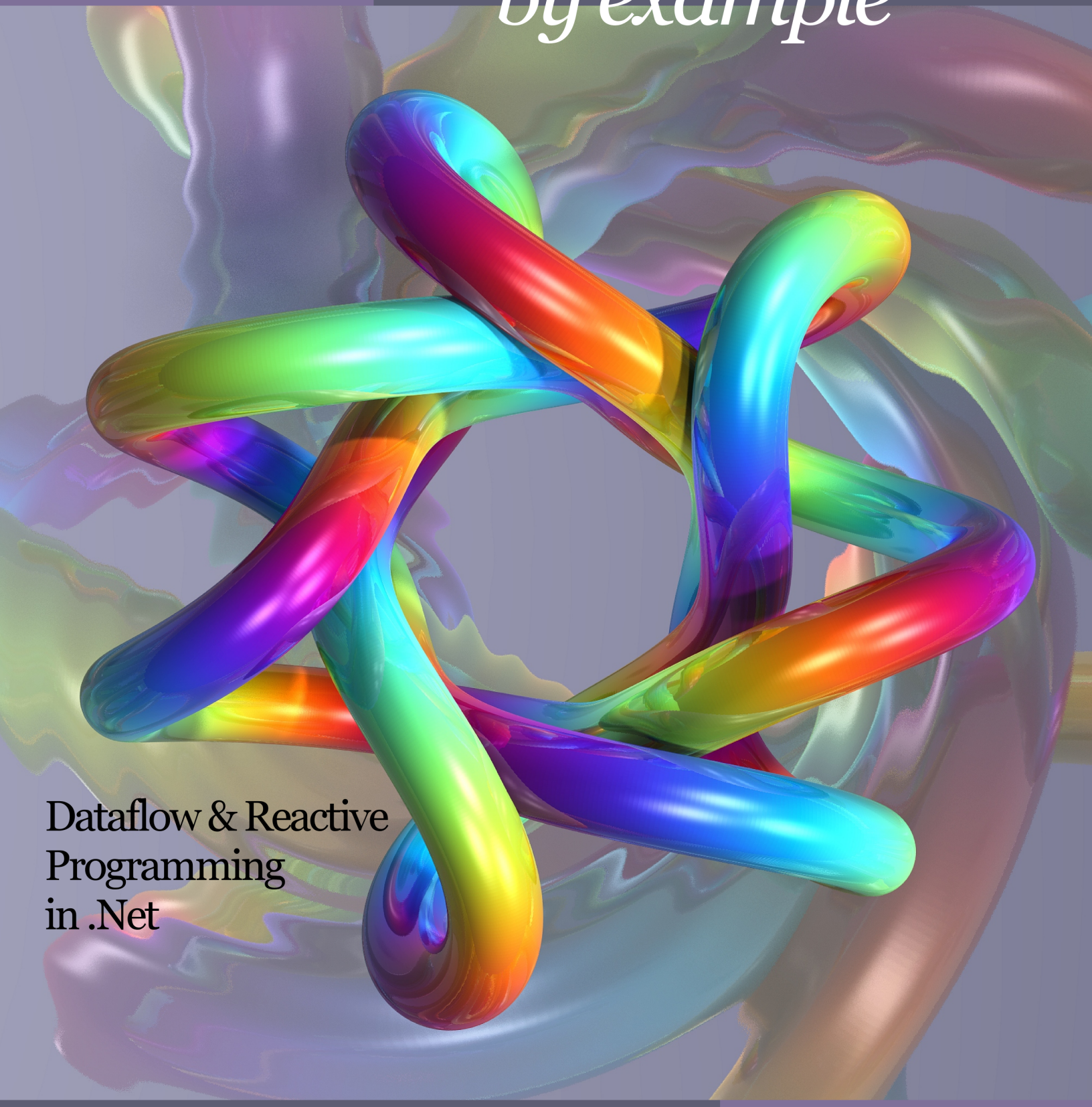
TPL Dataflow

by example



Dataflow & Reactive Programming in .Net

Matt Carkci

TPL Dataflow by Example

Dataflow and Reactive Programming in .Net

Matt Carkci

This book is for sale at http://leanpub.com/tpldataflowbyexample

This version was published on 2014-05-29

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

1 TPL Dataflow Basics

1.1 Blocks

In dataflow, blocks (or nodes) are entities that may send and receive data and are the basic unit of composition. The TPL Dataflow Library comes with a handful of predefined blocks, while they're very basic, they should cover 99% of your needs. Using these predefined blocks, you can build your own application specific blocks.

