QAsync: Asynchronous Functions for Qt

Henrik Hedberg

Asynchronous functions

- Synchronous functions run in the same thread as the caller
- Usually asynchronous functions return data to the caller by means of a callback function
  - Required typically with I/O operations, for example
- Here, by asynchronous function we mean
  - a function that syntactically looks similar to its synchronous counterpart but does not block the application execution
  - It allows the current thread to continue other activities, such as to go back to the event loop (yield)
  - The execution context is stored, and is restored later when the asynchronous method returns (resume)

Motivational example: introduction

- A TCP server
  - Accepts multiple connections
  - Writes all received characters into the console

```
class Server : public QTcpServer
{
    Q_OBJECT
public:
    Server(QObject *parent = 0);
    virtual ~Server() { }
private slots:
    void handleConnection();
    void handleRead();
};
```

```
int main(int argc, char** argv)
{
    QApplication application(argc,
                             argv);
    Server server;
    application.exec();
}
```

Motivational example: with plain Qt

- The connection handling must be split into two functions
  - read() causes application to block when no data available
  - We have to use a signal to wait the data to be received first

```
void Server::handleConnection()
{
    QTcpSocket *socket = nextPendingConnection();
    connect(socket, SIGNAL(readyRead()), SLOT(handleRead()));
}
void Server::handleRead()
{
    char buffer[1024];
    buffer[qobject_cast(QObject::sender())->read(buffer, 1023)] = 0;
    std::cout << buffer;
}
```

Motivational example: would be nicer?

- The flow of activities are expressed more naturally
  - For every new connection, a socket is taken and it is read forever
  - No need to implement state machines and separate functions

```
void Server::handleConnection()
{
    QTcpSocket *socket = nextPendingConnection();
    forever {
        char buffer[1024];
        buffer[socket->read(buffer, 1023)] = 0;
        std::cout << buffer;
    }
}
```

Motivational example: the challenge!

- QIODevice::read() is synchronous
  - The thread gets blocked when a connection is waiting for new data
  - The application is not returning into the event loop either

```
void Server::handleConnection()
{
    QTcpSocket *socket = nextPendingConnection();
    forever {
        char buffer[1024];
        buffer[socket->read(buffer, 1023)] = 0;
        std::cout << buffer;
    }
}
```

QAsync

- Asynchronous signal wait
  - Built on top of existing signals and slots mechanism
- Coroutines
  - Current prototype: every activated slot that has empty parameter list
  - Future plan: introduce a new keyword *async* and implement it into Meta-Object Compiler (moc)

Asynchronous signal wait

- A new class *QAsync* has the following functionality
  - int connect(const QObject *sender, const char *signal);
  - bool disconnect(const QObject *sender, const char *signal);
  - QPointer<QSignal> waitAsync();
  - static QPointer<QSignal> waitAsync(const QObject *sender,
    const char *signal);
- The new class *QSignal* provides the following functionality
  - int id() const;
  - template <typename Arg1> void arguments(Arg1 *arg1) const;
  - template <typename Arg1, typename Arg2>
    void arguments(Arg1 *arg1, Arg2 *arg2); ... // up to 10 arguments

Example: wait data to be ready

- Add *QAsync::waitAsync()* to perform asynchronous wait
  - Suspends the execution and returns to the event loop
  - The execution context is restored the *socket* emits *readyRead()*

```
void Server::handleConnection()
{
    QTcpSocket *socket = nextPendingConnection();
    forever {
        char buffer[1024];
        if (socket->bytesAvailable() == 0)
            QAsync::waitAsync(socket, SIGNAL(readyRead()));
        buffer[socket->read(buffer, 1023)] = 0;
        std::cout << buffer;
    }
}
```

Example: craft an utility function

- Asynchronous counterpart to *QIODevice::read()*

```
qint64 QIODevice::readAsyc(char *data, qint64 maxSize)
{
    if (bytesAvailable() == 0)
        QAsync::waitAsync(this, SIGNAL(readyRead()));
    return read(data, maxSize);
}
void Server::handleConnection()
{
    QTcpSocket *socket = nextPendingConnection();
    forever {
        char buffer[1024];
        buffer[socket->readAsync(buffer, 1023)] = 0;
        std::cout << buffer;
    }
}
```

