModule` class). (`MAPSComponent` and `MAPSModule` are declared in the included file

maps.hpp). Then a standard component declaration macro `MAPS_COMPONENT_STANDARD_HEADER_CODE(...)` makes some declarations that are common to all ^{RT}Maps components. The first argument of this macro is the name of the component C++ class (here, `MAPSMYCounter`). The new C++ class will also be called `MAPSMYCounter`.

Table 2.1 shows how this header file looks like at this stage.

```

////////////////////////////////////////
// RTMaps SDK Component header
////////////////////////////////////////

#ifndef _Maps_MyCounter_H
#define _Maps_MyCounter_H

// Includes maps sdk library header
#include "maps.hpp"

// Declares a new MAPSComponent child class
class MAPSMYCounter : public MAPSComponent
{
    // Use standard header definition macro
    MAPS_COMPONENT_STANDARD_HEADER_CODE(MAPSMYCounter)
private :
    // Place here your specific methods and
    // attributes
};

#endif

```

Table 2.1: The `maps_MyCounter.h` file as generated by the wizard.

2.2.5 The implementation file: `maps_MyCounter.cpp`

The file `maps_MyCounter.cpp` contains the implementation of the `MAPSMYCounter` class. At the beginning of the file, note that we include the header file defined above: `maps_MyCounter.h`. In this file you will specify:

- *Input definitions* with the help of macros: inputs are the data sources of your component.
- *Output definitions* with the help of macros: outputs allow you to send data to other (downstream) components.
- *Property definitions* with the help of macros: these properties allow you to change the behavior of your component at runtime.

- *Action definitions* with the help of macros: actions are kind of runtime function calls triggered by the ^{RT}Maps application user. They are useful to easily implement component actions due to user events.
- **MAPSMYCounter::Birth** method. This method is executed as soon as the component is born, that is when you enter the Run mode.
- **MAPSMYCounter::Core** method. This method is executed as often as possible to let the component behave the way it is intended to.
- **MAPSMYCounter::Death** method. This method is executed before the component dies, that is, when you leave the Run mode.

```

////////////////////////////////////
// RTMaps SDK Component
////////////////////////////////////

////////////////////////////////////
// Purpose of this module :
////////////////////////////////////

#include "maps.MyCounter.h" // Includes the header of
                           // this component

// Use the macros to declare the inputs
MAPS_BEGIN_INPUTS_DEFINITION(MAPSMYCounter)
    //MAPS.INPUT("iName",MAPS:: FilterInteger ,
    //              MAPS:: FifoReader)
MAPS_END_INPUTS_DEFINITION

// Use the macros to declare the outputs
MAPS_BEGIN_OUTPUTS_DEFINITION(MAPSMYCounter)
    //MAPS.OUTPUT("oName",MAPS:: Integer ,NULL,NULL,1)
MAPS_END_OUTPUTS_DEFINITION

// Use the macros to declare the properties
MAPS_BEGIN_PROPERTIES_DEFINITION(MAPSMYCounter)
    //MAPS.PROPERTY("pName",128,false,false)
MAPS_END_PROPERTIES_DEFINITION

// Use the macros to declare the actions
MAPS_BEGIN_ACTIONS_DEFINITION(MAPSMYCounter)
    //MAPS.ACTION("aName",MAPSMYCounter:: ActionName)
MAPS_END_ACTIONS_DEFINITION

// Use the macros to declare this component
// (MyCounter) behaviour
MAPS.COMPONENT_DEFINITION(MAPSMYCounter,"MyCounter","1.0",
    128,
    MAPS:: Threaded,MAPS:: Threaded ,
    0, // Nb of inputs

```

```

                                0, // Nb of outputs
                                0, // Nb of properties
                                0) // Nb of actions

void MAPSMYCounter::Birth()
{
    // Reports this information to the RTMaps console
    ReportInfo("MyCounter: Passing through Birth() method");
}

void MAPSMYCounter::Core()
{
    // Reports this information to the RTMaps console
    ReportInfo("MyCounter: Passing through Core() method");
    // Sleeps during 500 milliseconds
    // (500000 microseconds)
    Rest(500000);
}

void MAPSMYCounter::Death()
{
    // Reports this information to the RTMaps console
    ReportInfo("MyCounter: Passing through Death() method");
}

```

Table 2.2: The maps_MyCounter.cpp file as generated by the wizard.

2.2.6 Component structure definition

The new **MyCounter** component needs one output to transmit its counter value: its data type could be an integer for example. Declare it thanks to the `MAPS_OUTPUT` macro. This macro, which must be inserted between the `MAPS_BEGIN_OUTPUTS_DEFINITION` and `MAPS_END_OUTPUTS_DEFINITION` macros, takes 5 arguments:

1. The first argument defines the output name.
2. The three following arguments describe the output type:
 - standard type
 - named type
 - unit type.
3. The last argument defines the output vector size. This is useful if you need to exchange vector data (e.g. vector of floats for 2D

or 3D points coordinates, vectors of very high frequency data that are processed in packets...).

Here is the code corresponding to our output definition:

```
[...]
    MAPS_BEGIN_OUTPUTS_DEFINITION(MAPSMYCounter)
        MAPS_OUTPUT("outputInteger",MAPS::Integer, NULL, NULL
                    ,1)
    MAPS_END_OUTPUTS_DEFINITION
[...]
```

We can also define the properties of our component: between the `MAPS_BEGIN_PROPERTIES_DEFINITION` and `MAPS_END_PROPERTIES_DEFINITION` macros, we can insert properties definition by the mean of `MAPS_PROPERTY` macro which takes 4 arguments:

1. A character string that specifies the property name.
2. The property default value (and by the same way, this defines the property type).
3. A boolean that specifies whether this property has to be defined before the diagram begins to run. This will be explained in detail in [3.1.5](#).
4. A boolean that specifies whether this property may be modified while the diagram is running. This will be explained in detail in chapter [3.1.5](#).

Here is the code corresponding to our properties definition:

```
[...]
    MAPS_BEGIN_PROPERTIES_DEFINITION(MAPSMYCounter)
        MAPS_PROPERTY("startTime",0,false,false)
        MAPS_PROPERTY("period",1000000,false,true)
    MAPS_END_PROPERTIES_DEFINITION
[...]
```

The `startTime` property is an integer property with a default value of 0. This property defines the start time of the counter (in microseconds). It cannot be modified while the diagram is running.

The `period` property is also an integer property; its default value is 1 000 000. This property defines the interval between 2 increments of the counter (in microseconds). It can be defined while the diagram is running.

Now you need to complete your component definition according to the properties and outputs you already defined. This is done using the `MAPS_COMPONENT_DEFINITION` macro, which takes 10 arguments:

1. C++ class component name: here, it should be `MAPSMyCounter`, the name that was used in the outputs and properties macros until now.
2. Component model name: can be chosen freely but should not contain special characters such as spaces (it should be made of `[a-z][A-Z][0-9][_]` characters).
3. Version of the component.
4. Default priority of the component thread (if component is threaded, see Arguments 5 and 6).
5. Specify whether the component can run in **threaded** and/or **sequential** mode. In **threaded** mode, the component task will be performed in its own thread, whereas in **sequential** mode, it will run within the upstream component thread. Here, our component will only work in **threaded** mode. These two modes will be explained in detail in [3.1.9](#).
6. Specify the default mode of operation of the component (**threaded** or **sequential**). This is only relevant in case both modes are supported by the component (see the previous argument).
7. Number of inputs: should be 0 since none has been defined.
8. Number of outputs: should be 1 since we have defined 1 output.
9. Number of properties: should be 2 since we have defined 2 properties.

10. Number of actions: should be 0 since none has been defined.

```
MAPS_COMPONENT_DEFINITION( MAPSMYCounter, "MyCounter" ,
    "1.0" , 128 ,
    MAPS::Threaded, MAPS::Threaded ,
    0, // number of inputs
    1, // number of outputs
    2, // number of properties
    0) // number of actions
```

We have written above the C++ code to define the component structure: inputs, outputs, properties, and actions. What about the code for its run-time behavior? You may wonder what the component is doing in its current state. The answer to this question can be found by running the `MyCounter` component now. Here is the current `maps_MyCounter.cpp` file.

```
////////////////////////////////////
// RTMaps SDK Component
////////////////////////////////////

////////////////////////////////////
// Purpose of this module :
////////////////////////////////////

#include "maps_MyCounter.h" // Includes the header
                           // of this component

// Use the macros to declare the inputs
MAPS_BEGIN_INPUTS_DEFINITION(MAPSMYCounter)
MAPS_END_INPUTS_DEFINITION

// Use the macros to declare the outputs
MAPS_BEGIN_OUTPUTS_DEFINITION(MAPSMYCounter)
    MAPS_OUTPUT("outputInteger", MAPS::Integer, NULL, NULL, 1)
MAPS_END_OUTPUTS_DEFINITION

// Use the macros to declare the properties
MAPS_BEGIN_PROPERTIES_DEFINITION(MAPSMYCounter)
    MAPS_PROPERTY("startTime", 0, false, false)
    MAPS_PROPERTY("period", 1000000, false, true)
MAPS_END_PROPERTIES_DEFINITION

// Use the macros to declare the actions
MAPS_BEGIN_ACTIONS_DEFINITION(MAPSMYCounter)
MAPS_END_ACTIONS_DEFINITION

// Use the macros to declare this component
```

```

// (MyCounter) behavior
MAPS_COMPONENT_DEFINITION(MAPSMYCounter, "MyCounter", "1.0",
                          128,
                          MAPS::Threaded, MAPS::Threaded,
                          0, // Nb of inputs
                          1, // Nb of outputs
                          2, // Nb of properties
                          0) // Nb of actions

void MAPSMYCounter::Birth()
{
    // Reports this information to the RTMaps console
    ReportInfo("MyCounter: Passing through Birth() method");
}

void MAPSMYCounter::Core()
{
    // Reports this information to the RTMaps console
    ReportInfo("MyCounter: Passing through Core() method");
    // Sleeps during 500 milliseconds (500000 microseconds)
    Rest(500000);
}

void MAPSMYCounter::Death()
{
    // Reports this information to the RTMaps console
    ReportInfo("MyCounter: Passing through Death() method");
}

```

Table 2.3: The maps_MyCounter.cpp file (step 2).

Now you can build the `rtmaps_user_modules.pck` package containing this component (right click on the `rtmaps_user_modules` project and choose Build in Visual C++ or type `make` in the folder `~/rtmaps_sdk-/user_modules.u` under Linux).

Launch ^{RT}Maps Studio.