Think of the predefined blocks as being equivalent to keywords in C#. You build C# applications by using the keywords. The predefined blocks similarly define the basic operations of dataflow programs that you use to build your dataflow application.

Of the predefined blocks offered by the TPL Dataflow library, we can categorize them into three groups, blocks that process data (execution blocks), blocks that buffer or store data (buffer blocks) and blocks that group data together into collections (grouping blocks). In the follow sections we'll examine each category of block and discover how they work with simple code examples.

1.1.1 Execution Blocks

Execution blocks process data very similar to how methods accept data and possibly returns a value. At creation you pass either a `Func` or an `Action` that defines what the execution block will do with the data.

All execution blocks contain an internal buffer that defaults to an unbounded capacity.

1.1.1.1 ActionBlock<T>

An `ActionBlock<T>` has a single input and no output. It is used when you need to do something with the input data but won't need to pass it along to other blocks. It is the equivalent to the `Action<T>` class. In dataflow, this type of block is often called a "sink" because the data sinks into it like a black hole, never to emerge again.

At creation, an `ActionBlock<T>` accepts an `Action<T>` that is called when data arrives at the input.

An internal buffer is present on the input of an `ActionBlock<T>`. The buffer defaults to an unbounded capacity but this can be changed by using `DataflowBlockOptions` mentioned later.

ActionBlock<T> Example 1

Basic usage, Block threading, Post()

```
using System;
using System.Threading.Tasks.Dataflow;

namespace TPLDataflowByExample
{
    static class ActionBlockExample1
    {
        static public void Run() {

            var actionBlock = new ActionBlock<int>(n => Console.WriteLine(n));

            for (int i = 0; i < 10; i++) {
                actionBlock.Post(i);
            }

            Console.WriteLine("Done");
        }
    }
}
```

This example shows the basic usage of an `ActionBlock<T>` and how to send data to all types of blocks that accept inputs.

The `Post()` function sends data synchronously to blocks and returns `true` if the data was successfully accepted. If the block refuses the data, the function returns `false` and it will not attempt to resend it.

In this example the numbers 0 through 9 are pushed to `actionBlock`. The block takes each value and calls the `Action<T>` that was given at creation. Since, in this example, our action simply prints the received data, the output to the console looks like:

```
Done
0
1
2
3
4
5
6
```

```
7
8
9
```

Notice how “Done” was printed first. This is because `actionBlock` was executed in parallel to the main thread.

ActionBlock<T> Example 2

Basic usage with a delay

```
using System;
using System.Threading;
using System.Threading.Tasks.Dataflow;

namespace TPLDataflowByExample
{
    class ActionBlockExample2
    {
        static public void Run() {

            var actionBlock = new ActionBlock<int>(n => {
                Thread.Sleep(1000);
                Console.WriteLine(n);
            });
            for (int i = 0; i < 10; i++) {
                actionBlock.Post(i);
            }

            Console.WriteLine("Done");
        }
    }
}
```

This example is almost identical to example 1 except we are now sleeping for one second before printing the value to the console to simulate a long running action in the block. The output is the same as example 1.

1.1.1.2 TransformBlock<T1,T2>

A `TransformBlock<T1,T2>` is very similar to an `ActionBlock<T>` except it also has an output that you can connect to other blocks (linking blocks will be covered in a later section). It is equivalent to a `Func<T1,T2>` in that it returns a result. Similar to an `ActionBlock<T>`, it takes a function at creation that operates on the input data.

This block contains two buffers, one on the input and one on the output but it is best to think of it as only having a single buffer. Two buffers are needed to ensure that the data is transmitted in the same order as it arrived. The output buffer is used to restore the original ordering of the data. But this is an implementation detail that you should know about but not need to worry about normally.

