Introduction to TPL Dataflow

Stephen Toub, Microsoft
April 2011

Contents

TPL Dataflow (TDF) is a new .NET library for building concurrent applications. It promotes actor/agent-oriented designs through primitives for in-process message passing, dataflow, and pipelining. TDF builds upon the APIs and scheduling infrastructure provided by the Task Parallel Library (TPL) in .NET 4, and integrates with the language support for asynchrony provided by C#, Visual Basic, and F#.

Overview

The Task Parallel Library (TPL) was introduced in the .NET Framework 4, providing core building blocks and algorithms for parallel computation and asynchrony. This work was centered around the System.Threading.Tasks.Task type, as well as on a few higher-level constructs. These higher-level constructs address a specific subset of common parallel patterns, e.g. Parallel.For/ForEach for delightfully parallel problems expressible as parallelized loops.

While a significant step forward in enabling developers to parallelize their applications, this work did not provide higher-level constructs necessary to tackle all parallel problems or to easily implement all parallel patterns. In particular, it did not focus on problems best expressed with agent-based models or those based on message-passing paradigms. These kinds of problems are quite prevalent in technical computing domains such as finance, biological sciences, oil & gas, and manufacturing.

For TPL Dataflow (TDF), we build upon the foundational layer provided in TPL in .NET 4. TDF is a complementary set of primitives to those primitives delivered in TPL in .NET 4, addressing additional scenarios beyond those directly and easily supported with the original APIs. TPL Dataflow utilizes tasks, concurrent collections, tuples, and other features introduced in .NET 4 to bring support for parallel dataflow-based programming into the .NET Framework. It also directly integrates with new language support for tasks and asynchrony provided by both C# and Visual Basic, and with existing language support in .NET 4 for tasks provided by F#.

Background

In multiple vertical industries, including those in the technical computing domains, software systems are often best oriented around the flow of data. These “dataflows” are often large, infinite, or unknown in size, and/or require complex processing, leading to high-throughput demands and potentially immense computational load. To cope with these requirements, parallelism can be introduced to exploit system resources such as multiple cores. However, the concurrent programming models of today’s .NET Framework (including .NET 4) are not designed with dataflow in mind, leading to verbose code and low-level locking that complicates the source code, reduces system throughput, and introduces concurrency issues that can otherwise be avoided with a model more befitting to reactive systems. Furthermore, much of the data that enters these systems is the result of I/O operations. Developers tend to process this I/O synchronously, as asynchronous programming has historically been an all-around difficult activity. When a thread performs synchronous I/O, it essentially gives up control of the thread to the I/O device, often decreasing the responsiveness of the application and, again, hindering the overall throughput and scalability of the system.

Visual Studio 2010 targeted this problem space for native developers with the addition to the C Run-time (CRT) of the Asynchronous Agents Library (AAL). This model has proven to be very attractive and useful, and also shows up in the Concurrency & Coordination Runtime (CCR) available separately as a library contained in the Microsoft Robotics Studio. TDF can be thought of as a logical evolution of the CCR for non-robotics scenarios, incorporating CCR’s key concepts into TPL and augmenting the enhanced model with core programming model aspects of the Asynchronous Agents Library. This combination yields a technology that targets a wide range of scenarios. Most CCR-based code can be

brought forward and ported to TDF with a straightforward translation, composing well with workloads implemented with other primitives from TPL. Similarly, for developers desiring to port their native Asynchronous Agents-based code to managed, or to write a mixed-mode application that incorporates message passing on both sides of the divide, TPL Dataflow provides direct feature-to-feature mappings for most concepts.

TDF brings a few core primitives to TPL that allow developers to express computations based on dataflow graphs. Data is always processed asynchronously. With this model, tasks need not be scheduled explicitly – developers declaratively express data dependencies, and in return the runtime schedules work based on the asynchronous arrival of data and the expressed dependencies. In this manner, TDF is a generator of tasks just as is the Parallel class and PLINQ. In contrast, however, whereas Parallel and PLINQ are focused on more structured models of parallel computation, TDF is focused on unstructured, asynchronous models.