Register the `rtmaps_user_modules.pck` package which can be found in `%RTMAPS_PATH%/rtmaps_sdk/packages/(debug)/`. The `MyCounter` component should appear in a section called `rtmaps_user_modules.pck` in the 'Component list' window.

Place this `MyCounter` component in the diagram and run it.

The console information window notifies the creation of a new task (thread) for our component:

‘‘Starting main thread 0x1f8 for component MyCounter_1’’.

Thus the following succession of tasks occurs:

- **Birth** method: Cf ‘‘Info: Component MyCounter_1: Passing through Birth() method.’’ Birth() method is the place for the initialization phase of the module.
- It loops in the **Core** method: Cf. ‘‘Info: Component MyCounter_1: Passing through Core() method’’. Core() method is the place for normal and functional behavior of the module.
- **Death** method when you click on Shutdown button in ^{RT}Maps Studio toolbar. Death() method is the place for the closure of the component.

The messages ‘‘Info: Component MyCounter_1 ...’’ are printed in ^{RT}Maps console window by the means of the ReportInfo method. This method is inherited from the MAPSComponent class, the ancestor class of all ^{RT}Maps components. You can use this method whenever you want to report important information to the users of your applications.

2.2.7 Initialization

Now we go back to the code in order to complete the development of the MyCounter component.

At run-time, when the component starts its job, we want it to get the ‘startTime’ parameter (property) which can be set by the user through the ‘MyCounter_1’ properties dialog box. We place this code in the Birth method in order to execute it once before any further operations. We also define a new attribute in our MAPSMYCounter class in order to store this value.

```
class MAPSMYCounter : public MAPSComponent {
    // Use standard header definition macro
    MAPS_COMPONENT_STANDARD_HEADER_CODE(MAPSMYCounter)
private :
    // Place here your specific methods and attributes
    MAPSTimestamp m_Appointment;
    int m_Ticks;
};
```

Then we can implement the `Birth()` method as follows:

```
void MAPSMYCounter::Birth() {
    ReportInfo(" Initialization");

    //Initializes the m_Appointment member variable
    m_Appointment = GetIntegerProperty("startTime");
    //Initializes our counter (m_Ticks)
    m_Ticks = 0;
}
```

We use the `GetIntegerProperty` method to retrieve the user defined value for the property `startTime`. If nothing has been specified by the user, the default value defined above (in the `MAPS_PROPERTY` macro) is returned. Initialization has been done!

2.2.8 Component Core

Now we shall define the most important part of the component: the `Core` method. It will contain the instructions to increment the counter we are implementing, to wait for the period of time specified in the property interval and to write the counter value to the output of the component.

```
void MAPSMYCounter::Core() {
    //Waits for the next appointment
    Wait(m_Appointment);

    //Declaration of a new pointer to the standard
    //input/output data exchange structure.
    MAPSIOElt* ioElt;
    // Asks for writing a new data to the
    //output "outputInteger".
    ioElt = StartWriting(Output("outputInteger"));
    if (ioElt == NULL)
        return;
    //We've got a new data where to write, so write
    //the ticks value in there.
    ioElt->Integer() = m_Ticks;
    //Then send the data to inputs connected
    //to our output
    StopWriting(ioElt);

    //According to the parity of the m_Ticks variable
    if (m_Ticks%2 == 0)
        //Tells "tic"
```

```

        ReportInfo("tic");
    else
        //or "toc"
        ReportInfo("toc");

    //Increments our very elaborated counter
    m_Ticks++;

    //Define a new appointment: we get
    //user-defined "period" property
    m_Appointment+= GetIntegerProperty("period");
}

```

This C++ code needs some explanations:

MAPSComponent::Wait method allows the programmer to put the calling task into wait mode until a specified ^{RT}Maps time has been reached.

MAPSComponent::StartWriting method: the call to **StartWriting(Output(‘‘outputInteger’’))** returns a pointer to the next output buffer element of the output named ‘‘outputInteger’’. An output buffer element contains the data, and also qualifies it (timings information, ...). This element will only be written and sent to the connected inputs (other components or record/replay methods) when the **StopWriting** method will be called. **StartWriting** returns a pointer to a **MAPSIOElt** structure.

MAPSIOElt::Integer method: returns the integer data contained in the output buffer element. You can use it directly because it returns a reference on this data (see a C++ documentation for more information about references or pointers).

ioElt->Integer() = m_Ticks sets the data to have the same value as counter ticks.

MAPSComponent::StopWriting method: this method validates the data you have been editing, and sends it to the connected outputs.

In order to fully test our component, we have to change the ‘MyCounter_1’ properties in the diagram (double click on the ‘MyCounter_1’ component in the diagram window). For example you can set the **startTime** to 4000000 (second 4.000), and the **period** to 2000000 (2 seconds) as shown in figure 2.8. We may add an



Figure 2.8: The MyCounter properties.

Oscilloscope² and a Recorder component to improve our ^{RT}Maps application. The result should look like shown in figure 2.9.

The Studio console window reports each counter increment, while the oscilloscope window shows the counter value growing. The Recorder also records your component output data. The database will be created in the ^{RT}Maps process current directory, i.e. the directory from which ^{RT}Maps was launched (it can be `%RTMAPS_PATH%/maps_pck_sdk/user_modules.u/` in case you launch ^{RT}Maps directly from Visual C++), in a new folder whose name has the following format: `date.time.recordername`.

2.2.9 Conclusion

This MyCounter sample introduced you with the different steps of the creation and use of your own ^{RT}Maps component. These steps are:

1. Use the wizard to create a new component skeleton.
2. Define your inputs, outputs, properties and actions with the help of ^{RT}Maps definition macros.
3. Define your component settings with the help of `MAPS_COMPONENT_DEFINITION` macro.

²Properties chosen for the Oscilloscope can be: `scroll=true`, `inputA.scale=10.0`, `inputA.color=255`.

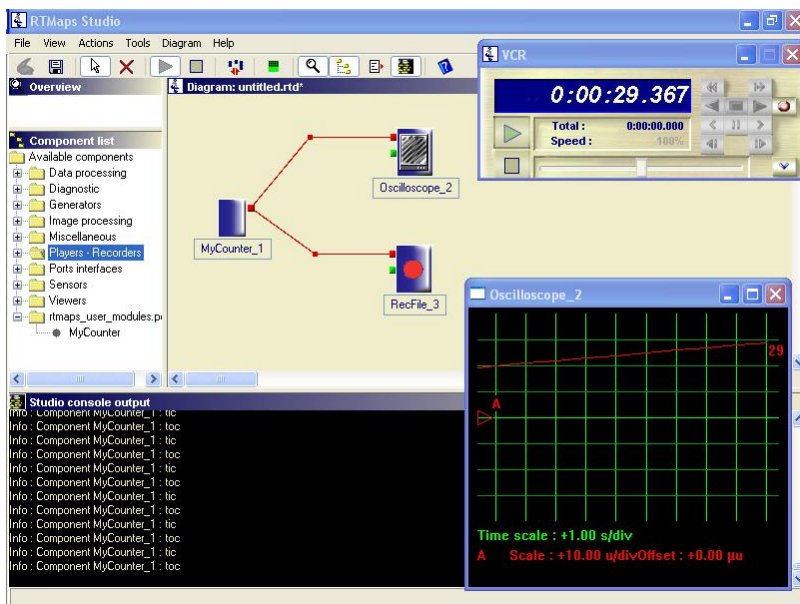


Figure 2.9: A simple application with our `MyCounter` component.

4. Program your component behavior: initialization in `Birth()` method, closure in `Death()` method, and functional behavior in the `Core()` method. Insert in the class `MAPSYourComponentName` whatever methods or attributes you need.
5. Launch the ^{RT}Maps Studio, register the package you created (or placed it in the list of default packages that are registered automatically at ^{RT}Maps startup) and make your diagram.
6. Run and test your component!³

³Step by step debugging is possible through `Visual C++` on Windows or `ddd` on Linux for example, following the usual procedure. See [3.5](#) for details.

2.3 Your first Record/Replay Method (RRM)

In this section you will follow step by step procedures to program your own Record/Replay Method in case you need a specific recording format for your data.

2.3.1 The Record/Replay method skeleton

As for component creation, you first have to use the wizard to create a new RRM template. MyRRM is the name of the Record/Replay Method we are going to create. A glance at the files created by the wizard is a good way to start understanding RRMs.

```

////////////////////////////////////
// RTMaps SDK Component
////////////////////////////////////

#ifndef _MAPSMYRRM.H
#define _MAPSMYRRM.H

// Includes maps sdk library header
#include "maps.hpp"

// Declares a new MAPSRecordReplayMethod child class
class MAPSMYRRM: public MAPSRecordReplayMethod
{
    // Use standard header definition macro
    MAPS_RECORD_REPLAY_METHOD_STANDARD_HEADER_CODE(
        MAPSMYRRM);
private :
    // Place here your specific methods and attributes
    MAPSString hint;
};

#endif

```

Table 2.4: The maps_MyRRM.h file as generated by the wizard.

The file `maps_MyRRM.cpp` is composed of 3 main parts:

1. The definition part.
2. The record-specific part.
3. The replay-specific part.

2.3.2 The RRM definitions part

```

////////////////////////////////////
// RTMaps SDK Component
////////////////////////////////////

////////////////////////////////////
// Purpose of this module :
////////////////////////////////////

#include "maps.MyRRM.h"

////////////////////////////////////
// Record/Replay common definitions
////////////////////////////////////

// Properties definition
MAPS_BEGIN_PROPERTIES_DEFINITION (MAPSMYRRM)
MAPS_END_PROPERTIES_DEFINITION

// Actions definition
MAPS_BEGIN_ACTIONS_DEFINITION (MAPSMYRRM)
MAPS_END_ACTIONS_DEFINITION

// RRM definition
MAPS_RECORD_REPLAY_METHOD_DEFINITION (MAPSMYRRM, "MyRRM", "1.0",
                                     MAPS::FilterInteger,
                                     false,
                                     false,
                                     0,0)

```

Table 2.5: The definitions in maps_MyRRM.cpp file as generated by the wizard.

The common definition part is composed of:

1. Property definitions: you have already seen such properties in the **MyCounter** sample. The principle is the same here for record/replay methods.
2. Action definitions.
3. RRM definition: specify here the name of the method, and other behavior settings like the data it can manage, the number of properties for the record method, the number of properties for the replay method.

