The Sigslot Concept

Using hierarchical components
to create complex applications
out of re-usable code fragments

1 Contents

2 Introduction (2005-07-15)

Signal-Slot is a well known technique to connect different modules. Often it is used by frameworks or libraries to implement graphical user interfaces (GUIs). Examples: QT from Trolltech, KDE based on QT, etc.

The idea behind signal-slot-connections is to implement modules independent of each other in a way that a module does not know anything about other modules. In that way, it is most likely that a module can be put into a completely other application and be used for another purpose without changing anything within the module.

This document introduces a special implementation of the signal slot concept that is called **"Sigslot"**. This implementation follows different goals than most common implementations:

- small overhead compared to direct function calls
- optimized for static systems with static connections between components known at compile time
- scalable from small embedded systems to large and complex systems

The most important goal that is achieved by Sigslot is the "Basic Sigslot Paradigm":

- make it as simple as possible for a developer to implement a software module (called "component" within the Sigslot language).

Whenever there has been a design decision for Sigslot that makes it easier to implement a component and more complex to use that component, this decision has been taken. There are some tools that support the usage of components to handle that increased complexity.

3 The Concept behind Sigslot (2005-08-17)

3.1 Basic Terms

"Figure 1: Some First Terms" explores some of the basic terms used by the Sigslot concept. You might also take a look at the glossary in the appendix of this document. That appendix "Glossary (2005-07-18)" gives some more definitions for the important terms regarding the signal slot concept.

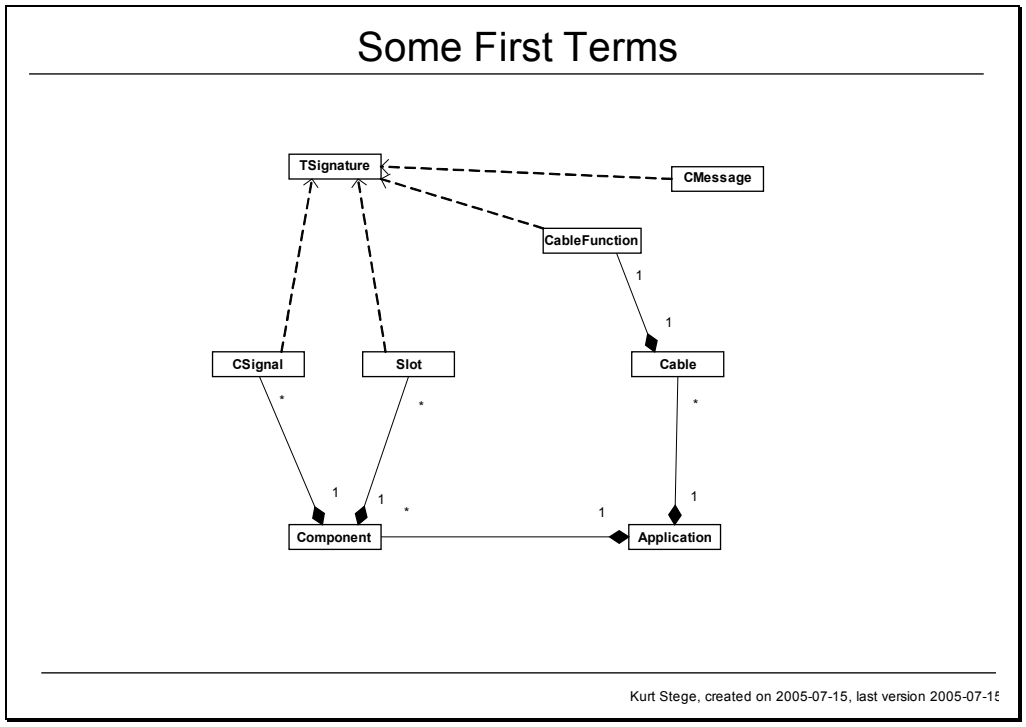

Figure 1: Some First Terms

TODO: "Composite Component"

3.2 Basic Example (2005-09-13)

The first terms discussed in section "3.1 Basic Terms" discuss the Sigslot concept from the viewpoint of the C++ language. This first example shows the complete source code for a basic example. It includes the source code of some components, the connections between these components and even the relevant source code for the implementation of the Sigslot library. The purpose of this example is to demonstrate the simplicity of the concept. This simplicity is important for example when debugging components or applications.

Let us begin with a component that converts speed information data from meters per second (mps) to kilometres per hour (kmh).

I would like to write the source code like this:

```
class CConvert
{
```

```
    slot   input  (double speed_in_mps);
    signal output (double speed_in_kmh);
};

/*!
    Gets a speed value in meters per second,
    converts it to kilometers per hour, and
    fires the output signal with the result.
*/

slot CConvert::input (double speed_in_mps)
{
    output (speed_in_mps * 3.6);
}
```

This uses two new identifier “signal” and “slot” with a difficult semantics regarding the underlying language of C++.

“slot” is easy. It could be typedef’d to void. This is exactly the semantics of a slot. A slot is a function implemented by the component that gets parameters specified by the signature of the slot and returns void.

“signal” is not so easy. Its semantics is: This is a function (or an object that behaves like a function) only defined and used by the component, but implemented by the compiler (or anywhere out of the component). If I could find a way to implement that function in an easy way I would like to switch to that notation.

Meanwhile, actually components are written like this:

```
#include "Sigslot/TSignature.hpp"

typedef NSigslot::TSignature1<double> tSig_speed_in_kmh;

class CConvert
{
    public:

    void SLOT_input  (double speed_in_mps);
    tSig_speed_in_kmh::CSignal SIGNAL_output;
};

/*!
    Gets a speed value in meters per second,
    converts it to kilometers per hour, and
    fires the output signal with the result.
*/
void CConvert::SLOT_input (double speed_in_mps)
{
    SIGNAL_output (speed_in_mps * 3.6);
}
```

As a code convention, signals and slots have got a prefix in their name. The identifier “slot” has been replaced by “void”. The Sigslot concept provides a class CSignal. Thus, the signal is defined as an ordinary member variable. This member variable is public, for it is part of the interface of the component. Applications that use the component have to access the signal to connect it to the receivers.

Some detailed explanations: All identifiers defined by the Sigslot concept are put in the namespace "NSigslot". The header file "Sigslot/TSignature.hpp" defines a template class for signatures. This template gets the list of types for the messages as parameters. To be more precise, for each number of parameters there is an own template called TSignature<n>, where n is the number of parameters (from 0 to some reasonable large value like 9). This template class TSignature is a trait class that does not have any members or functions on its own. You are not supposed to create members of a TSignature class. Instead, the template class provides type and class definitions for messages, cables, connections, and signals for the given signature. The example uses a typedef to give the signature a name. That seems to be useful but is not necessary. Defining the signal as "NSigslot::TSignature1<double>::CSignal SIGNAL_output;" works as well.

Note: "SLOT_input" and "SIGNAL_output" are *not* reasonable good names for a signal or a slot of a component. Better use component specific reasonable names. When a component has just one signal and/or one slot, and is such generic that there is no specific name, I suggest to use "operator()" for the slot and just the sole prefix "SIGNAL" for the signal.

A graphical UML representation of the component is shown in Figure 2: Example Component "Convert".

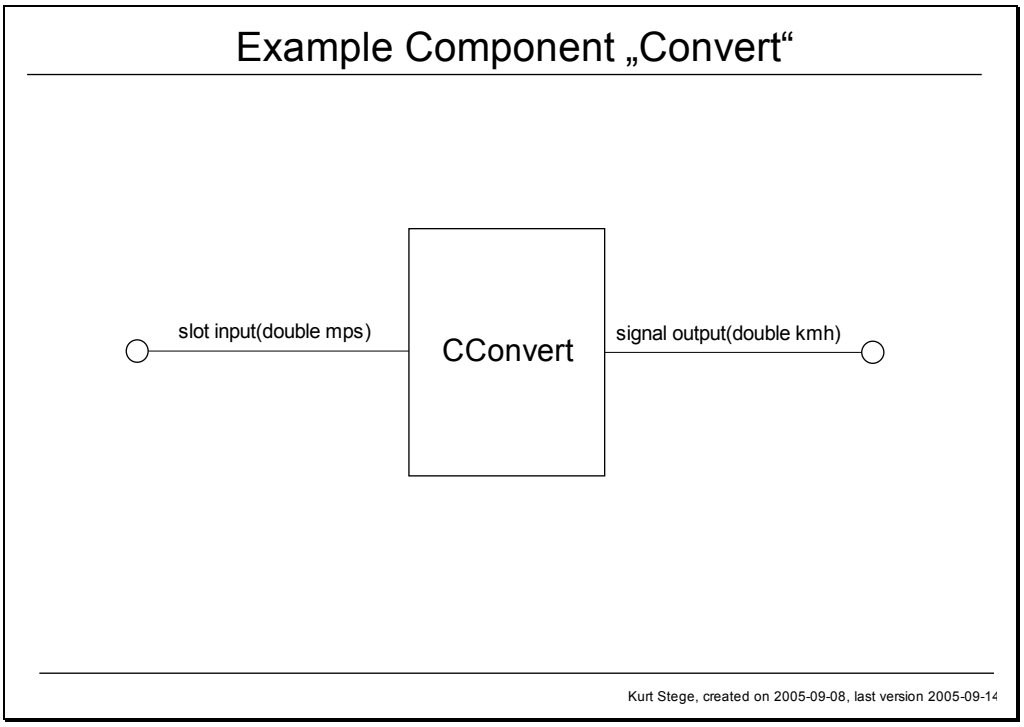

Figure 2: Example Component "Convert"