TransformBlock<T1,T2> Example 1

Receive()

```
1  using System;
2  using System.Threading;
3  using System.Threading.Tasks.Dataflow;
4
5  namespace TPLDataflowByExample
6  {
7      class TransformBlockExample1
8      {
9          static public void Run() {
10             Func<int, int> fn = n => {
11                 Thread.Sleep(1000);
12                 return n * n;
13             };
14
15             var tfBlock = new TransformBlock<int, int>(fn);
16
17             for (int i = 0; i < 10; i++) {
18                 tfBlock.Post(i);
19             }
20
21             for (int i = 0; i < 10; i++) {
```

```
22                  int result = tfBlock.Receive();
23                  Console.WriteLine(result);
24              }
25
26              Console.WriteLine("Done");
27          }
28      }
29  }
```

In this example we create a `TransformBlock<T1,T2>` with a function that squares the input value after a one second wait to simulate a long running process.

To extract data from a `TransformBlock<T1,T2>` (or any block with an output) you use the `Receive()` method that operates synchronously. If no data is available, the thread will be suspended until data is available. We highlight that fact in this example.

Executing this code should display…

```
0
1
4
9
16
25
36
49
64
81
Done
```

The `for` loop passes the numbers `0` through `9` to the `tfBlock`. The function that we passed in at creation time then squares each value and sends the result to the output where we `Receive()` them.

Notice that for this example "Done" is only printed *after* all the output values have been printed. This is because the `Receive()` method operates synchronously in the same thread as the `for` loops.

TransformBlock<T1,T2> Example 2

ReceiveAsync(), Task.Result()

```
using System;
using System.Threading;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;

namespace TPLDataflowByExample
{
    class TransformBlockExample2
    {
        static public void Run() {
            Func<int, int> fn = n => {
                Thread.Sleep(1000);
                return n * n;
            };

            var tfBlock = new TransformBlock<int, int>(fn);

            for (int i = 0; i < 10; i++) {
                tfBlock.Post(i);
            }

            // RecieveAsynch returns a Task
            for (int i = 0; i < 10; i++) {
                Task<int> resultTask = tfBlock.ReceiveAsync();
                int result = resultTask.Result;
                    // Calling Result will wait until it has a value ready
                Console.WriteLine(result);
            }

            Console.WriteLine("Done");
        }
    }
}
```

This example shows how to receive data, asynchronously, from all blocks with outputs using the aptly named `ReceiveAsync()` method. Since it operates asynchronously, the method does not return a value like the `Receive()` method does. Instead the `ReceiveAsync()` method returns a `Task<T>` that represents the receive operation. Calling the `Result()` method on the returned `Task` forces the program to wait until data becomes available essentially making it a synchronous operation like the previous example with the same console output. The next example shows how to create a completely asynchronous receive.

TransformBlock<T1,T2> Example 3

ReceiveAsync(), Task.ContinueWith()

```
using System;
using System.Threading;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;

namespace TPLDataflowByExample
{
    class TransformBlockExample3
    {
        static public void Run() {
            Func<int, int> fn = n => {
                Thread.Sleep(1000);
                return n * n;
            };

            var tfBlock = new TransformBlock<int, int>(fn);

            for (int i = 0; i < 10; i++) {
                tfBlock.Post(i);
            }

            Action<Task<int>> whenReady = task => {
                int n = task.Result;
                Console.WriteLine(n);
            };

            for (int i = 0; i < 10; i++) {
                Task<int> resultTask = tfBlock.ReceiveAsync();
                resultTask.ContinueWith(whenReady);
                // When 'resultTask' is done,
                // call 'whenReady' with the Task
            }

            Console.WriteLine("Done");
        }
```

```
36      }
37  }
```

If we modify the previous example slightly, we can receive data from blocks asynchronously. The addition of a continuation with the `ContinueWith()` method allows our main thread to proceed without having to wait for data to be available to read.

A continuation is just something that will be done after the `Task` is completed. In this case our continuation is the `whenReady` action that will print the result to the console.

When run, the example displays…

```
Done
0
1
4
9
16
25
36
49
64
81
```

We again have "Done" printed first since the main thread doesn't have to wait to receive data.

1.1.1.3 Block Configuration

All of the pre-defined blocks in the TPL Dataflow library can be configured by passing an options object to the blocks' constructor. Execution blocks use the `ExecutionDataflowBlockOptions` class, grouping blocks use the `GroupingDataflowBlockOptions` class and buffering blocks use the `DataflowBlockOptions` class. `ExecutionDataflowBlockOptions` and `GroupingDataflowBlockOptions` both inherit from the `DataflowBlockOptions` class (described in the Grouping Blocks section).

