JOANNEUM RESEARCH
Forschungsgesellschaft mbH

Institut für Informationssysteme &
Informationsmanagement

Leiter
Dipl.-Ing. W. Haas

Steyrergasse 17
A-8010 Graz, Austria
Tel.: +43 316 876 - 1119
Fax: +43 316 876 - 1191

Module Developer's Guide

An introduction to module development

Status: 19th March 2007

Abstract: This document consists of overviews of the various paradigms, which are used by the analysis framework architecture. For specific information please refer to the class references, which are generated by usage of doxygen out of the framework's source code.

Keywords: creation generation ModuleWizard inputstreams outputstreams timeslots timeline input output streams `Module` `ModuleResult` `ModuleState` `Transformer` `Factory` DLL stub code threads execution configuration files graph framework

DOCUMENT HISTORY

Version	*Date*	*Reason of change*
1	1st March 2007	Document created
1.1	19th March 2007	Document updated

Table of Contents

1 Graph

The basic paradigm used here is based on a unidirectional graph. A graph consists of nodes and unidirectional interconnections between them (they are one-way communication channels). A graph is a distributed architecture by nature. The various nodes do not have any information regarding each other and there is also no central control centre. A graph allows a high degree of flexibility and modularity.

Information output by a node flows through a unidirectional communication channel (called 'connection' or 'interconnection' furtheron) according to the connection's direction to the next node and enters there a new node. Please take a look below at the example. In this context each node is called "module" and the whole graph itself is called "module graph".

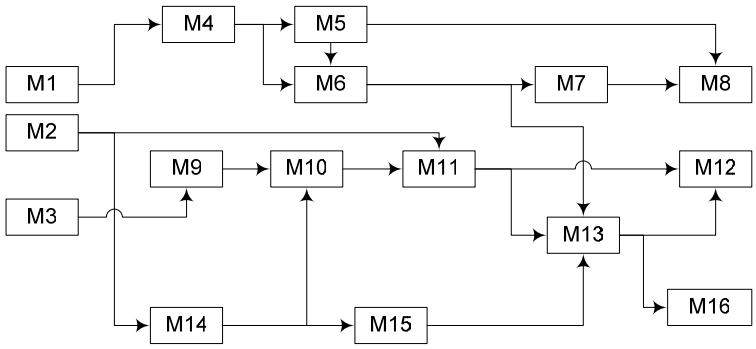


Figure 1: General module graph

As shown in the figure above, the data flows through an interconnection according to the arrow's direction. An interconnection between modules may be considered as a data buffer in this context, because it buffers data output by a module for input purposes of the next module. Each module is responsible to read data from those buffered interconnections and to remove those data, which has been already read. There is no external check which would avoid a run out of memory. As shown in the previous figure each module is associated with a unique name within a graph. Although each module must have a unique name, it is still possible that multiple instances of the same module are used within the same graph.

2 Module concept

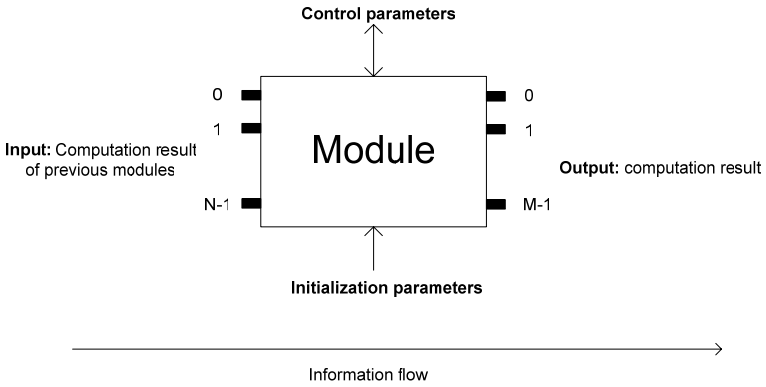


Figure 2: `Module` model

A module may be considered of being something similar to an electronic chip. It consists of some (optional) input pins on the left side, of a "black box" in the middle and some (optional) output pins on the right side. Data enters the module on the left side via the input pins, is modified within the "black box" and leaves the module on the right side via the output pins. In other words: The "black box" reads input from the input pins, performs any arbitrary computation and writes the output to an output pin. A module may have any arbitrary number of input pins and any arbitrary number of output pins, e.g. a module without any input pins and without any output pins or with only one input pin is still a valid module. Any combination of input pins and output pins is possible. There is only one restriction: The number of input and output pins is considered to be a constant module's property. **During the module's lifetime the number of input-/output pins can not be changed**.

For initialization purposes so called "initialization" parameters are sent to the module, which contain initialization values required by a module to enter a first valid state. E.g. a module which reads data from an mpeg stream out of a file, needs the path and filename of the mpeg-file. The path and filename are sent to the module as initialization parameters.

During computation it may be desirable to control the module, e.g. to change threshold values required for generating black & white images out of a colour image. Those "control parameters" are sent to the module during computation and are passed to the module's computation method (will be explained in more detail below).

Note: The module was designed to be merely a computation module without any graphical user interface. Whole communication to the "external" world is done via "control parameters". The module does not need to take care about any graphical user interface issues. The one and only task of a module is only a computation task, according to the paradigm: "input, process, output", which means read data (from the input pins), compute something and output the result (to the output pins). That's it.

3 Graphs & Links

The graph shown previously is a very general graph sketched in a simplified way. The connections between modules may be much more complex. The allowed links may be derived from one general link rule and one implicit assumption:

1. **"A connection starts at an output pin and ends at an input pin."**
2. **"An input pin is only allowed to be connected with one output pin."**

These rules allow that a module's input-/output pin is not connected at all and that an output pin may be connected to several different input pins. Regardless of the number of input and output pins, every module is treated in the same way. E.g. a module without any input pins is treated exactly in the same way as a module with three input pins and one output pin.

Forbidden links:

- An input pin must not be connected with an input pin (breaks Rule 1).
- An output pin must not be connected with an output pin (breaks Rule 1).
- An input pin must not be connected with multiple output pins (breaks Rule 2).

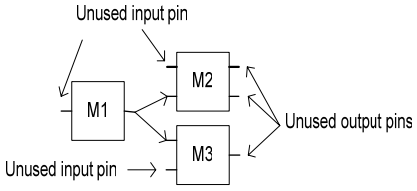


Figure 3: Correct Graph

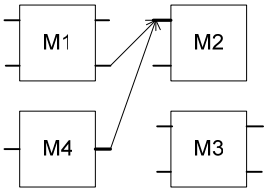


Figure 4: Incorrect Graph, two output pins are connected to the same input pin

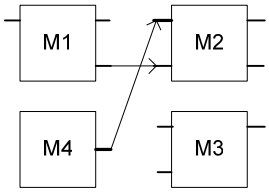


Figure 5: Correct Graph, although multiple input and output pins are unconnected, the module graph is still valid. Note that even one module ("M3") has no input connections at all, this graph is valid!

Note: A graph may also contain cycles. Although this feature is not completely implemented it will be supported by the next framework's version.

4 Graph example

To demonstrate the explained concepts more clearly a concrete example of a graph is presented below. This graph comes from practice and is a typical application example.

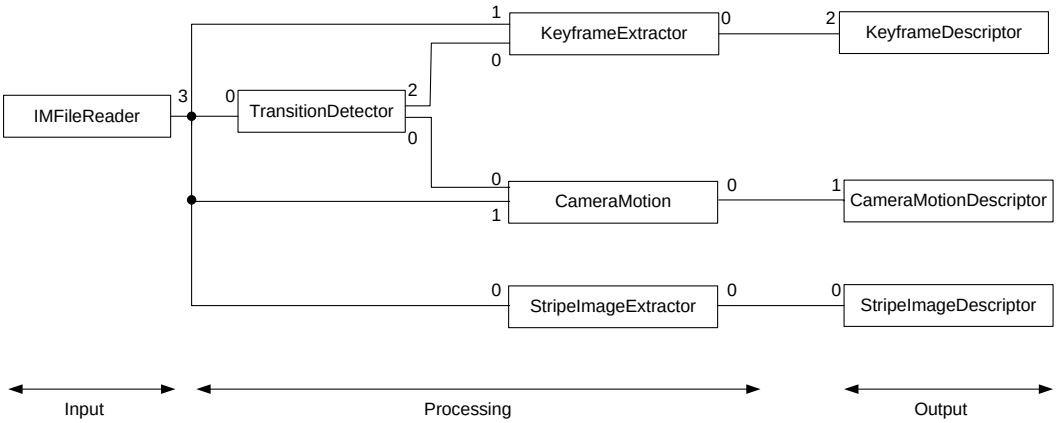


Figure 6: Concrete graph example from practice

In a graph typically three types of modules are used:

- Modules, which generate data, so called "input modules", e.g. a module that decodes a video (Mpeg-2 file) and forwards the single video frames and/or audio data to subsequent modules. In the example above the "IMFileReader" modules serves for this purpose. An informal naming convention specifies that "IM" stands for "Input Module".
- Modules, which process data, so called „processing modules". In the example above there are several modules of this type: TransitionDetector, KeyframeExtractor, CameraMotion, StripeImageExtractor. E.g. the transition detection module ("TransitionDetector" above) gets the single frames of the video and computes so called shot borders (cuts between scenes/different cameras). More generally these processing modules extract some kind of metadata out of the video frames. This metadata is then written in subsequent "descriptor" modules in an Mpeg7-document (XML file). An informal naming convention specifies that "PM" stands for "Processing Module".
- Modules, which output data, so called "output modules". In the example above there are three output modules: KeyframeDescriptor, CameraMotionDescriptor, StripeImageDescriptor. These three modules "output" the incoming data to an Mpeg7-document (XML-file). In other terms they "describe" the content, therefore they got the name "descriptor" modules. An informal naming convention specifies that "OM" stands for "Output Module".

These three types of modules have the same programming interface are treated by the framework in the absolutely same way. It is only the functionality in the modules that differs!

The digits on a module's side denote the pin number. If the digit is on the module's box left side, the input pin number is denoted. If the digit is on the module's box right side, the output pin number is denoted. E.g. a connection starts from IMFileReader's output pin 3 and ends at TransitionDetector's input pin 0. A module's output pin may be multiply connected to subsequent modules input pins (e.g. IMFileReader's output pin 3 is connected to input pins of modules KeyframeExtractor, TransitionDetector, CameraMotion and StripeImageExtractor).

5 Module Execution

The modules are managed by a framework in the background, which is responsible for all the management tasks, like module instantiation, module execution, data buffering between the modules, establishing of communication links, synchronisation…

A module provides a few call-back methods known by the framework. The most important one is the module's `Process()` method. Computation is done in general step by step by calling the module's `Process()` method in and 'endless' loop. The framework calls the `Process()` method, if at least one of the following conditions is true:

1. New data from a previous module becomes available,
2. The module has output a result in a previous Process() step,
3. Cyclic call period elapsed (optional)
4. It is the first time (break through cyclic dependencies) to call Process()
5. A connection has been closed by a previous module

The **first criterion** is true, when a previous module has output something to an output pin, which is connected to this module's input pin.

Code example: `_execInfo->NewInput()` returns true if new data is available on any input pin, `_execInfo->NewInput( "input pin" )` returns true if new data is available on the specified input pin

Second criterion: A very common example for this criterion is a module, which reads data from an mpeg file and outputs the single images. This module does not need to react on any input, the module's purpose is only to "generate" data (the images) out of a data source (the mpeg file). Modules of this type typically have only output pins and no input pins, and the first criterion would never become true. This criterion is also of importance, when a module wants to flush its internal cache.

In general: Whenever a module has output something to its output pin, it is called immediately again. This is especially of interest for data source modules, which generate data ("active" modules). Those modules typically do not have any input pins, they generate data out of a source and output those created data to one of the output pins.

Code example: `_execInfo->OwnOutput()` returns true

Third criterion: A module may want to indicate the framework that it wants to be called periodically, with a period time specified by the module during initialization. Whenever this specified period of time has elapsed, the framework will call the module's `Process()` method. The sketch below shows a time line based on assumption that a module wants to be called periodically with time period t. The first sketch below shows the theoretical points in time, when the module would be called by the framework.

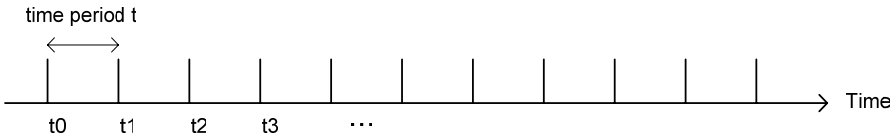


Figure 7: Theoretical points in time to call the module periodically

Figure 8: „Real" points in time when the module is called by the framework

When a module's computation time is longer than time period t (this happens in the figure above for time point t1) the framework immediately calls the module again (at "shifted" time point t2). The framework adds time period t to "shifted" t2 to compute the next point in time ("shifted" t3) when the module has to be called again. In other words: Time period t starts to count from the beginning of a module call and the theoretical points in time to call the module may be shifted backwards when the module takes longer for a computation cycle than expected.

Code example: `_execInfo->CycleTimeout()` returns true

Fourth criterion: If a graph contained a cycle, each module would wait for another one's first activity (deadlock). To break through this cyclic dependency, each module is called the first time without regard to the other three criterions.

Code example: `_execInfo->FirstCall()` returns true

Fifth criterion: If a module has finished its computation (due to whatever reason) all output streams are closed by the framework. "Close" means that no module results will be routed through this connection. A subsequent module, which is connected to such a "closed" output stream, will be called in order to signal the "close" operation on the output stream (=input stream of the subsequent module).

Code example: `_execInfo->InputStreamClosed()` returns true

Module execution termination

The module execution is terminated by the framework, when the module

1. Indicates an error (by a return code) or
2. Causes an exception or
3. Is unable to compute anything futhermore.

   This requirement is true, when a module

   - Is not connected to any other module or when all input streams are already closed.