Now for some other equally simple components.

```
typedef NSigslot::TSignature1<double> tSig_speed_in_mps;

class CSensor
{
    public:

    tSig_speed_in_mps::CSignal SIGNAL_mps;
```

```
    void measureSpeed();
};

void CSensor::measureSpeed()
{
    // This demo assumes 23 miles per hour.

    SIGNAL_mps (23.0);
}
```

The component CSensor just has one signal (and no slots) that emits the current speed (in meters per second) of whatever the sensor is measuring. CSensor has a public function “measureSpeed()” as part of its interface. It is important to recognize that signals and slots are not everything on the world. When some other means of communication is more appropriate, it might be better to use them. A Sigslot component may have other interfaces as well.

Note: In this example, technically “void measureSpeed()” is a slot. It just hadn’t got it in the name.

The last two components implement each a slot that gets a speed value and prints it onto the screen.

```
#include <iostream>

class CPrintKMH
{
    public:

    void SLOT_kmh (double speed_in_kmh);
};

void CPrintKMH::SLOT_kmh (double speed_in_kmh)
{
    std::cout << "The speed is " << speed_in_kmh << " kmh.\n";
}


class CPrintMPS
{
    public:

    void SLOT_mps (double speed_in_mps);
};

void CPrintMPS::SLOT_mps (double speed_in_mps)
{
    std::cout << "The speed is " << speed_in_mps << " meters per
second.\n";
}
```

These four components can easily be wired together to an application, as shown in Figure 3: Application Example "Convert".

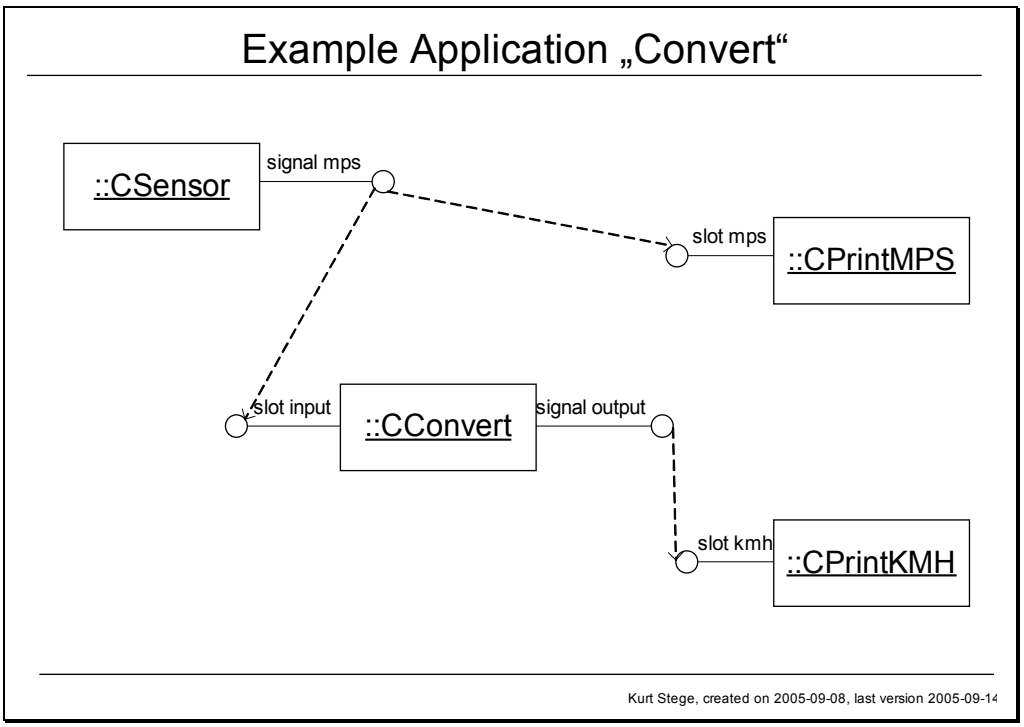

Figure 3: Application Example "Convert"

The measured speed of the sensor (in miles per hour) is sent to the printer and to the converter. The converter converts the speed into kilometres per hour, and sends it to a second printer.

Note: It would be possible to create one class CPrint that is configured during run time about the unit or that simply has two slots (one for each unit) and create two instances of that class. An application has its focus on components as objects of a component class, and not on component classes itself.

TODO: Which order is best? First the application code showing cables and connection, and then the implementation of TSignature.hpp, or the other way around? I show first the application. We are in an example, and thus the whole example is tidied together.

The Sigslot concept tries to make the development of components as easy as possible. The components listed above listed show that this goal is achieved at least for this target. Later on we will see that this holds even for complex components requiring several threads and asynchronous communications. The drawback of this strategy is that the usage of components is more complex than otherwise necessary. That drawback is taken by code generator Cogen that generates the composite components with all its cables and connections, and thus removes this complex effort completely from the developer. Despite this fact, this example shows the whole truth and shows the application of Figure 3: Application Example "Convert" in all its complexity. It is somewhat mysterious, but not *that* worse.

Note: The code shown in this example is written by hand. The tool Cogen has not yet been developed.

Note: The Sigslot concept allows nearly infinite possibilities to write such an application and the connections between the components. This example chooses one solution and does not mention any alternatives. Other strategies are discussed in chapter TODO.

```
class CApp
{
```

```
    private:

    // The components:

    CSensor     m_sensor;
    CConvert    m_convert;
    CPrintMPS   m_printMPS;
    CPrintKMH   m_printKMH;

    // The cables:

static void cable_sensor_mps  (void *context, const
tSig_speed_in_mps::CMessage &msg);
static void cable_convert_output(void*context,const
tSig_speed_in_kmh::CMessage &msg);

    public:

    void run_application();
};
```

The application creates for each of its components an object, in this example as member variables. Further, for each signal that has to be connected to one or more slots, the application implements a static function that handles the call of that signal. Each signal is able to store just one pointer to such a static function, called cable function, and one void pointer that may be used to provide some context data for that cable.

At last, the application declares a function “run_application()” that lets the application do some work.

```
void CApp::run_application()
{
    // plug the cables into the signals

    m_sensor.SIGNAL_mps     =
tSig_speed_in_mps::CSignal(cable_sensor_mps, this);
    m_convert.SIGNAL_output =
tSig_speed_in_kmh::CSignal(cable_convert_output, this);

    // let the application do some application specific work

    m_sensor.measureSpeed();
}
```

The constructor of CApp creates the components. The connections between the components are created and configured by the function “run_application()”. After plugging the cables into the signals, the application calls the function “measureSpeed()” of its sensor component. As we have already seen, the sensor pretends to measure some speed and emits the result using its signal. In this example it is a dummy sensor, so the result will be 23 miles per hour, every time.

The signals are member variables of its components, and thus already constructed when its components are constructed. The default constructor of a signal leaves it in an unconnected state without any receivers. This application uses the assignment operator of signals to change that state. “CSignal(cable, this)” creates a temporary object, a signal, that will call the given cable function (“cable”) with the given pointer (this, points to the application itself). Thus, the line “signal = CSignal(cable, this);” copies the temporary object into signal. Thus, signal will call the provided cable function, when it emits a message.

Using this way, both signals of the application are connected with the cable functions.

The first cable function looks like this:

```
void CApp::cable_sensor_mps (void *context, const
tSig_speed_in_mps::CMessage &msg)
{
    CApp *this_ = static_cast<CApp *>(context);
    this_->m_printMPS.SLOT_mps(msg.m_par1);
    this_->m_convert.SLOT_input(msg.m_par1);
}
```

It is a static function that gets the "this" pointer to the object explicit as first parameter. Thus this cable function is only a pseudo-static function. The reason for this lies deep hidden in the language of C++. The cable function retrieves the original type of the "this" pointer, has thus access to the components of the application and calls all the slots as required by the connection diagram.