Astute readers may notice some similarities between TPL Dataflow and Reactive Extensions (Rx), currently available as a download from the MSDN Data developer center. Rx is predominantly focused on coordination and composition of event streams with a LINQ-based API, providing a rich set of combinators for manipulating IObservable<T>s of data. In contrast, TPL Dataflow is focused on providing building blocks for message passing and parallelizing CPU- and I/O-intensive applications with high-throughput and low-latency, while also providing developers explicit control over how data is buffered and moves about the system. As such, Rx and TPL Dataflow, while potentially viewed as similar at a 30,000 foot level, address distinct needs. Even so, TPL Dataflow and Rx provide a better together story. (In fact, TPL Dataflow includes built-in conversions that allow its “dataflow blocks” to be exposed as both observables and observers, thereby enabling direct integration of the two libraries.)

Architecture

At its heart, TDF is based on two interfaces: ISourceBlock<T> and ITargetBlock<T>. Sources offer data, and targets are offered data; nodes in a dataflow network may be one or the other, or both. A handful of concrete implementations of these interfaces are provided in the library, and developers may further build their own in order to target more advanced scenarios (though it is expected that the built-in implementations suffice for the majority of developer needs). Put together, these interfaces and implementations enable patterns for parallel stream processing, including multiple forms of data buffering; greedily and non-greedily receiving, joining, and batching data from one or more sources; selecting a single datum from multiple sources; protecting concurrent operations without explicit locking by executing tasks within a declarative reader/writer model; automatically propagating data from one operation to another, all with as much concurrency as the system and developer jointly allow; and more. Each of these patterns can be used standalone, or may be composed with others, enabling developers to express complex dataflow networks. The primitives provided by TDF not only compose well at the programming model level with tasks, parallel loops, observables, and language features in support of asynchrony, they also compose at the system level to enable efficient execution that doesn’t oversubscribe the system.

While TDF is reminiscent of actor-oriented models, it is important to note that it makes no built-in guarantees of isolation. It is an expressive and flexible messaging substrate and task generation system (using TPL as the scheduling infrastructure) on top of which higher-level models may be built.

Getting Started

TPL Dataflow is encapsulated as a single managed DLL, available as System.Threading.Tasks.Dataflow.dll. To get started, add a reference to this DLL from your managed project. The majority of the functionality in this DLL is exposed from the System.Threading.Tasks.Dataflow namespace, with a small number of types also contained in the System.Threading.Tasks namespace.

Introductory Examples

One of the most common ways into TPL Dataflow is the ActionBlock<TInput> class. This class can be thought of logically as a buffer for data to be processed combined with tasks for processing that data, with the "dataflow block" managing both. In its most basic usage, we can instantiate an ActionBlock<TInput> and "post" data to it; the delegate provided at the ActionBlock's construction will be executed asynchronously for every piece of data posted:

```
var ab = new ActionBlock<TInput>(delegate(TInput i)
{
    Compute(i);
});
…
ab.Post(1);
ab.Post(2);
ab.Post(3);
```

Another common entry point into TPL Dataflow is the BufferBlock<T> type, which can be thought of as an unbounded buffer for data that enables synchronous and asynchronous producer/consumer scenarios. For example, combined with new language features for asynchrony in C# and Visual Basic, we can write code to asynchronously pass data between multiple producers and consumers:

```
private static BufferBlock<int> m_buffer = new BufferBlock<int>();

// Producer
private static void Producer()
{
    while(true)
    {
        int item = Produce();
        m_buffer.Post(item);
    }
}

// Consumer
private static async Task Consumer()
{
    while(true)
    {
        int item = await m_buffer.ReceiveAsync();
        Process(item);
    }
```

```
}
```

Foundational Features

TPL Dataflow is comprised of "dataflow blocks," data structures that buffer, process, and propagate data. They can be either sources, targets, or both, in which case they're referred to as propagators.

