Multithreading

Multithreading and WPF

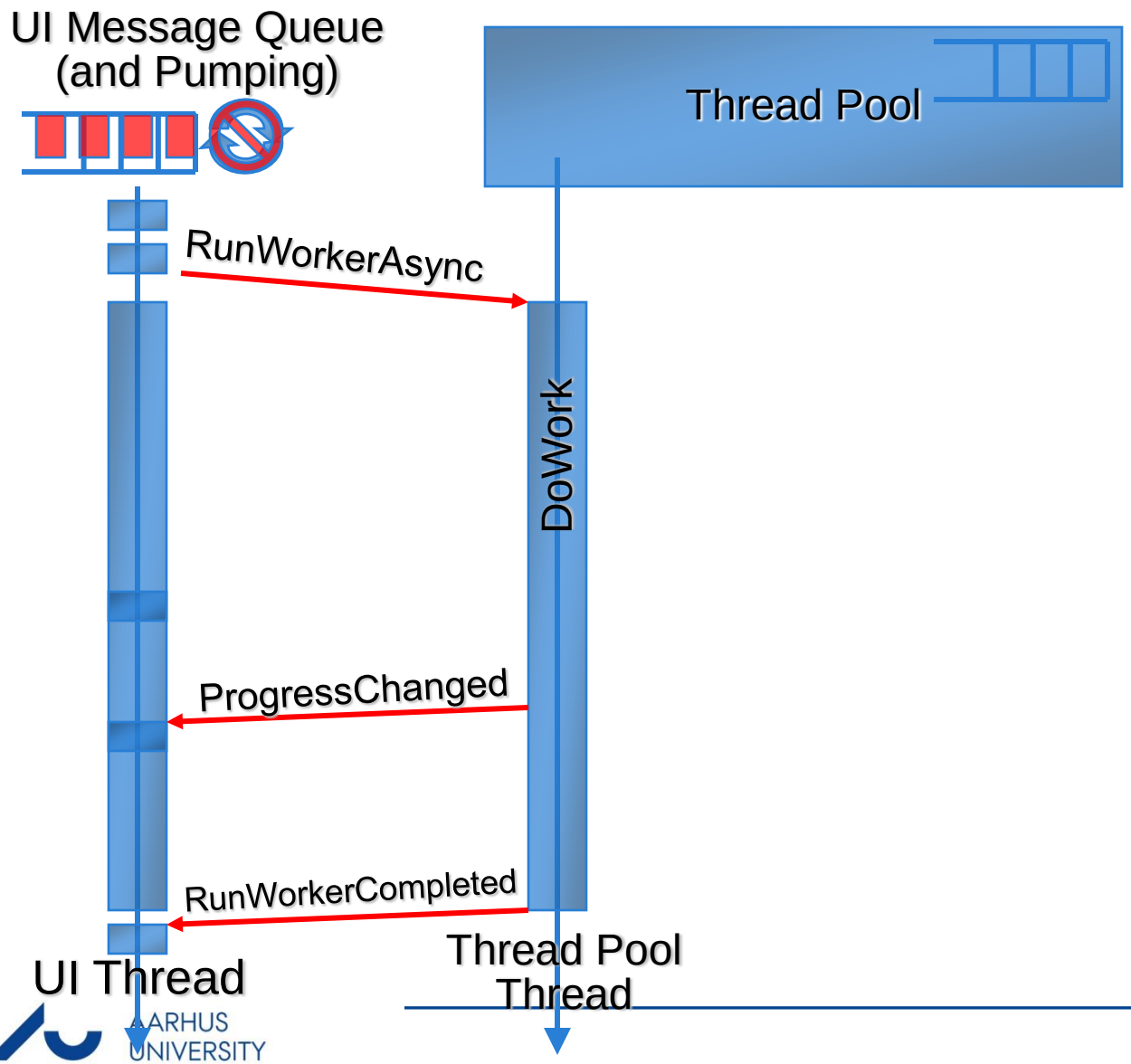
Agenda

- UI and threads – Overview
- The Dispatcher
- Background Worker
- Task Parallel Library