The second and last cable function works the same.

```
void CApp::cable_convert_output(void *context, const
tSig_speed_in_kmh::CMessage &msg)
{
    CApp *this_ = static_cast<CApp *>(context);
    this_->m_printKMH.SLOT_kmh(msg.m_par1);
}
```

Now, the application is complete. Just add a main function…

```
int main()
{
    CApp app;

    app.run_application();
}
```

…and let it run. The output is:

```
The speed is 23 meters per second.
The speed is 82.8 kmh.
```

That is really all. Provide the header file "Sigslot/TSignature.hpp" and compile the source code. There is even no library required, for this far all the stuff is template classes.

TODO: Provide a reference to the chapter that discusses the content of TSignature.hpp.

3.3 Important Extensions

Keywords: Composite Components, Threads, Synchronization, Asynchronous Connections.

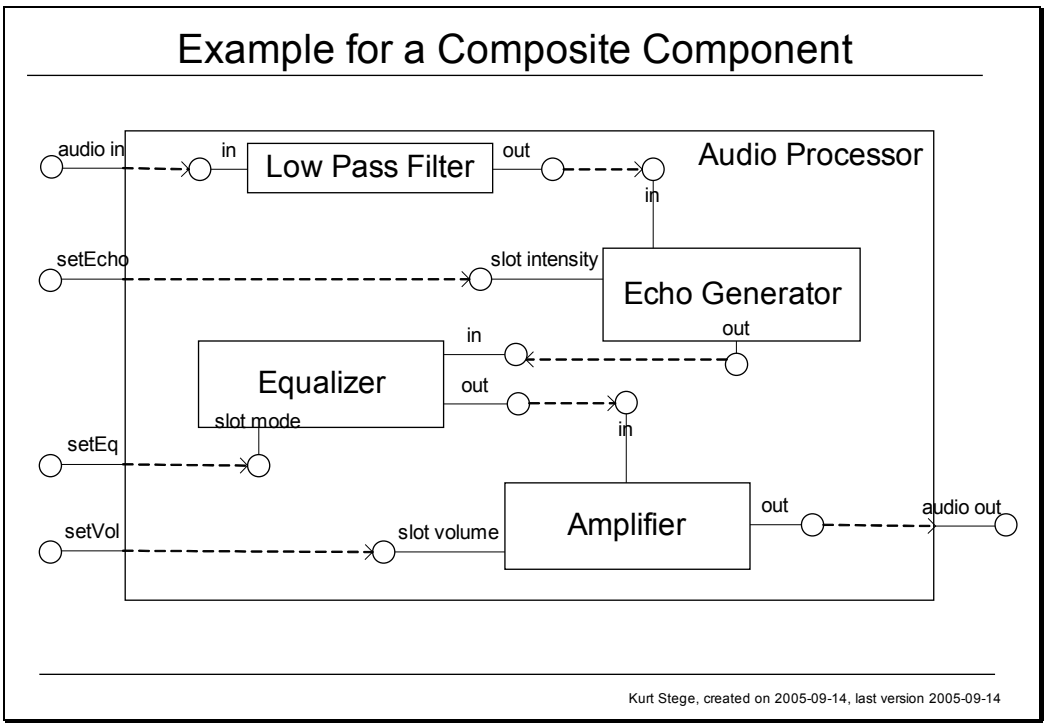

Figure 4: Example for a Composite Component

3.4 Example for a Multi Threading Application

3.5 Code Generation Tools

There are several tools available that help to use the Sigslot concept for complex applications. The most important tool at all is called (TODO: Find a good name) "Cogen" and generates composite components and applications.

3.5.1 Cogen – Composite Component Generator

Input: Definition of a composite component (list of inner components, list of public signals and slots, list of connections).

Output: Source code for a class that implements the defined component. This source code can be used like any other source code of any simple component.

Special: Typical, the definition of a composite component lists further composite component definitions as inner components instead of already generated composite components. Thus, Cogen typically is called just once for the outermost composite component (that is the application) and not for each inner composite component. While creating the outermost composite component Cogen reads the inner composite components and generates the source code for that components as well, recursively through all levels. Even more, Cogen is able to eliminate all inner composite components and transforms the whole application into an equivalent flat form with just one composite component and lots of inner simple components wired together.

3.5.2 Cogen – Application Generator

To generate an application you can use Cogen with an additional parameter. It generates an ordinary composite component and gives it additional functions (like start() and stop(), details are under development) that handle start up and shut down of the components.

4 Examples (2005-07-18)

xxx

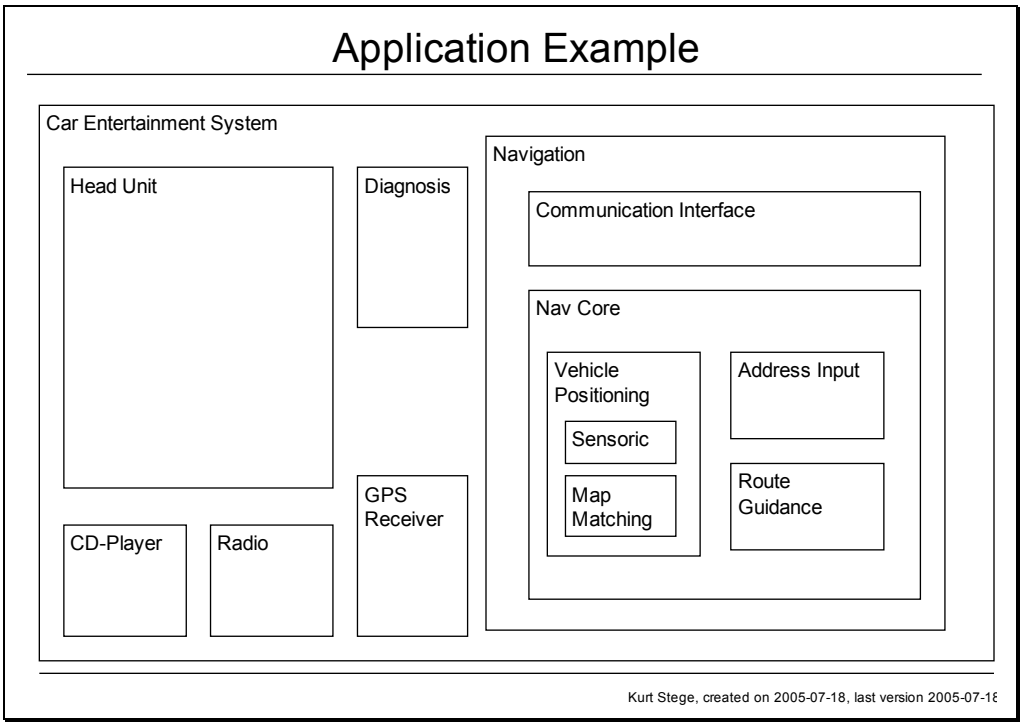

Figure 5: Compostion of a Complex System

yyy

5 Attributes (2005-08-05)

This chapter discusses the attributes of components, signals, slots and parameters. Attributes are a mean of a component designer to specify some requirements due to implementation details to the user of the component.

Each attribute is specified by the attribute name and the value, and of course the object (e.g. the signal) it refers to. Technically, the value is a string, but some attributes may interpret the value as an integer, an enumeration value, as a boolean value, or even discard it at all.

5.1 Attributes of Components

At this moment, there are no attributes for components planned to be introduced with the first release of the Sigslot concept. In later releases there might be extensions to give some informal non-technical attributes like the author, the version and the release date of the component. Furthermore it might be reasonable for a component to specify if it may be instantiated as often as required, or only once for some of its implementation relies on singletons, or even not at all for all the implementation is done as static functions.

5.2 Attributes of Signals

Name	Value	Default Value	Description	Comment
Allow delayed return	<Boolean>	false	Specifies if the component expects the signal to return “immediately” (default) or accepts the receivers to take “a long time”.	
Init	<type of init signal>	(None)	Special signals used for start-up and shut-down of the component. See chapter “Life Cycles (2005-08-08)” for details.	

5.3 Attributes of Slots

Name	Value	Default Value	Description	Comment
Returns immediately	<Boolean>	false	Specifies if the component expects possibly a long execution time of the slot (default) or if it guarantees the slot to return “immediately”.	
Init	<type of init signal>	(None)	Special slots used for start-up and shut-down of the component. See chapter “Life Cycles (2005-08-08)” for details.	

5.4 Attributes of Parameters

Attributes of parameters for signals and slots:

- Lifetime: How long is the parameter valid? Minimum is until the signal returns, maximum is forever.
- Environment: In which environment keeps the parameter valid? Same thread, other thread, other process, other processor?
- Copy Algorithms: How may the parameter be copied? Copy constructor, assignment operator, memcpy(), some other well known mean of serializing or copying (e.g. a clone() method). Please note that lifetime and environment have to be specified for the result of each copy algorithm.
- Special required handling. For example, a sender component could require that the parameter "someClass&return_value" has to be written by exactly one connected receiver, and that a slot of a receiver component declares to handle that parameter by writing a value. No harm is done when the same signal is connected to a second receiver that just logs the system but does not write to the return_value.

