# Dynamic Signals and Slots

**by Eskil Abrahamsen Blomfeldt**

**Signals and slots are declared at compile-time, and normally you cannot add new signals and slots to a meta-object at run-time. In some situations, it is useful to extend a meta-object while an application is running to obtain truly dynamic function invocation and introspection.**

The need for the dynamic addition of signals and slots arises when binding a script interpreter, such as Qt Script for Applications (QSA), to the C++ code in your application. In QSA, it is possible to use Qt's meta-object system to communicate between objects created in the C++ code and objects created in a script. Qt 4's meta-object system makes this sort of extension possible, with a little bit of hackery.

This article will get you started writing a dynamic signals and slots binding layer for Qt 4. We will review an abstract base class, `DynamicQObject`, that offers the required functionality. The "miniframework" presented here can serve as a foundation for doing more advanced extensions of Qt's meta-object system.

**[Download source code]**

## Expanding the Q_OBJECT Macro

Any QObject subclass that declares signals and slots must contain the Q_OBJECT macro in its definition. This is a magic macro that expands to a set of private declarations:

```
// reimplemented from QObject
const QMetaObject *metaObject() const;
void *qt_metacast(const char *className);
int qt_metacall(QMetaObject::Call call, int id,
                void **arguments);

// static members
static QString tr(const char *sourceText,
                  const char *comment = 0);
static QString trUtf8(const char *sourceText,
                      const char *comment = 0);
static const QMetaObject staticMetaObject;
```

Q_OBJECT is "magic" because qmake and moc will recognize it and automagically generate code for the class's meta-object during compilation of the project.

We want to write a `DynamicQObject` class that mimics part of the behavior of the `moc` but that does not limit us to statically declared signals and slots. Since we need to prevent the `moc` from compiling our class, we can't use the Q_OBJECT macro; instead, we will copy its definition and insert it into our code.

The Q_OBJECT macro is defined in `src/corelib/kernel/qobjectdefs.h` (which is included by

<QObject>). It declares a number of functions that must be reimplemented to fully support Qt's meta-object system. For the purposes of this article, it will be sufficient to reimplement just one of these functions:

```
int qt_metacall(QMetaObject::Call call, int id,
                void **arguments);
```

When a signal is emitted, Qt uses `qt_metacall()` to invoke the slots connected to the signal. The first parameter, `call`, is then set to QMetaObject::InvokeMetaMethod. (The `qt_metacall()` function is also used for other types of access to the meta-object, such as setting or getting properties.)

The second parameter is an index that uniquely identifies a signal or slot in the hierarchy of classes inherited by the current object. The last parameter is an array that contains pointers to arguments, preceded by a pointer to the location where the return value should be placed (if any).

## The DynamicQObject Class Definition

Here's the definition of the `DynamicQObject` class:

```
class DynamicQObject : public QObject
{
public:
    int qt_metacall(QMetaObject::Call call, int id,
                    void **arguments);
    bool emitDynamicSignal(char *signal);
    bool connectDynamicSlot(QObject *obj, char *signal,
                            char *slot);
    bool connectDynamicSignal(char *signal, QObject *obj,
                              char *slot);

    virtual DynamicSlot *createSlot(char *slot) = 0;

private:
    QHash<QByteArray, int> slotIndices;
    QList<DynamicSlot *> slotList;
    QHash<QByteArray, int> signalIndices;
};
```

The intended usage pattern of `DynamicQObject` involves subclassing it and adding some functionality. Whenever we add a slot to an object, we would expect this slot to execute code when it is called. Since this part of the mechanism is dependent on the language to which we want to bind, it must be implemented in a subclass.

The pure virtual function `createSlot()` returns an instance of an appropriate subclass of the abstract class `DynamicSlot` that handles function invocation for your specific language binding. `DynamicSlot` is defined as follows:

```
    class DynamicSlot
    {
    public:
        virtual void call(void **arguments) = 0;
    };
```

`DynamicSlot` subclasses must reimplement the `call()` function to handle a slot invocation. The `arguments` array's first entry is a pointer to a location where we can put the return value. This pointer is null if the signal's return type is `void`.

The rest of the array contains pointers to arguments passed to the slot, in order of declaration. We must cast these pointers to the correct data types before we can use them. For example, if the slot expects a parameter of type `int` as its first parameter, the following code would be safe:

```
    int *ptr = reinterpret_cast<int *>(arguments[1]);
    qDebug() << "My first parameter is" << *ptr;
```

## Implementation of connectDynamicSlot()

The `connectDynamicSlot()` function connects a static signal declared in an arbitrary QObject instance to a dynamic slot in the current `DynamicQObject` instance. The function takes a pointer to the sender object (the receiver is always `this`) and the signatures of the signal and slot. The function returns `false` on error.

```
bool DynamicQObject::connectDynamicSlot(QObject *obj,
                                        char *signal,
                                        char *slot)
{
    QByteArray theSignal =
        QMetaObject::normalizedSignature(signal);
    QByteArray theSlot =
        QMetaObject::normalizedSignature(slot);
    if (!QMetaObject::checkConnectArgs(theSignal,
                                       theSlot))
        return false;
```

The first step is to normalize the signatures passed to the function. This will remove meaningless whitespace in the signatures, making it possible for us to recognize them. Then, we check whether the signatures are compatible, to avoid crashes later.

```
  int signalId =
            obj->metaObject()->indexOfSignal(theSignal);
    if (signalId == -1)
        return false;
```

If the argument lists are compatible for the two functions, we can start resolving the signal and slot indexes. The signal is resolved using introspection into the provided QObject object. If the signal was not found, we return `false`.