UI AND THREADS

Overview

Threads, UI, Thread Pool



Threads and Controls

```
public partial class MainWindow : Window
{
   ..
   // Created on UI thread
   private TextBlock tblkThreadStatus;

   // Doesn't run on UI thread
   private void RunsOnWorkerThread()
   {
      DoSomethingSlow();
      tblkStatus.Text = "Finished!"; // BAD!!
   }
}
```

Threads and Controls

- The WPF threading model has strict rules about the use of threads:
  - **You must never use any member of a WPF user interface element on any thread other than the one that created it.**
    (with very few exceptions)

- WPF types with thread affinity derive from the DispatcherObject base class.
- The DispatcherObject class defines a few members, all of which are exempt from the thread affinity rule
  - You can use them from any thread.
  - Only the functionality added by classes that derive from DispatcherObject is subject to thread affinity.

- ***This means that you cannot call any methods on anything in the user interface unless the documentation for that method says that it's OK.***

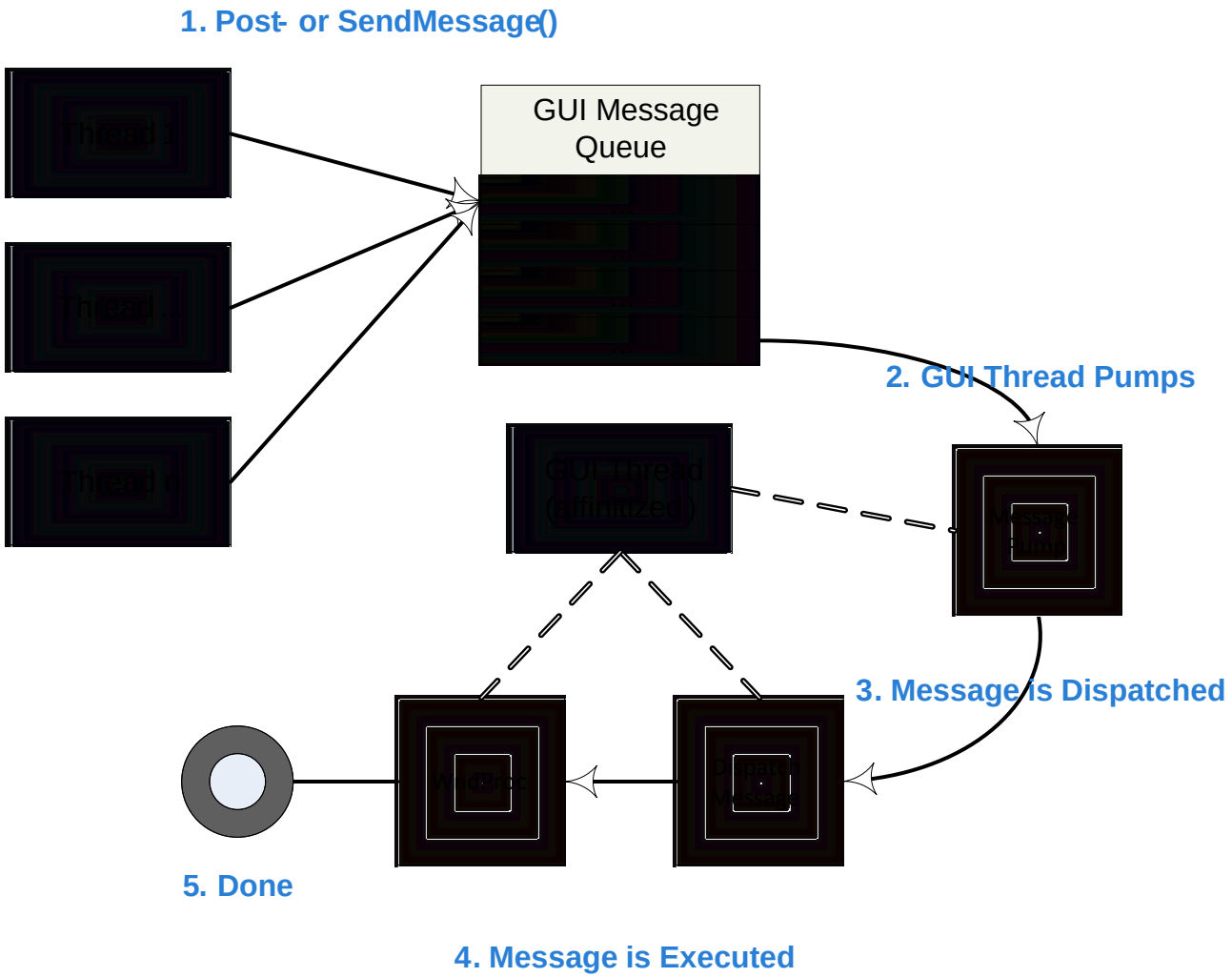
- *The WPF threading model is similar to the Windows.Forms threading model and COMs Single Threaded Apartment (STA) model.*

