Midge Documentation

Release v3.9.2

Project 8 Collaboration

Jul 07, 2021

Contents

Contents:

CHAPTER 1

Signals and Slots

1.1 Introduction

The signal/slot pattern allows function calls to created and destroyed at runtime. The signal is the caller, and the slot is the callee. A signal can call multiple slots and a slot can be connected to multiple signals.

Signals and slots in Midge are intended to allow nodes to talk to one another without needing to use the strict and limited capabilities of the data buffers. For example, signals can be added to nodes to pass useful information downstream for initialization.

Just like any function call, signals and slots have a particular function signature. Any arguments can be used, but the return type has to be `void`. The signature of the signal must match the signature of the attached slots.

A signal/slot function call is created by "connecting" the slot to the signal. That connection can later be disconnected.

Signal and slot objects in Midge can optionally have an owner. This is used to allow the signals and slots to register themselves with the owner.

1.2 Quick Start

1. Include file: `signal_slot.hh`
2. Create a slot. This example uses a lambda function that prints out a string and an int:

```
m_slot< std::string, int > lambda_slot( "", [](std::string arg1, int arg2) {
    std::cout << "(lambda) " << arg1 << " " << arg2 << std::endl;
} );
```

3. Create a signal. This example has the same signature as the slot above:

```
m_signal< std::string, int > the_signal( "" );
```

4. Connect the signal to the slot:

```
int connection = the_signal.connect( &lambda_slot );
```

5. Emit the signal to call the slot:

```
the_signal.emit( "The answer:", 42 );
```

6. Disconnect the slot from the signal:

```
the_signal.disconnect( connection );
```

1.3 The Rules

1.4 Use in a Node

When using signals and slots in a node, they're added as member variables of the node. They're used in the `owner` mode so that the node knows about all of its signals and slots, and so that that they can be connected at runtime via the diptera object.

1. Add the signal or slot object to your node as a private or protected member variable. Use classes `m_signal` and `m_slot`, whose template arguments are the arguments of the signal/slot function call.

   Signal example:

   ```
   m_signal< std::string, int > f_my_signal;
   ```

   Slot example:

   ```
   m_slot< std::string, int > f_my_slot;
   ```

2. Initialize the signal or slot in the node constructor. Be sure to use one of the "owner" constructors.

   Signal example:

   ```
   f_my_signal( "my-sig", this );
   ```

   Slot example:

   ```
   f_my_slot( "my-slot", this, &my_class::print_values, this );
   ```

   In both examples, the last argument `this` is the owner of the signal/slot. The pointer is passed to the constructor to allow it to register itself. The first argument in both examples is the name of the signal/slot, which is passed to the owner for registration.

   In the slot example, the third argument is the member function that will be called as the slot, and the second argument (also `this`) is the object on which the member function is called.

3. In the object with the signal, use `m_signal::emit()` to emit the signal.

1.5 Details

1.5.1 Signals

Non-owned-signal constructor:

```
m_signal( const string_t& p_name );
```

Owned-signal constructor:

```
template< typename x_owner >
m_signal( const string_t& p_name, x_owner* p_owner );
```

Copy and move constructors are not available.

Ownership call (owner must have this function):

```
if( p_owner ) p_owner->signal_ptr( this, p_name );
```

Connect using a slot object:

```
unsigned connect( slot* p_slot );
```

Connect a slot function without using slot object:

```
template< typename T >
unsigned connect_function( T *inst, void (T::*func)( x_args... ) );

template< typename T >
unsigned connect_function( T *inst, void (T::*func)( x_args... ) const );

unsigned connect_function( const std::function< signature > & slot ) const;
```

Connection functions all return the connection ID, which can be used to disconnect the particular connection.

Disconnect slots:

```
void disconnect( unsigned id ) const;

void disconnect_all() const;
```

Emit the signal:

```
void emit( x_args... args );

void operator()( x_args... args );
```

1.5.2 Slots

Non-owned-slot constructors:

```
m_slot( const string_t& name, const std::function< signature >& sig );

template< typename T >
m_slot( const string_t& name,  T *inst, void (T::*func)( x_args... ) );

template< typename T >
m_slot( const string_t& name,  T *inst, void (T::*func)( x_args... ) const );
```

Owned-slot constructors:

```
template< typename x_owner >
m_slot( const string_t& name, const std::function< signature >& sig, x_owner* owner );

template< typename T, typename x_owner >
m_slot( const string_t& name,  T *inst, void (T::*func)( x_args... ), x_owner* owner␣
↪);

template< typename T, typename x_owner >
m_slot( const string_t& name,  T *inst, void (T::*func)( x_args... ) const, x_owner*␣
↪owner );
```

Copy and move constructors are not available.

Ownership call (owner must have this function):

```
if( p_owner ) p_owner->slot_ptr( this, p_name );
```

Disconnect this slot from connected signals:

```
void disconnect_all();
```

1.6 Thread Safety

Thread safety is not guaranteed by the Midge library, and is therefore up to the user to ensure. In particular, be careful that signals and slots are not being (dis)connected while signals are being emitted, and slot functions themselves should be thread safe in a multi-threaded environment.

Full Doxygen API Reference