2.3.3 The record-specific part

```

////////////////////////////////////
// Record specific implementation
////////////////////////////////////

void MAPSMYRRM::BirthRecord(void)
{
    // Reports this information to the RTMaps console
    ReportInfo("Passing through BirthRecord() method");
}

void MAPSMYRRM::DeathRecord(void)
{
    // Reports this information to the RTMaps console
    ReportInfo("Passing through DeathRecord() method");
}

// Hint() method: is called each time a data has to be
// recorded (IOelt) in the .rec file.
// Put the data to record in the returned string
const char* MAPSMYRRM::Hint(MAPSIOelt& IOelt)
{
    // Reports this information to the rtmaps console
    ReportInfo("Passing through Hint() method");

    // Prints the integer data as a decimal string in str
    MAPSStreamedString str;
    str<<MAPSSManip::dec<<IOelt.Integer();

    // Returns the decimal string in the non-temporary
    //(very important!) hint variable
    hint=str;
    return hint; // This string will be stored
                // in the .rec file
}

// Store method: can be used to store data in a
// programmer-defined file format when recorded
// an output data
void MAPSMYRRM::Store(MAPSIOelt& IOelt)
{
    // Reports this information to the RTMaps console
    ReportInfo("Passing through Store() method");
    // Unlocks the I/O element
    // (compulsory and very important!)
    Unlock(IOelt);
}

```

Table 2.6: The record-specific part in maps.MyRRM.cpp file as generated by the wizard.

You may recognize in this part the first two methods: `BirthRecord` and `DeathRecord` allow the programmer to make initialization and destruction in record mode.

The last two methods need more explanations. The better one might be to run this RRM first with this simple diagram (when connecting the `Randint` component to the `Recorder`, choose `MyRRM` as recording method) as shown in figure 2.10.

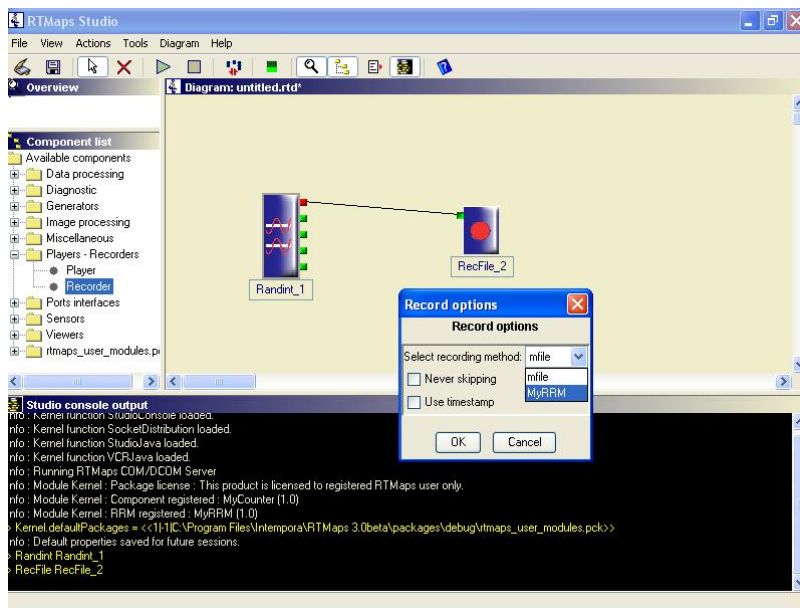


Figure 2.10: Recording with MyRRM.

Run this diagram. The `MyRRM` record/replay method generates information in the console window:

```

[... ]
Info : Component ri.outputInteger.MyRRM : Passing through BirthRecord() method
[... ] I
Info : Component ri.outputInteger.MyRRM : Passing through Hint() method

```

```

Info : Component ri.outputInteger.MyRRM : Passing through Store() method
Info : Component ri.outputInteger.MyRRM : Passing through Hint() method
Info : Component ri.outputInteger.MyRRM : Passing through Store() method
[...]
Info : Component ri.outputInteger.MyRRM : Passing through DeathRecord() method []

```

The **Hint** method is called each time a sample reaches the Recorder. This method is used to specify what is to be recorded in the `.rec` file (the main synchronization and data file). The `.rec` file may contain data itself according to the RRM we are talking about⁴. It is the case, for example, when using the `mfile` RRM, but not for the `jseq` (jpeg image compression RRM) method.

The **Store** method is also called each time a sample reaches the Recorder. It is mainly used to record data in separate files according to programmer-defined file formats.

2.3.4 The replay-specific part

```

////////////////////////////////////
// Replay specific implementation
////////////////////////////////////

void MAPSMYRRM::BirthReplay(void)
{
    // Reports this information to the RTMaps console
    ReportInfo("Passing through BirthReplay() method");
}

void MAPSMYRRM::DeathReplay(void)
{
    // Reports this information to the RTMaps console
    ReportInfo("Passing through DeathReplay() method");
}

// Replay() method: is called in replay mode when the
// user wants to replay previously recorded data by this
// RRM. Put in IOElt the data to replay, hint contains
// the string recorded by Hint() method in the .rec file.
bool MAPSMYRRM::Replay(MAPSIOElt& IOElt, int rec,
                       const char *hint, MAPSTypeInfo& type)
{
    // Reports this information to the RTMaps console
    ReportInfo("Passing through Replay() method");
}

```

⁴Data contained in the `.rec` file can only be stored in the form of character strings, and should be quite small, with no line breaks.

```

// The replay work is quite simple here :
// we just have to convert content of hint
// into an integer value. We set the data
// (integer) contained in IOElt to this value.
IOElt.Integer()=MAPS::Atoi(hint);

// And returns OK!
return true;
}

```

Table 2.7: The replay-specific part in maps_MyRRM.cpp file as generated by the wizard.

Experience what the replay of data means: launch the ^{RT}Maps Studio and instantiate a **Player** component in your diagram. Open the previously recorded database in the **Player**. Connect the created output on the **Player** to an **Oscilloscope** component. Run the diagram. You will get such an output in the console window at run time:

```

[...]
Info : Component counter.outputInteger.MyRRM : Passing through BirthReplay() method.
[...]
Info : Component counter.outputInteger.MyRRM : Passing through Replay() method.
Info : Component counter.outputInteger.MyRRM : Passing through Replay() method.
Info : Component counter.outputInteger.MyRRM : Passing through Replay() method.
Info : Component counter.outputInteger.MyRRM : Passing through Replay() method.
[...]
Info : Component counter.outputInteger.MyRRM : Passing through DeathReplay() method.
[...]

```

Once again, we have the **BirthReplay** and **DeathReplay** methods, in which we can do initialization and closure of variables. The **Replay** method is called by ^{RT}Maps each time a data has been reached in the database: when the ^{RT}Maps clock reaches the recorded time of data, this method is called with the following arguments:

IOElt is a container for the data to send to the connected inputs.

rec contains the rank of this data (the first data of this kind to be recorded was labeled “1”, ...).

hint contains the ASCII string recorded in the `.rec` file by the `Hint()` method. This string may be used to store the data itself in the simple cases. If the data has to be stored in another file, this string should give an index or any other information useful to know where the data is really stored.

type is set by `RTMaps` to the type of the data to replay. This information is retrieved from the `.rec` file.

2.3.5 MyRRM implementation

Above we have presented the architecture of Record Replay Methods. The sample we are going to build here is a record/replay method we could use with our `Counter` component. Our `MyRRM` will store data in two files:

- In the synchronization file (`.rec`): all data will be stored here using the `Hint()` method. Thus, the replay part of the RRM will be very easy to write.
- In a user-defined file and format: here for example, we will create a separate Matlab file, and store data using this software file format⁵.

2.3.6 The Header file

In the header file, we have to add some attributes in our class:

- `myMatlabFileHandle` will contain the handle to the custom file in the record mode.
- `firstTime` is a flag (boolean value) used to indicate if we have already written data into the files (in record mode).

```
#ifndef _MAPSRRMMyDataFormat_H
#define _MAPSRRMMyDataFormat_H

#include "maps.hpp"

class MAPSMYRRM : public MAPSRecordReplayMethod {
    MAPS_RECORD_REPLAY_METHOD_STANDARD_HEADER_CODE(
        MAPSMYRRM);
};
```

⁵The result will be pretty much the same as the `mfile` RRM included in the `RTMaps` Standard distribution.

```

    MAPSString m_Hint;
    MAPSFileWriteHandle *m_MyMatlabFileHandle;
    bool m_FirstTime;
};

#endif

```

Table 2.8: The new maps_MyRRM.h header file for the record-specific part.

2.3.7 Implementation of the Record part

In the **BirthRecord**, we have to create the separate file in which we want to write data in user-defined format.

The **FileName** method is used to get the full path of this file. This is a standard **MAPSRecordReplayMethod** method.

^{RT}Maps SDK is provided with a complete API to create, write and read files. This is because ^{RT}Maps also monitors file I/O performances. Thus, we use the **FileOpen4Writing** and **FileWrite** method to open and write to the file.

In the **DeathRecord** method, we have to close the separate file we created in **BirthRecord**. We use the **FileClose** method and the file handle.

In the **Store** method, we also use **FileWrite** to store each data and its timestamp (in milliseconds). The **MAPSStreamedString** class is helpful to manage strings as standard C++ streams.

The **Hint** method is also used to store the data in the **.rec** file.


```

////////////////////////////////////
// Record specific implementation
////////////////////////////////////

void MAPSMYRRM::BirthRecord(void)
{
    // Defines our specific file name
    MAPSSString myMatlabFileName=FileName() +
        MAPSSString(".m");
    // Opens this file in write mode
    if ((m_MyMatlabFileHandle=FileOpen4Writing(
        myMatlabFileName,
        0))==NULL)
        // Stops the RRM, and reports an error.
        Error("Error while creating the file");

    MAPSSString str="myVariable=";
    // Writes 'str' string into the file
    FileWrite(m_MyMatlabFileHandle, str, str.Length());
    // Sets firstTime to true
    m_FirstTime=true;
}

void MAPSMYRRM::DeathRecord(void)
{
    MAPSSString str="];\r\n\
        'Here we insert specific matlab \
        code\r\n\
        plot(myVariable, '+')';\r\n";
    // Writes 'str' string into the file
    FileWrite(m_MyMatlabFileHandle, str, str.Length());
    // Closes the file
    FileClose(m_MyMatlabFileHandle);
}

// Hint() method: is called each time a data has to be
// recorded (IOElt) in the .rec file.
//Put the data to record in the returned string
const char* MAPSMYRRM::Hint(MAPSIOElt& IOElt)
{
    // Returns the data (integer number) contained
    //in IOElt as a string
    m_Hint.Format("#", IOElt.Integer());
    return m_Hint;
}

// Store method: can be used to store data in a
// programmer-defined file format when recorded
// an output data
void MAPSMYRRM::Store(MAPSIOElt& IOElt)
{
    MAPSStreamedString str;
    // Prints in str:
    // - the integer value of the data contained
    //   in IOElt
    // - the timestamp of this data in milliseconds
    //   (rounded to the nearest millisecond)

```

```

// - ';' and newline{ascii(10)+ascii(13)} if this is
// not the first time we store data
if (m.FirstTime) // There is only 1 first time!
    m.FirstTime=false;
else
    str<<" ;\r\n";
str<<IOElt.Integer()<<" , "<<((int)((IOElt.Timestamp()
+500)/1000));

// Writes 'str' string into the file
FileWrite(m.MyMatlabFileHandle,(const char *)str,
    str.Length());

// Unlocks the I/O element
// (compulsory and very important!)
Unlock(IOElt);
}

```

Table 2.9: Implementation of the record part.