6 Life Cycles (2005-08-08)

This chapter discusses the start-up and shut-down of components and applications.

6.1 General Concept for Start-Up and Shut-Down

The start-up- and shut-down-concept of the Sigslot concept assumes that several components are connected to a complex structure called application. The start-up of the application is:

- Create the components
- Create the connections and cables between the components
- Start-up the components
- Declare the application as running

Shut-Down tears all these things down in reverse order:

- Declare the application as stopped
- Stop the components
- Destroy the connections and cables
- Destroy the components

Like all other complex systems, start-up is simple compared to shut-down. While starting up the system, nothing is running, and the application just can start all things in the order and still keep complete control over the whole system. Shut-down is complex for there are many components running on their own, often using own working threads. There has to be a communication between the component and the shut-down-controller for the component may not be deleted as long as it is still active.

To handle this communication, each single component can implement some special signals and slots, called init signals and init slots. A normal component does not have to implement any of these signals and slots. They are just a way provided for the component to get informed or influence the start-up- and shut-down-phases.

In general, each single component can be in any of these states:

- non-existent
- existent, but not able to send signals
- fully operable, including all connections

In addition, when all components of an application are fully operable, the application is declared as running.

The Sigslot concept assumes that simple components are used to create composite components that these composite components may be used to create further composite components and that on some level there is exactly one outermost composite component called “application”. For all innermost simple components the “application is running” signal refers to the same outermost application and not to the composite component.

The init signals and slots of a component are optional. When a component behaves in a standard way, it does not have to implement these special signals and slots. Thus, a component may declare and implement the special signals and slots as required by its implementation. They are part of the “private interface” of the component. The user of a component, that is the composite component, has to respect that private interface. The init signals and slots are designed in a way that a composite component can declare its own

init signals and slots to handle the requirements of its inner components. The code generation tool that creates the composite components handles that part automatically.

6.2 Init Signals and Slots

Signals and slots of a component may have the attribute “init” and a value that specifies the type of the init signal or init slot. All init signals and slots in common is that they are not part of the public interface of the component and just notify about the requirements of the implementation (“private interface”). The init slots are supposed to be called by the owner of the component (that is the composite component), and the init signals are supposed to be connected to some listener within the owner. It is not supposed to connect init signals or slots to any other components.

This section lists all the available types of init signals and init slots and specifies its exact semantics from the viewpoint of the component. Furthermore, a composite component is supposed to implement init signals and slots on its own to be able to handle the init signals and slots of its inner components as required. This section also explains what a typical composite component does within its own init signals and slots.

Please note that the developer creating composite components just specifies the inner components and the connection tables between these components and the public interface of the composite component. All attributes of the public signals and slots, including the creation and implementation of init signals and slots, are done automatically by the tool that generates the code for the composite components.

6.2.1 Constructor

Yes, I know, the constructor is not an init slot. But the constructor is the first thing of a component that is called during its existence. The constructor is just the normal C++ constructor of the component class and used in its ordinary way.

As most components are created at system start-up, and in most systems, start-up times are a critical element, it is supposed to make the constructor of a component as light and fast as possible. Alas, this requirement has nothing to do with the Sigslot concept. The constructor is not allowed to call any signals of the component, for the connections are not yet created.

A composite component can use the constructor to construct all its inner components, and even to create the cables and connect the components. But to create a light-weight-constructor the composite component is supposed to delay this work until its first init slot (of type “start up”) is called.

6.2.2 Start Up

After the constructor this is the first call to the component. While the constructor might already be called when starting the executable (depending of the owner of the component) this init slot is supposed to be called when the application is due to be started.

The component may use this init slot to do some internal initializations. Alas, the component has to obey two rules:

- Do not call any signals. They are not yet connected anyway. Please behave, as if other threads might access the signals to store some connections, and if the access to signals is not multi threading safe. (In most applications, the first is not true, but the second *is* true.)
- Return within a reasonable short time. Please assume that all start-up-functions of every component of the application is called one by one in one main thread. Thus, avoid complex

calculations for several seconds or even blocked waiting for some external events. Use other means (e.g. threads or timer slots) to achieve this goal.

A composite component will use this init slot to call each start-up slot of the inner components and to create the cables and connect the inner component.

6.2.3 Start Up 2 (not implemented)

This init slot will probably not be implemented. The idea is to separate the initialization of the components and the creation of connections.

The semantics of "start up" would be: "Make now your (first) initialization."

The semantics of "start up 2" would be: "From all components of the application the "start up" has been called. From no component, any working thread is running. You may complete your initialization. You may not send any signals. The connections will be created next."

6.2.4 Signal Propagator Thread

A component may implement one or more slots as signal propagator thread. The owner of the component has to create a working thread for each signal propagator thread and call these init slots from within that working thread. The slots may return any time they want, but it is normal behaviour when the slots do not return at all until the component is shutting down.

A signal propagator thread may not send signals on its own from the beginning on. It is allowed to send signals, when the sending of signals is allowed for the whole application. The component can detect this fact due to one of several events:

- The init slot "you may send signals" is called.
- The component receives any signal (that is, any ordinary slot is called).
- The first signal initiator thread is started.

Signal propagator threads may be used by the component for any reason. The name comes from a typical usage: One or more slots of the component receive signals and either stores the messages into some queues or evaluate the messages and send the results on their own. When a slot is called the component can rely on the fact that the signal propagator threads are already running. (This is not guaranteed for the signal initiator threads.) When the signal propagator thread finds some messages in the queue it can evaluate these messages and publish the results via signals. Using a signal propagator thread in this way is safe, for it cannot "initiate" the sending of signal from the void (before any signals have been received) but only "propagate" signals as a reaction on incoming signals.

TODO: Specify the behaviour when a working thread returns. Possible solutions:

- The init slot is called again immediately, except the component is about being shut down. This behaviour is the default behaviour and will be available.
- The init slot will not be called again. The underlying thread terminates instead.
- The init slot will be called again after some wake-up-event (timer event or by firing a wake-up-signal?)

TODO: Specify how the component changes between the different behaviours. Using attributes during compile time? Using a return code during run time? Possible solution: Configuration signal that is connected to the configuration slot of the used thread component.

A composite component does not use any working threads by itself. Even more, it does not create any working threads that call the component. Instead it creates a helper component of type “thread class” (see chapter 7.1 “Thread Component”) and thus delegates the system specific tasks.

6.2.5 Your Signal Propagator Threads Are Running

This init slot of a component is called, when all its signal propagator threads have been started. A component may implement this slot even when it has none signal propagator threads. The slot gets called after the threads would have been started. Like the init slot “start up” (and most other init slots) the function has to return reasonable fast.

A composite component uses this init slot to first create the signal propagator threads of its inner components and than call their init slot “your signal propagator threads are running”.

TODO: It would be possible to unify the init slots “your signal propagator threads are running” and “you might receive signals”. Decide if it is useful to keep that signals separate.

6.2.6 You Might Receive Signals

Like most other init slots, this slot has to return reasonable fast. The component can rely on the fact that none (ordinary) slot will be called until this init slot has returned. That is, even while the slot is called, the component does not receive any messages. This is guaranteed eliminate the component from the burden of synchronizing access to variables during the start up time.

The component is not yet allowed to send any signals. It just has to be prepared to receive signals from now on.

A composite component implements this init slot and just calls the init slot of its inner components. Of course, when no inner component has declared this init slot, the composite component would implement an empty loop and is supposed to omit the declaration (and implementation) of that init slot at all.

From the viewpoint of the Sigslot application a virtual switch is turned on when this init slot has been called for each single component. From now on it is allowed and possible to send and receive signals. The first action is to call “you may send signals” for each component.

6.2.7 You May Send Signals

This init slot notifies the component that it may send any signals from now on, even from within this init slot. Alas, the slot call has to return immediately, like all other init slots.

Please note that the component may receive signals by calls to ordinary slots even before this init slot is called. If that happens, the component is allowed to send signals from that moment on.

A composite component implements this init slot just to call the init slot of all its inner components. Like “you might receive signals” the composite component is encouraged to not declare that init slow when none of its inner components has declared that slot.

6.2.8 Signal Initiator Thread

A component may implement one or more slots as signal initiator thread. The owner of the component has to create a working thread for each signal initiator thread and call these init slots from within that working

thread. The slots may return any time they want, but it is normal behaviour when the slots do not return at all until the component is shutting down.

Opposed to signal propagator threads a signal initiator thread may send signals on its own from the beginning on. For further details see section 6.2.4 “Signal Propagator Thread”.

6.2.9 Your Signal Initiator Threads Are Running

This init slot of a component is called, when all its signal initiator threads have been started. A component may implement this slot even when it has none signal initiator threads. The slot gets called after the threads would have been started. Like the init slot “start up” (and most other init slots) the function has to return reasonable fast.