Calling a Control on the Right Thread

- If your application causes multiple threads to be created, either through explicit thread creation or implicitly through the use of asynchronous APIs, you should avoid touching any user interface objects on those threads.

- If you need to update the user interface as a result of work done on a different thread, you must either:
  - Use the ***dispatcher*** to get back onto the UI thread.
  - Use **BackgroundWorker** to encapsulate the worker thread.
  - Use **data binding** to update the UI.
  - Use **await** (where possible).

GUIs and Messages



THE DISPATCHER

The Dispatcher

- Each thread that creates user interface objects needs a Dispatcher object.
- This effectively owns the thread, running a loop that dispatches input messages to the appropriate handlers.
  - The Dispatcher maintains a prioritized queue of work items for a specific thread.
- As well as handling input, the dispatcher enables us to get calls directed through to the right thread.
- When a Dispatcher is created on a thread, it becomes the only Dispatcher that can be associated with the thread, even if the Dispatcher is shut down.
- The Dispatcher class belong to the System.Windows.Threading namespace.

Verify Access

- DispatcherObject provides a couple of methods that let you check whether you are on the right thread for the object:
  - `CheckAccess`
    returns true if you are on the correct thread, false otherwise.
  - `VerifyAccess`
    throws an exception if you are on the wrong thread.

- Many WPF types call VerifyAccess when you use them.
  - They check in enough places that you are unlikely to get very far on the wrong thread before the problem becomes apparent.

Obtaining a Dispatcher

- All WPF objects with thread affinity derive from the `DispatcherObject` base class.
- This class defines a Dispatcher property, which returns the Dispatcher object for the thread to which the object belongs.
- You can also retrieve the Dispatcher for the current thread by using the `Dispatcher.CurrentDispatcher` static property.
- If you attempt to get the CurrentDispatcher for the current thread and a Dispatcher is not associated with the thread, a Dispatcher will be created.

Getting Onto the Right Thread with a Dispatcher

- You can use either Invoke or BeginInvoke.
- Both of these accept any delegate and an optional list of parameters.
- They both invoke the delegate's target method on the dispatcher's thread, regardless of which thread you call them from.
- **Invoke**
  - does not return until the method has been executed (synchronous call).
  - Simpler, but there is a risk of deadlock.
- **BeginInvoke**
  - queues the request to invoke the method, but returns straight away without waiting for the method to run (asynchronous call).
  - No risk of deadlock, but the order of events is less predictable.

Using Dispatcher - Delegate

```
public partial class MyWindow : System.Windows.Window
{
  public delegate void MyDelegateType();
  ...
  // This function is called from another thread
  void RunsOnWorkerThread()
  {
    MyDelegateType methodForUiThread = delegate
      {
        tblkStatus.Text = "Finished.";
      };

    this.Dispatcher.BeginInvoke(DispatcherPriority.Normal,
      methodForUiThread);
  }
...
```

Using Dispatcher – Inline Delegate

```
public partial class MyWindow : System.Windows.Window
{
  public delegate void MyDelegateType();
  ...
  // This function is called from another thread
  void RunsOnWorkerThread()
  {
     this.Dispatcher.BeginInvoke(new Action(
        delegate()
        {
            tblkStatus.Text = "Finished Again!";
        }
        ));
  }
...
```