Test this Record Method with the **Counter** component written in 2.2.

- Launch the ^{RT}Maps Studio.
- Instantiate a **Counter** component and a **Recorder** in your diagram. Connect them together and choose **MyRRM** as recording method.
- Run the diagram.

The synchronization file (**.rec**) will contain something like the following ASCII records:

```

[Data]
00:00.0197 / MyCounter_1.outputInteger#0=
00:00.0544 @Record MyCounter_1.outputInteger(MyCounter_1.outputInteger[0x0000000000000002,
    ,16,1]) as MyRRM
00:01.0341 / MyCounter_1.outputInteger#1=1
00:02.0008 / MyCounter_1.outputInteger#2=2
00:02.9998 / MyCounter_1.outputInteger#3=3

```

```
00:03.9998 / MyCounter_1.outputInteger#4=4
00:04.9997 / MyCounter_1.outputInteger#5=5
00:05.9997 / MyCounter_1.outputInteger#6=6
00:07.0007 / MyCounter_1.outputInteger#7=7
```

In the same directory as the synchronization file, you should also find the data file stored by MyRRM.

```
myVariable=[0,20;
1,1034;
2,2001;
3,3000;
4,4000;
5,5000;
6,6000;
7,7001];
    'Here we insert specific matlab code
    plot(myVariable,'+');
```

2.3.8 Implementation of the Replay part

Here, to replay data stored with MyRRM, we have nothing to add to the BirthReplay, DeathReplay, and Replay methods. The whole replay job is done by the `IOElt.Integer()=MAPS::Atoi(hint);` command line in the `Replay()` method. See 3.2.2 for more details about these method parameters.

```
////////////////////////////////////
//  Replay specific implementation
////////////////////////////////////

void MAPSMYRRM::BirthReplay(void)
{
    // We have nothing to do here...
```

```

}

void MAPSMYRRM::DeathReplay(void)
{
    // We have nothing to do here...
}

// Replay() method: is called in replay mode when
// the user wants to replay previously recorded data
// by this RRM. Put in IOElt the data to replay,
// hint contains the string recorded by Hint() method
// in the .rec file
bool MAPSMYRRM::Replay(MAPSIOElt& IOElt, int rec,
                      const char *hint, MAPSTypeInfo& type)
{
    // The replay work is quite simple here: we just
    // have to convert the content of hint into an
    // integer value. We set the data (integer)
    // contained in IOElt to this value.
    IOElt.Integer()=MAPS::Atoi(hint);

    // and returns
    return true;
}

```

Table 2.10: Implementation of the replay part.

To replay the previously recorded database:

- Launch the ^{RT}Maps Studio.
- Instantiate a **Player** and an **Oscilloscope**
- Connect them together.
- Run the diagram.

2.3.9 Conclusion

This MyRRM sample introduced you with the different steps of the creation of your own ^{RT}Maps RRM and how to use it. These steps are:

- Use the wizard to create a new RRM template.
- Define your RRM settings using macros.

- Program your RRM Record behavior: initialization in `BirthRecord()` method, closure in `DeathRecord()` method, and its functional behavior in the `Hint()` and `Store()` methods. Insert whatever methods or attributes you need in the class `MAPSYourRRMName`.
- Register your package in ^{RT}Maps Studio, define your diagram, then run and test your RRM!

2.4 Coding conventions

The conventions proposed here are mainly common sense:

- Make as many modules as possible. That is, distinguish different tasks such as *data acquisition*, *data (pre-)processing*, *data fusion*, *data visualization*, and implement each of them in its own module.
- Do not ever store data in a file by other means than a RRM.
- Always use ^{RT}Maps macros when available. Do not ever try to declare modules without the help of macros.
- Always use ^{RT}Maps API methods instead of the OS specific methods.
- Always use ^{RT}Maps API methods instead of the standard C functions.
- Always use ^{RT}Maps API data types instead of the standard C data types.
- Give long and explicit names to your variables, modules, inputs, outputs, properties and so on.
- As soon as you begin using anything (such as an object and a FIFO slot) using an ^{RT}Maps primitive, think that there is often another primitive to call when you are through (to unlock the FIFO, free the memory and so forth).
- Other standard coding conventions that can be interesting (these are a subset of the *Hungarian Notation*):
 - Variable names start with a lowercase character (ex: `MAPSString hint;`).
 - Member variable names start with a ‘m_’ (ex: `bool m_FirstTime;`).
 - Classes and structures start with an uppercase character (ex: `class MAPSMYRRRM : public MAPSComponent`).
 - Class method names start with an uppercase character (ex: `void Dynamic();`).

Chapter 3

In the heart of ^{RT}Maps

At this point, we went through the step by step programming of two very simple ^{RT}Maps modules: a `Counter` component, and the `MyRRM` record/replay method. We have understood the basic (and most important) points for programming specific ^{RT}Maps modules. However, ^{RT}Maps presents many more powerful functionalities enabling you to fully customize your modules.

We will now go through the description of the in-depth concepts of ^{RT}Maps.

3.1 ^{RT}Maps components in detail

In the following section, we describe more in depth the structure of the ^{RT}Maps components, the ones which are used to make sensor acquisition, processing or HMI components. They are called `MAPSComponent`. We also describe the main objects that are always associated with them: the inputs, outputs, properties and actions, respectively called `MAPSInput`, `MAPSOutput`, `MAPSPROPERTY` and `MAPSACTION`.

3.1.1 Data flow and data links

The ^{RT}Maps Component Model implements links between components using the *one writer - several readers* paradigm. This means that the output of a component can be connected to several inputs, but one input cannot be connected to several outputs (see Figure 3.1).

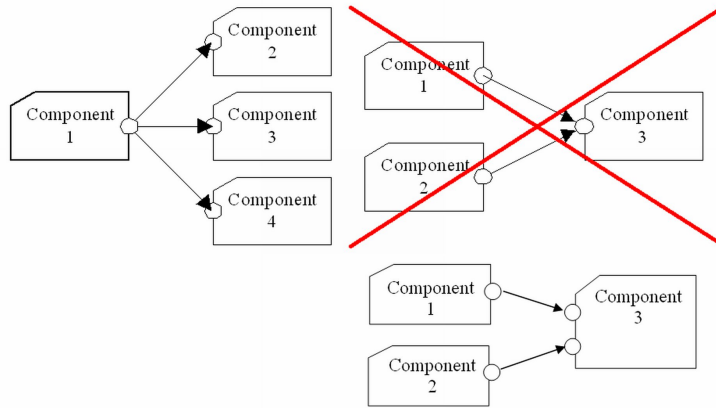


Figure 3.1: The data flow model in ^{RT}Maps (one writer - several readers).

We will first discuss the output topic, since outputs are the place where data are being generated and stored. After that, we will discuss inputs, which are the places where data are being consumed. We will end by discussing properties and actions.

3.1.2 The MAPSIOElT class

Data is exchanged between the components through the one and only structure: `MAPSIOElT`. This structure can encapsulate any type of data and gives access to the data itself, but also additional information such as the `Timestamp` or the `VectorSize`.

Both the `StartReading` and the `StartWriting` functions return a pointer to a `MAPSIOElT` structure. You can *read* the data contained in a `MAPSIOElT` returned by a `StartReading` call, and you can *write* data into a `MAPSIOElT` returned by a `StartWriting` call.

WARNING:

- NEVER WRITE DATA IN A *SOURCE* `MAPSIOElT` (RETURNED BY `StartReading`): YOU MAY CORRUPT THIS DATA FOR OTHER COMPONENTS WHICH INPUTS ARE CONNECTED TO THE SAME OUTPUT.

- NEVER WRITE DATA IN A *DESTINATION* `MAPSIOElt` (RETURNED BY `StartWriting`) *AFTER* THE CALL TO `StopWriting`: YOU MAY MODIFY DATA WHILE OTHER COMPONENTS ARE READING THE SAME BUFFERS.

Here are some useful methods of the `MAPSIOElt` class:

- `MAPSIOElt::Type`: retrieves the type of data contained in the `MAPSIOElt` object. This function can be used in conjunction with `MAPS::TypeFilter`.
- `MAPSIOElt::Timestamp` gets/sets the timestamp of the data. (Refer to the ^{RT}Maps User's manual for a detailed description of the `Timestamp` and `TimeOfIssue` fields.)
- `MAPSIOElt::TimeOfIssue`: gets/sets the time of issue. (Refer to the ^{RT}Maps User's manual for a detailed description of the `Timestamp` and `TimeOfIssue` fields.)
- `MAPSIOElt::VectorSize`: gets/sets the number of elements contained in the `MAPSIOElt`.
- `MAPSIOElt::BufferSize` gets the maximum number of elements that can be contained in the `MAPSIOElt`¹.
- `Integer(int index=0)`, `Integer64(int index=0)`, `Float(int index=0)`, `Stream8()`, `IplImage()`, `MAPSImage()`, `CANFrame(int index=0)`, ...: access to the data itself.
- `MAPSIOElt::Frequency`: for sampled signals, you can set/get the sampling frequency using this field. This frequency is expressed in `mHz`.
- `MAPSIOElt::Quality`, `Misc`: use this field to get/set additional information for particular data.

Refer to ^{RT}Maps online reference for more detailed information about the `MAPSIOElt` class.

¹This value is set when the output buffer is allocated (see 3.1.3).

3.1.3 Outputs or how to send data

First of all, we should concentrate on *FIFO buffers*. As you may know, FIFO means “*First In First Out*”. In ^{RT}Maps, the principle is one output, one FIFO buffer.

Each output is attached to a FIFO buffer where the component that owns the output will **write** some data, and where the components the inputs of which are connected to this output will **read** some data (see Figure 3.2).