     This means that no new input will be available in the future for processing (first criterion of the call reasons will be always false).
   - The module has not output anything in the last call to its output streams.
   - There is no cyclic timer, which calls the module.

   If all of these requirements are true, the framework stops execution of the module, even if the module has not indicated its own computation end.

6 Input and Output

6.1 Computation result

Till yet in this description only the very general term "data" has been used. What is "data" in this context? The module and framework design does not make any assumptions about the "data" which flows from one module to another, no analysis or anything else is done by the framework on this "data". Because the design is completely object-oriented, "data" is also encapsulated in a specific subclass, which has to be derived from an abstract class called "`ModuleResult`". When data flows from one module to another one, the framework simply passes pointers to object instances of type "`ModuleResult`" (because it does not know the specific subclass). If multiple subsequent modules are connected to the same output pin of another module, the framework passes the same pointer value to all of the subsequent modules. All the subsequent modules share access to the same "`ModuleResult`" object instance originating from the previous module.

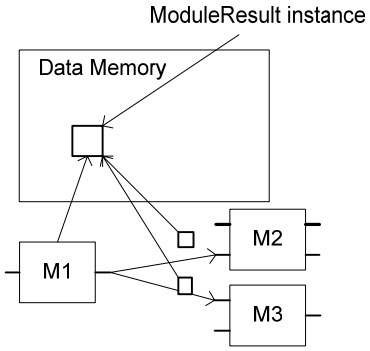


Figure 9: M2 and M3 share access to the same „`ModuleResult`" object instance originating from M1

The previous figure shows a memory area called "Data memory" where a "`ModuleResult`" object instance resides. This object instance has been generated by M1 within a process call to output computation results. When M1 outputs this object instance on an output pin, the framework takes ownership of this object instance and forwards the address of this object instance to M2 and M3. Both M2 and M3 share access to the same "`ModuleResult`" object.

A "`ModuleResult`" object instance is only allowed to be modified in the same `Process()` method call, in which this "`ModuleResult`" object is created. Once the "`ModuleResult`" has been output to an output pin and the module has left its `Process()` call, no module is allowed any more to alter this object instance's contents. This is especially true for M1. M1 is not allowed to store any references to the "`ModuleResult`" object instance in order to modify this object's content in a future `Process()` call ("fire and forget"). Only read-access is allowed furtheron.

As long as a module remains in its `Process()` method, the "`ModuleResult`" object instances created within this `Process()` call, are not visible by subsequent modules. When the module leaves its `Process()` method, the framework passes the pointers to the subsequent modules and makes the created instances visible. From this point in time only read operations are allowed on the "`ModuleResult`" instance furtheron. The framework knows by usage of a reference counter when a "`ModuleResult`" may be destroyed (or reused).

Because each module is executed in its own thread, access to the "`ModuleResult`" object instances must be thread-safe. Although only read-operations are allowed, the module developer must ensure that reading of computation results stored in a "`ModuleResult`" instance is thread-safe.

6.2 Output streams

The timeline which is considered in this context is divided into discrete time-slots.

When a module outputs a computed result, it must associate a time slot to the result. This association is visible to all subsequent modules via their input streams.

Each input (output) pin is associated with a so called input (output) stream. Each pin's stream is managed completely independent of each other. Both input and output streams are based on the discrete time slots paradigm. For reading and writing purposes a module has to specify the concrete time slot to address a discrete point in time, for which the operation is valid (mostly to establish an association between a time slot and a `ModuleResult` object instance for a write operation and to get the associated `ModuleResult` object instance at this particular time slot for a read operation).

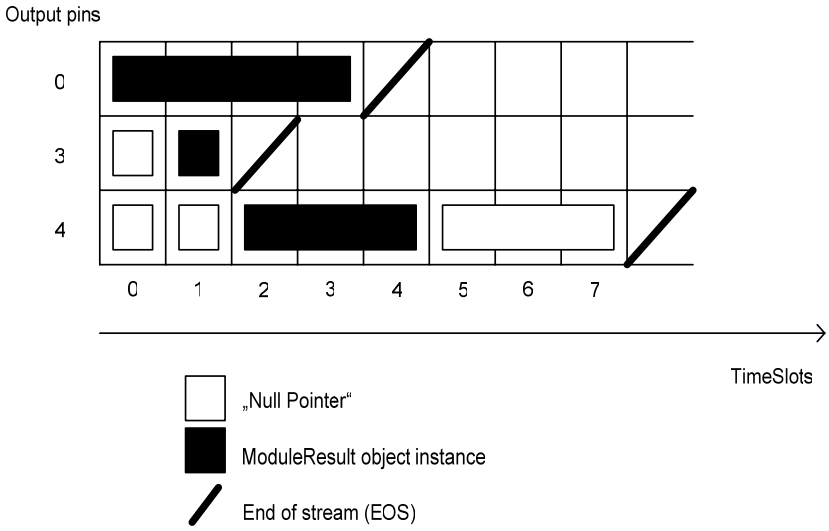


Figure 10: Structure of an output stream

When a module outputs a computation result, this computation result may be a "Null Pointer" or a real `ModuleResult` subclass object instance. The framework offers the possibility to output a "Null Pointer" in order to indicate that no "real" result is available for this specific time slot, e.g. when a module omits some input data due to any reason, it may output a Null Pointer for those omitted time slots. The framework offers a second possibility to save computation time: A `ModuleResult` (and Null Pointers too) may be associated with multiple contiguous time slots. Such a contiguous block of time slots associated to the same `ModuleResult` is called a 'segment'. (A time slot may be considered as a segment of length 1.) A module may address the input stream by segment indices or by (relative or absolute) time slot values. A module is only able to append a new `ModuleResult` object instance or a Null Pointer at the end of the stream (EOS).

Classes:

`OutputStreams`

Module member:

```
OutputStreams* _outputStreams;
```

6.3 Input streams

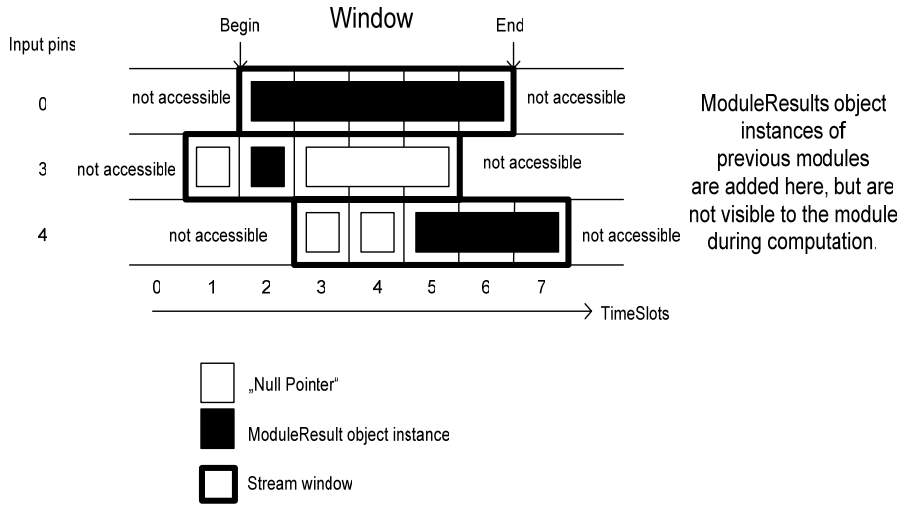


Figure 11: Structure of input streams

Each input pin's associated stream is managed completely independent of the other input pins. An input stream consists of accessible and not accessible areas. A window defines a contiguous block of time slots, which are accessible (and visible) by the module. A module is only able to shift the window's begin, not the end. When a module returns from its `Process()` call, the framework appends all `ModuleResult` object instances at the end of the input streams, which have been arrived in the meantime. After appending these object instances and adapting the end of the window appropriately, the module is called again (because new input data is available). Now the module is able to access the new `ModuleResult` object instances. `ModuleResult` object instances, which fall out of the window due to a shift of the window's begin, are removed from memory by the framework.

As explained above, each input pin is associated with an independent input stream and only those elements within the window are visible. As depicted in the figure below, multiple windows may overlap the same contiguous block of time slots. Such an overlapping region is called "common window". This common window exists only, when all input stream windows contain the same time slots.

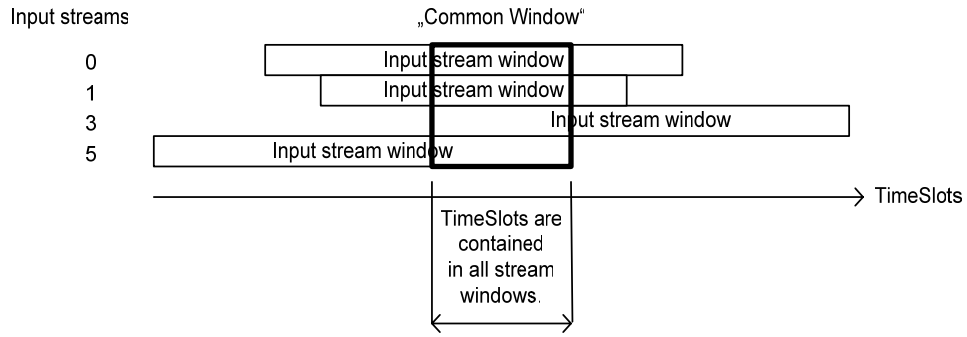


Figure 12: Illustration of a common window

Classes:

`InputStreams`

Module member:

```
InputStreams* _inputStreams;
```

7 Base classes

[This chapter is intended mainly for programmers, who want to get a deeper technical background.]

A module developer has to implement basically five subclasses derived from abstract interfaces:

- `Factory`
- `ModuleResult`
- `ModuleState`
- `Module`
- `Transformer`

Note: For a detailed class documentation, please refer to the documentation which is shipped with the module developer SDK. The SDK contains a detailed documentation on a per method and member variable basis and is generated by usage of doxygen directly out of the framework's source code.

7.1 Factory

The framework does not know how to create any subclass implemented by a module developer (there is no "a-priori" knowledge contained in the framework). Therefore it makes use of a so called "`Factory`". The `Factory` object has the knowledge **how** to create and dispose "`ModuleResult`", "`ModuleState`" and "`Module`" object instances required by the framework. (At first the framework needs a `Factory` object, this is done by the module DLL and will be explained in detail below.) The framework knows **when** these objects have to be created and disposed. The `Factory` does not need to keep track of the created objects for memory management purposes. The knowledge when an object becomes unused and when it has to be removed from memory is the framework's responsibility. This subclass' implementation can be created automatically by usage of the "`ModuleWizard`" (described later in more detail).

7.2 ModuleResult

`ModuleResult` encapsulates a module's computation result. For this purpose the module developer has to subclass the abstract "`ModuleResult`" interface to add the datastructures required to store a computation result. The design specifies that each output pin is associated to one `ModuleResult` subclass (please take a look at the figure below). Therefore the module developer has to implement up to as many `ModuleResult` subclasses as module's output pins. A common example is an mpeg-reader module, which outputs the images from the mpeg stream on one output pin and the audio data on another output pin. For this purpose the module developer has to implement two `ModuleResult` subclasses. One for containing the audio data and one for storing an image. Those object instances are then forwarded to subsequent modules, which have to "understand" those classes.

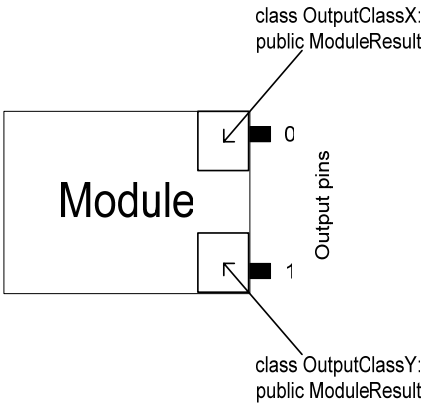


Figure 13: Association of a module's output pins to specific `ModuleResult` subclasses. In the module example depicted in the figure above, the module developer has to implement two subclasses of `ModuleResult`: OutputClassX (e.g. for audio data) and OutputClassY (e.g. for visual data, most probably an image).

When a subsequent module's input pin is connected to a previous module's output pin, the subsequent module has to be able to interpret the incoming data (more specifically: the `ModuleResult` subclass) on it's input pin. The design specifies that the module developer has to provide those `ModuleResult` subclasses, which are required by subsequent modules either as a compiled library plus header files or as complete source code. The module developer of a subsequent module links either the library or adds the source code to his own module's code. The subsequent module is now able to interpret those `ModuleResult` subclass object instances on it's input pins correctly.

Note: The framework has no a-priori knowledge regarding the various `ModuleResult` subclasses. The framework handles always only with pointers to `ModuleResult` object instances. Therefore all methods provided by the framework's classes only offer type "`ModuleResult`". The module developer itself must downcast to the desired `ModuleResult` subclass type when necessary (e.g. when reading from its input pin).

Note: Modules are executed in it's own threads and run completely asynchronously. This means that reading the datastructures of a module result object instance has to be thread-safe! Rule of thumb to provide thread-safe reading methods without locking mechanisms: Use only local variables for write purposes (e.g. altering the index in an array) and make your read-methods const (without help of mutable member variables)!

