



Message Queues are a very convenient form of IPC (Inter Process Communication). This is an example of how to use a message queue to send messages from one process to another.

The Message Queue Handler is meant to be a part of an utility DLL which is linked into both the server and client projects. The server can use it to send messages and the client can use it to observe for messages from the server.

Contents

- 1 The code
 - ◆ 1.1 File:
[messagehandlerbase.h](#)
 - ◆ 1.2 File:
[messagehandlerbase.cpp](#)
 - ◆ 1.3 File:
[messageobserver.h](#)
 - ◆ 1.4 File:
[messagehandler.h](#)
 - ◆ 1.5 File:
[messagehandler.inl](#)
- 2 Usage example
 - ◆ 2.1 File:
[testmessage.h](#)
 - ◆ 2.2 File:
[typedefs.h](#)
 - ◆ 2.3 File:
[server.cpp](#)
 - ◆ 2.4 File: [client.h](#)
 - ◆ 2.5 File:
[client.cpp](#)
- 3 Limitation

The code

The message queue handler uses the thin-template idiom to be able to handle any type of message. Symbian has already implemented **RMsgQueue** as a thin template.

There are different ways how they can be implemented. The base class can have either a protected or a public API. **RArrayBase**, the base class for **RArray** declares protected functions which means they are accessible only to deriving classes. **RMsgQueueBase** on the other defines public functions which means the thin-template wrapper can easily be unwrapped and the base class used directly. That is what the message queue handler does. The base class uses **RMsgQueueBase** without the **RMsgQueue** wrapper defines a new wrapper.

File: messagehandlerbase.h

```

#ifndef CMESSAGEHANDLERBASE_H
#define CMESSAGEHANDLERBASE_H

#include <e32base.h>
#include <e32msgqueue.h> // For RMsgQueueBase

class CMessageHandlerBase : public CActive
{
public:
    ~CMessageHandlerBase();

    virtual void StartNotify( TAny* aObserver );
    void CancelNotify();
    virtual TInt SendMessage( const TAny* aMsg, TInt aMsgLength );

protected:
    CMessageHandlerBase();
    void ConstructL( const TDesC& aQueueName, TInt aSize, TInt aMsgLength );

    void DoCancel();
    void RunL();
        TInt RunError( TInt aError );

/**
    * Deriving class must override this to read a message from queue
    * and notify the observer. Base class cannot do that because it
    * doesn't know the message type or observer type.
    */
    virtual void ReadMessageAndNotify() = 0;

protected: // data

    /**
    * Pointer to message observer.
    * Not own.
    */
    TAny* iObserver;

    /**
    * ESG message queue.
    * Own.
    */
    RMsgQueueBase iQueue;

};

#endif // CMESSAGEHANDLERBASE_H

```

File: messagehandlerbase.cpp

```

#include "messagehandlerbase.h"

```

Message_Queue_Handler

```
CMessageHandlerBase::CMessageHandlerBase()
    : CActive( CActive::EPriorityStandard )
    {
    }

void CMessageHandlerBase::ConstructL( const TDesC& aQueueName,
                                      TInt aSize,
                                      TInt aMsgLength )
    {
    if ( iQueue.CreateGlobal( aQueueName, aSize, aMsgLength ) )
        {
        User::LeaveIfError( iQueue.OpenGlobal( aQueueName ) );
        }

    CActiveScheduler::Add( this );
    }

CMessageHandlerBase::~CMessageHandlerBase()
    {
    Cancel();
    iQueue.Close();
    }

void CMessageHandlerBase::StartNotify( TAny* aObserver )
    {
    __ASSERT_DEBUG( aObserver, User::Panic( _L("MessageHandler"),
                                           KErrArgument ) );

    if ( !IsActive() )
        {
        iObserver = aObserver;
        iQueue.NotifyDataAvailable( iStatus );
        SetActive();
        }
    }

void CMessageHandlerBase::CancelNotify()
    {
    Cancel();
    }

TInt CMessageHandlerBase::SendMessage( const TAny* aMsg,
                                       TInt aMsgLength )
    {
    return iQueue.Send( aMsg, aMsgLength );
    }

void CMessageHandlerBase::DoCancel()
    {
    iQueue.CancelDataAvailable();
    }

void CMessageHandlerBase::RunL()
    {
    if ( iStatus.Int() == KErrNone )
        {
        ReadMessageAndNotify();
        }

    iQueue.NotifyDataAvailable( iStatus );
    SetActive();
    }
}
```

Message_Queue_Handler

```
TInt CMessageHandlerBase::RunError( TInt /*aError*/ )
{
    return KErrNone;
}
```

File: messageobserver.h

```
#ifndef MMESSAGEOBSERVER_H
#define MMESSAGEOBSERVER_H

template<typename MSG>
class MMessageObserver
{
public:
    virtual void MessageReceived( const MSG& aMsg ) = 0;
};

#endif // MMESSAGEOBSERVER_H
```

File: messagehandler.h

```
#ifndef CMESSAGEHANDLER_H
#define CMESSAGEHANDLER_H

#include "messagehandlerbase.h"
#include "messageobserver.h"

template<typename MSG, typename OBS> // MSG = Message, OBS = Observer
class CMessageHandler : public CMessageHandlerBase
{
public:

    inline static CMessageHandler<MSG,OBS>* NewL( const TDesC& aQueueName,
                                                    TInt aSize );
    inline static CMessageHandler<MSG,OBS>* NewLC( const TDesC& aQueueName,
                                                    TInt aSize );

    /**
     * Starts message notification
     * @param aObserver Observer to be notified
     */
    inline void StartNotify( OBS* aObserver );

    /**
     * Sends a message to the message queue
     * @param aMsg Message to be sent
     */
    inline TInt SendMessage( const MSG& aMsg );

private:

    /**
     * Reads a message from queue and notifies the observer
     */
    void ReadMessageAndNotify();
};
```