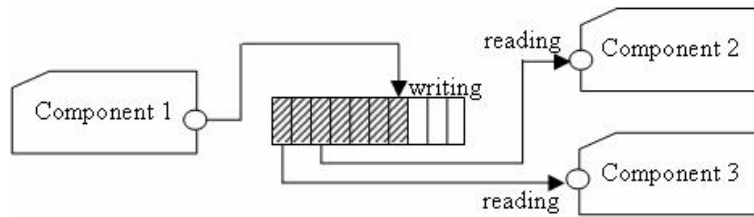


Figure 3.2: One output, one FIFO buffer.

Each of the sections of the FIFO buffer contains a `MAPSIOEl_t` object, which itself contains specific data among other common information. You will have to use this structure to access it (see 3.1.2).

In ^{RT}Maps, outputs have one VERY important responsibility: to allocate the memory for the FIFO buffer. This is usually done automatically, but sometimes, for complex data (such as images), the component programmer needs to specifically call the `MAPSOutput::AllocOutputBuffer` or any other similar function (ex: `MAPSOutput::AllocOutputBufferIplImage(...)`).

Sending data to the downstream component actually means writing data into a piece of the FIFO buffer. This is done through the `StartWriting` and `StopWriting` functions. Calling `StartWriting` gives a handle to one piece of the FIFO buffer (automatically chosen among the available ones), with write access (through a `MAPSIOEl_t` structure – see 3.1.2). Calling `StopWriting` means that you finished filling the

buffer with your piece of data, then it releases it to make it readable by all the downstream components.

Bear in mind that the FIFO buffer deallocation is done automatically by ^{RT}Maps. This will be one memory leak avoided...

Outputs definition macros

MAPS_BEGIN_OUTPUTS_DEFINITION

Opens the outputs definition section for the component specified as argument. Must be placed in the `.cpp` file, before the module definition.

Name	Type	Arguments
		Description
<i>Class name</i>	string	Name of the component C++ class.

MAPS_OUTPUT

Must be placed in a component outputs definition section. Defines an output for this component.

Name	Type	Arguments
		Description
<i>Output name</i>	string	Symbolic name of the output. This is the name that can be used in the execution script.
<i>Type</i>	long int	Standard data type. See the online reference for a detailed description of the data types and filters available.
<i>Data type name</i>	string	This is an alternative to the preceding argument that allows the designation of a data type by a string. Usually, you will use the preceding argument and let this one be NULL.
<i>Unit</i>	string	Represents the unit in which the value is expressed. (ex: "kmph").
<i>Element size</i>	int	Maximum size of the output vector in case this output is 'vectorizable'.

MAPS_END_OUTPUTS_DEFINITION

Ends the outputs definition section.

The **MAPS_OUTPUT_FIFOSIZE** macro is also available with an extra argument which enables you to set the default FIFO buffer size (the number of `MAPSIOEl_t` objects that can be queued in the output buffer). Refer to the online reference.

3.1.4 Inputs or how to receive data

When you give the description of an input, you need to specify its name – simply a character string –, a filter – indicating what kind of data can be sent to this input –, and a behavior. In the following example of input definition, the *name* is simply ‘‘input’’, the *filter* is `MAPS::FilterTextAscii`, which means that only ascii text can be sent to this input, and the default *behavior* of the input is `MAPS::FifoReader`.

```
MAPS_BEGIN_INPUTS_DEFINITION(MAPSTextComponent)
    MAPS_INPUT("input", MAPS::FilterTextAscii,
               MAPS::FifoReader)
MAPS_END_INPUTS_DEFINITION
```

Many other filters are available such as `FilterIntegers`, `FilterFloats` or `FilterNumbers` to handle numbers, `FilterIplImage` to handle uncompressed images that fit the standard `IplImage` structure, or `FilterMAPSImage` for other types of images, `FilterCANFrames` or `FilterStream8`. You can even make your own type of data and filter it: Refer to source code samples:

- `UserDefinitions.h`
- `maps_UserStructureOutput.h`
- `maps_UserStructureOutput.cpp`
- `maps_UserStructureInput.h`
- `maps_UserStructureInput.cpp`

Reading samples arriving on your component input(s) can be done through the `StartReading` functions.

Input “reader types”

The behavior (or *readerType*, `MAPS::FifoReader` in this case) does not only specify the way an ^{RT}Maps component handles the data that comes on its input, but it determines how it can be synchronized with the other components too. It determines the behavior of the `StartReading`

function: i.e. whether it has to block or not, and which sample (or `MAPSIOElT`) is retrieved from the output among all those that are present in the output FIFO buffer.

In the current ^{RT}Maps Component Model definition, five different behaviors are defined. They are `MAPS::FifoReader`, `MAPS::Wait4NextReader`, `MAPS::SamplingReader`, `MAPS::NeverSkippingReader` and `MAPS::LastOrNextReader`². We shall inspect them in detail.

The `FifoReader` type

This is the most common behavior for an input in ^{RT}Maps. As its name implies, this behavior makes use of the FIFO buffer which is connected to the input. This behavior is selected by specifying `MAPS::FifoReader` in the input definition macro.

Principle:

- If the circular buffer is empty or if all the samples within the circular buffer have already been read, a **StartReading** instruction on the input will *block* until some data arrives in the buffer (i.e. until the *writer* component outputs some new data).
- If the circular buffer contains some data that has not already been read by the input, the **StartReading** instruction will immediately return the oldest data in the buffer *that has not been read yet*. Once all the components connected to that output have read a piece of data, it is removed from the buffer.
- If the circular buffer gets full, i.e. if the *reader* component is not fast enough to read the data from the buffer, the new data arriving in the buffer will crush the oldest data – although the latter have never been read by the component. In this case, the component will never have access to these crushed data and the next **StartReading** will return the oldest data still present in the buffer. Thus as some data is missed by the component, a warning signal (`'FIFO overflow on output ...'`) is emitted by the ^{RT}Maps console.

Consequences:

²Note that, most of the time, this macro argument only defines the default behavior. The ^{RT}Maps user will be able to switch between `FifoReader`, `NeverSkippingReader` and `LastOrNextReader` if one of these is the default, through the `readerType` input property.

- If the downstream component calls **StartReading** at a higher frequency than that of the upstream component writes data, the buffer will be empty all the time. Therefore this will synchronize the execution of the reading component with the execution of the writing component, with no latency and no data loss.
- If for some reason the data flow arriving on the input is for a short period higher than what the component can actually consume, then the FIFO buffer will temporarily store the data still to be read. The reader will simply get it later: some following **StartReading** calls will get immediately this data. This is the purpose of the FIFO buffer: to avoid a data loss by providing a transitory storage place. Note that the data will be read with some lateness. This lateness is called latency time. Note also that since **StartReading** calls are not blocking during the catching up with the missed data, the synchronization with the writing component is momentarily lost.
- When the reading component is permanently too slow to cope with the arriving data flow, the circular buffer will be continuously full. Not only some data will be lost – leading to a lot of ‘**FIFO overflow**’ messages emitted by ^{RT}Maps –, but the time spent by the data in the buffer before being read will have significance. The latency time will be proportional to the size of the buffer. In addition to that, no synchronization will be kept between the reader and the writer, since the reader will always be late and try to keep on reading to catch up for its lateness. In brief, this is a very bad case that *must* be avoided. To fix this behavior, two solutions are available:
 - The first one is to specify a subsampling rate on the input. This will divide the arriving flow with a factor of *n*, that is, you will choose to consider only one piece of data out of *n*. The reader will lose some data, but you will know at which rate, and the synchronization with the writer will be restored.
 - The second one is to choose another behavior but ‘**FifoReader**’.

Applications:

- Data logging.
- Real-time processing of asynchronous data.

Note that the processing time of a component is very rarely constant, since it depends on the task switching on the processor, on the disk and I/O accesses, ... Unless a component is running at maximum thread priority and is the only one at this priority, it is very difficult to guarantee an execution time for a component task, even on a real-time system, since the interactions between tasks can be difficult to evaluate. What you can easily measure is an average processing time for a task. If the average processing time of a reading component is lower than the period of the data flow on its input, then the **FifoReader** behavior is very well suited: the component will process the whole data and the circular buffer will help dealing with the fickleness of the processing time of the component, thus avoiding any data loss.

The **LastOrNextReader** type

Principle:

- When calling **StartReading**:
 - If the buffer is empty, **StartReading** blocks until a new data arrives (same behavior as the **FifoReader**).
 - Otherwise, if the buffer contains unread samples, it will discard everyone of them but the most recent one and return this most recent **MAPSIOElt**.

As its name says, this means that **StartReading** either retrieves the last **MAPSIOElt** from the output FIFO buffer, or blocks and waits for the next one.

Consequences:

- This behavior is pretty much equivalent to the **FifoReader** but it can discard data even if the circular buffer is not full, and it cannot induce any latency when reading data (data read is always the most recent). If data is skipped and discarded, no warning message will be output.

Applications:

- Data display

The `SamplingReader` type

Principle:

- Calling `StartReading` on an input will get the last received data in the output buffer right away, even if this data has already been read. `StartReading` never blocks (actually it may block only once when ^{RT}Maps switches to the Running state when the output buffer is totally empty and until the first sample is output by the upstream component! You can test if there is or not a sample in the buffer with the `MAPSComponent::DataAvailableInFIFO(...)` function (see the SDK online reference).

Consequences:

- If the reading component is able to read at a higher frequency than that of the incoming data flow, the component will read several times the same data.
- In the opposite situation, if the reading component tries to read at a lower frequency than that of the incoming data flow, the component will miss the data that came in while it was processing the previous data.
- The `SamplingReader` behavior does not provide any synchronization between the reader and the writer. The synchronization is generally supplied by another reading attempt on another input of the same component with a synchronizing behavior (`FifoReader`, `Wait4NextReader` or `NeverskippingReader`) or by an *explicit pause* within the processing loop.

Applications:

- Re-sampling
- Actuation and system control

The `WaitForNextReader` type

Principle:

- The input does not consider any longer what is in the connected circular buffer. The reading is done directly on the arriving data flow. Thus, a `StartReading` call on the input always waits for a

new piece of data to arrive and gets it. It is always a blocking call. If some data arrived just before the call to `StartReading`, this data is *discarded*.

Consequences:

- If the reading component is fast enough to process the incoming data flow, it keeps on being synchronized with the writing component, and no data should be lost. Note that if the processing time of the reading component happens to be higher than the period of the incoming data flow, the input may miss some data. This missed data will permanently be lost.

Applications:

- This behavior is thus well suited for components that can tolerate to miss some data without any significant effect and that need to deal with up-to-date samples.