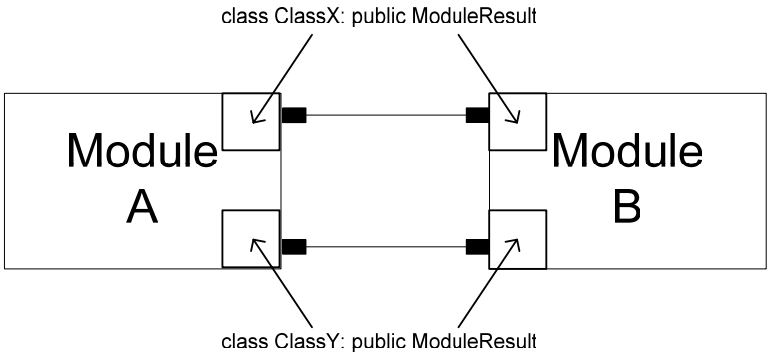


Figure 14: `Module` B is able to interpret module A's output data encapsulated in subclasses ClassX and ClassY. ClassX and ClassY have been provided by A's developer and integrated in `Module` B (either by linking ClassX and ClassY code into `Module` B or by compiling ClassX's and ClassY's source code into B.)

7.2.1 Link Verification

In order to avoid a link between one module's output pin and a subsequent module's input pin, where the subsequent module does not recognize the specific `ModuleResult` subclass, the design specifies a "link compatibility" check. When the framework tries to establish a connection between one module's output pin and another module's input pin, it asks the target module whether it is able to interpret this specific `ModuleResult` subclass. This check is performed by passing one sample `ModuleResult` object instance (originating from the source module) to the target module. The target module is responsible to determine the `ModuleResult's` subclass (C++ `dynamic_cast<x*>(z)`). If the target module fails, the framework refuses to establish this connection and does not execute this graph (seems to be an invalid module graph). Otherwise the link seems to be okay and the connection is established.

Note: If the ModuleWizard has been used to create a module skeleton, this "validation" code has been created automatically.

7.3 ModuleState

The basic paradigm used here is based on input, compute and output, where output is completely defined as a function of input and the module's inner state. A module's state may be considered as it's memory. A computation step may change a module's inner state due to any reason, e.g. a threshold value may have been exceeded which means that the module uses another algorithm in the next computation step. The design associates each computation step with a specific module state. The current `ModuleState` object instance is passed to the module as a parameter for computation. The framework keeps track of the various `ModuleState` object instances in the past and it's associations to specific computation steps. When the framework rewinds back a module, it just passes the old module state to the module and triggers the module's computation again.

In C++ terms all member variables, which influence a module's behaviour during it's life time, have to be encapsulated in this `ModuleState` class.

7.4 Module

`Module` is the main interface and specifies all required methods for a useful module. The "real" computation on input data is done by this class. Computation is performed step by step, which means that the module is able to compute only one result during one process call requested by the framework.

7.5 Transformer

The purpose of this class is explained in chapter "User interaction".

8 Memory management

When a module outputs a computation result encapsulated in a `ModuleResult` subclass object instance, the framework associates a **reference counter** with this instance and passes it to the subsequent modules. Each time a module shifts an input pin's stream window, the framework determines those object instances, which have been fallen out of the window. Located outside of the window means that this `ModuleResult` instance cannot be accessed any more and is invisible to the module. The framework decreases the associated reference counters of those 'invisible' `ModuleResult` instances. When the object instance has become invisible to all subsequent modules, the reference counter is zero and the `ModuleResult` instance is removed from memory by the framework.

If a module does not shift the windows of the input streams, no `ModuleResult` instances are removed from memory and memory will run out of due to new `ModuleResult` instances, which have been appended to the input streams. Therefore every module is responsible to regulate the used amount of memory by shifting the input stream windows properly.

Note: The framework depends on a correct window shifting implementation within the modules (sorry, this is a design error)! Although a module may implement the window shifting technique properly, there is still a possibility for a memory overflow. This memory overflow takes place, when a module takes too much time for computation within a single `Process()` call, because the module has no knowledge about any new module results in its input streams, which may be output in the mean time by a concurrently executed module:

> *Detailed explanation:* Before the framework calls a module's `Process()` method, the framework freezes the module's context and creates a "snapshot" of the current input/output stream and control parameter values. Within the Process() call the module is executed with the snap shot context. As a consequence the module is only able to read those `ModuleResult` instances in its `Process()` method afterwards (via class `InputStreams`), which are contained in this snapshot. While a module is computing something within `Process()`, other concurrently executed modules, which are connected to this module, may output new module results. But these new output module results are not visible to the module in the current `Process()` method (via class `InputStreams`), because they are not part of the snapshot. In order to overcome this challenge, class `InputStreams` provides a method to take a look behind the scenes to get knowledge about the number of hidden `ModuleResult` instances, which have been inserted in the meantime. If the returned value becomes larger and larger, the module should speed up to keep track with the computation speed of the other modules.

Summary:

In order to avoid a memory overflow a module developer has to keep two things in mind (both of them are only necessary, when any input pin is connected):

1. A module must shift the windows of the input streams accordingly.
2. A module should not take too much time for computation in the `Process()` method.

9 User interaction

9.1 Parameter objects

As described previously, a module does not to take care about user interface topics. For communication purposes with the external world (mainly user interface) the framework provides so called parameter objects. Parameter objects are objects, which encapsulate an atomic datastructure well known from informatics, e.g. lists, maps, integers, lines... Those parameter objects may be divided into several groups:

- basic types (integer, double, bool),
- structured types (map, list, set, selection list),
- graphical types (point, line, rectangle, polygon),
- image types (image),
- file system related types (file name, path).

These module parameter objects are used for three purposes:

1. As **output parameters**: This application is intended to give a (graphical) feedback to the user (e.g. rectangles, lines…) of the computation result. For this purpose a ModuleResult object instance is translated to a set of ModuleParameter objects which are known by the framework. From the semantic point of view, these parameters are only intended as output parameters, which can not be modified by the user.
2. As **control parameters** in order to influence a module's behaviour. These control parameters allow the user to influence a module's behaviour by modification of the control parameters' values. These control parameters are sent in both directions: from the module to the user and back from the user to the module. It is only the semantic which is changed in contrast to the output parameters application. Control parameters are sent in both directions and may be changed by the user, but there are still the same classes from the programmer's point of view.
3. As **initialization parameters** in order to set a module to a first valid state. Initialization parameters are only sent from the user to the module. They "flow" only in one direction, but may be also modified by the user in order to provide the module with meaningful initialization values.

Programmer's info: Parameter object class names have prefix "`ModuleParameter`" and are subclassed from the same parent called "`ModuleParameterValue`". Each parameter object has a unique class ID, which is publicly available. Parameter objects are created by usage of a so called `ModuleParameterFactory`. This `Factory` is passed the unique class ID of the desired parameter object class, and the appropriate object is returned (for more details please refer to the class reference generated by doxygen).

9.2 Output parameters

The design emphasizes on one hand simplicity of module creation and on the other hand module implementation. This means that the module developer should focus his attention on implementation of (most probably digital image processing) algorithms (on the "real" module's task) instead of having to deal with low-level management tasks. Therefore the design delivers the module developer completely from all graphical user interface tasks. Graphical user interface tasks are handled behind the scenes by the framework or by special applications. But the module developer does not need to read the latest user input from the console or somewhere else.

Nevertheless graphical output is necessary to give the user a feedback about what is going on and what has been computed by the module. The framework does not provide any means for graphical

user interface input & output. All graphical output is done via so called "parameter objects". Those parameter objects are understood by the framework and can be displayed by an arbitrary application. The missing link between the member variables stored within a `ModuleResult` object instance and the parameter objects is filled by a special class called `Transformer`. As the name implies the `Transformer` takes a module result object instance and creates various parameter objects.

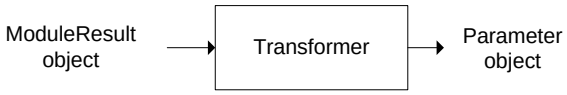


Figure 15: Main task of a `Transformer`, take a `ModuleResult` object instance and create parameter objects out of it

For future compatibility additional `Transformer` capabilities are foreseen:

- Transformation should also be possible with a `ModuleState` object instance (same semantic)
- Re-Transformation of parameter objects to a `ModuleResult` object instance
- Re-Transformation of parameter objects to a `ModuleState` object instance

The parameter objects generated out of a `ModuleResult` object instance are only intended for display to the user. The user is not able to modify any values of those parameter objects. Those parameter objects are "read-only" objects. This means that the user can not overwrite a module's computed result.

9.3 Control parameters

Next to graphical feedback to inform the user what has been computed by a module, it is necessary to give the user a possibility to control a module's behaviour (e.g. to change a threshold value) **during runtime**. To control a module, some kind of input from a user is required. The module developer does not need to take care about communication with the user, this is also done by the framework and is handled behind the scenes. Control of a module is achieved again by parameter objects. Control parameter values "flow" in both directions, from the user to the module and back. When the user modifies a control parameter value, e.g. to change a threshold value, the modified value is sent to the module, which will influence the future module's results. The module is also allowed to change the control parameter values. Those control parameter values are sent to user interface and displayed to the user.

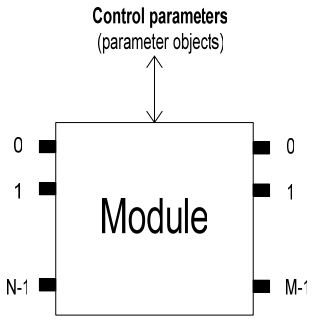


Figure 16: Control parameters are composed of parameter objects

Note: Modification of a control parameter's value is not a call reason. The current control parameter values are always passed to the `Process()` method as a method argument.

9.4 Initialization parameters

A module's initialization is also achieved by usage of parameter objects. Initialization parameters may be stored in a separate file or even retrieved from the module itself (method

`GetDefaultInitParameters()`), if no file could be found. Afterwards those initialization values are displayed to the user to offer a possibility to assign meaningful values to initialization parameters (e.g. which mpeg-file should be decoded). When the user has finished adapting the initialization values, those initialization parameters are sent to the module. It is the module's responsibility to interpret the initialization parameter values and to set the internal member variables according to them.

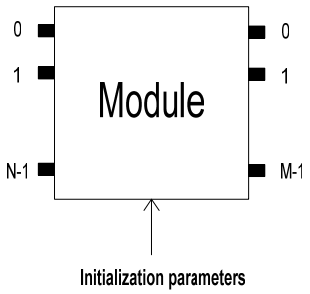


Figure 17: Initialization parameters are also composed of parameter objects

10 Utility, Tools

10.1 Runtime statistics

During runtime the framework offers various statistics to a module, which comprehend miscellaneous time values (execution time, overhead time, waiting times), fill levels (number of available time slots on each input pin in the past), number of modules in the graph and the reason, why the module has been called (new input, module has output something, cyclic time out or first time). For a more detailed description please refer to class `ExecutionInformation`'s doxygen documentation.

Classes:

`ExecutionInformation`

Module members:

```
ExecutionInformation* _execInfo;
```

10.2 Logging facilities

The framework offers to the module developer a logging facility for debug purposes. Class `Logger` provides methods for all basic data types to write them into a log file. The module developer can not change the name of the log file, the log file's name is specified by the application, which executes the framework.

Normally each call to a log method generates a separate log entry (line) in the log file. If several values have to be appended to compose one single line, just call method `Logger::IncOutStart` (inc is an abbreviation for incremental) to start a line and call method `Logger::IncOutEnd()` to close the line.

Classes:

`Logger`

Module members:

```
Logger* _logger;
```

11 Module DLL

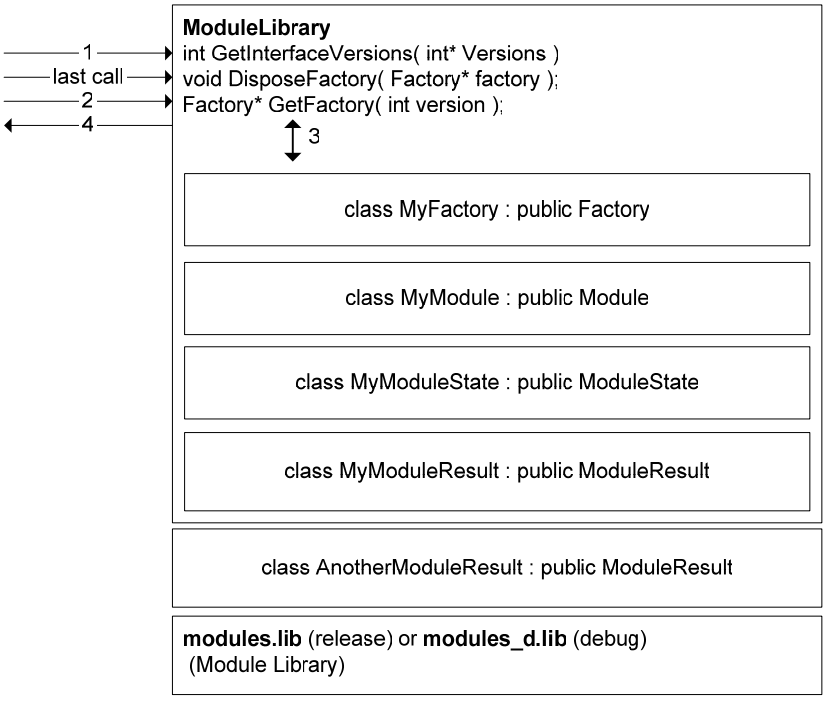
All abstract interfaces (like those for the class module, `Transformer`…) and all classes, which have to be implemented by the module developer, and some more (will be explained later) are compiled into one shared library (DLL on windows platforms). This DLL is loaded dynamically by the framework on demand, when the whole module graph is initialized.

Such a DLL contains at least the following compiled code:

- Abstract interfaces (shipped with the module developer SDK as library `modules_d.lib` as debug or `modules.lib` as release version) of all classes, with which a module may have to deal with, e.g. `Module` and `ModuleState` interfaces, parameter object interfaces, input streams and output streams interfaces…
- Implementation of `Module`, `ModuleState`, `ModuleResult`, `Factory` and `Transformer` interfaces done by the module developer
- DLL stub code (Functions `GetInterfaceVersions`, `GetFactory`, `DisposeFactory`)

Note: The code of the subclassed `Factory` class and the DLL stub code are generated automatically when using the `ModuleWizard`.