Using Dispatcher – Lambda

```
public partial class MyWindow : System.Windows.Window
{
  public delegate void MyDelegateType();
  ...
  // This function is called from another thread
  void RunsOnWorkerThread()
  {
    Dispatcher.BeginInvoke(new Action(() =>
        { tblkStatus.Text = "All Done!"; }));
  }
...
```

BeginInvoke

```
public DispatcherOperation BeginInvoke(
                    DispatcherPriority priority,
                    Delegate method )
```

priority

The priority, relative to the other pending operations in the Dispatcher event queue, the specified method is invoked.
Values such as: Send, Normal, Background, ApplicationIdle, or SystemIdle.

method

The delegate to a method that takes no arguments, which is pushed onto the Dispatcher event queue.

Return Value

An object, which is returned immediately after BeginInvoke is called, that can be used to interact with the delegate as it is pending execution in the event queue.

BeginInvoke Overload

```
public DispatcherOperation BeginInvoke(
                              DispatcherPriority priority,
                              Delegate method
                              Object[] args )
```

priority

The priority, relative to the other pending operations in the Dispatcher event queue, the specified method is invoked.

method

A delegate to a method that takes multiple arguments, which is pushed onto the Dispatcher event queue.

args

An array of objects to pass as arguments to the specified method.

BACKGROUND WORKER

BackgroundWorker

- BackgroundWorker provides event-based threading interface
  - based on *Asynchronous* design pattern

- BackgroundWorker perfect for independent, background tasks:
  - Event-based model integrates nicely with event-driven UIs
  - Can pass parameters to thread & harvest return values
  - Reduces cost of multithreading by using .NET's ***thread pool***
    - *threads are reused from pool instead of created & destroyed each time*
  - Ability to cancel task
  - Exceptions are caught for you — you process or ignore

Using the BackgroundWorker Class

- Executive summary:
  - create an instance of BackgroundWorker
  - handle *DoWork* event to run thread code
  - handle *ProgressChanged* & *RunWorkerCompleted* as appropriate
  - call *RunWorkerAsync* to run
  - call *CancelAsync* to cancel

```
using SCM = System.ComponentModel;
.
.
.
worker = new SCM.BackgroundWorker();
worker.DoWork += this.PerformCall;

worker.ProgressChanged    += new SCM.ProgressChangedEventHandler(DisplayProgress);
worker.RunWorkerCompleted += new SCM.RunWorkerCompletedEventHandler(DisplayResults);

worker.RunWorkerAsync(parameters);   // start!
.
.
.
}

private void PerformCall(...)
{  // do thread code here  }
```

```
public class Window1
{
    public void DisplayProgress(…)
    { … }

    public void DisplayResults(…)
    { … }
```

ASYNCHRONY WITH ASYNC/AWAIT

New in C# 5

Task-based Asynchronous Pattern

- .Net v. 4.5 exposes asynchronous versions of a great many operations, following a newly-documented **Task-based asynchronous pattern**.
- The WinRT framework used to create Windows Store Apps enforces asynchrony for all long-running (or potentially long-running) operations.

- In short:

the future is asynchronous

Asynchronous Functions

- C# 5 introduces the concept of an **asynchronous function** in the language.
- This is always either a method or an anonymous function which is declared with the `async` modifier,
  - and can include `await` expressions.

```
private async void btnGetHtml_Click(object sender,
                                    RoutedEventArgs e)
{
    tbxLength.Text = "Fetching...";
    string url = tbxUrl.Text;
    HttpClient client = new HttpClient();
    string text = await client.GetStringAsync(url);
    tbxLength.Text = text.Length.ToString();
}
```

Task<TResult>

- If we split the call to HttpClient.GetStringAsync from the await expression we can see the types involved:

```
HttpClient client = new HttpClient();
Task<string> task = client.GetStringAsync(url);
string text = await task;
```