A composite component uses this init slot to first create the signal initiator threads of its inner components and than call their init slot “your signal initiator threads are running”.

6.2.10All Components Are Running

This init slot is called when all working threads of all components of the application are running and all init slots “your signal initiator threads are running” have been called for all components.

A composite component implements this slot, when at least one of its inner components has that init slot implemented. In that case it just calls that slot.

TODO: This init slot is not really useful. Decide if to delete this init slot or if to implement the support within the composite components.

6.2.11The Application Is Running (probably not implemented)

At this stage of the start up, when “all components are running” is called for all components, all components are running and working, but the application is still in a formal state “not running”. From now on, the components may be conflicted with three situations:

- The components keep running and active, and the application status keeps “not running”.
- The application status changes to “running” indicated by the init slot “the application is running”.
- The components are shutting down. When this begins by calling “stop sending signals”, the shut down proceeds until “all components are shut down” and cannot terminate before.

Most components are ignorant about the “application is running”-status. The semantics of “application is running” is dependent of the application. Thus, a component that implements one of the application init slots is a component specific for that application and cannot be used in other applications. A typical usage for the running status is for selected components to control or to cancel or delay the termination of the application.

TODO: All this application stuff seems to be clumsy, highly specialized for a specific purpose, and not really part of a Sigslot concept. Introduce other strategies that allow application wide communications to handle tasks like application shut down or general mode switching (e.g. switching between “record”, “play” and “stop”).

6.2.12The Application Is Shutting Down (probably not implemented)
6.2.13The Application Is Stopped (probably not implemented)
6.2.14Stop Sending Signals

When the application begins to shut down, it calls the init slot “stop sending signals” of all components.

6.2.15Shut Down
6.2.16All Components Are Shut Down
6.2.17Destructor
6.3 Sequence Diagrams

6.4 Thumb Rules or Invariants

- The owner of a component is the composite component (or application) it resides in.
- The owner of a component is responsible for construction and destruction of the component, for obeying its “private interface” (attributes and init signals and slots), for starting up and shutting down the component.
- A component publishes an “active” flag. This is false (or “inactive”) per default and may be set by the component. Whenever this flag is set, there might be an external reference to the component. Therefore, the component may not be destructed by its owner. The component is responsible to clear that flag when the system is shutting down. This “active” flag refers to external references only. References within the Sigslot application are known by the owner anyway and need not to be published by this flag. As long as there are internal references (that is within the application) to the component, e.g. cables pointing to slots of the component or even some slots that are called at the moment), the owner may not delete the component even when the active flag is cleared.
- A component publishes a “signal” flag. This flag is false per default and may be set by the component using a special init signal. This flag denotes that something outside the Sigslot-application-context might call (e.g. send) any signals of the component. As long as this flag is set, the owner of the component may not remove or change the cables connected to the signals of the component. The component is responsible to clear that flag while the system is shutting down.
- A component may send signals from within an ordinary slot and from within a signal-initiator-thread at any time. In addition, a component may send signals from its signal-propagator-threads, when the start-up has reached the appropriate step. In addition, a component may send signals from any other (external) context, when both the start-up has reached the appropriate step, and the component publishes this fact using its “signal” flag. (TODO: Handle signal-propagator-threads during shut-down.)

7 Component Library (2005-09-14)

This chapter discusses components that are provided by a component library. These components serve different goals:

- The code generation tools for composite components may use these components to regard the requirements of the components specified by attributes.
- The architect or designer of the application may use these components to change the behaviour of the application or parts of it.
- The designer of a component may partition the component into smaller sub-components and use some of the components of the library to implement part of the functions.

Some of these components (and attributes) are supposed to implement complex, error prone, or operating system specific behaviour (multi-threading, synchronizing, usage of mutexes, inter-process-communication) and thus make the hand-written components simpler, more secure, and easy re-usable.

7.1 Thread Component

The thread component, see Figure 6: Library Component "Thread", implements just one SIGNAL_thread() and a SLOT_start(). When the slot start is called, the component generates a new (operating system specific) thread and calls the signal from the context of that thread. When the signal returns, it is called immediately again, within a loop. The loop ends, when the system is shutting down.

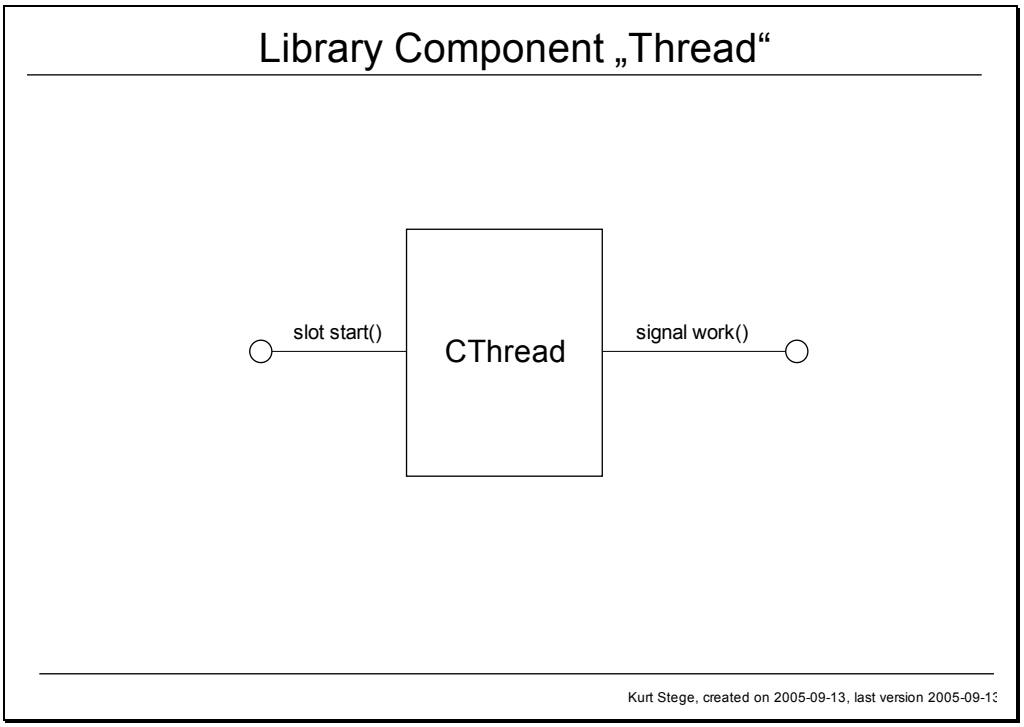

Figure 6: Library Component "Thread"

The component does not declare any slots as (signal propagator or signal initiator) thread using the terms from the Sigslot concept but implements that feature on its own using some kind of magic. Thus, the code

generation tool for composite components can use thread components to transform the composite component in a form that does not have any Sigslot threads anymore.

However, the thread component has to use the Sigslot means to declare the component as “active” as long the internal thread is existing.

Even more, this component will be used (beside other use cases) to start a signal propagator threads of another component. This happens at a time when in general the sending of signals is not yet allowed. From the viewpoint of the thread component, its slot gets called, and thus the thread component may create its thread and call its signal. The composite component that uses this component has to treat this component with special care, and not as an ordinary inner component. Therefore, the thread component is supposed to uses as few init slots as possible, and not to change the required init slots without checking the code generation tools.

7.2 Wait Component

The Wait Component as shown in Figure 7: Library Component "Wait" implements a (operating specific) blocking wait.

- SLOT_work() is called within a context that expects this slot to return only after a long time. The execution of this slot waits (in a blocking form) until a wake-up condition arises, then calls SIGNAL_work(), then returns.
- SIGNAL_work() is called from the execution of SLOT_work(), when a call to SLOT_wakeUp() has occurred. When SIGNAL_work() returns, SLOT_work() will return as well to the caller (and probably be called again immediately).
- SLOT_wakeUp() notifies the wait component to call SIGNAL_work() as soon as possible. In most times, SLOT_work() is already called and waiting. In that case, it awakes and calls SIGNAL_work() from the thread context of SLOT_work(). In all other cases (SIGNAL_work() is still running or SLOT_work() is not running) the notification is stored somewhere, and when SIGNAL_work() is called the next time, it calls SIGNAL_work() immediately without waiting for another call of SLOT_wakeUp().

Several consequent calls to SLOT_wakeUp() without actually calling SIGNAL_work() in between will result in calling SIGNAL_work() only once at all, and not once for each call to SLOT_wakeUp().