The `NeverskippingReader` type

Principle:

- The principle of this behavior is similar to the `FifoReader` behavior in that it makes full use of the circular buffer located between the input and output connections. The difference is that the `NeverskippingReader` does not tolerate any FIFO buffer overflow. It means that if the FIFO happens to be full, a `StartReading` call on a `NeverskippingReader` input will block the writing component until the circular buffer has a new place where to write.

Consequences:

- This behavior is the only one that can lead to a direct influence on another component but the reading component. Indeed, the writing component can be blocked by the reading component. This must be considered with care, and thus this behavior must be restricted to critical components that cannot tolerate any data miss at any price.

Applications:

- Post-processing for data conversion. Warning: when performing post-processing data conversion, make sure the inputs on the processing chain are set as `NeverskippingReader` otherwise, data may be discarded elsewhere.

Dealing with multiple inputs

Refer to source code samples:

- `maps_FusionSampleWithMultipleStartReading.h`
- `maps_FusionSampleWithMultipleStartReading.cpp`
- `maps_FusionSampleWithMultipleSamplingReaders.h`
- `maps_FusionSampleWithMultipleSamplingReader.cpp`
- `maps_StereoVisionIPUSample.cpp`

When programming components with several inputs, you will have to take much care about each input `readerType` and the choice of the suited `StartReading` function(s) called:

- Asynchronous reading on multiple inputs can be performed through the overloaded version of the `StartReading` function:

```
MAPSInput* myInputs[3] = {&Input("in1"),&Input("in2"),&Input("in3")};
int inputThatAnswered;
MAPSIOElt* ioElt = StartReading(3,myInputs,&inputThatAnswered);
```

- Synchronized reading on several inputs can be performed through the call to `SynchroStartReading` (see the online SDK reference for more details about this function).
- Triggering the reading on multiple inputs by the arrival of a sample on one particular input can be performed using several successive `StartReading` calls where the first input is set as `FifoReader`, and all the others are set as `SamplingReader`.

Inputs definition macros

MAPS_BEGIN_INPUTS_DEFINITION

Opens the inputs definition section for the component specified as argument. Must be placed in the `.cpp` file, before the module definition.

Arguments		
Name	Type	Description
<i>Class name</i>	string	Name of the component C++ class.

MAPS_INPUT

Must be placed in a component inputs definition section. Defines an input for this component.

Arguments			
Name	Type		Description
<i>Input name</i>	string		Symbolic name of the input. This is the name that can be used in the execution script.
<i>Type filter</i>	MAPSTypeFilterBase		^{RT} Maps filter describing the different data types the input may accept. Refer to the SDK online reference for a detailed description of the available data types and filters.
<i>Reader type</i>	MAPS::FifoReader, MAPS::Wait4NextReader, MAPS::Never-SkippingReader, MAPS::SamplingReader or MAPS::LastOrNextReader		Input behavior as described just above (3.1.4).

MAPS_END_INPUTS_DEFINITION

Ends the inputs definition section.

3.1.5 Properties

Properties are used to choose permanently (or at least for a long time) among the different behaviors a module may have. They may be accessed through the ^{RT}Maps control interface, unlike C++ attributes which may be called only in the C++ code.

Five types of properties are possible:

- *boolean* (**true** or **false**)
- *integer* (64-bit integers),
- *float* (**double** type, i.e. 64-bit floating numbers)
- *string*
- *enumeration* (a limited number of possible strings)

The default value and the type of the property is determined through the second parameter in the `MAPS_PROPERTY` definition macro. Ex:

- `MAPS_PROPERTY("myProp",true,false,false)` defines a boolean property with a default value set to **true**.
- `MAPS_PROPERTY("myProp",4,false,false)` defines an integer property with a default value set to 4.
- `MAPS_PROPERTY("myProp",4.0,false,false)` defines a float property with a default value set to 4.
- `MAPS_PROPERTY("myProp","myfile.txt",false,false)` defines a string property with a default value set to "myfile.txt".
- Enumeration properties are defined through a specific macro:
`MAPS_PROPERTY_ENUM("myProp","val1|val2|val3",1,false,false)` defines an enumeration property with "val1", "val2" and "val3", as the possible values. The default selected one is "val2" and is determined by the third argument of the macro ("1") which is the zero-based index of the default value in the table of possible values.

The two last boolean arguments of these macros respectively define:

- Whether the property has to be explicitly set before switching to the Running state. If set to **true**, this often means that the programmer cannot provide a default value for the property (ex: the 'file' property of the **Player** component).

- Whether the property can be changed while ^{RT}Maps is in the Running state.

To get the current value of a property within your code, call one of the following functions according to the property type:

- `GetBoolProperty`
- `GetIntegerProperty`
- `GetFloatProperty`
- `GetStringProperty`
- `GetEnumProperty`

In order to execute some code each time a property is changed, you may overload the `Set(MAPSPProperty& p, ... value);` virtual functions. Here is a short example that intercepts value changes for integer properties:

```
//Called each time the value of an integer property is
//changed.
void MAPSMyComponent::Set(MAPSPProperty& p, MAPSInt64
    value)
{
    //Check which integer property has changed.
    if (&p == &Property("myIntProp1")) {
        //Whatever...
        m_MyProp1Value = value;
    }
    else if (&p == &Property("myIntProp2")) {
        //Whatever...
        m_MyProp2Value = value;
    }
    //Don't forget to call the parent class
    //"Set" property since we overloaded a virtual
    //function:
    MAPSComponent::Set(p, value);
}
```

Refer to source code samples:

- `maps_SetProperty_Overload.h`
- `maps_SetProperty_Overload.cpp`

Read-only properties

Read-only properties can be used in order to display information to the ^{RT}Maps Studio user, without enabling him to change their values. For this, use the `MAPS_PROPERTY_READ_ONLY` declaration macro.

Properties definition macros

MAPS_BEGIN_PROPERTIES_DEFINITION

Opens the properties definition section for the component specified as argument. Must be placed in the `.cpp` file, before the module definition.

Name	Type	Arguments
		Description
<i>Class name</i>	string	Name of the component C++ class.

MAPS_PROPERTY

Must be placed in a component properties definition section. Defines a property for this component. Can be used for boolean, integer, float, or string properties. For enumeration properties, use the MAPS_PROPERTY_ENUM macro.

Arguments		
Name	Type	Description
<i>Property name</i>	string	Symbolic name of the property. This is the name that can be used in the execution script.
<i>Default value</i>	string, int, float, or boolean	The property default value (and by the same way defines the property type).
<i>needs2Be-Initialized</i>	boolean	Must the property be declared before ^{RT} Maps switches to Running state?
<i>canBe-Changed-AfterInstantiation</i>	boolean	May the property be modified while ^{RT} Maps is in Run mode?

MAPS_PROPERTY_ENUM

Must be placed in a component properties definition section. Defines an *enumeration* property for this component.

Name	Type	Arguments
		Description
<i>Property name</i>	string	Symbolic name of the property. This is the name that can be used in the execution script.
<i>Default possible values</i>	string	Different possible strings accepted for this property, separated with the ‘ ’ character. Ex: ‘‘value1 value2 value3’’.
<i>Selected string index</i>	int	Zero-based index defining which is the default selected value among the possible ones.
<i>needs2Be-Initialized</i>	boolean	Must the property be declared before ^{RT} Maps switches to Running state?
<i>canBe-Changed-AfterInstantiation</i>	boolean	May the property be modified while ^{RT} Maps is in Run mode?

MAPS_PROPERTY_READ_ONLY

Must be placed in a component properties definition section. Defines a *read-only* property for this component.

Name	Type	Arguments
		Description
<i>Property name</i>	string	Symbolic name of the property.
<i>Default value</i>	string, int, float, or boolean	The property default value (and by the same way defines the property type).

MAPS_END_PROPERTIES_DEFINITION

Ends the properties definition section.

3.1.6 Actions

Refer to source code samples:

- `maps_ComponentWithActions.h`
- `maps_ComponentWithActions.cpp`

Actions are used to alter immediately the behavior of a module. They may be called through the ^{RT}Maps control interface (GUI, scripts or COM/DCOM interface), unlike C++ methods which may be called only in the C++ code.

Actions definition macros

MAPS_BEGIN_ACTIONS_DEFINITION

Opens the actions definition section for the component specified as argument. Must be placed in the `.cpp` file, before the module definition.

Name	Type	Arguments
		Description
<i>Class name</i>	string	Name of the component C++ class.

MAPS_ACTION

Must be placed in a component actions definition section. Defines an action for this component. Using this macro, the action can only be executed while ^{RT}Maps is in the Running state. Otherwise, use the `MAPS_ACTION2` macro.

Name	Type	Arguments
		Description
<i>Action name</i>	string	Symbolic name of the action. This is the name that can be used in the execution script.
<i>Method name</i>	function pointer	C++ name of the module C++ method that must be called by the action.

MAPS_ACTION2

Must be placed in a component actions definition section. Defines an action for this component. Using this macro, the action may be executed while ^{RT}Maps is in the Standby state.

Name	Type	Arguments
		Description
<i>Action name</i>	string	Symbolic name of the action. This is the name that can be used in the execution script.
<i>Method name</i>	function pointer	C++ name of the module C++ method that must be called by the action.
<i>allow-when-Dead</i>	boolean	If true , the action can be executed while ^{RT} Maps is in the Standby state. Otherwise, it is equivalent to an action declared with the MAPS_ACTION macro.

MAPS_END_ACTIONS_DEFINITION

Ends the actions definition section.

3.1.7 The component itself

Now the component inputs, outputs, properties and actions have been declared, it is time to describe the component itself.

Component definition macro

MAPS_COMPONENT_DEFINITION

Defines the component itself. Must be placed in the `.cpp` file, after the inputs, outputs, properties and actions definitions, and before any method implementation.

Arguments

Name	Type	Description
<i>Class name</i>	string	Name of the component C++ class.
<i>Component name</i>	string	Symbolic name of the component. This is the name that will appear in the Component list window in the ^{RT} Maps Studio.
<i>Version</i>	string	The version of the component or any simple comment describing the component.
<i>Default priority</i>	integer	Priority of the thread to create if the component is threaded (range: 0 - 255).
<i>Accepted threading behavior</i>	'MAPS::Threaded' or 'MAPS::Sequential' or 'MAPS::Threaded MAPS::Sequential'	Determines if the component can be respectively threaded, sequential, or can accept both modes. See 3.1.9 for details.
<i>Default threading behavior</i>	MAPS::Threaded or MAPS::Sequential	Default starting mode of the component. Must match the possibilities defined in the preceding argument.
<i>Nb inputs</i>	integer	Number of inputs to create.
<i>Nb outputs</i>	integer	Number of outputs to create.
<i>Nb properties</i>	integer	Number of properties to create.
<i>Nb actions</i>	integer	Number of actions to create.