The following figure illustrates those DLL components:



The module DLL stub code consists of the following functions:

- `GetInterfaceVersions()`
- `CreateFactory()`
- `DisposeFactory()`

The following definitions mentioned here may be subject of change in future definitions. However the library's "`GetInterfaceVersions()`" must exist in every library of any interface version. The first function called by the framework after loading of the module library will be always "`GetInterfaceVersions()`", all other call sequences depend on the returned value.

11.1.1 GetInterfaceVersions()

Note: This function's code is generated automatically when using ModuleWizard.

`GetInterfaceVersions()` is the first method called by the framework and is intended to negotiate the behaviour between framework and module. As the specifications and module's interface may change over time in future releases (e.g. other or additional member methods, other or additional parameters to methods…), it is defined here that in all future releases DLL stub code function `GetInterfaceVersions()` will be called at first. All further calls launched by the framework vary depending on this function's result. Each module interface whenever defined somewhere else must be assigned a unique number. (The module interface defined and mentioned here has been assigned number "1"). When `GetInterfaceVersions()` is called by the framework, **a list of supported module interface versions is returned.** This handshake procedure avoids execution of modules not supported by a framework, e.g. if an old framework release, which is able to handle with modules of interface versions "2", "3" and "5", loads such a library, `GetInterfaceVersions()` informs the framework that the module contained in the library supports interface versions "4" and "6". In this case the framework refuses to load and execute the module and will inform the user that the framework is not able to co-operate with this module.

The library's "`GetInterfaceVersions()`" must be aware that no other functions of the library may be called and therefore must not allocate any resources.

Example code of a possible `GetInterfaceVersions()` implemention:

```
int GetInterfaceVersions( int * Versions )
{
      if ( Versions == NULL ) return 2;

      Versions[ 0 ] = 1;
      Versions[ 1 ] = 3;
      return -1;
}
```

`GetInterfaceVersions()` is called by the framework twice. The first time the framework passes a NULL pointer to indicate that the function should return the number of interfaces supported by this module. Then the framework allocates memory big enough to hold the number of interface versions and passes this address to the `GetInterfaceVersions()` function. Now the versions pointer is not equal to NULL and the function fills in as many version entries as returned in the first call. The framework guarantees to the function that Versions is a valid pointer.

This implementation informs the framework that the module contained in the library supports interface versions `1` and 3. The framework chooses one of them and calls the library's `CreateFactory()` function.

Note: With this handshake procedure the behaviour between framework and module is negotiated. The handshake procedure does not impose any restrictions which types of modules may be interconnected. It depends on the framework's capability whether it supports interconnections of modules with different interface versions. E. g. a module which supports interfaces 2, 5, 8, 12 may be interconnected with a module which supports interfaces 3, 4. In this case the framework has to be able to support one of the interface versions 2, 5, 8, 12 as well as one of interface versions 3, 4.

Note: The module does not know the list of interfaces supported by the framework and does not need to take care about.

11.1.2 CreateFactory()

Note: This function's code is generated automatically when using ModuleWizard.

`CreateFactory()` returns an instance of a `Factory` object contained in the module library. In C++ terms this `Factory` is a subclass derived from `Factory`, which is passed to the framework. The `Factory` knows how to create a module object or a module state object and a lot of other objects needed by this specific module. The framework uses the `Factory` object to create all the specific objects, e.g. module, module state object, module result... The `Factory` object returned by `CreateFactory()` is disposed by a call to `DisposeFactory()`.

```
Factory* CreateFactory( int Version );
```

Parameter `Version` passed to `CreateFactory()` is an element out of the result list returned by the module library's `GetInterfaceVersions()` function. This value is chosen by the framework and defines the further framework's behaviour. `CreateFactory()` returns the corresponding `Factory` object according to the requested interface version. The returned `Factory` object must also behave according to the requested interface version as specified by input parameter `Version` for function `CreateFactory()`. E.g. if a `Factory` object is returned for `Version` equal to two, the `Factory` object must also return objects which support interface version two.

11.1.3 DisposeFactory()

Note: This function's code is generated automatically when using ModuleWizard.

`DisposeFactory()` knows how to destroy the `Factory` created by `CreateFactory()`. The framework calls this function, if the `Factory` object is not needed any more.

12 Base classes reference

The module interface specifies several methods. Some of them are pure virtual (abstract) and have to be overwritten, some of them are virtual and are already implemented by the interface but may be overwritten.

Module is the abstract interface of a node in an arbitrary graph, which is executed by the framework.

A module developer has to subclass this abstract interface and to implement the abstract methods. Additionally a module developer has to subclass the following interfaces:

- `ModuleResult`
- `Factory` (implemented automatically when using ModuleWizard)
- `Transformer` (optional)
- `ModuleState` (not used yet)

ModuleWizard is a convenient tool to create a first compilable module skeleton. All interfaces are subclassed automatically and a workspace is created too. ModuleWizard provides a fully functionable implementation for interface `'Factory'`.

In the following sections only the most important member methods of the various abstract classes are listed. For a full documentation please refer to the doxygen documention shipped with the module SDK.

12.1 Factory Class Reference

Class Factory serves as an interface, which has to be subclassed and implemented by a module developer.

The required methods are specified below. Factory is - as the name already implies - responsible to create the specific object instances, developed by a 3rd party module provider. Factory does not need to implement for efficiency reasons some caching strategy, the framework reuses object instances already. The framework assumes that the CreateXXX methods contain only the new operator, which returns the created object instance. The same is true for the various overloaded Dispose methods.

12.1.1 Public Member Functions

virtual **InterfaceManager** * **CreateInterfaceManager** ()=0

Method CreateInterfaceManager creates an object instance of an interface manager.

virtual **Module** * **CreateModule** ()=0

*Method **CreateModule()** should create an object instance of a module.*

virtual **ModuleResult** * **CreateModuleResult** (unsigned int outputPin)=0

Method CreateModuleResult creates an object instance of a module result for the specified output pin.

virtual **ModuleState** * **CreateModuleState** ()=0

Method CreateModuleState creates an object instance of a module state.

virtual **Transformer** * **CreateTransformer** ()=0

Method CreateTransformer creates an object instance of a transformer.

virtual void **Dispose** (**InterfaceManager** *interfaceManager)=0

*Method **Dispose(InterfaceManager* interfaceManager)** destroys an interface manager.*

virtual void **Dispose** (**Transformer** *transformer)=0

*Method **Dispose(Transformer* transformer)** destroys a transformer object instance.*

virtual void **Dispose** (**ModuleResult** *moduleResult)=0

*Method **Dispose(ModuleResult* moduleResult)** destroys a module result object instance.*

virtual void **Dispose** (**ModuleState** *moduleState)=0

*Method **Dispose(ModuleState* moduleState)** destroys a module state object instance.*

virtual void **Dispose** (**Module** *module)=0

*Method **Dispose(Module* module)** destroys a module object instance.*

Note: These methods are automatically implemented when using the ModuleWizard (contained in the Module SDK).

12.2 Module Class Reference

Class Module is the abstract interface of a node in an arbitrary graph, which is executed by the framework. The most important

12.2.1 Public Member Functions

virtual bool **AcceptInputType** (unsigned int inputPin, const **ModuleResult** *moduleResult) const=0

The framework asks the module, whether this specific 'moduleResult' instance is accepted on the given input pin. The module must try to downcast the 'moduleResult' instance to the expected specific class at this input pin. Depending on the outcome of the downcast operation either true or false must be returned. This method is typically called by the framework during creation of the module graph. Each connection in the module graph is validated by the framework by calling this method. If the module returns true, the framework establishes an input stream between the previous module's output pin and this module's input pin.

***Note:** This method is implemented automatically when using ModuleWizard.*

virtual bool **Deinit** (**ModuleParameterFactory** *paramFactory)=0

Deinit is the inverse method to init, it notifies the module that the module should release all resources, which have been locked during the Init() call.

virtual ModuleParameters* **GetDefaultInitParameters** (const **ModuleParameterFactory** * paramFactory) const)=0

The module should return default initialization parameters. This method is only called if no initialization parameters could have been loaded from a module configuration file. The initialization parameters are intended to be displayed to the user, who may change them. After change those initialization parameters will be used for the **Init()** method call, which launches the "real" module's initialization.

virtual unsigned int **GetInputPinCount** () const=0

Returns the module's number of input pins. This property is assumed to remain constant for the whole life time of a module's instance.

virtual unsigned int **GetOutputPinCount** () const=0

Returns the module's number of output pins. This property is assumed to remain constant for the whole life time of a module's instance.

virtual bool **Init** (InitInfo *initInfo)=0

Init() is called to initialize the module according to the initialization values contained in InitInfo. The framework calls all modules to initialize themselves according to the initialization sequence, regardless of module crashes in the Init() call.

Note: COM initialization must be done in Init(), otherwise COM initialization will fail. The current framework implementation calls Init() in the context of the *main* process/thread, which is required for a successful COM initialization. Other methods like Start(), Stop(), Pause(), Resume() and Process() are called in another thread context..

virtual bool **Pause** ()

Pause is called when module's execution is suspended due to any reason (e.g. user pressed the pause button). Pause is only intended as an event notification method to inform the module that execution will be suspended for a longer period of time. This is useful when database connections are used, they may be terminated within pause to prevent the database from terminating the connection due to a time-out. by the database. No processing or some other computation is allowed to be done in this method. Non-const method calls to _inputStreams and _outputStreams result in a **ModuleException**. The input and output streams are locked by the framework to prevent modification of them during the pause method (The same is true for the Resume method).

virtual int **Process** (const **ModuleParameters** *ctrlParameters)=0

Process is the main method, in which all computation takes place. This method is called by the framework repeatedly until -the module generates an uncaught exception, or -the module indicates an error or -the module has finished computation. -any input stream is closed

virtual bool **Resume** ()

Resume is called when module's execution is resumed due to any reason (e.g. user pressed the resume button). Resume is only intended as an event notification method to inform the module that execution will be resumed. This is useful when database connections are used, because those connections may be timed out by the database and they may be revived within Resume. No processing or some other computation is allowed to be done in this method. Non-const method calls to _inputStreams and _outputStreams result in a **ModuleException**. The input and output streams are locked by the framework to prevent modification of them during the resume method (The same is true for the Pause method).

virtual void **Start** ()

This method is called by the framework in order to inform the module that now an arbitrary number of **Process()** method calls will follow immediately.

E.g. here a database connection may be established. Establishing a database connection within the **Init()** method may not be a good idea, because there is no guarantee when a **Process()** method call will happen after module initialization. And the time period between **Init()** and the first **Process()** call may cause a closing of the database connection performed by the database.

virtual void **Stop** ()

Method Stop serves only to notify the module that the framework does not launch any further **Process()** method calls any more due to any reason.

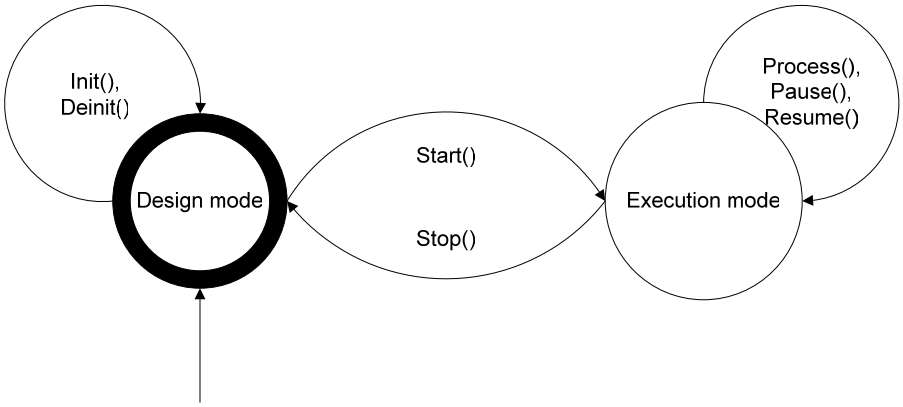
Stop() is the inverse method to **Start()**.

12.2.2 Method call sequence

The most important module methods are:

- Init()
- Start()
- Process()
- Pause()
- Resume()
- Stop()
- Deinit()

The mentioned method calls are not called completely arbitrarily, there are certain restrictions when a specific method is called by the framework. The framework specifies two modes: a design mode and an execution mode. In design mode, the module is mainly asked to initialize and deinitialize itself (Init/Deinit, AcceptInputType, GetInputPinCount, GetOutputPinCount and the various member setter methods). When the framework enters the execution mode, the module is signaled by a Start() call to prepare itself for execution. Once the framework has entered the execution mode, the module methods Resume, Pause and Process are allowed to be called. Pause and Resume are reverse methods, which signal the module either that execution is suspended for an arbitrary time or that execution is resumed.



12.2.3 Exact method call sequence

From construction to initialization:

SetLogger SetExecMode SetMethodProgress SetTotalProgress SetOutputStreamsProgress SetModuleState SetLogger SetUtility GetInputPinCount GetOutputPinCount GetModuleName GetModuleInfo GetModuleVersion GetModuleDeveloper SetSharedMemory SetExecMode GetDefaultInitParameters Init

From initialization to execution:

SetExecInfo SetInputStreams SetOutputStreams **Start … Pause Resume Process (N times) Pause Resume Process (M times) … Stop** SetExecInfo SetInputStreams SetOutputStreams

From execution to initialization:

Deinit

Note: The exact method call sequence regarding the various SetXXX() methods may vary between framework implementations.

Note: In design mode the module is neither able to write something to its output streams nor to read something from any input stream.