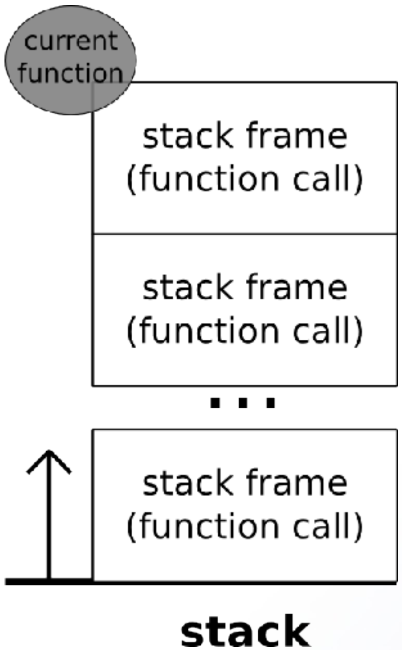
The execution of an async function

- An async function “returns” when it starts to wait a signal
- The function is “restarted” when the signal is emitted

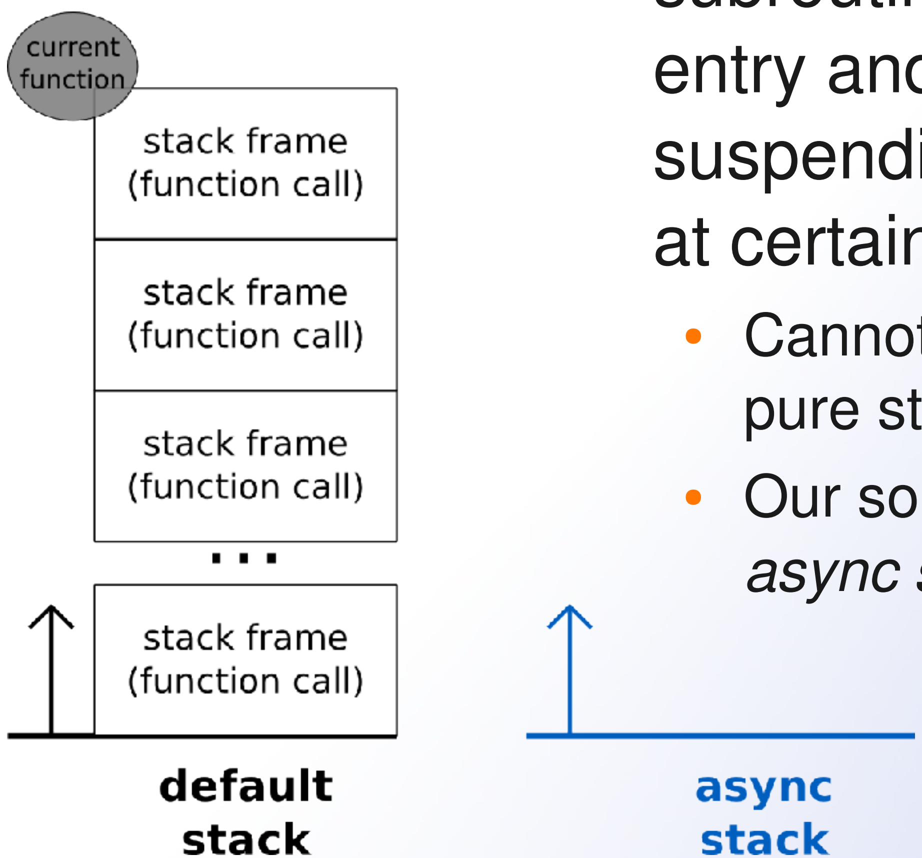
```
void Server::handleConnection()
{
    QTcpSocket *socket = nextPendingConnection();
    forever {
        char buffer[1024];
        buffer[socket->readAsync(buffer, 1023)] = 0;
        std::cout << buffer;
    }
}
```