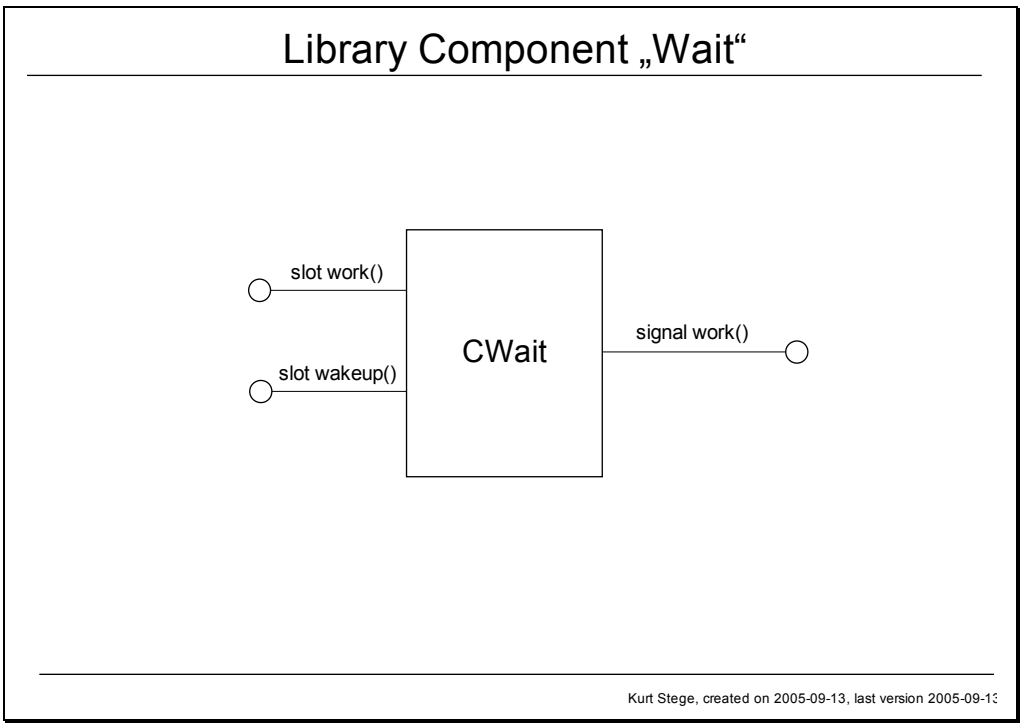

Figure 7: Library Component "Wait"

From the moment on SIGNAL_work() is called, a call to SLOT_wakeUP() will trigger another call to SIGNAL_work() later on. In fact, the moment to switch behaviour is a little bit earlier before calling SIGNAL_work().

7.3 Queue Component

The Queue Component in Figure 8: Library Component "Queue" is a component that implements a queue for objects of any data type, usually the tMessage of a given signature.

- The SLOT operator()(message) puts a new element into the queue.
- SIGNAL_notify() is triggered, whenever the reader should be called, i.e. when a new element has been put into the queue, or when the reader returns while the queue is not empty.
- SLOT_read() checks, if the queue is empty. When not, it moves one element from the queue and calls the SIGNAL_message with it.
- SIGNAL_message(message) is called to deliver a message to the receivers.

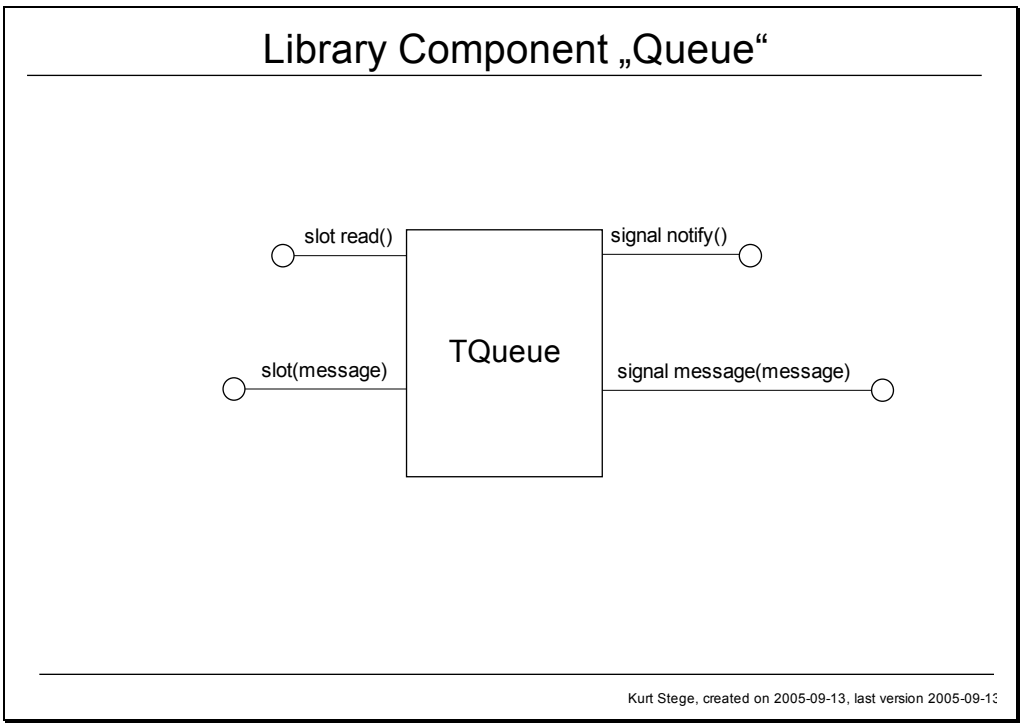

Figure 8: Library Component "Queue"

Figure 9: Asynchronous Connection demonstrates how to use a Queue Component to create asynchronous connections between a sender and a receiver. To synchronize several slots with each other (and keep the ordering of the incoming messages) the queue component may be surrounded by a dispatcher and an enpatcher.

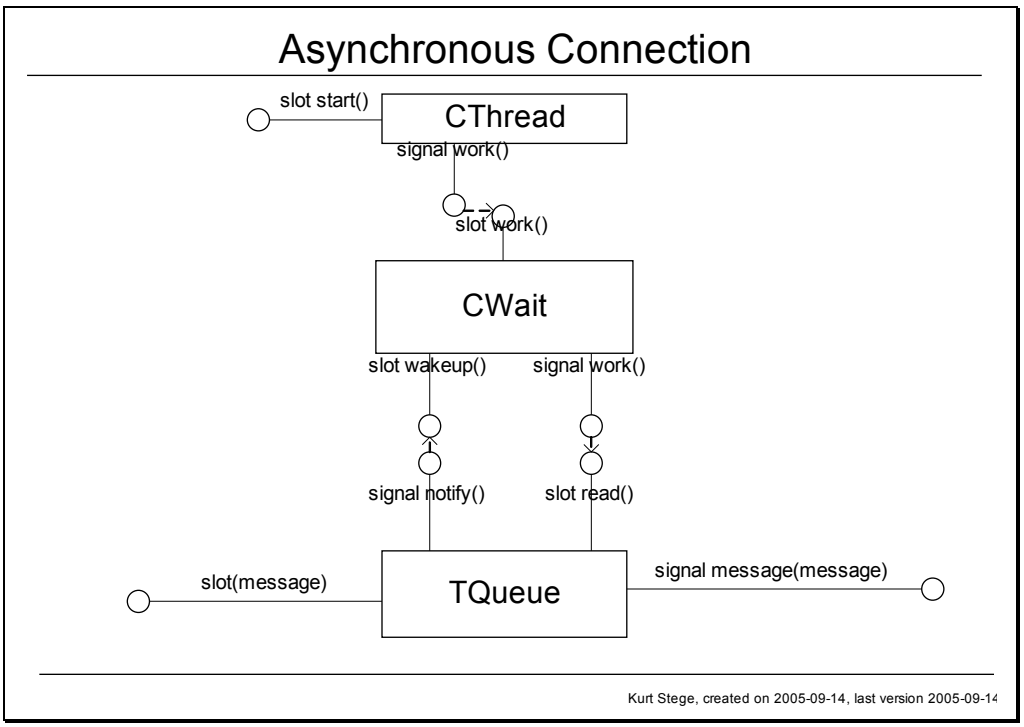

Figure 9: Asynchronous Connection

7.3.1 Message Union

A message union is just a simple structure that stores a union that holds a message of one of several possible signatures, and a flag that tags from which signature the message is stored.

For some technical reasons, namely the handling of constructors and destructors, the message union is not implemented as a union but as a full featured class. A setValue() function is used to store one of the several types, and a getValue() function retrieves the last stored value (and type).

7.3.2 Dispatcher Component

The dispatcher component has a slot that gets a message union as parameter and for each possible message a signal that is called when the incoming message union stores a message of that type.

7.3.3 Enpatcher Component

The enpatcher component (TODO: What is the opposite term for dispatcher?) has for each type of the message union a signal, stores each incoming message into a message union and delivers it using its one slot to the receivers.

7.4 Timer Component
7.5 Junction Box Component

See Figure 10: Library Component "Junction Box". Mainly used for hand-written connections that are distributed in several places. Whenever the junction box gets a message, it calls both its signals with that message and thus duplicates it.