12.2.4 Protected Attributes

Every module inherits from its parent class a few members, which are set to meaningful values by the various set methods (e.g. `SetInputStreams`):

- **ExecutionInformation** * **_execInfo**
- **AF::EXECUTION::MODE _execMode**
- **InputStreams** * **_inputStreams**
- **Logger** * **_logger;**
- Progress * **_methodProgress**
- **ModuleState** * **_moduleState**
- **OutputStreams** * **_outputStreams**
- **OutputStreamsProgress** * **_outputStreamsProgress**
- **SharedMemory** * **_sharedMemory**
- Progress * **_totalProgress**
- Utility * **_utility**
- char * **_workingDir**

12.3 ModuleResult Class Reference

Base class for a module's computed result. A module developer is responsible to subclass ModuleResult to encapsulate a computation result, which is forwarded by the framework to subsequent modules when the module returns from it's Process() call.

In the current framework implementation each module is executed in it's own thread, so all modules receive and forward module results completely asynchronously. When a module is connected to subsequent modules, those subsequent modules may read at the same time the same module result object instance in parallel. Therefore the module developer has to assure, that the module result's reading operations are thread-safe. The concept foresees that subsequent modules are only allowed to perform reading operations on module results received on the input streams. Modules should never modify a module result's values originating from any previous module! Rule of thumb for making thread-safe operations: "Const" methods normally do not change an object's internal values. Therefore reading operations should be based completely on those "const" methods, which are typically all kinds of some "get" methods. It is highly recommended not to use "mutable" member variables, because those variables make the constant "get" methods not thread-safe!

12.3.1 Public Member Functions

Media samples (visual and audio) typically have different sample rates, e.g. 25 frames per second for a video and 44.1kHz for audio. In order to support both types of sample rates in a single module result instance, a module result stores them separately in private members “audioSampleRate” for audio samples and “visualSampleRateNumerator” respectively “visualSampleRateDenominator”. The visual sample rate is not stored as a single float number, instead the sample rate is stored as a numerator and a denominator, which form a fraction, e.g. 30000/1001 equals about 29.97 frames per second. In this example 30000 denotes the visualSampleRateNumerator and 1001 denotes the visualSampleRateDenominator. The “audioSampleRate”, “visualSampleRateNumerator” and “visualSampleRateDenominator” are set by the corresponding Get() and Set() methods, which are publicly available in a module result instance.
In order to have a common basis of these different sample rates, module result supports additionally a so called “clockRate”. Clock rate is intended to be used as a single common sample rate, valid for all sample rates, instead of having to take into account the various values for different audio and visual sample rates. For this purpose the clock rate is computed as least common multiple (lcm) of the used audio and visual sample rate and is set by the SetClockRate() method. Member “clockRate” is undefined by default.

JRS-internal Note: In the meantime a few modules have been written (especially TransitionDetection, KeyFrameExtraction…), which use the above mentioned topics of a common clock rate and various sample rates extensively. These modules use formulas for computation of the sample (frame) number out of a sample rate number, which cause confusion, when people have to deal with them for the first time.
Definition: The clock rate is the least common multiple (LCM) of the most frequently used sample rates (44.1kHz, 48 kHz, 25 fps, 29.97 fps…). Therefore the following formula may be stated:

*clockRate = videoToClockRatio * videoSampleRate*

(where videoSampleRate = visualSampleRateNumerator / visualSampleRateDenominator)
videoToClockRatio is just a simple factor, which has to be multiplied with the used videoSampleRate in order to get the clock rate (other stated: the "upsample" factor or: How many times the video sample rate has to be "oversampled" in order to get the clock rate).
Assumption: The clockRate is set correctly by the previous module, the same is true for visualSampleRateNumerator and visualSampleRateDenominator values (explained below).
The time slots in the input- and outputstreams are given on the clockRate basis. If computation works on a sample (frame) number basis, the given time slot values have to be transformed into sample numbers (look at the JRS-internal example below).

*timeSlotValue = videoToClockRatio * sampleNumber*

It is important to note, that sampleNumber is not equal to the number of elapsed sample ticks. E.g. 29.97 fps may be formulated as 30000/1001, which means that there are 30000 sample ticks per second, but only after every 1001 sample ticks, the sample number is increased. Only after 1001 triggered sample ticks, a new sample is created and the sample number is increased. Therefore the sampleNumber has to be multiplied by the visualSampleRateDenominator (e.g.1001) in order to get the number of elapsed sample ticks:

*sampleTicks = sampleNumber * visualSampleRateDenominator*

The JRS-internal modules use for time stamps the IplJrsMediaTimePoint class, which is based on sampleTick values. It requires usage of the above mentioned nominator and denominator values.
IplJrsMediaMediaTimePoint::FractionsOneSecond: This value is equal to visualSampleRateNumerator (e.g. 30000 for 29.97 fps or 25 for 25 fps).
IplJrsMediaTimePoint::SetFractionIncrement: This value is equal to visualSampleRateDenominator (e.g. 1001 for 29.97 fps or 1 for 25 fps)
IplJrsmediaTimePoint::TotalNrOfFractions: This value is equal to sampleTicks.
JRS-internal example:
Assumption: *myModuleResult* has been extracted from the input stream and points to my current ModuleResult and *from1* indicates the begin of the segment.

```
_clockRate= myModuleResult->GetClockRate();
_visualSampleRateNumerator= myModuleResult ->GetVisualSampleRateNumerator();
_visualSampleRateDenominator= myModuleResult ->GetVisualSampleRateDenominator();
_audioSampleRate= myModuleResult ->GetAudioSampleRate();
_videoToClockRatio= (_clockRate *_visualSampleRateDenominator) / _visualSampleRateNumerator;

iplJrsMediaTimePoint.SetFractionsOneSecond(_visualSampleRateNumerator);
iplJrsMediaTimePoint.SetFractionIncrement(_visualSampleRateDenominator);
iplJrsMediaTimePoint.SetTotalNrFractions((from1/_videoToClockRatio)*
       _visualSampleRateDenominator);
```

JRS-internal Note: The whole fractions and sample ticks stuff is only necessary for sample rates, which have to be noted as a fraction like NTSC sample rate 29.97fps.

unsigned int **GetAudioSampleRate** (void)
Returns the audio sample rate

unsigned _int64 **GetClockRate** (void)
Returns the clock rate.

void * **GetMetaData** () const
This method is used by the framework and provides a way to associate some meta data with this object.

virtual unsigned int **GetSize** () const=0
Returns the number of bytes allocated by this object instance directly and indirectly.

unsigned int **GetVisualSampleRateDenominator** (void)
Returns the visual sample rate denominator.

unsigned int **GetVisualSampleRateNumerator** (void)
Returns the visual sample rate numerator.

virtual void **Reset** ()=0
Due to the cost of dynamic memory allocation the framework reuses object instances of type ModuleResult.

void **SetAudioSampleRate** (unsigned int audioSampleRatePar)
Sets the audio sample rate.

void **SetClockRate** (_int64 clockRatePar)
Sets the clock rate.

void **SetMetaData** (void *ptr)
This method is used by the framework and provides a way to load some meta data associated with this object.

void **SetVisualSampleRateDenominator** (int visualSampleRateDenominatorPar)
Sets the visual sample rate denominator (e.g. 1 for 25 fps)

void **SetVisualSampleRateNumerator** (int visualSampleRateNumeratorPar)
Sets the visual sample rate numerator (e.g. 25 for 25 fps)

virtual const char * **ToString** () const=0
Provides a string representation of the result encapsulated by an object instance of this type.

12.3.2 Protected Member Functions

ModuleResult ()
A module result object instance is created by the factory.

virtual **~ModuleResult** ()
A module result object instance is disposed by the factory.

12.4 Transformer Class Reference

Class Transformer "translates" data structures, which are stored in a **ModuleState** or **ModuleResult** object instance to data structures, which are "understood" by the framework.

In the current framework implementation only the method listed above is used.

12.4.1 Public Member Functions

virtual **ModuleParameters** * **Transform** (const **ModuleResult** *moduleResult, unsigned int outputPin, **ModuleParameterFactory** *paramFac, **ModuleParameterStorage** *storage, Utility *utility)=0

Transforms a module result object instance to a set of ModuleParameter(s), which are stored in a ModuleParameters object.

12.4.2 Protected Member Functions

Transformer ()
A transformer object instance is created by the factory.

virtual **~Transformer** ()
A transformer object instance is disposed by the factory.

13 Performance

[This chapter requires an update and is intended for JRS internal use.]

13.1 Observations

Analysis framework performance profiling on a pentium D (2 logical CPUs). All percentage values give the ratio = modules computation time / (module computation times + module waiting times). The percentage values indicate the CPU usage.

A "standard JRS graph" consists of modules

- IMFileReader, PMCameraMotion, PMCameraMotionDescriptor, PMFaceDescriptor, PMFaceDetector, PMKeyframeDescriptor, PMKeyframeExtractor, PMSimpleMotionActivityDescriptor, PMSimpleMotionActivityExtractor, PMStripeImageDescriptor, PMStripeImageExtractor, PMTransitionDescriptor, PMTransitionDetector, PMVisualFeatureExtractor

13.1.1 Offline mode

Offline mode is characterized by active waiting, modules wait on each other (most/all JRS modules have been written for this use case). A module sends a computation result to all subsequent modules and waits until all of them have sent an acknowledgment that the computation result has been received.

Different graphs, depending on parallelism, from 48% to 97% total percent CPU usage

1. Standard JRS graph: ~64% +/-3% (Debug, long&short videos), 52% +/-2% (Release, long&short videos)
2. Well parallelized graph: up to ~97% (both Debug & Release)

13.1.2 Online mode

No active waiting, modules do not wait on each other when forwarding computation results. A module sends a computation result to all subsequent modules and does not wait for any acknowledgment.

(Buffer restriction: JRS IMFileReader module decodes a video pretty fast -> images are buffered -> memory overflow). Only short videos can be analyzed due to buffer restriction.

3. Standard JRS graph: ~83% +/-2% (Debug), ~64% +/-2% (Release)
4. Well parallelized graph: up to ~97% (both Debug & Release)

13.1.3 Runtimes

Offline mode, Standard JRS graph

5. Debug: video length 00:03:24 -> execution 01:07:37 (hh:mm:ss) = Factor 19
6. Debug: video length 00:00:12 -> execution 00:03:13 (hh:mm:ss) = Factor 16
7. Release: video length 00:03:24 -> execution 00:27:08 (hh:mm:ss) = Factor 8
8. Release: video length 00:00:12 -> execution 00:01:18 (hh:mm:ss) = Factor 7

Online mode, Standard JRS graph

9. Release: video length 00:00:12 -> execution 00:01:01 (hh:mm:ss) = Factor 5

13.2 Hypotheses

13.2.1 Test case 1

The synchronization overhead between module threads takes place in kernel mode, which is assumed to be constant. Modules' release code is optimized with respect to speed, therefore the part of module code execution gets smaller in relationship to the whole execution time compared to modules' non-optimized debug code. Therefore release version's CPU usage is worse than the debug version (release: less module computation, constant sync time; debug: more module computation, constant sync time).

13.2.2 Test case 2

CPU usage seems to depend to a large degree on the parallelizing possibilities of the modules. If the modules do not depend much on each other and can work most of the time completely independent, CPU usage is relatively high. To demonstrate this, a "well parallelized graph" has been constructed. This graph consists of several (4 – 9) modules (they compute visual features of a frame), which work completely in parallel, relying only on one previous module (video decoding module).

13.2.3 Test casse 1 & 2

Comparison of test case 1 and test case 2 demonstrate the possible CPU usage if module execution can be parallelized well.

13.2.4 Test case 1 & 3

CPU usage differences between online and offline mode for the same graph indicate the amount of time that is spent by active waiting. Online mode seems to improve CPU usage by 10% (release) to 20% (debug).

13.2.5 Test cases 5, 6 versus 7, 8

Execution of release module code results in a performance boost of about factor 2.

13.2.6 Test cases 8 & 9

Test case 8 & 9 reflect the amount of time spent by active waiting. Test case 8 uses active waiting, test case 9 does not, the rest of the configuration should be the same. The different graph execution times seem to reflect the different synchronization implementations. A performance gain of about 20% based on offline execution times seem to be theoretically feasible if online mode is used.

14 Module native GUI

[This chapter deals with the concept of a module's native GUI and may be considered as optional information for most module developers.]

The concept of a module described so far has already mentioned the concept of the transformer and module parameters, which may be used for display of module results, initialization and control parameters. However these module parameters may be not sufficient and/or inefficient for display of specific types of data. For this purpose the tasks of module result visualization and user interaction may be completely outsourced to a specialized software component: *module native GUI*

The module native GUI is responsible for the whole interaction between the module and the user. If a module native GUI exists, the framework does neither take care of any module result visualization nor any user interaction tasks.

The software architecture of a module's native GUI is similar to that of a module. The design has defined a few interface classes, which are inherited by the new native GUI subclass. The interface classes are contained in two packages: a listener and a GUI package

The listener package contains a set of a few interface classes, which may be considered as callbacks. These callbacks are completely independent of any GUI issues and are therefore contained in a separate package with its own header and library files.

The GUI package contains the interface classes, which have to be subclassed and implemented by the module native GUI developer.

A module native GUI may be considered as a passive software component, which inherits and implements a few listener interface (callback) classes. These callback methods are invoked by the framework in turn of any module activity.

Example: A module performs an operation on its input streams, e g. reading a module result. The invoked method of the input streams implementation calls the native GUI listener, which in turn has to react properly, e.g. display of the read module result content.

Module execution (start/stop/pause/resume) still remains the application's responsibility by usage of the framework, but nevertheless the native GUI is allowed to talk with the module directly (therefore a member variable '_module' with a pointer to the module instance is defined in the GUI interface).