File: messageobserver.h

Message_Queue_Handler

```
};  
  
#include "messagehandler.inl"  
  
#endif // CMESSAGEHANDLER_H
```

File: messagehandler.inl

```
template<typename MSG, typename OBS> // MSG = Message, OBS = Observer  
inline CMessageHandler<MSG,OBS>* CMessageHandler<MSG,OBS>::NewL(  
    const TDesC& aQueueName,  
    TInt aSize )  
{  
    CMessageHandler<MSG,OBS>* self = CMessageHandler<MSG,OBS>::NewLC(  
        aQueueName, aSize );  
    CleanupStack::Pop( self );  
    return self;  
}  
  
template<typename MSG, typename OBS> // MSG = Message, OBS = Observer  
inline CMessageHandler<MSG,OBS>* CMessageHandler<MSG,OBS>::NewLC(  
    const TDesC& aQueueName,  
    TInt aSize )  
{  
    CMessageHandler<MSG,OBS>* self = new( ELeave ) CMessageHandler<MSG,OBS>();  
    CleanupStack::PushL( self );  
    self->ConstructL( aQueueName, aSize, sizeof( MSG ) );  
    return self;  
}  
  
template<typename MSG, typename OBS> // MSG = Message, OBS = Observer  
inline void CMessageHandler<MSG,OBS>::StartNotify( OBS* aObserver )  
{  
    CMessageHandlerBase::StartNotify( aObserver );  
}  
  
template<typename MSG, typename OBS> // MSG = Message, OBS = Observer  
EXPORT_C TInt CMessageHandler<MSG,OBS>::SendMessage( const MSG& aMsg )  
{  
    return CMessageHandlerBase::SendMessage( &aMsg, sizeof( MSG ) );  
}  
  
template<typename MSG, typename OBS> // MSG = Message, OBS = Observer  
void CMessageHandler<MSG,OBS>::ReadMessageAndNotify()  
{  
    MSG message;  
    if ( iQueue.Receive( &message, sizeof( MSG ) ) == KErrNone )  
    {  
        static_cast<OBS*>( iObserver )->MessageReceived( message );  
    }  
}
```

Usage example

This is a simple example of how the message queue handler is used. First a message class is defined. It must be a simple T class that contains the data that is to be sent. Because the templates make the class declarations a bit long, typedefs are used to clean up the code. When the message queue handler is created, the message queue name must be specified. It will be a global queue so its name must be unique in the entire system. Here a simple `_L` macro is used to specify the name, but in a real system it should be a `_LIT` in some header that contains all the constants in the system. Also the number messages the queue can hold must be specified. The example specifies 10 but that should be decided based on the needs of the system.

File: testmessage.h

```
#ifndef TTESTMESSAGE_H
#define TTESTMESSAGE_H

#include <e32std.h>

class TTestMessage
{
public:

    TTestMessage()
        : iData( 0 )
        {
        }

public: // data

    // Payload
    TInt iData;

};

#endif // TTESTMESSAGE_H
```

File: typedefs.h

```
#ifndef TYPEDEFS_H
#define TYPEDEFS_H

class TTestMessage;

// Typedef for the observer
template<typename MSG> class MMessageObserver;
typedef MMessageObserver<TTestMessage> MTestMsgObserver;

// Typedef for the message handler
template<typename MSG, typename OBS> class CMessageHandler;
typedef CMessageHandler<TTestMessage, MTestMsgObserver> CTestMsgHandler;

#endif // TYPEDEFS_H
```

File: server.cpp

```
void CSomeServer::ConstructL()
{
    ...
    iMsgHandler = CTestMsgHandler::NewL( _L("TestMsgQueueName"), 10 );
}

void CSomeServer::SendMessageToClient()
{
    TTestMessage msg;
    msg.iData = 42;
    iMsgHandler->SendMessage( msg );
}
```

File: client.h

```
#include "messageobserver.h"
#include "testmessage.h"
#include "typedefs.h"

class CSomeClient : public CBase
                  , public MTestMsgObserver
{
    ...
private:

    void MessageReceived( const TTestMessage& aMsg );

private: // data

    ...
    CTestMsgHandler* iMsgHandler;
};
```

File: client.cpp

```
void CSomeClient::ConstructL()
{
    ...
    iMsgHandler = CTestMsgHandler::NewL( _L("TestMsgQueueName"), 10 );
    iMsgHandler->StartNotify( this );
}

void CSomeClient::MessageReceived( const TTestMessage& aMsg )
{
    RDebug::Print( _L("Message received: Data %d"), aMsg.iData );
}
```

Limitation

As mentioned in the [Message Queues](#) article, there can only be one process listening to a queue at a time. If a second process calls **NotifyDataAvailable()** on the queue, the thread that did it will be panicked. Also there is no API to check whether or not there already is a process listening to the queue. This makes it hard to

Message_Queue_Handler

discreetly handle the situation when the queue cannot be observed. It is possible to create a special thread to check for this and then monitor if it is panicked or not, but that kind of trickery is not really recommendable. You're better off designing a system where this problem never arises. If you need to send messages to multiple processes, use multiple queues.