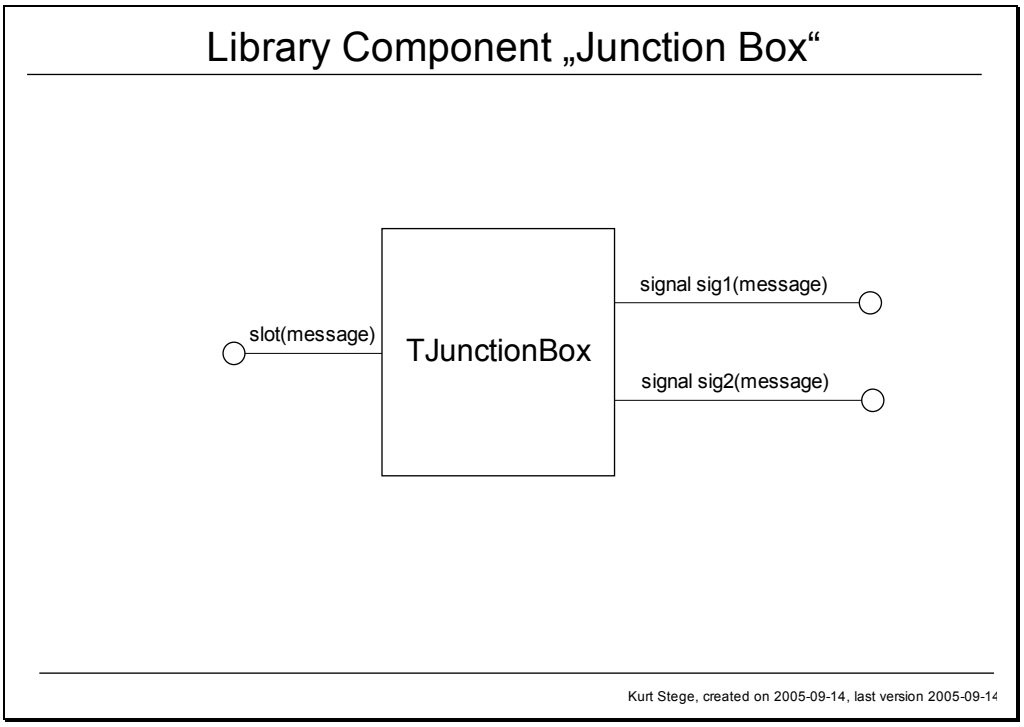

Figure 10: Library Component "Junction Box"

A signal can be connected to several slots without using a junction box, so the junction box seems useless. Alas, a signal can hold only one cable, and this cable is able to call several slots. A junction box allows several cable functions to be used with one slot.

For example, assume some ordinary library that provides a signal for call back usage. Several clients of the library would like to use that one signal, but don’t know about each other. Now, each client can provide an own junction box, move the original connection of the signal to “sig1” of the junction box, listens itself to “sig2”, and connects the slot of the junction box to the signal of the library. The junction box even has a constructor that gets the library signal and the cable function of the client as parameters and does all this plugging as described.

8 Background Stuff (2005-07-21)

This chapter discusses details about some design decisions, reasoning, motivation, and history about this Sigslot implementation.

8.1 Variants for the Implementation of a Signal Slot Concept

Most signal slot implementations are optimized for dynamic usage. When connected, both signals and slots know each other and have dynamically allocated lists of each other. Thus, both signals and slots are complex types. This has, for applications within embedded systems, some disadvantages:

- When debugging some code, it is obfuscating and a long way to see which slots are actually called when a signal fires. Using Sigslot, there are only five to ten lines of code executed between the calling of the sending signal and the calling of the receiving calls.
- Dynamic allocation of memory, as used by most signal slot implementations, are critical in embedded systems, whenever an allocation might fail due to memory shortage or memory fragmentation. The Misra-C-Rules even explicitly forbid any usage of dynamic memory allocation. When the connections are all known during compile time, it is possible to avoid each dynamic allocation. Sigslot provides the means to achieve this.
- During start-up, when all components and connections are created, ordinary signal slot implementations use many processor cycles (and program code) to create the connections. One of the problems of nowadays complex applications in embedded system is the very long start-up time. Sigslot works with precompiled connections where most of the work is already done during compile time. For each signal just one pointer has to be set during run-time at start-up, even when there are several slots connected to that signal.
- There is a memory and run-time overhead for signals, slots and connections for ordinary signal slot implementations compared to Sigslot. Therefore, the penalty for using Sigslot is small and Sigslot components may be used even for many small components or for heavy used connections.

Of course, the Sigslot concept also has many disadvantages compared to the ordinary implementations:

- When used in heavy dynamic contexts, Sigslot has to implement all that complex stuff within the connection that the ordinary implementations already implemented directly in the signal and slot classes. In that context, Sigslot looses all the above mentioned advantages and pays its flexibility using an extra static layer between the signal and some dynamic cable.
- Sigslot is proprietary and not well known compared to QT or Boost Signals.

Please don’t forget, that Sigslot is much more than just the implementation of signals and slots. What is most important for Sigslot is the concept of components and the long list of attributes the developer of a component can use to specify requirements of the component that are dependent of its implementation.

Thus, Sigslot decides to use for signals and slots some easy and small implementation. A signal calls just exactly one cable function, and a slot is even more simple, namely just an ordinary member function of the component class.

8.1.1 Variants for Slot Implementations

For slots, there are not so many different implementations discussed. The reason is that most possible implementations are rejected for the same reason.

8.1.1.1 Ordinary Member Function

When a component gets a message and wants to process that signal, it has to do it eventually using a function. Thus, when a developer implements a component class, most probably he has to implement for each slot a member function to handle the incoming signals.

That is, for the developer of a component the most easy decision is to define any ordinary member function as a (potentially) slot.

Alas, for the implementation of Sigslot, this decision is expensive. Using member functions as slots is that a slot is not a data type but only a term. Within the C++ language, there is not one data type “member function pointer” (with a given signature), but there is for each class an own distinct data type “member function pointer to a member function of class “name” with a given signature”.

As a result, signals must be able to call objects (slots) of a type not yet known when the code for the signal is compiled.

8.1.1.2 Member Variables

A slot could be an ordinary class with any implementation to be discussed. A component defines its slots as ordinary member variables of this slot type. In addition, the component class implements some function that handles the incoming signal. Most times, this “signal handler” has to access the component object (and not only the slot object within the component). In general, there are two ways to provide this:

- Either, the signal handler is implemented as a member function of the component, and thus has any access to the whole component. Than, the slot has to delegate the processing of the signal message to that signal handler and has to find the component. The slot object has to store a reference to the component that embeds the slot, and the slot has to be configured to call the right member function of the component.
- Or, the signal handler is implemented within the slot object by the developer of the component. The slot has to store a reference to the component, and the component developer has to access this reference.

Anyway, each slot object within a component has to store a reference to the component. This overhead might be worth its value, when the developer of the component has some profit. But it is not the component developer but the developer of connections and cables that might profit using this approach. Even more, the component developer has to pay with an additional line of code, for he has not only to write the member function, but to provide a slot and connect the slot to the member function in some way.

Thus, the decision for Sigslot is to use any member function as a slot without any additional means within the component.

8.1.2 Variants for Signal Implementations

This section introduces some different approaches to implement signals and connection for the Sigslot concept. All these approaches assume the decision that slots are ordinary member functions of a receiving component.

Most of these suggestions use the strategy that a signal is a small class that stores a pointer to one cable function and some context that is given to the cable function when the signal is called. In addition to the context, the cable function gets the parameters of the signal as function parameters.

When a connection is created, both a cable function and a context (specific for that cable function) is provided. That cable function has to call *all* connected components (either static or using dynamic lists), for a signal itself does not provide memory to store a complete list of receivers.

The differences for the possible signal implementations lie in the amount of context and in the kind of cable function. For the context, the scale ranges from:

- None context at all.
- A single pointer to void. This pointer may point to any type of object the cable expects.
- A pointer to void and an additional byte array (of a fixed size), used by the cable for any cable specific reasons.

The "cable function" stored in a signal can be implemented in many different ways:

- Nothing stored at all. (All signals share the same cable function and are configured using the context).
- Pointer to a global (or static) cable function.
- Pointer (or reference) to a cable object that has a (probably virtual) deliver function that is called by the signal.

Some combinations of these choices are examined closer.

TODO: Replace the current description using exact example source code to demonstrate the possible solution.

8.1.2.1 Static Cable Functions without Context

(A signal stores just one pointer to a global or static cable function and calls this function just with the signal parameters but without any additional context information. For each connection the code generator generates a static cable function that calls all the connected slots. The problem is: The cable function has no way to find the connected receivers but using some global (singleton) pointers to the receivers. Most times, this is OK and not a really draw back. But for complex systems, where composite components that include cables and connections are used in several instances, the cables need some kind of context to decide which instance of the receiver is to be called.)