- The type of `GetStringAsync` is `Task<string>`.
- But the type of the `await` task expression is just string.
  - **The `await` expression performs an "unwrapping" operation.**

await

- A new language construct that allows you to "await" an asynchronous operation.
- **The main purpose of await is to avoid blocking while we wait for a time-consuming operation to complete.**
- This "awaiting" looks very much like a normal blocking call, in that the rest of your code won't continue until the operation has completed
  - ***but it manages to do this without actually blocking the currently executing thread***.

await uncovered

- Await's trick is the method actually returns as soon as we hit the await expression.
  - Up until that point, it executes synchronously on the UI thread just as any other event handler.
- When await is reached, the code checks whether the result is already available, and if it's not it schedules a **continuation** to be executed when the awaited operation has completed.
- The continuation is executed on the GUI thread.

```
private async void btnGetHtml_Click(object s, RoutedEventArgs e)
{
    string url = tbxUrl.Text;
    HttpClient client = new HttpClient();
    string text = await client.GetStringAsync(url);
    tbxLength.Text = text.Length.ToString();
}
```

Continuation

Return types from async methods

- Async methods are limited to the following return types:
  - `void`
  - `Task`
  - `Task<TResult>`

- Task and Task<TResult> types, represent an operation which may not have completed yet.
  - Task<TResult> represents an operation which returns a value of type T.
  - Task will not produce a result.

- Note:
  - You cannot use the out or ref modifiers on the parameters to an async operation declaration.

TASK PARALLEL LIBRARY
TPL

Introduced in .Net 4.0

Parallel Extensions

- Parallel Extensions is a .NET Library that supports:
  1. **Parallel LINQ (PLINQ)**

     Declarative and imperative data parallelism.
  2. **Task Parallel Library (TPL)**

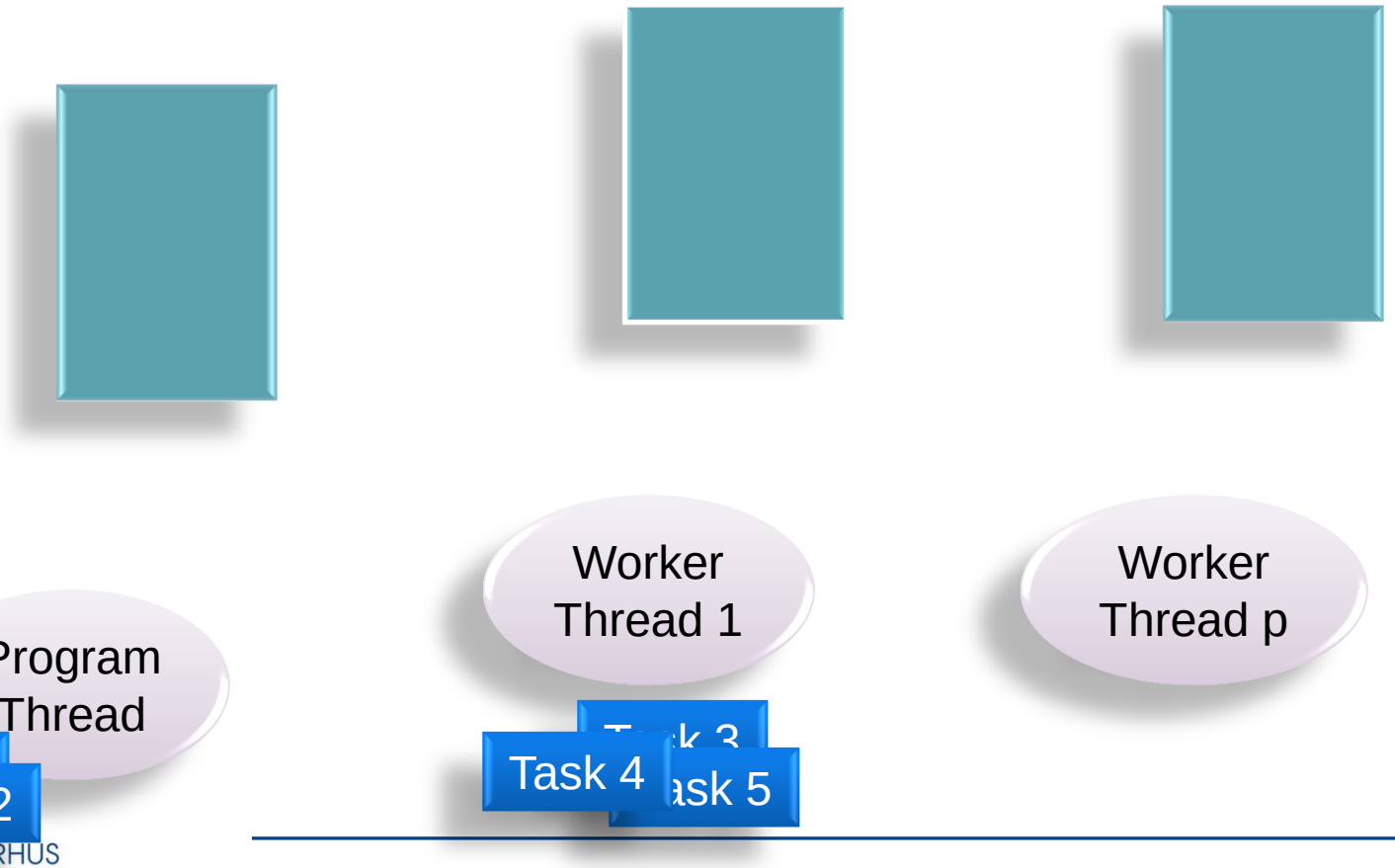
     Imperative task parallelism.
  3. **Coordination Data Structures (CDS)**