A module's native GUI is registered by the framework by an entry in the module's configuration file. The configuration files are described in detail in chapter "Configuration files".

14.1 Listener interface classes

The listener interface classes specify callbacks, which are invoked by the framework, e.g. there are callback methods which are invoked when a module performs some operations on its input or output streams. It is the native GUI's task to react properly when such a listener method is called by the framework.

Note: A listener's callback method may be called by the framework from another thread's context than the main application's GUI thread. If not explicitly otherwise stated, this is true for all defined listener interfaces. A listener has to be implemented in a thread-safe way.

14.1.1 AbstractListener

Base class for all listener interfaces. It provides storage of a module ID, which is convenient to know, from which module the event comes from. It is the framework developer's responsibility to set the module ID. The framework guarantees that a listener, which is subscribed for a module A, will never receive any notification from any other module.

It is up to the listener implementation to ensure proper locking when entering a listener method and unlocking when leaving a listener method.

14.1.2 ErrorListener

Interface definition of all methods, which are called by the framework in case of an error.

14.1.3 GraphListener

Set of all callback methods, which are notified by the framework, whenever the module graph is changed. A graph listener does not have to be thread safe implemented, but this may change in future versions.

Note: The native module GUI subclass does not inherit this interface class because this interface is foreseen for a central graph component. It is mentioned here for the purpose of completeness.

14.1.4 InputStreamsListener

Set of interface methods, which are called by the framework each time the module performs some operation on its input streams. The methods of class InputStreams are mapped as close as possible to methods of class InputStreamsListener. Additionally the return value of the called input streams method is passed as first parameter 'result' to the listener callback methods.

Some methods of class InputStreams are provided only for convenience, e.g. ShiftAllWindowsXXX(). These methods, which are just a combination of other basic methods (e.g. ShiftAllWindowsXXX() calls method ShiftWindowXXX() for all input pins), are not explicitly listed as a listener method, because the basic methods (e.g. ShiftWindowXXX()) are anyway supported as listener methods. Apart from the ShiftAllWindowsXXX() methods, methods

- PopFrontSegment() (composed of GetSegment() and ShiftWindowBeginAbsolute()),
- ReadDataRelative() (converted to a ReadDataAbsolute() call)
- ShiftWindowBeginRelative() (converted to a ShiftWindowBeginAbsolute() call)

are also not supported as input streams listener callback methods. The relative method calls (ReadDataRelative() and ShiftWindowBeginRelative()) are converted internally to absolute method calls (ReadDataAbsolute() and ShiftWindowBeginAbsolute()), which in turn call the appropriate listener callback methods. And method PopFrontSegment() is composed of a GetSegment() followed by a ShiftWindowBeginAbsolute() call, where each method calls the appropriate listener callback method again. An InputStreamsListener has to be thread-safe, the callback methods are called from any other thread contexts than the main application's GUI thread. Class ModuleResult is wrapped by a ModuleResultReferenceCounter, which contains a reference counter, for usage by the listener implementation. If a module result must be locked for later use by the listener, the reference counter has to be increased within the appropriate callback method immediately. Later, when the module result instance is not used any more, the reference counter must be decreased again. Modification of the reference counter is implemented in a thread-safe way by the framework. The passed reference counter is NULL, when the module has output a NULL result by methods OutputStreams::AppendNullRelative() or OutputStreams::AppendNullAbsolute(). Although the corresponding method in class InputStreams is called "IsConnected(...)", the listener method is named "IsInputPinConnected(...)" in order to avoid nameclashes, when using multiple inheritance from the OutputStreamsListener and InputStreamsListener interfaces (method "IsConnected(...)" in class OutputStreamsListener is named "IsOutputPinConnected(...)").

14.1.5 LogListener

Set of interface methods, which are called by the framework each time the module logs something. The methods defined in class Logger are mapped to listener methods with the same signature. A LogListener implementation has to be thread-safe.

14.1.6 ModuleListener

Set of interface methods, which are called by the framework each time *after* the corresponding module method call. The module methods are mapped as close as possible to the listener methods, additionally the module's method return value is passed as parameter 'result' to the listener's

methods. The various GetXXX()/SetXXX() methods to set a module's member variables are not supported as listener methods. A ModuleListener implementation has to be thread-safe.

14.1.7 OutputStreamsListener

Set of interface methods, which are called by the framework each time the module performs some operation on its ouput streams.

The methods of class OutputStreams are mapped as close as possible to methods of class OutputStreamsListener. Additionally the return value of the called output streams method is passed as first parameter 'result' to the listener callback methods. Methods

- AppendDataRelative()
- AppendNullRelative()
- AppendDataAbsolute()
- AppendNullAbsolute()

are mapped to listener method AppendAbsolute(). The relative values passed as arguments to AppendDataRelative() and AppendNullRelative() are converted to absolute values. If the module uses either AppendNullRelative() or AppendNullAbsolute(), then parameter 'refCounter' is NULL in listener method AppendAbsolute() and AppendFinished().

If a module uses OutputStreams::AppendDataRelative(), the output streams implementation provided by the framework returns a ModuleResult instance, which is then filled up by the module with some data. The implementation code within OutputStreams::AppendDataRelative() notifies the listener by calling method AppendAbsolute() (the relative values are converted to absolute values). When the listener is notified by this method, the ModuleResult instance is still empty. When the module finishes its Process() method, then the framework notifies the listener again but by calling method AppendFinished(). Now the ModuleResult instances should have been filled with meaningful values and the listener is now able to interpret these data.

To sum up the difference between AppendAbsolute() and AppendFinished():

- AppendAbsolute(): Immediate notification of the listener (module is still within the Process() method), enables to keep track the module's output streams operations. The passed ModuleResult instances are empty.
- AppendFinished(): Notification of the listener with filled ModuleResult instances, after the module has left the Process() method.

Although the corresponding method in class OutputStreams is called "IsConnected(...)", the listener method is named "IsOutputPinConnected(...)" in order to avoid nameclashes, when using multiple inheritance from the OutputStreamsListener and InputStreamsListener interfaces (method "IsConnected(...)" in class InputStreamsListener is named "IsInputPinConnected(...)").

An OutputStreamsListener implementation has to be thread-safe.

14.1.8 ProgressListener

Set of interface methods, which are called by the framework each time the module indicates a new progress (either based on method, output streams or total progress).

A ProgressListener implementation has to be thread-safe.

14.2 GUI interface classes

The GUI interface classes define the interface for the module's native GUI. The interface classes consist of two classes: GUIFactory and QModuleGUI.

14.2.1 GUIFactory

GUIFactory is called by the framework to create and dispose a native GUI component instance. This class is automatically generated when using ModuleWizard.

14.2.2 QModuleGUI

QModuleGUI is the base class for a module's native GUI. Q is a reminder, that the native module's GUI depends on Qt. QModuleGUI in turn inherits the following listeners:

- ErrorListener
- InputStreamsListener
- LogListener
- ModuleListener
- OutputStreamsListener
- ProgressListener

Due to the empty default implementations of the listeners, it is up to the native GUI to implement as many callbacks as desired.

15 CentralGraphComponent

[This chapter deals with the concept of a module's native GUI and may be considered as optional information for most module developers.]

In the concepts described so far there is still one missing point: Sometimes it may be necessary to get a global overview of all the ongoing activities within a graph. For this purpose a central graph component has been designed. A central graph component resides in its own DLL and may be considered as a passive software component, which is notified by the framework each time an operation has been performed on the graph, e.g. a module has been added or removed.

A central graph component is registered in the graph configuration file. Multiple central graph components may be registered at the same time for the same graph. It is guaranteed by the framework that all central graph components are loaded before the first module is loaded. The framework also unloads the central graph components after unloading of all modules. There is no sequence defined in case of loading multiple central graph components.

15.1 Interface classes

A central graph component consists of two classes:

- CentralGraphComponentFactory
- CentralGraphComponent

15.1.1 CentralGraphComponentFactory

CentralGraphComponentFactory

is called by the framework to create and dispose a central graph component instance. This class is automatically generated when using ModuleWizard.

15.1.2 CentralGraphComponent

Base interface class for all central graph components and inherits listener GraphListener. GraphListener is a set of all callback methods, which are notified by the framework, whenever the module graph is changed.

Note: A graph listener (respectively a central graph component) does not have to be thread safe implemented, but this may change in future.

16 Configuration files

16.1 Graph configuration file

A graph configuration is stored in an XML file and contains the following information:

- Graph execution mode
- A list of used modules associated with a unique name
- Reference for each module to the module's own configuration file
- Interconnections between modules (one module's output pin to another module's input pin)
- `Module` initialization sequence (in which sequence the modules have to be initialized by the framework)
- (Optional) A list of central graph components
- (Optional) How a global graph progress is computed

As indicated previously the module's unique name is defined in the graph configuration file. The associated unique name is only used internally by the framework and is completely transparent to the module.

A sample graph configuration is given below:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<GraphConfiguration>
	<ExecutionMode>
		OFFLINE_MODE
	</ExecutionMode>

	<CentralGraphComponent>
			<CentralGraphComponentID>
				Central_1
			</CentralGraphComponentID>
			<CentralGraphComponentLibrary>
				..\..\CGCTestCentralGraphComponent_d.dll
			</CentralGraphComponentLibrary>
	</CentralGraphComponent>

	<Module>
		<ModuleID>
			TEST_MODULE_0
		</ModuleID>
		<ModuleConfiguration>
			..\ModuleConfig\testModuleConfig.txt
		</ModuleConfiguration>
		<Activated>
			true
		</Activated>
	</Module>

	<Module>
		<ModuleID>
			TEST_MODULE_1
		</ModuleID>
		<ModuleConfiguration>
			..\ModuleConfig\testModuleConfig.txt
		</ModuleConfiguration>
		<Activated>
			true
		</Activated>
```

```
	</Module>

	<Interconnection>
		<SrcModuleID>
			TEST_MODULE_0
		</SrcModuleID>
		<OutputPin>
			1
		</OutputPin>
		<TgtModuleID>
			TEST_MODULE_1
		</TgtModuleID>
		<InputPin>
			7
		</InputPin>
	</Interconnection>

	<GlobalProgress>
		<Operand>
			ADD
		</Operand>
		<ModuleID>
			TEST_MODULE_0
		</ModuleID>
		<ModuleID>
			TEST_MODULE_1
		</ModuleID>
	</GlobalProgress>

	<InitSequence>
		<Item_0>
			TEST_MODULE_0
		</Item_0>
		<Item_1>
			TEST_MODULE_1
		</Item_1>
	</InitSequence>
</GraphConfiguration>
```

Section ExecutionMode contains one of two possible entries: `ONLINE_MODE` and `OFFLINE_MODE` The module may get knowledge in which mode it is executed, but it is not recommended to the module developer to implement different module behaviours depending on the module execution mode.

The various `Module` entries assign a unique module ID to the module and specify the concrete module configuration file, which contains the module's initialization parameters.

Each interconnection tag specifies a distinct link between one module's output pin and another module's input pin.

`InitSequence` specifies the sequence in which the modules are initialized by the framework (at first <Item_0>, <Item_1>…).

Central graph components may be considered as graph listeners, which are notified by the framework each time the graph is modified/updated, e.g. a module is added or removed, an interconnection is added or removed, a module is initialized or deinitialized.

GlobalProgress: To give some feedback to the user about the current work progress, a progress bar is often used. The framework provides a feature to define how this 'global work progress' is computed, which is done by the various modules in the graph. The global progress section may contain two different tag types:

1. <Operand>: Indicates which operand is used to compute the current overall progress of the mentioned modules. Allowed values: MINIMUM, MAXIMUM, ADD, MULTIPLY The <Operand> tag is allowed to exist only once within <Progress>
2. <ModuleID>: This tag may occur more often than once. Each occurrence of a <ModuleID> contains a unique ID of a module. The set of all <ModuleID> tags defines those modules, which are used to compute the 'global work progress' of the graph.

16.2 Module configuration file

A module configuration is stored in an XML file and contains the following information:

- Reference to the module's DLL
- Initialization parameters
- (Optional) Reference to module's log file (if not existing, it is created), whether it should be used and when it should be reset
- (Optional) Reference to module's GUI DLL
- (Optional) Whether the module's GUI should be used or not.
- (Optional) The module's working directory

Instead of having to define the initialization parameters and the associated values each time before the module is initialized, those parameters may be specified once and stored in a module configuration file.

If no initialization parameters are contained in the module configuration file, the module is asked for default initialization parameters. (This approach seems to be strange, but is useful because the initialization parameters may be shown to the user, who is able to modify them before the module is initialized.)

The structure of a module configuration file is given below:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ModuleConfiguration>

      <ModuleLibrary>
            ..\..\Modules\samples\ \ComputeHistoModule_d.dll
      </ModuleLibrary>

      <LogFile>
            ..\ LogFiles\TestModule.log
      </LogFile>

      <ResetLogFile>
            MODULE_LOAD
      </ResetLogFile>

      <ModuleGUILibrary>
      ..\ guiLib\TestModuleGUI_d.dll
      </ModuleGUILibrary>

      <UseModuleGUI>
            true
      </UseModuleGUI>

      <UseLogFile>
            YES
      </UseLogFile>

      <WorkingDirectory>
            ../TemporaryFiles/TestModule
      </WorkingDirectory>
```



```
      <PrettyName>
            Camera motion estimation
      </PrettyName>

      <ModuleParameter>
            <Name>
                  Source file
            </Name>
            <Label>
            </Label>
            <Description>
            </Description>
            <User>
                  true
            </User>
            <Value>
            …
            </Value>
      </ModuleParameter>
</ModuleConfiguration>
```

ModuleLibrary's entry specifies the path to the module DLL either absolute or relative to the module configuration file.