1.1.1.4 Execution Block Options

In addition to the options provided by its base class, `ExecutionDataflowBlockOptions` also includes the options, `MaxDegreeOfParallelism` and `SingleProducerConstrained`.

ExecutionDataflowBlockOptions Example 1

MaxDegreeOfParallelism

```
using System;
using System.Threading;
using System.Threading.Tasks.Dataflow;

namespace TPLDataflowByExample
{
    class ExecutionDataflowBlockOptionsExample1
    {
        static public void Run() {

            var generator = new Random();
            Action<int> fn = n => {
                Thread.Sleep(generator.Next(1000));
                Console.WriteLine(n);
            };
            var opts = new ExecutionDataflowBlockOptions {
                MaxDegreeOfParallelism = 2
            };

            var actionBlock = new ActionBlock<int>(fn, opts);

            for (int i = 0; i < 10; i++) {
                actionBlock.Post(i);
            }

            Console.WriteLine("Done");
        }
    }
}
```

Blocks can be configured to operate on more than one piece of data at a time. The default is for each value to be processed one at a time. The `MaxDegreeOfParallelism` option tells the computer to operate on multiple values at a time in parallel.

This example is a modification of the ActionBlock Example 2. We added a random delay to `actionBlock` to more closely approximate a real world situation. Running this example shows how the output order of values differs from the input order due to different delays.

On my machine running the example produces...

```
Done
1
0
2
3
4
5
7
6
9
8
```

ExecutionDataflowBlockOptions Example 2

SingleProducerConstrained

```
using System;
using System.Diagnostics;
using System.Threading;
using System.Threading.Tasks.Dataflow;

namespace TPLDataflowByExample
{
    // http://blogs.msdn.com/b/pfxteam/archive/2011/09/27/10217461.aspx
    class ExecutionDataflowBlockOptionsExample2
    {
        static public void Benchmark1() {
            var sw = new Stopwatch();
            const int ITERS = 6000000;
            var are = new AutoResetEvent(false);

            var ab = new ActionBlock<int>(i => { if (i == ITERS) are.Set(); });
            while (true) {
                sw.Restart();
                for (int i = 1; i <= ITERS; i++) ab.Post(i);
```

```
20                  are.WaitOne();
21                  sw.Stop();
22                  Console.WriteLine("Messages / sec: {0:N0}",
23                      (ITERS / sw.Elapsed.TotalSeconds));
24              }
25          }
26          static public void Benchmark2() {
27              var sw = new Stopwatch();
28              const int ITERS = 6000000;
29              var are = new AutoResetEvent(false);
30
31              var ab = new ActionBlock<int>(i => { if (i == ITERS) are.Set(); },
32                  new ExecutionDataflowBlockOptions {
33                      SingleProducerConstrained = true
34                  });
35              while (true) {
36                  sw.Restart();
37                  for (int i = 1; i <= ITERS; i++) ab.Post(i);
38                  are.WaitOne();
39                  sw.Stop();
40                  Console.WriteLine("Messages / sec: {0:N0}",
41                      (ITERS / sw.Elapsed.TotalSeconds));
42              }
43          }
44      }
45  }
```

The option `SingleProducerConstrained` is an optimization for situations where there is only a single block feeding data to another block. The creator of the TPL Dataflow library, Stephen Toub, explains its usage:

> Dataflow blocks by default are usable by any number of threads concurrently. While flexible, this also places more synchronization requirements, and therefore cost, on the blocks than might otherwise be necessary. If a block is only ever going to be used by a single producer at a time, meaning only one thread at a time will be using methods like Post, OfferMessage, and Complete on the block, this property may be set to true to inform the block that it need not apply extra synchronization. For blocks that observe this property, you can significantly reduce synchronization overheads by setting this property to true. Right now, only ActionBlock pays attention to this property, but more blocks could in the future as necessary.
>
> (from http://blogs.msdn.com/b/pfxteam/archive/2011/09/27/10217461.aspx)

Using his code (above) for a performance comparison, he measured an `ActionBlock<T>` throughput of 10,942,715 without the `SingleProducerConstrained` option (`Benchmark1()`), and 37,456,691 with the option set (`Benchmark2()`).