     A set of data structures that make coordination easier.

Tasks and Futures

- **Task<T>** represents a logical unit of work
  - Latent parallelism
  - May be run serially
  - Parent/child relationships
- **Future<T>** is a task that produces a value
  - Accessing Value will:
    - Runs it serially if not started
    - Block if it's being run
    - Return if the value is ready
    - Throw an exception if the future threw an exception
- Can wait on either (Wait, WaitAll, WaitAny)
  - Runs the task "inline" if unstarted

“Work Stealing” in Action



Parallel Static Class

When program statements are independent…

```
StatementA();
StatementB();
StatementC();
```

…they can be parallelized

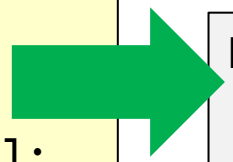
```
Parallel.Invoke(
   () => StatementA(),
   () => StatementB(),
   () => StatementC() );
```

Parallel.For

- Executes a for loop in which iterations may run in parallel.
  (has 12 overloads).

```
public static ParallelLoopResult For(
        int fromInclusive,
        int toExclusive,
        Action<int> body
 )
```

```
int[] A, B, C;
.
.
.
for(i=0; i<N; i++)
{
   C[i] = A[i] + B[i];
}
```



```
Parallel.For (0,  N,
  (i) =>
  {
     C[i] = A[i] + B[i];
  }
);
```

PLINQ

- Parallel LINQ (PLINQ) enables developers to easily leverage many core with a minimal impact to existing LINQ programming model

```
var q = from p in people.AsParallel()
        where p.Name == queryInfo.Name &&
              p.State == queryInfo.State &&
              p.Year >= yearStart &&
              p.Year <= yearEnd
        orderby p.Year ascending
        select p;
```

Resources - Links

- .NET Framework Developer's Guide
  Managed Threading
  http://msdn2.microsoft.com/library/3e8s7xdd.aspx
- Give Your .NET-based Application a Fast and Responsive UI with Multiple Threads
  http://msdn.microsoft.com/library/default.asp?url=/msdnmag/issues/03/02/multithreading/toc.asp
- **UI Dispatcher in MVVM**
  http://stackoverflow.com/questions/4621623/wpf-multithreading-ui-dispatcher-in-mvvm
- **Parallel Programming in the .NET Framework**
  http://msdn.microsoft.com/en-us/library/dd460693.aspx