Function calls utilises the stack

```
value = function(argument);
```

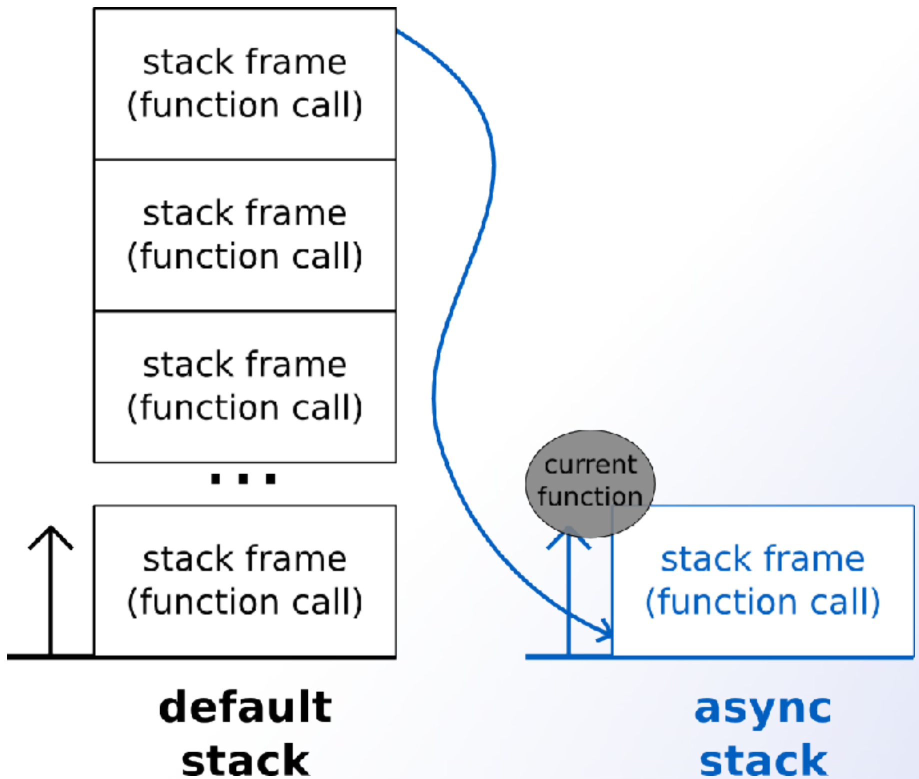


Async slots are coroutines



- Coroutines are generalisations of subroutines to allow multiple entry and exits points for suspending and resuming execution at certain locations
  - Cannot be implemented effectively with pure stack-based solution
  - Our solution introduces a separate *async stack* for asynchronous slots

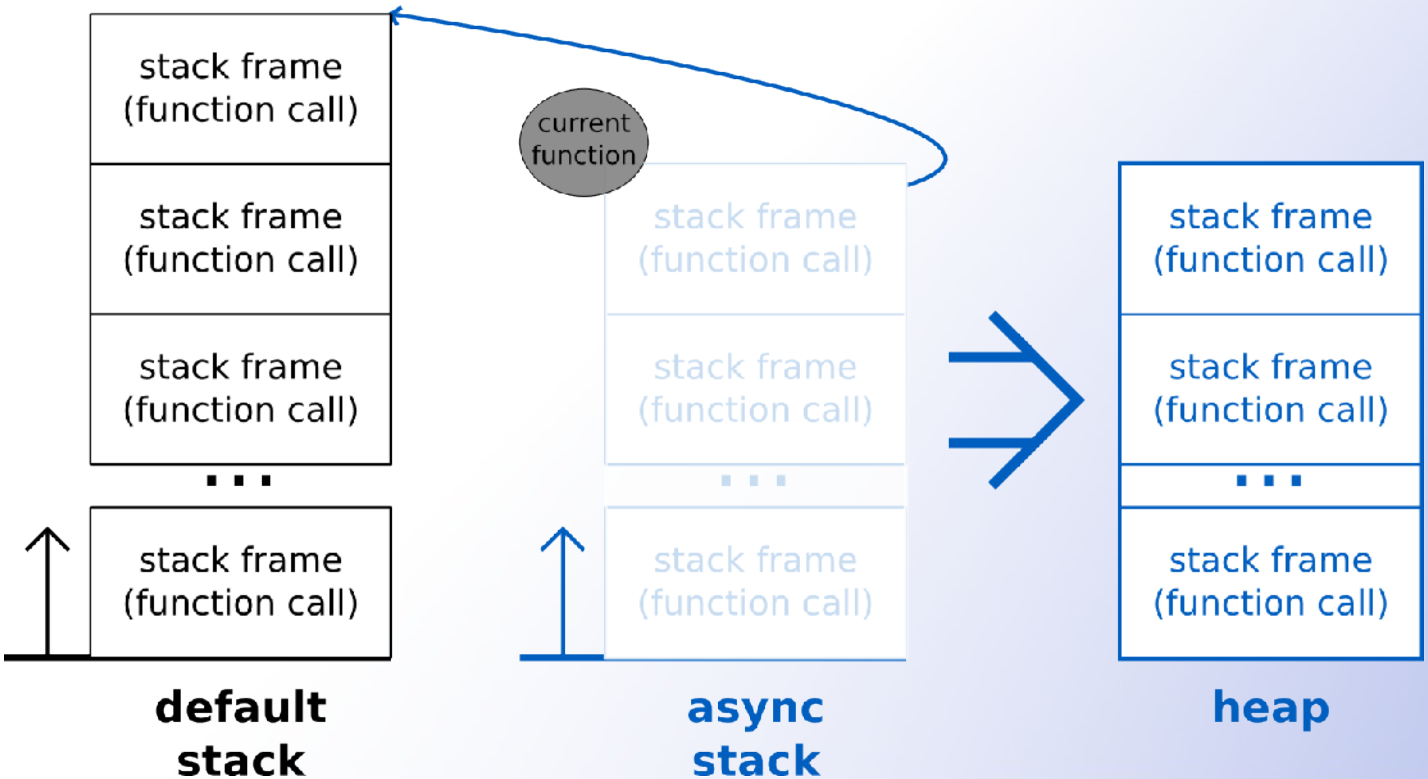
Activating an async slot




```
class Example : public QObject
{
...
public slots:
    void slotAsync();
};

void Example::slotAsync()
{
...
}

connect(someObject, signal(),
        example, slotAsync());

someObject: emit signal();
```