3.1.8 Dynamically creating inputs/outputs/properties/actions

Refer to source code samples:

- `maps_DynamicInputs.h`
- `maps_DynamicInputs.cpp`

Until now, we have seen how to create components with a constant number of inputs, outputs, properties and actions. However, as you may have noticed, some components can create them dynamically (have a look at the `VectorViewer` component and its `nbofInputs` property for example). This can be done within the `MAPSComponent::Dynamic()` function which has to be overloaded in your component.

Try and create inputs dynamically (knowing that the procedure for outputs, properties and actions is similar).

- First of all, you will have to determine what are the different types of inputs that will have to be created dynamically. Then, define each one of them in the inputs definition section. These inputs will not be created automatically when instantiating your component IF the number of inputs declared in the `MAPS_COMPONENT_DEFINITION` macro is lower than the inputs effectively defined. Here is an example:

```
MAPS_BEGIN_INPUTS_DEFINITION(MAPSMYComponent)
    MAPS_INPUT("staticInput", MAPS::FilterIplImage,
              MAPS::FifoReader)
    MAPS_INPUT("dynamicInput", MAPS::FilterInteger,
              MAPS::SamplingReader)
MAPS_END_INPUTS_DEFINITION
...
MAPS_COMPONENT_DEFINITION(MAPSMYComponent, "MyComponent",
                          "Version 1", 128,
                          MAPS::Threaded, MAPS::Threaded,
                          1, 2, 2, 0)
```

In this example, two inputs are defined, but only the first one will be created by default, since the component definition only specifies

1 input. The second definition (for the ‘‘dynamicInput’’ input will be used to create dynamically additional inputs that will be based on this model.

- Now, overload the `Dynamic` virtual method in your component code and use the `NewInput` method in order to create your additional inputs. Example:

```
void MAPSMyComponent::Dynamic()
{
    int nbExtraInputs =
    (int)GetIntegerProperty("nbXtraInputs");
    for (int i=0; i<nbExtraInputs; i++)
    {
        MAPSStreamedString sx;
        sx << "dynInput" << i;
        NewInput(1, sx); // "1" is the zero-based index
                        // of the input definition to use.
    }
}
```

This `Dynamic` function will be called by the ^{RT}Maps engine whenever the component is created, one of its property values is changed AND ^{RT}Maps is in the Standby state, or just before ^{RT}Maps switches to Running state. Thus, we cannot create additional inputs in Run mode.

As you may have noticed, there is no need to care about destroying the inputs. They are automatically destroyed by the ^{RT}Maps engine after `Dynamic` if ever they have not been ‘rebuilt’ within `Dynamic`.

WARNING: DO NOT EVER CALL `NewInput`, `NewOutput`, `NewProperty` OR `NewAction` OUTSIDE OF THE `Dynamic` METHOD.

3.1.9 The components threading model

Threads can be considered sub-processes. Though they share the same memory, they execute independently one from another. Using threads is a powerful way to execute several parcels of code in parallel in the same

application. Using threads is compulsory when dealing with multiple asynchronous data sources. That is why in ^{RT}Maps, most of the time, each component runs in its own *thread*. This means, it performs its job independently from the other components on the diagram, waiting for data on its input(s) from the upstream component(s), and writing data on its output(s), thus releasing the downstream component(s).

With ^{RT}Maps, developers do not have to deal with usual multi-threaded programming problems (such as concurrent data access or mutual exclusion zones) when they develop components: the ^{RT}Maps engine does it for you. However you must understand how the different components behave and synchronize each other.

When a component is *threaded*, it also has a *priority*. Component priorities determines which component will be able to interrupt the job of another one if they need to work at the same time. And synchronization of the processing is assured by the exchange of data most of the time³.

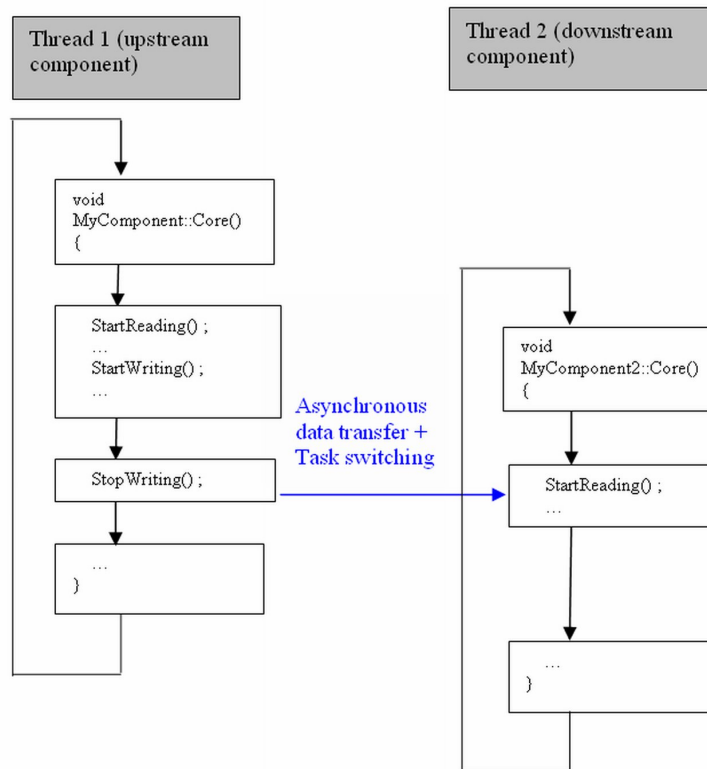
However, some components also support the **Sequential** mode (setting their **threaded** property to **false**). This means that the component **Core()** function will be performed in the same thread as its upstream component when the latter outputs a sample by calling **StopWriting**.

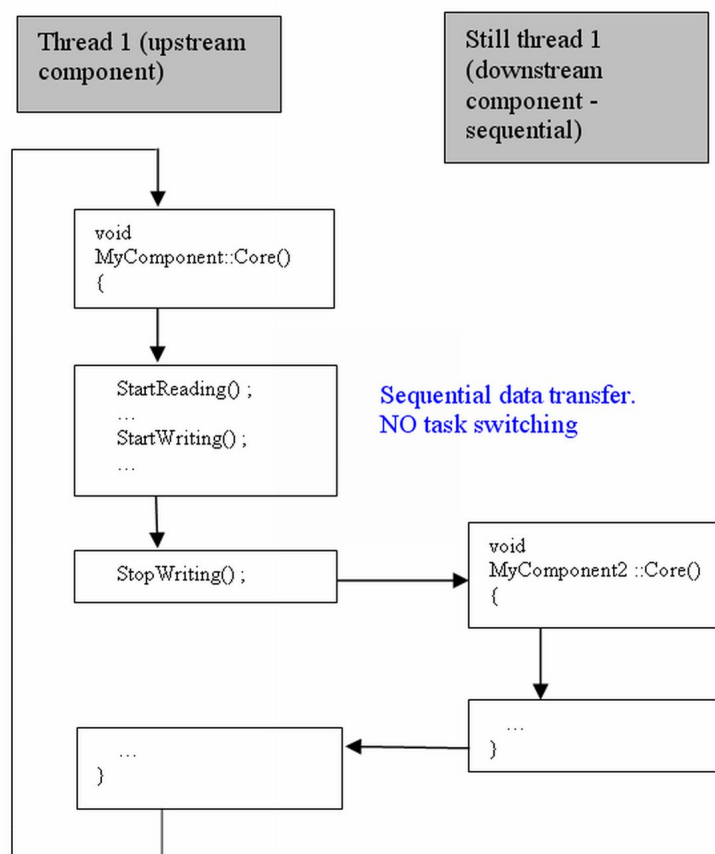
Warning: In **sequential** mode, a component adds some CPU load to its upstream component. Therefore, the use of sequential component is reserved to components that perform basic and very fast calculations or pre-processing, thus avoiding an additional thread within the ^{RT}Maps application and the associated thread switches.

Figures 3.3 and 3.4 show the difference between threaded and sequential execution for a component.

On the other hand, you may need to run 2 different threads within a single component. This is possible through the **CreateThread** function. Provide it with a function pointer and this function will be called within a new thread. In order to loop in that function until shutdown, you may use a **while(!IsDying()) ...; loop**.

³For every type of reader except **SamplingReader3.1.4**

Figure 3.3: *Threaded* execution.

Figure 3.4: *Sequential* execution.

You can use the `MAPSEvent` and `MAPSMutex` classes to synchronize your threads. Refer to the online reference documentation for a detailed description of these classes and their respective methods.

Refer to source code samples:

- `maps_MultiThread.h`
- `maps_MultiThread.cpp`
- `maps_MultiThreadAndSynchronisation.h`
- `maps_MultiThreadAndSynchronisation.cpp`

WARNING: USE ADDITIONAL THREADS WITHIN COMPONENTS WITH GREAT CARE. Before starting programming a multi-threaded component, check if no other solution is acceptable (such as the asynchronous `StartReading` function on multiple inputs).

3.1.10 Constructor/destructor of a component

Until now, we did not have to write some code to be run when the component was constructed (i.e. created in a diagram) or destroyed (i.e. removed from a diagram). However this can be done:

- In the header file of your component, replace the `MAPS_COMPONENT_STANDARD_HEADER_CODE` macro with the `MAPS_COMPONENT_HEADER_CODE_WITHOUT_CONSTRUCTOR`.
- Now declare your constructor (we are still in the `.h` file). Its arguments cannot be changed:

```
MAPSMYComponent(const char* componentName,
                 MAPSComponentDefinition& cd);
```

- Implement the constructor/destructor in the component `.cpp` file:

```
MAPSMYComponent::MAPSMYComponent(const char*
                                   componentName, MAPSComponentDefinition& cd) :
MAPSComponent(componentName, cd)
{ ... }
```

3.2 ^{RT}Maps record/replay methods in detail

Each *RRM* (Record/Replay Method) is implemented by a C++ class. This class derives from the `MAPSRecordReplayMethod` class. To help the user declare a new RRM, macros were designed. We will first browse those macros, and then discuss in detail the seven methods the component class must overload: the `BirthRecord`, `BirthReplay`, `DeathRecord`, `DeathReplay`, `Store`, `Hint` and `Replay` methods. A full description of the other methods of the class may be found in the ^{RT}Maps SDK online reference.

3.2.1 RRM definition macros

Class macro

The first thing is to declare a class which derives from `MAPSRecordReplayMethod` in the `‘.h’` file. The header of this class definition is made thanks to a macro. Component-specific attributes and methods declaration should be written thereafter.