Resolving the (dynamic) slot is a bit more work:

```
        int slotId = slotIndices.value(theSlot, -1);
          if (slotId == -1) {
              slotId = slotList.size();
              slotIndices[theSlot] = slotId;
              slotList.append(createSlot(theSlot));
          }
```

We check whether the slot signature has been registered before in the same `DynamicQObject` instance. If it hasn't, we add it to our list of dynamic slots.

We use a QHash<QByteArray, `int>` called `slotIndices` and a QList<DynamicSlot `*>` called `slotList` to store the information needed for the dynamic slots.

```
        QMetaObject::connect(obj, signalId, this,
                 slotId + QObject::metaObject()->methodCount());
    }
```

Finally, we register the connection with Qt's meta-object system. When we notify Qt about the connection, we must add the number of methods inherited from QObject to our `slotId`.

The `connectDynamicSignal()` function is very similar to `connectDynamicSlot()`, so we won't review it here.

## Implementation of qt_metacall()

Qt's meta-object system uses the `qt_metacall()` function to access the meta-information for a particular QObject object (its signals, slots, properties, etc.).

```
    int DynamicQObject::qt_metacall(QMetaObject::Call call,
                                    int id, void **arguments)
    {
        id = QObject::qt_metacall(call, id, arguments);
        if (id == -1 || call != QMetaObject::InvokeMetaMethod)
```

```
            return id;

        Q_ASSERT(id < slotList.size());
        slotList[id]->call(arguments);
        return -1;
    }
```

Since the call may indicate access to the meta-object of the QObject base class, we must start by calling QObject's implementation of the function. This call will return a new identifier for the slot, indexing the methods in the upper part of the current object's class hierarchy (the current class and its subclasses).

If the QObject::qt_metacall() call returns -1, this means that the metacall has been handled by QObject and that there is nothing to do. In that case, we return immediately. Similarly, if the metacall isn't a slot invocation, we follow QObject's convention and return an identifier that can be handled by a subclass.

In the case of a slot invocation, the identifier must be a valid identifier for the `DynamicQObject` object. We don't allow subclasses to declare their own signals and slots.

If all goes well, we invoke the specified slot (using `DynamicSlot::call()`) and return -1 to indicate that the metacall has been processed.

## Implementation of emitDynamicSignal()

The last function we need to review is `emitDynamicSignal()`, which takes a dynamic signal signature and an array of arguments. The function normalizes the signature then emits the signal, invoking any slots connected to it.

```
bool DynamicQObject::emitDynamicSignal(char *signal,
                                       void **arguments)
{
    QByteArray theSignal =
            QMetaObject::normalizedSignature(signal);
    int signalId = signals.value(theSignal, -1);
    if (signalId >= 0) {
        QMetaObject::activate(this, metaObject(),
                              signalId, arguments);
        return true;
    } else {
        return false;
    }
}
```

The arguments array can be assumed to contain valid pointers. A simple example of how to do this can be found in the accompanying example's source code, but as a general rule, QMetaType::construct() is your friend when you are doing more advanced bindings.

If the (dynamic) signal exists (i.e., it has been connected to a slot), the function simply uses QMetaObject::activate() to tell Qt's meta-object system that a signal is being emitted. It returns `true` if the signal has previously been connected to a slot; otherwise it returns `false`.

## Pitfalls and Pointers

Having explained the architecture of this simple framework, I'd like to conclude by explaining what the framework *doesn't* do and how you can expand on it or improve it to suit your needs.

- Only a subset of the features in Qt's meta-object system is supported by `DynamicQObject`. Ideally, we would reimplement all the functions declared by the Q_OBJECT macro.
- Declaring new signals and slots in `DynamicQObject` subclasses will cause an application to crash. The reason for this is that the

meta-object system expects the number of methods declared in a meta-object to be static. A safer workaround would be to use a different pattern than inheritance to employ dynamic signals and slots.

- The framework is sensitive to misspellings. For instance, if you try to connect two distinct signals to the same dynamic slot and misspell the signature of the slot in one of the connect calls, the framework will create two separate slots instead of indicating a failure. To handle this in a safer way, you could implement a `register()` function and require the user to register each signal and slot before connecting to them.
- The framework cannot handle connections where both the signals and the slot are dynamic. This can easily be written as an extension of the `DynamicQObject` class.
- Removing connections is not possible with the current framework. This can easily be implemented in a similar way to establishing connections. Make sure not to change the order of entries in `slotList`, because the list is indexed by slot ID.
- In real-world applications, you may need to do type conversions before passing parameters between functions declared in scripts and functions declared in C++. The `DynamicQObject` class can be expanded to handle this in a general way by declaring a set of virtual functions that can be reimplemented for different bindings.