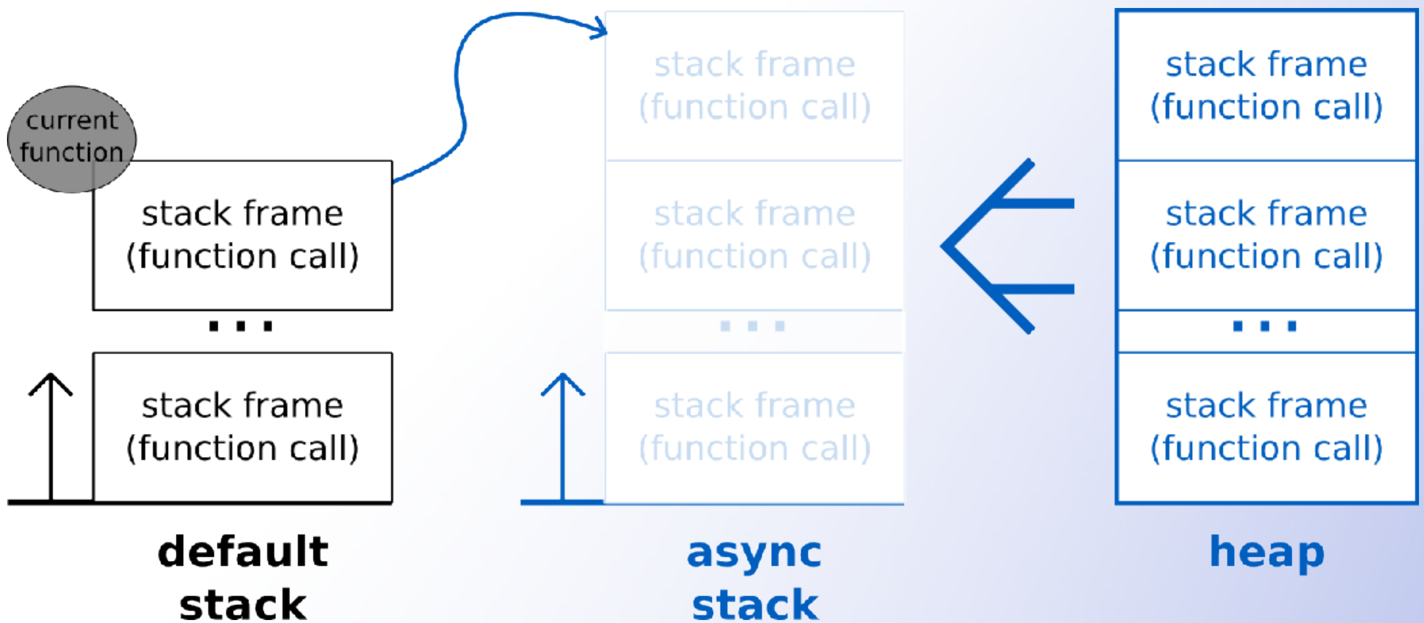
Suspending an async slot

```
QAsync::waitAsync(otherObject,
                  Q_SIGNAL(otherSignal(void*)));
```