Interfaces

All dataflow blocks implement the IDataflowBlock interface:

```
public interface IDataflowBlock
{
    void Complete();
    void Fault(Exception error);
    Task Completion { get; }
}
```

This interface serves three primary purposes. First, it helps to identify a class as a dataflow block, making it possible to store a collection of blocks regardless of their purpose in life as a source or a target. Second, it enables any block to receive a request to be shutdown, either successfully in the form of a call to the Complete method, or in fault through a call to the Fault method. Finally, it ensures that every dataflow block exposes a System.Threading.Tasks.Task instance that represents the block's complete processing asynchronously. This Task may be used to determine whether a given dataflow block has completed, and if it has, whether it completed in a successful, canceled, or faulted state. Due to the compositional nature of Task, this enables consuming code to await the completion of a block, to be notified when one of or all of many blocks have completed, to perform additional operations when an individual block completes, and so on.

While all blocks implement IDataflowBlock, a block that only implements this interface isn't very useful. Several interfaces that inherit from IDataflowBlock are provided and deliver much more value. First, all source blocks implement the ISourceBlock<TOutput> interface, which itself inherits from IDataflowBlock:

```
public interface ISourceBlock<out TOutput> : IDataflowBlock
{
    bool TryReceive(out TOutput item, Predicate<TOutput> filter);
    bool TryReceiveAll(out IList<TOutput> items);

    IDisposable LinkTo(ITargetBlock<TOutput> target, bool unlinkAfterOne);

    bool ReserveMessage(
        DataflowMessageHeader messageHeader, ITargetBlock<TOutput> target);
    TOutput ConsumeMessage(
        DataflowMessageHeader messageHeader, ITargetBlock<TOutput> target,
        out bool messageConsumed);
    void ReleaseReservation(
        DataflowMessageHeader messageHeader, ITargetBlock<TOutput> target);
}
```

By implementing ISourceBlock<TOutput>, all sources provide several key kinds of functionality. Sources have the ability to be linked to targets. This enables "dataflow networks" to be built, where blocks may be connected to one another such that sources automatically and asynchronously propagate any data

they contain to targets. Sources may be linked to zero or more targets, targets may be linked from zero or more sources, and these linkings may be added and removed dynamically at run time, with all thread-safety aspects handled by the implementation of the dataflow block (blocks may be used by any number of threads concurrently). When sources push data to targets, the protocol employed may allow the target to simply accept and keep the offered data, or it may require the target to communicate back to the source. In doing so, sources and targets may participate in a two-phase commit protocol whereby a target can reserve data from a source prior to consuming it. This allows a target to atomically consume data from multiple sources, and is enabled by the ReserveMessage, ConsumeMessage, and ReleaseReservation APIs (if you're familiar with transactions, you can logically think of these as being related to "prepare," "commit," and "rollback," respectively).

Targets provide a relatively simple interface. As with ISourceBlock<TOutput>, ITargetBlock<TInput> implements IDataflowBlock:

```
public interface ITargetBlock<in TInput> : IDataflowBlock
{
    DataflowMessageStatus OfferMessage(
        DataflowMessageHeader messageHeader, TInput messageValue,
        ISourceBlock<TInput> source, bool consumeToAccept);
}
```

As mentioned when discussing source blocks, sources and targets engage in a protocol for transferring messages between them. The OfferMessage method is the mechanism through which a source block passes messages to a target. The OfferMessage implementation decides whether or not to accept the message, and potentially calls back to methods on the source block. It also returns a DataflowMessageStatus that provides status on the offering of a message. If the target block accepted the message and now has ownership of it, it'll return Accepted; conversely, if it wants nothing to do with the message and leaves ownership with the source, it'll return Declined. Other values are possible as well: The target may decide that it potentially wants the message in the future but can't use it now, in which case it may return Postponed, indicating to the source that the target may come back to it later (through the aforementioned methods like ConsumeMessage) to attempt to consume the message. Or the target may inform the source that, not only is it declining this message, it will be permanently declining all future messages, and the source should stop pestering it with future offered messages.