8.1.2.2 Static Cable Functions with Basic Context

(The signal stores a pointer to a static cable function. The context is just single void pointer. Some cable functions don't use that context. Most cable functions use that pointer either as an explicit this pointer to the cable object or as a pointer to some destination object or function.)

8.1.2.3 Static Cable Functions with Full Context

(The signal stores a pointer to a static cable function. The context has a void pointer and a byte array of some 4 byte. All the cables using only a basic context are possible without any changes. In addition, new cable types can be introduced that stores even more info in the context. Most useful are cables that store a pointer to the destination object in the void pointer and a pointer to the member function in the byte array. See [Hickey1994] for details. Problem is that the data type of the member function pointer is dependent of the class the pointer is pointing to. The pointer has to be copied with memcpy() into the byte array. That is ugly and dangerous. Even more, in theory, the compiler may choose the size of that pointer different for each destination class. Therefore, it is difficult to choose a reasonable size of the byte array for this

purpose. In practice, a member function pointer has a size of 4 bytes like a normal pointer (for 32-Bit-machines). Alas some compilers use 8 bytes to store member function pointers when DLLs are used.)

8.1.2.4 Abstract Cable Class

9 TODO

9.1 Documentation

This section discusses issues that have to be done within the development of the Sigslot concept, mainly within this document.

- Some more introductions.
- Usage of Sigslot for static systems.
- Usage of Sigslot for dynamic systems.
- Code Generation Tools.
- Attributes (“private Interface”).
- Discuss hand-wired code versus generated code.
- Discuss components and other support provided for a construction kit (“Baukastensystem”).

9.2 Implementation

This section lists the work that has to be done within the implementation.

- Introduce Eclipse JET technology.
- Create the Sigslot/TSignature.hpp header file.
- Create some demo applications.
- Implement some cable classes, especially a junction box and/or a cable to a member function that allocates storage for the member function pointer within some dynamically allocated cable object.
- Create test suites.
- Create template classes for some connections and cables.
- Create template components (e.g. TJunctionBox) used both by hand-written code and generated code.

10 Glossary (2005-07-18)

This chapter defines most important and important terms used by the Sigslot concept. The most important terms are written in bold face. Most of these terms are not new invented words but terms taken from a more general environment. Whenever it would not be clear from the context that the special Sigslot term is mentioned, it might be prefixed with "Sigslot" to specify the context.

Term	Definition
Application	Components, even composite components, are following the basic Sigslot paradigm and thus they are not really simple to use. An application, or Sigslot application, is (or uses) a composite component and provides an additional interface to support an easy start-up and shut-down of the component.
Attribute	Attributes may be used for components, signals, slots and parameter of signals and slots to specify properties of the implementation of the components. See chapter "TODO" or "private interface" for details.
Basic Sigslot Paradigm	"Make it as easy as possible to create and implement components, even when this makes them difficult to use."
Cable	A cable is used to connect signals with slots. See "connection". The tool that generates the source code for a composite component implements and uses the cables as well.
Cable Function	A cable function is a static function provided by the cable that is called by the signal and calls all connected slots. To provide as little overhead as necessary for static systems, a signal provides enough local memory to store up to one single cable function. For automatically generated static systems that cable function just lists all the receiving slots and calls them. For dynamically created connections the cable function has to provide support to dynamic connect and disconnect the components.
Component	A module (or part of the complete system). A component has a Sigslot interface that is defined by its signals and slots. Components that are not purely academic may implement or use other (non-Sigslot) interfaces as well. Of course, this might reduce the chances for a re-usage of that component. A Sigslot component may come in any size and complexity. A very tiny component could be a class written in ten lines of code as part of a source code file. Components are scalable and may be put together from smaller components and get as large as the most complex application you can imagine. See also: simple component, composite component, application.
Composite Component	A composite component is a component that is created by putting together some other (smaller) components and the connections between these inner components. The inner components may be simple components, or composite components. Typically, a composite component has (public) signals and slots of its own, that are "mirrors" or "proxies" of signals and slots of some of the inner

	components. There is a code generator that creates composite components out of a description that lists the inner components, the (public) signals and slots and the connections between the components. Thus, a composite component does never implement functions on its own; it always just uses and connects already implemented (inner) components.
Connection	A connection between components connects a signal of one component with a slot of another component. (In some cases, sender and receiving component are identical.) A signal may have connections to several slots, and a slot may be connected to several signals. A signal may be connected to any slot with a matching signature. The signatures don't have to be exactly identical, it is sufficient when the implicit type conversions done by the compiler for an ordinary function call makes them compatible. Connections are implemented by cables.
Init signal	A signal with the attribute "init" of a component used to control its start-up and shut-down. See chapter 6 "Life Cycles (2005-08-08)" for more details.
Init slot	A slot with the attribute "init" of a component used to control its start-up and shut-down. A most typical init slot is a slot declared as working thread that is called just once at the start-up time and supposed to return during the shut-down. See chapter 6 "Life Cycles (2005-08-08)" for more details.
Interface	The (public) Sigslot interface of a component is specified by its signals and slots. This public interface is specified by the architect of the application and tells which components might be connected at all. In addition, the Sigslot concept provides a private interface that uses "attributes" to tell about some implementation details or some requirements to the user of the component that come from this implementation of the component. This private interface is provided by the developer of the component and not by the architect. Of course, the architect might give some "Vorgaben" to the developer. See chapter 5 "Attributes (2005-08-05)" for a detailed discussion of the private Sigslot interface of a component. Some attributes move complete signals or slots to the private interface of a component. Especially the "init" attribute allows a component to control start-up and shut-down of the component and the whole application. This is discussed in detail in chapter 6 "Life Cycles (2005-08-08)".
Message	The parameters similar to the parameters given to an ordinary function call that are transmitted by a signal and received by a slot.
Private Interface	See "interface". The private interface of a component specifies requirements of the implementation of the component and is declared using special attributes within the interface definition of the component.
Signal	The term "signal" is used in two situations. First, it is used as a synonym for "message", that is the actual data transmitted

	by a signal-slot-connection. Second it is the technical implementation for a component to send this message. This sender is an instance of the class CSignal and is part (member variable) of the component. Sending a signal is just like calling an ordinary function that returns void. The class CSignal provides the operator() to emit the signal (or message).
Signature	The signature of a signal or slot is, just like the signature of an ordinary function, specified by the number and types of parameters. The C++ implementation of Sigslot uses a template class TSignature<> that gets the n parameter types as template parameters. This template provides classes like CSignal for signals, slots, messages, cables etc.
Sigslot	Abbreviation for “signal slot”. A concept introduced by this document to partition complex software systems into small reusable components that are supposed to be most easy re-usable. Whenever a special term of the Sigslot concept might be confused with the general term, it might be prefixed with “Sigslot”.
Simple Component	A simple component is a (small) component, typically written by hand, that is not created out of some even smaller components.
Slot	A slot is the receiver of a signal-slot-connection. From the viewpoint of the technical solution, a slot is just an ordinary member function of the receiving component.

11 Index

12 References (2005-07-18)

12.1 Internal References

[Sigslot2005]: Kurt Stege
"The Sigslot Concept"
http://www.goto.onlinehome.de/sigslot/index.htm
This paper, enriched with source code, presentation material, implementations, and examples.

12.2 External References

[Boost]: "The Boost C++ Libraries"
http://www.boost.org
A collection of well known libraries that are candidates to be integrated into future standards of the C++ language.

[Gregor2004]: Douglas Gregor
"Boost.Signals"
http://www.boost.org/doc/html/signals.html
This article discusses the signal implementation that is part of the Boost libraries.

[Popma2004]: Remko Popma
"JET Tutorial"
http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html
http://www.eclipse.org/articles/Article-JET/jet_tutorial2.html
Tutorial for the usage of Java Emitter Templates (JET) in the Eclipse Modeling Framework (EMF).

[KDE]

[Hickey1994]: Rich Hickey
"Callbacks in C++ Using Template Functors"
http://www.tutok.sk/fastgl/callback.html
This old article describes a technical solution to call any member function from a signal.

[Thompson2002]: Sarah Thompson
"sigslot – C++ Signal/Slot Library"
http://sigslot.sourceforge.net/sigslot.pdf
http://sigslot.sourceforge.net/
A typical signal-slot implementation using C++ templates.

[TTL2005]: Eugene
"Tiny Template Library"
http://sourceforge.net/projects/tinytl
Template libraries that include a lightweight signal slot implementation similar to the boost technology.

[QT]

13 History

(ks) = Kurt Stege, kurt-stege@online.de

2005-07-14 (ks): Document created.
2005-09-14 (ks): current version; document is still in development.