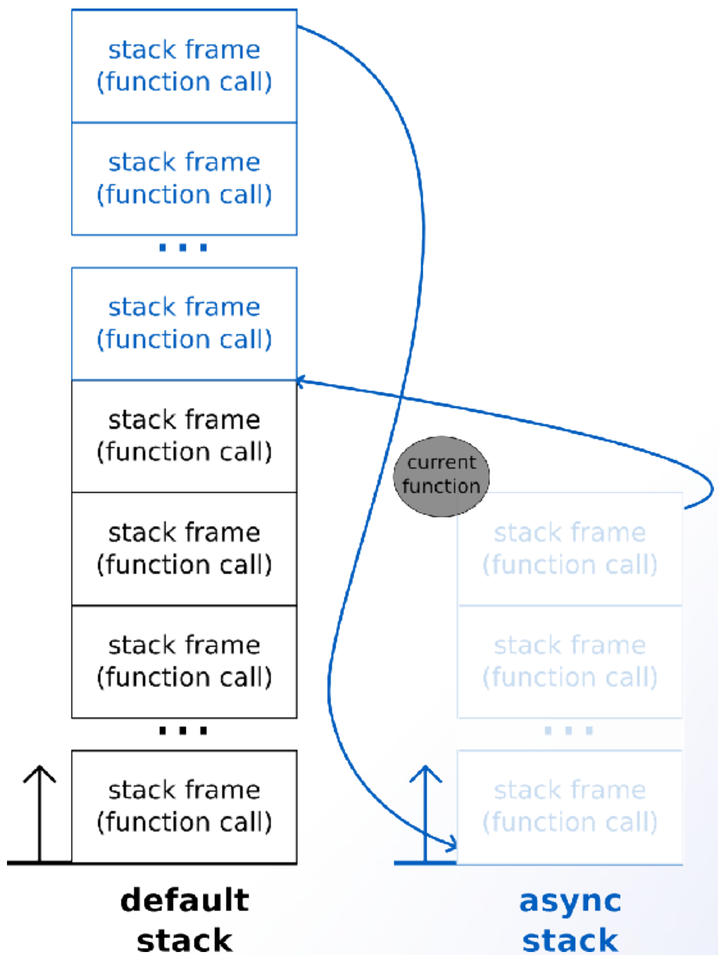


Resuming to an async slot

`otherObject: emit otherSignal(data);`



Special: From async to async slot



- When a function that is executed in the async stack activates other async slot, the same stack must be reused
  - The solution is to copy the current async stack into the default stack
  - This corresponds to suspending an async slot
  - The async stack is restored when the other async slot returns

Caveats

- Care must given when referencing signal arguments or other variables that may have been allocated from the stack
- The object of which signal is waited must not be deleted before the corresponding section of the asynchronous slot function has returned into the event loop
- Variables allocated from the asynchronous stack must not be referenced outside the asynchronous slot function itself

Evaluation

- Performance
  - memcpy may be expensive but also optimised (including caches)
  - Example: Intel Core i7-920, 2.67 GHz
    - Emit a plain Qt signal to an empty slot => 320 ns
    - Emit a signal that resumes an async slot => 3600 ns
    - char *s = "...65 characters..."; while (*s != 0) s++; => 360 ns
- Other solutions
  - Pure signals and slots are better for relatively independent activities
  - Nested *QEventLoop*s must be terminated in opposite starting order
  - Threads introduce synchronisation and other issues
  - Pure coroutine implementation (Qt Labs) lacks semantics and have a separate stack for each coroutine (run out of virtual memory)

Twitter: @qasync

http://qasync.henrikhedberg.com

henrik.hedberg@iki.fi