After section ModuleLibrary multiple `ModuleParameter` entries in the list comprise the initialization parameters. Each `ModuleParameter` entry defines one initialization parameter. A parameter exists of a name, a label, a description and a value. Entry "User" specifies, whether this initialization value must be displayed to a user or not (this may be useful when the user must specify a file path unknown a priori to the module). Label and Description are optional values, which are not evaluated by the framework. Value contains subentries, which specify the initialization value in more detail.

Values for `ResetLogFile:`

`MODULE_LOAD`: LogFile is reset when the module's DLL is loaded
`MODULE_UNLOAD`: LogFile is reset when the module's DLL is unloaded
`MODULE_INIT`: LogFile is reset before `Module::Init()` is called
`MODULE_DEINIT`: LogFile is reset before `Module::Deinit()` is called
`MODULE_START`: LogFile is reset before `Module::Start()` is called
`MODULE_STOP`: LogFile is reset before `Module::Stop()` is called
`MODULE_PAUSE`: LogFile is reset before `Module::Pause()` is called
`MODULE_RESUME`: LogFile is reset before `Module::Resume()` is called

17 ModuleWizard

17.1 Module creation

The module wizard is a tool, which assists the module developer in creation of a module.

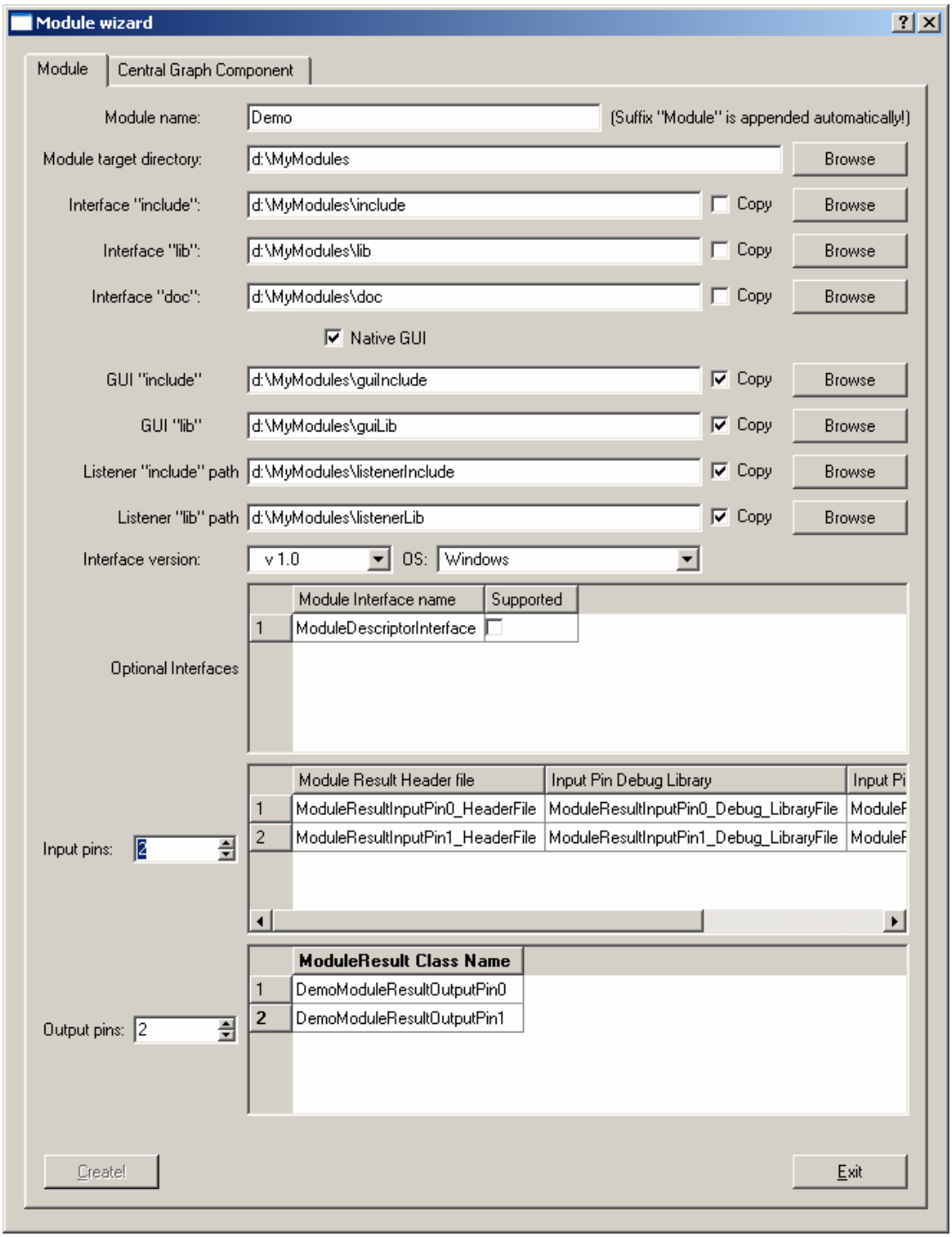
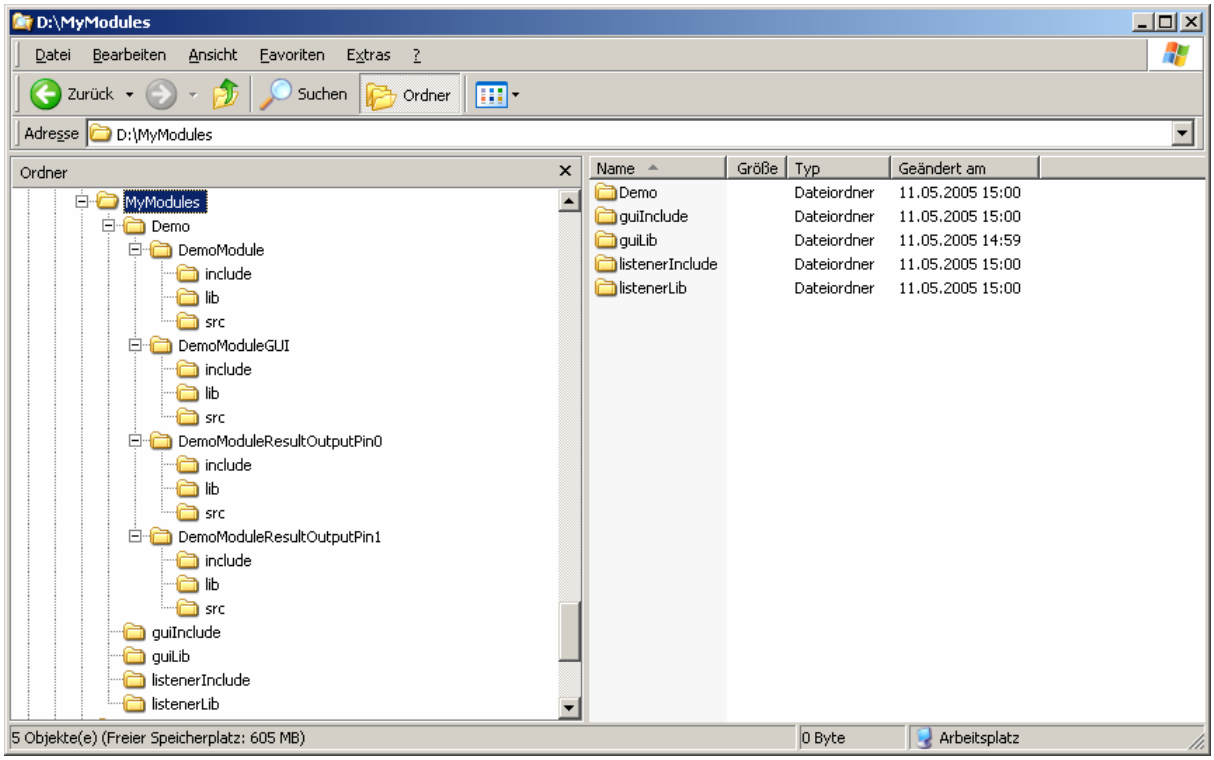


Figure 18: Screenshot of **`ModuleWizard`**

The `ModuleWizard` creates a code skeleton for the subclasses, which have to implemented by the module developer. Additionally the DLL stub code and the implementation of the `Factory` subclass is generated automatically. It is highly recommended to use the `ModuleWizard` to avoid programming errors. The framework assumes a DLL stub code and `Factory` subclass implementation code, which is generated by the `ModuleWizard`.
Note: The `ModuleWizard` contains templates for module skeletons in one of its subdirectories. Do not change these template files!
`ModuleWizard` generates automatically a project file for Microsoft's Developer Studio Version 6.0 including project and workspace files with correct compiler and linker settings.
`ModuleWizard` generates a directory structure (below the *MyModules* directory) according to the example shown in the previous figure:



For each specified module result class a separate directory below the module name directory ("Demo") is created. These directories contain the include, lib and src subdirectories of the module's result classes. Each module result "lib" directory contains the compiled static library. These static libraries have to be specified at ModuleWizard's input pins section.

17.1.1 Module Target Directory

The "root" for all further module related subdirectories.

17.1.2 Module Name

The text entered into the "Module Name" field determines the module's name and the "root" directory for all subdirectories related to this new module. The module's "root" directory is created below the "module target directory".

17.1.3 Interface "include"

This text field specifies the directory where the module interface header files are located. If these header files do not already exist somewhere, the ModuleWizard offers the optional functionality to copy the shipped module interface header files to the specified directory. In any case the

ModuleWizard generates automatically the compiler's include paths to the specified include directory. If the header files exist already in another directory, it is just necessary to specify this directory as "include" directory without checking the "Copy" checkbox, otherwise these module interface header files would be overwritten.

17.1.4 Interface "lib"

This text field specifies the directory where the module interface library files are located. If these library files do not already exist somewhere, the ModuleWizard offers the optional functionality to copy the shipped module interface library files to the specified directory. In any case the ModuleWizard generates automatically the linker's input paths to the specified lib directory. If the library files exist already in another directory, it is just necessary to specify this directory as "lib" directory without checking the "Copy" checkbox, otherwise these module interface library files would be overwritten.

17.1.5 Interface "doc"

This text field specifies the directory where the module interface documentation files have to be copied too. If "Copy" is not checked, nothing is done. The module interface documentation files are shipped with the ModuleWizard and have been generated by doxygen out of the module interface header files.

17.1.6 Native GUI

If checked the ModuleWizard generates also a code skeleton for a module's native GUI, which is contained in a completely separated DLL. A module's native GUI programming environment also requires directories for the interface header and library files as shown in the example above (guiInclude, guiLib, listenerInclude, listenerLib, DemoModuleGUI).
If "Native GUI" is checked, the text fields GUI "include", GUI "lib", Listener "include" path and Listener "lib" path are enabled and an additional GUI module class is created, which is inherited from various listener classes of the listener and GUI package.

17.1.7 GUI "include" path

This text field specifies the directory where the gui interface header files are located. If these header files do not already exist somewhere, the ModuleWizard offers the optional functionality to copy the shipped gui interface header files to the specified directory. In any case the ModuleWizard generates automatically the compiler's include paths to the specified include directory. If the header files exist already in another directory, it is just necessary to specify this directory as "include" directory without checking the "Copy" checkbox, otherwise these gui interface header files would be overwritten.

17.1.8 GUI "lib" path

This text field specifies the directory where the gui interface library files are located. If these library files do not already exist somewhere, the ModuleWizard offers the optional functionality to copy the shipped gui interface library files to the specified directory. In any case the ModuleWizard generates automatically the linker's input paths to the specified library directory. If the library files exist already in another directory, it is just necessary to specify this directory as "library" directory without checking the "Copy" checkbox, otherwise these module interface library files would be overwritten.

17.1.9 Listener "include" path

This text field specifies the directory where the listener interface header files are located. If these header files do not already exist somewhere, the ModuleWizard offers the optional functionality to copy the shipped listener interface header files to the specified directory. In any case the ModuleWizard generates automatically the compiler's include paths to the specified include directory. If the header files exist already in another directory, it is just necessary to specify this directory as

"include" directory without checking the "Copy" checkbox, otherwise these listener interface header files would be overwritten.

17.1.10 Listener "lib" path

This text field specifies the directory where the listener interface library files are located. If these library files do not already exist somewhere, the ModuleWizard offers the optional functionality to copy the shipped listener interface library files to the specified directory. In any case the ModuleWizard generates automatically the linker's input paths to the specified library directory. If the library files exist already in another directory, it is just necessary to specify this directory as "include" directory without checking the "Copy" checkbox, otherwise these module interface library files would be overwritten.

17.1.11 Interface version

ModuleWizard supports creation of modules and native GUIs only according to interface version 1.0.

17.1.12 OS

ModuleWizard supports creation of modules and native GUIs only for Windows (Linux has not been tested).

17.1.13 Optional Interfaces

The generated module class may be also inherited from those interfaces, which are selected from the list of supported additional interfaces. Currently one additional module interface ("ModuleDescriptorInterface") exists. This interface has been introduced mainly in order to discriminate between modules, whether they are able to handle MPEG-7 documents.