MAPS_RECORD_REPLAY_METHOD_STANDARD_HEADER_CODE

Must be placed in the .h file, at the very beginning of the class definition.

Arguments		
Name	Type	Description
<i>Class name</i>	string	Name of the RRM C++ class.

Properties macros

The same macros as those used for the components may be used to define the properties of a RRM at the beginning of the ‘.cpp’ file. See [3.1.5](#).

The RRM definition macro

Once every property has been defined, the RRM itself should be declared.

MAPS_RECORD_REPLAY_METHOD_DEFINITION

Defines the RRM itself. Must be placed in the .cpp file, after the properties definitions, and before any method implementation.

Name	Type	Arguments
		Description
<i>Class name</i>	string	Name of the RRM C++ class.
<i>RRM name</i>	string	Symbolic name of the RRM. This is the name that can be used in the execution script.
<i>Version</i>	string	The version of the RRM or any simple comment describing it.
<i>Filter</i>	<code>MAPS::Filter*</code>	^{RT} Maps filter describing the different data types the RRM may store. See the SDK online reference for a detailed description of the data types and filters available.
<i>Record threaded</i>	boolean	Indicates whether the RRM has its own thread in record mode. Should be true unless the storing method is very small.
<i>Replay threaded</i>	boolean	Indicates whether the RRM has its own thread in replay mode. Should be true unless the replay method is very small.
<i>Nb record properties</i>	int	Number of properties described above that will be used in recording mode.
<i>Nb replay properties</i>	int	Number of properties described above that will be used in replay mode. Starts after the last property definition used for record.

3.2.2 The RRM methods

Now, what remains to be done is to implement the virtual methods that have to be overloaded.

BirthRecord, DeathRecord

BirthRecord and **DeathRecord** methods are executed respectively at the very beginning of the diagram execution (at run) and at the very end of it (at shutdown) when the RRM is used in record mode. They must be overloaded. The main tasks one might expect from these methods are to open and to close files. Remember to use **MAPSFileIO**⁴ methods to do this.

Store

The **Store** method receives a **MAPSIOElt** object and should write its content into a file. Remember to use **MAPSFileIO** methods to do this. Once it has been written, do not forget to call the **unlock** method to release your grip on the **MAPSIOElt** as soon as you do not need to access it anymore.

Hint

The **Hint** method receives the **MAPSIOElt** and should return a string. **MA**ke sure you return a string that is not local to the function. This string will be recorded in the **.rec** file. In the easiest cases, it is possible to store the whole data in the **.rec** file using only the **Hint** method. However, in more complex cases, **Hint** will only store an address in the **.rec** file. The purpose of this address is to help the replay method to find the whole data in a specific file.

BirthReplay, DeathReplay

BirthReplay and **DeathReplay** are executed respectively at the very beginning and at the very end of the execution of a diagram when the RRM is used in replay mode. They must be overloaded. The main tasks one might expect from these methods are to open and to close files. Remember to use **MAPSFileIO** methods to do this.

⁴See the ^{RT}Maps SDK online reference for more details about this class.

Replay

The `Replay` method must construct a `MAPSIOElt`. It receives most of the information available in the `.rec` file as parameters: the hint that was recorded, the record number, and information on the awaited data type.

3.3 Time handling functions

While writing your ^{RT}Maps component, you may need to access some specific functions to deal with *time* such as retrieving the current time in order to set a timestamp or pausing your component execution for some time.

First of all, remember that all timestamps, duration, timeouts are expressed in *microseconds*. Use the `MAPSTimestamp` and `MAPSDelay` structures to represent them. These structure are actually 64-bit integers.

Here are some methods that can be used to handle time:

MAPS::CurrentTime: Retrieves the current ^{RT}Maps time (i.e. this is the time that is displayed in the VCR).

MAPSComponent::Rest(MAPSDelay d): Pauses the current component thread for the duration 'd'. This duration is based on the ^{RT}Maps time clock. This means that if time speed is set to 50%, a `Rest(1000000)`; will actually pause the execution during 2 seconds. *Pay attention to the difference between this function and the `MAPS::Sleep` function which is based on the absolute time.*

MAPS::Sleep(MAPSDelay d): Pauses the current thread for the duration 'd'. This duration is based on the absolute time and not on the ^{RT}Maps clock. This means that the ^{RT}Maps time speed will have no effect on the real pause duration. *Pay attention to the difference with the `MAPSComponent::Rest` function.*

When calling this function within your code, make sure you explicitly call the method of the MAPS class (writing `MAPS::Sleep(...)` instead of `Sleep(...)`), since the `Sleep` function may also be available in the standard namespace of your environment (the `Sleep` function on Windows takes an argument expressed in milliseconds).

`MAPSComponent::Wait(MAPSTimestamp t)`: This function pauses the execution of the code until the ^{RT}Maps time reaches the specified timestamp (`t`).

3.4 Controlling the ^{RT}Maps clock

The ^{RT}Maps standard clock is based on one of the internal host machine clocks: the **Performance Counter** under Windows, and the **RTC** – Real-Time Clock – (or **System clock** in case you do not have the administrator rights) under Linux. These clocks, like any other clock, are not absolute: they *shift*. This shift depends on the quality of the oscillators, their temperature, and so on. The shift is not an issue for most applications, but in some cases, you may need to apply corrections periodically, or even implement your own clock (based on a special clock provided by a hardware board or a sound card for example). You might also want to implement for some reason a clock whose time speed varies with external parameters.

For all this, there are two different ways to influence the behavior of the ^{RT}Maps clock from within a home-made component: you can either *synchronize* the main clock (i.e. periodically apply time corrections to the ^{RT}Maps main clock), or implement your own custom clock. In this case, your own clock will be queried each time someone in the ^{RT}Maps application asks for the current time (the VCR for example, or the ^{RT}Maps components when they need to timestamp data).

3.4.1 Applying corrections to the standard clock

Refer to source code samples:

- `maps_synchronizer.h`
- `maps_synchronizer.cpp`

Imagine you want to synchronize periodically the ^{RT}Maps standard clock to avoid time shifting. You can do this using the **MAPSSynchronizer** object: remember that there is one and only one instance of type **MAPSSynchronizer** living in the ^{RT}Maps Engine. The aim is to retrieve

a pointer to it (through `MAPS::GetSynchronizer`), use it (through periodical calls to `MAPSSynchronizer::SignalSynchronizationCommand`), and release it (through `MAPS::ReleaseSynchronizer`) when you do not need it anymore, so that someone else can use it.

Note that as soon as a `Recorder` component is instantiated on your diagram, the ^{RT}Maps time cannot jump backward. If you call the `SignalSynchronizationCommand` with a timestamp argument that would make a jump backwards in time, the ^{RT}Maps clock just pauses for the corresponding delay until it catches up.

3.4.2 Implementing your own custom clock

Refer to source code samples:

- `maps_customclock.h`
- `maps_customclock.cpp`

When discrete clock corrections to the standard clock does not fit your need in terms of “*clock customization*”, you can implement your own custom clock in a specific ^{RT}Maps component. In this case, once this component is instantiated in the current diagram, the ^{RT}Maps Engine will be able to query *your clock* instead of the standard clock when anyone needs the current timestamp (through `MAPS::CurrentTime` for example).

For this, make your component inherit from the `MAPSBaseClock` class in addition to `MAPSComponent`. Then, you will need to implement some virtual functions from `MAPSBaseClock` such as `InitClock`, `RunClock`, `ShutdownClock` and last but not least: `CurrentTime`. You will also need to specify (even approximately) the timespeed of your clock through the `MAPSBaseClock::SetAbsoluteTimeSpeed` method.

InitClock is called when the component is created. Can be empty.

RunClock is called when ^{RT}Maps enters the Running state. From this time, your clock time has to flow.

CurrentTime will be called each time someone (the ^{RT}Maps Engine, the **Recorder** component, or any other component in the diagram) asks for the current time. Therefore, this function implementation must be simple *and fast!!!*

ShutdownClock is called when the diagram has finished its execution. The ^{RT}Maps time can stop flowing.

Additionally to these implementation, you will have to call the **SetAbsoluteTimeSpeed** method to tell the ^{RT}Maps Engine what is the “*real*” time speed of your clock (since you can implement virtual clocks that run slower or faster than real-time, the Engine has to know it). This can be an approximation, but always specify an *upper bound* of the time speed you are creating!

Example: if your clock runs 50% slower than real-time, you can specify a timespeed of 550, but not 450 (1000 corresponds to real-time). Specifying a lower value than the real one would alter the accuracy of pausing functions such as **Wait** and **Rest** (the more you underestimate, the more inaccurate it will be). Specifying higher values implies a little more CPU calculations for these same methods, but do not alter their accuracy.

If the effective time speed of your custom clock varies with time, you can call **SetAbsoluteTimeSpeed** as many times as you wish in order to update the maximum real time speed (but make sure your clock remains at lower time speeds that the last one specified through this function).

3.5 Debugging an ^{RT}Maps module

Once the code has been written in your ^{RT}Maps component or RRM and this code compiles correctly, you will often need to debug the code. It is of course possible to perform step by step debugging of an ^{RT}Maps component that has been compiled in debug mode. Here is the way of doing it.

3.5.1 With Microsoft Visual Studio

- First of all, compile the project containing your ^{RT}Maps component in *Debug* mode. The .pck file will be generated in the .../packages/debug folder of your ^{RT}Maps SDK directory.

- Make sure that this project is the active one (it appears in bold characters in the list of projects of the solution). Otherwise, right click on the project in the **Projects explorer** window, then choose ‘‘Set as active project’’.
- Add breakpoints within your code where you want the debugger to catch the execution.
- Start the debug session (F5). If you are asked to choose an executable, browse for **rtmaps.exe** in the **bin** folder of your ^{RT}Maps installation directory. (Do not pay attention if you are warned that ^{RT}Maps does not contain any debug information).
- Register the previously compiled package file. (Make sure you register the Debug version and not the Release one).
- Then run the diagram. When the execution reaches the first breakpoint, the debugger pauses the execution and you can start step by step execution, but also watch at the variables, the call- stack, the memory state...

3.5.2 Under Linux, using ddd

- Compile your package in debug mode.
- Launch ddd.
- In ddd, choose menu **File | Open program...**, then browse for the package.
- In ddd, choose menu **File | Open source...**, then open your component source file and add breakpoints wherever you want to inside it.
- Launch ^{RT}Maps *independently* from ddd.
- Register your package in ^{RT}Maps (Make sure you register the Debug version and not the Release one).
- In ddd, choose menu **File | Attach to process...**, then select **RTMaps** among the available processes.
- Run the diagram. ddd should pause the execution when reaching one of your breakpoints.