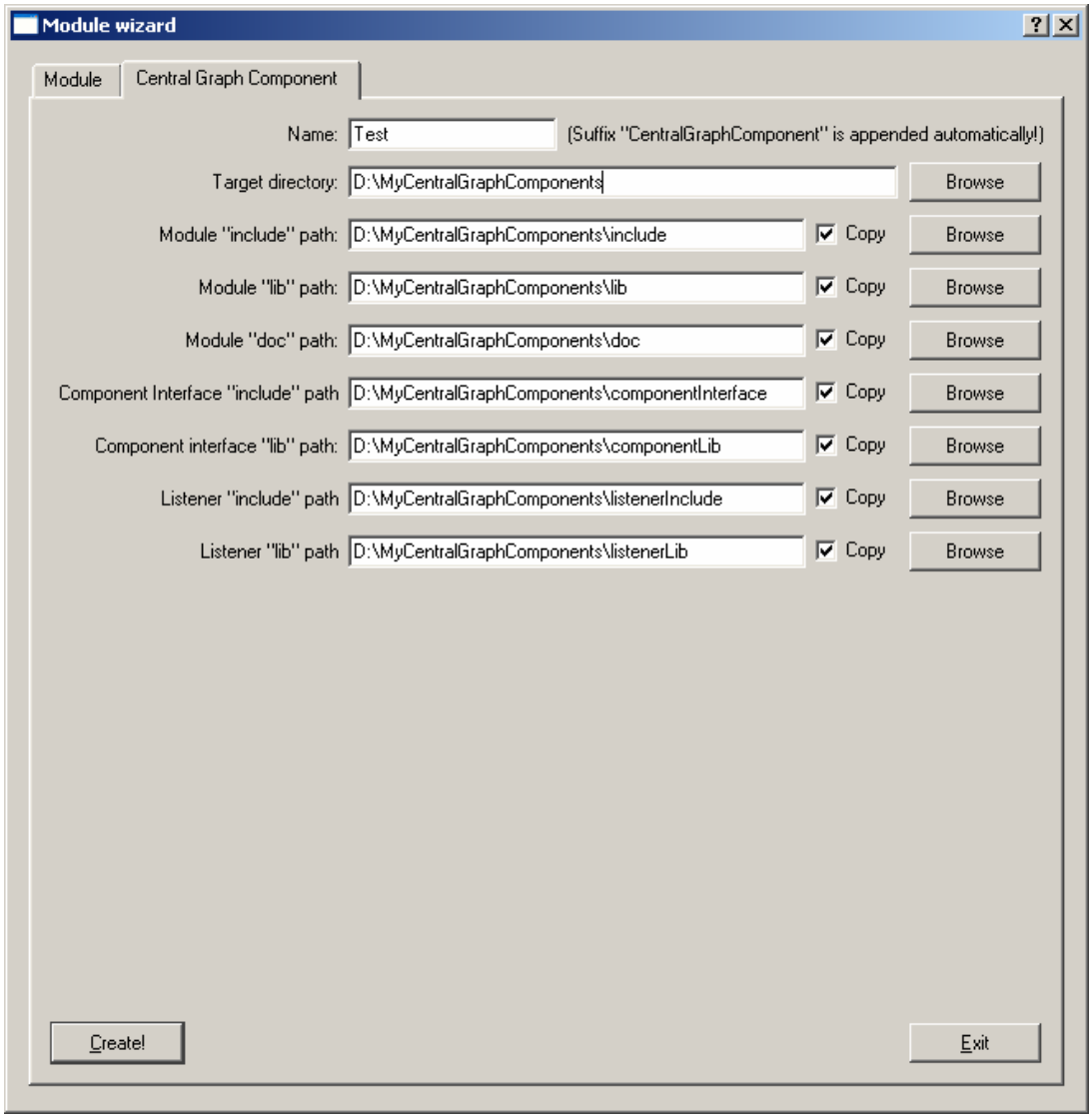
17.1.14 Input pins

The input pins table consists of three columns
Module Result Header File: The header file of the module result class, which will be included in the module's cpp file.
Input Pin Debug Library: The debug library file of the module result class, which will be linked to the module DLL.
Input Pin Release Library: The release library file of the module result class, which will be linked to the module DLL.

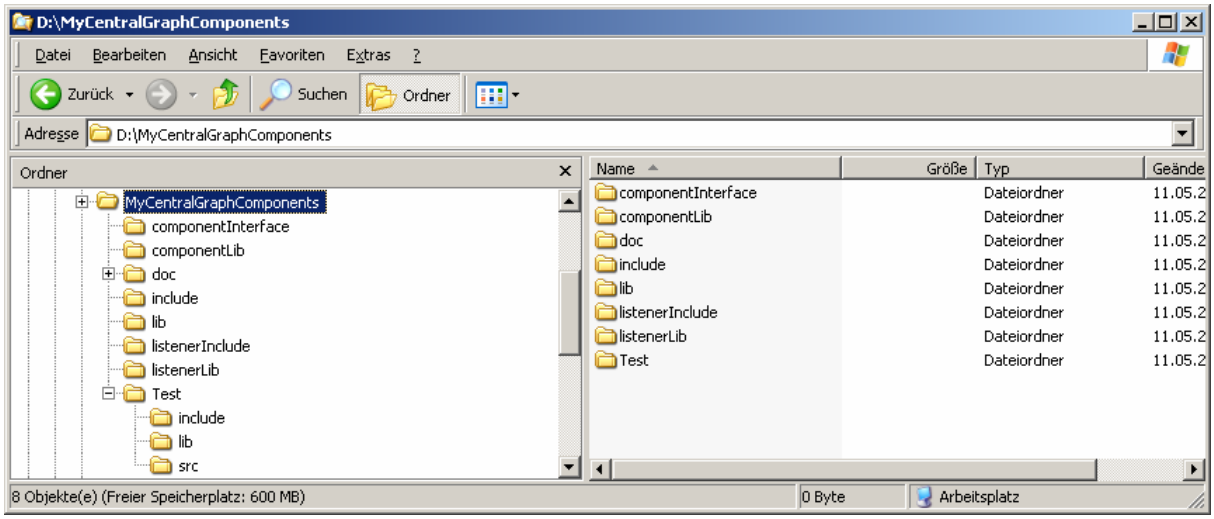
17.1.15 Output pins

The output pins table consists of the "ModuleResult class name" column. This column specifies the name of the module result subclass, which is also the name of the directory below the module's name directory ("Demo").

17.2 CentralGraphComponent



Creation of a central graph component is supported by the ModuleWizard too. Similar to a module's native GUI a central graph component is contained in a completely separated DLL. A central graph component programming environment requires a few directories, which contain the interface header and library files.
The directory structure created by the ModuleWizard looks like in the following screenshot according to the entries in the example above:

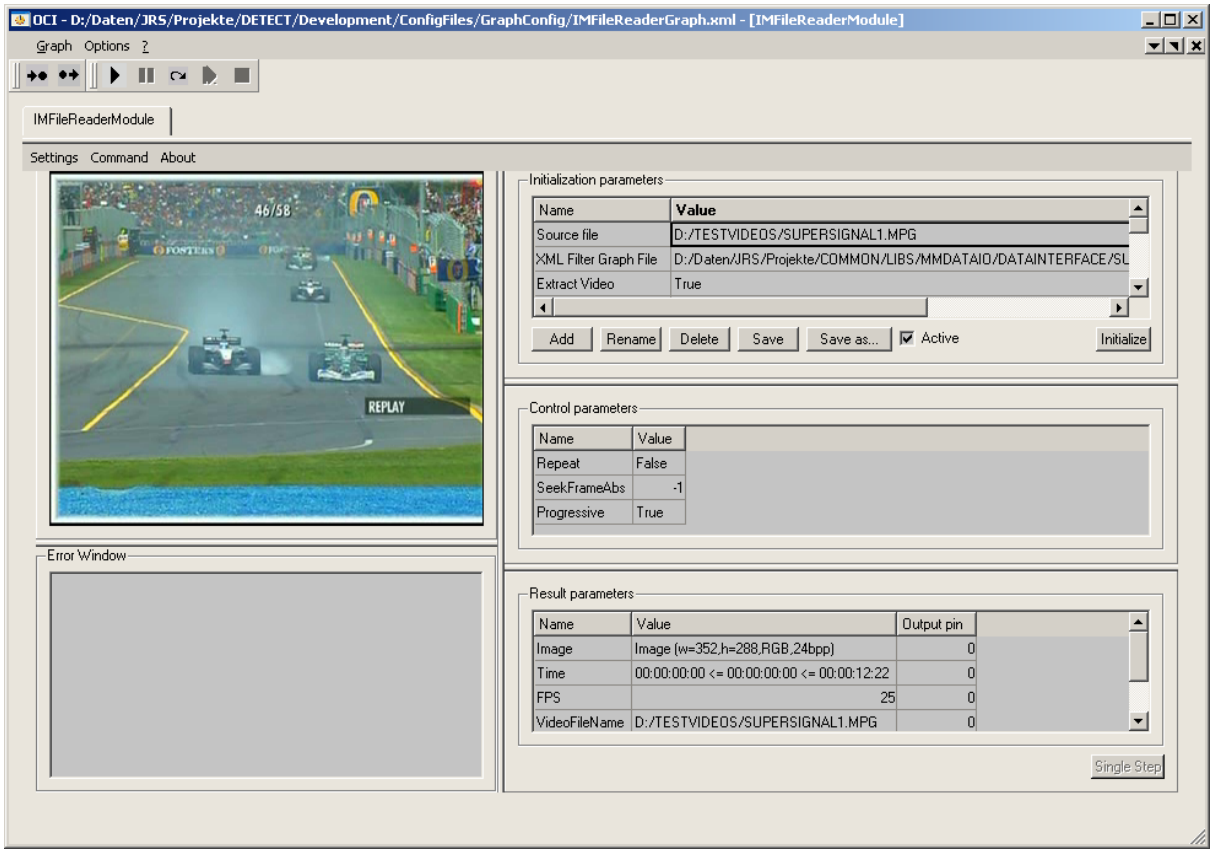


Target directory, Module "include", Module "lib", Module "doc", Listener "include", Listener "lib" entries have been already described in the previous sections of creation of a module respectively native module GUI. Entries Component Interface "include" path and Component interface "lib" path have to be considered in a similar way. These directories contain the central graph interface header and library files.

18 Observation & Control Interface (OCI)

The module development kit contains an application, which may be considered as a graphical frontend based on the framework. OCI offers:

- Basic graph execution commands (init, play, pause, single step, resume, stop, deinit)
- Display of initialization, control and result parameters
- Interactive modification of initialization and control parameters
- Time synchronized display of control and result parameters originating from different modules (overlay)



Modification of initialization and control parameters is achieved by double clicking on the desired table cell. Depending on the parameter type a dialog will appear, that allows modification of the parameter.

18.1 OCI menu "Options"

18.1.1 Submenu "Update Settings…"

OCI polls the framework for new events of any type. The "Update Settings…" define the period of time elapsed between two poll requests. Lower values mean that GUI update is faster, but less computation time is left the framework for module execution (on a single CPU machine).

18.2 Module menu "Settings"

18.2.1 Submenu "Select output pins…"

This command tells OCI which module results of which output pins shall be displayed.

18.2.2 Submenu "Route graphical output to…"

If multiple modules are contained in the graph, OCI offers the possibility to overlay module results originating from different modules within one graphical window. For this purpose menu "Settings" command "Route graphical output to…" has to be selected from the module's menu, which results have to be routed to another module's graphical output window.

Note: Rerouting of module results is a bit risky and may end in a crash of OCI due to tremendous memory consumption. Technical background: Each module is executed by the framework in its own thread, which means they are executed completely asynchronously. As a consequence different modules may have output results for different time slots. If multiple module results are routed to the same output window, OCI displays them altogether at once only when module results exist for the same time slot (synchronized output). But: OCI is unable to determine whether a module will output any module result on an output pin in the future or not. If an output pin of a module is selected for display and this module is rerouted to another module but does not output any result on the specified output pin, OCI will buffer and buffer and buffer module results from other modules and will cause finally a tremendous memory leak resulting in a crash. This endless buffering of module results happens due to the fact, that OCI does never get any module result from such an undefined output pin and will never be able to proceed with displaying of module results for the next time slots.

18.3 Console parameters

Because OCI is intended for module developers to debug their modules, it is a nice feature to control the application by console parameters in order to avoid waste of time required for repeated tasks like graph loading, initialization, execution start…

For this purpose OCI offers the following console parameters:

Usage: OCI -open graphfile -init -start -pause [msec] -wait [msec] -singlestep [times] -resume [msec] -stop [msec] -deinit WhenFinished -close -exit [msec]

Note: [Values in brackets are optional]

Note: All given console parameters are considered as a list of user interaction commands and are treated in the same way. If any command is not possible at the specified index in the list of interaction commands, the command is ignored, e.g. open followed immediately by deinit (-open … -deinit), and execution of the remaining commands is continued.

Note: While OCI executes the interaction commands passed by the console, a user may still interact with OCI, but the user not able to stop the execution of the interaction commands. This feature is foreseen for the next OCI version.

18.3.1 -open graphfile

Opens the specified "graphfile", which may be either relative or absolute. Same as choosing "Graph" / "Open Graph"

18.3.2 -init

The same as pressing the "Init All" button in OCI's Toolbar.

18.3.3 -start

Same as pressing the "Start" button in OCI's Toolbar.

18.3.4 -pause [msec]

Waits [msec] milliseconds, until graph execution is paused. [msec] is optional and should be positive.

18.3.5 -wait [msec]

Waits [msec] milliseconds while the graph's state (loaded, initialized, running, paused…) remains the same.

18.3.6 -singlestep [times]

Same as pressing "Single Step" button in OCI's Toolbar [times] times.

18.3.7 -resume [msec]

Waits [msec] milliseconds and then resumes graph execution. Same as pressing "Resume" button in OCI's Toolbar.

18.3.8 -stop [msec]

Waits [msec] milliseconds and then stops graph execution. Same as pressing "Stop" button in OCI's Toolbar.

18.3.9 -deinit WhenFinished

Deinitializes the graph. If WhenFinished is specified, it is the same as pressing "Deinit All" button in OCI's Toolbar. If not a deinitialization of the graph is forced regardless of the current graph state.

18.3.10 -close

Closes the graph. Sames as choosing "Graph" / "Close Graph"

18.3.11 -exit [msec]

Waits [msec] milliseconds and exits OCI afterwards. Same as choosing "Graph" / "Exit"