Blocks can be both sources and targets; in fact, such "propagator" blocks are the most common kind of block. Propagators are represented by the IPropagatorBlock<TInput,TOutput> interface, which simply inherits from the aforementioned interfaces without adding any additional members:

```
public interface IPropagatorBlock<in TInput, out TOutput> :
    ITargetBlock<TInput>, ISourceBlock<TOutput> { }
```

Finally, sources may also provide additional capabilities around supporting the extraction of their data. A block may implement the IReceivableSourceBlock<TOutput> interface, which inherits from ISourceBlock<TOutput>, in order to support TryReceive and TryReceiveAll methods. A consumer may call TryReceive or TryReceiveAll in order to extract a single item or all available data atomically. When

extracting a single item, a filter may optionally also be specified that enables the source to atomically determine whether or not to remove and provide the next item available.

```
public interface IReceivableSourceBlock<TOutput> : ISourceBlock<TOutput>
{
    bool TryReceive(out TOutput item, Predicate<TOutput> filter);
    bool TryReceiveAll(out IList<TOutput> items);
}
```

Built-in Dataflow Blocks

While implementing custom dataflow blocks is supported, TPL Dataflow strives to provide the most common and needed blocks built-in such that most developers should not need to build custom blocks. The built-in blocks fall into three categories: pure buffering blocks, whose primary purposes in life are to buffer data in various ways; execution blocks, which execute user-provided code in response to data provided to the block; and grouping blocks, which group data across one or more sources and under various constraints.

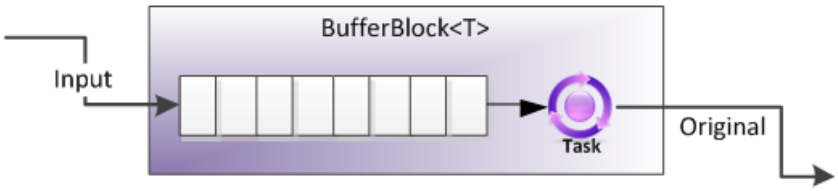
Pure Buffering Blocks

BufferBlock<T>

Arguably, the most fundamental block in TPL Dataflow is a propagator known as BufferBlock<T> (if you're familiar with the CCR, think of BufferBlock<T> as being the direct equivalent of Port<T>, or if you're familiar with the native Asynchronous Agents library in Visual C++ 2010, think of BufferBlock<T> as being the direct equivalent of unbounded_buffer<T>.) In short, BufferBlock<T> provides an unbounded or bounded buffer for storing instances of T. You can "post" instances of T to the block, which cause the data being posted to be stored in a first-in-first-out (FIFO) order by the block. You can "receive" from the block, which allows you to synchronously or asynchronously obtain instances of T previously stored or available in the future (again, FIFO). You can configure the block to asynchronously push any available data to target blocks to which it's linked (more on this momentarily). You can get a Task that represents the lifetime of the block. You can inform the buffer that it won't receive any further data. And so on.

BufferBlock<T>'s whole raison d'être is to buffer an arbitrary number of messages provided to it, and to make that data available later for consumption, either by code explicitly going to the buffer to retrieve the data, or by more actively pushing it to interested and listening parties. In this manner, it is a core primitive used to build agent-based systems that communicate through message passing.

BufferBlock<T> will hand out an individual element to only one receiver. If multiple targets are linked from the buffer, it will offer a message to each in turn, allowing only one to consume each. And if code manually receives from the buffer, doing so will remove the received element from the collection.



BufferBlock<T> Examples

Synchronous Producer/Consumer

```
// Hand-off through a BufferBlock<T>
private static BufferBlock<int> m_buffer = new BufferBlock<int>();

// Producer
private static void Producer()
{
    while(true)
    {
        int item = Produce();
        m_buffer.Post(item);
    }
}

// Consumer
private static void Consumer()
{
    while(true)
    {
        int item = m_buffer.Receive();
        Process(item);
    }
}

// Main
public static void Main()
{
    var p = Task.Factory.StartNew(Producer);
    var c = Task.Factory.StartNew(Consumer);
    Task.WaitAll(p,c);
}
```

Asynchronous Producer/Consumer

```
// Hand-off through a BufferBlock<T>
private static BufferBlock<int> m_buffer = new BufferBlock<int>();

// Producer
private static void Producer()
{
    while(true)
    {
        int item = Produce();
        m_buffer.Post(item);
    }
}

// Consumer
private static async Task Consumer()
```

```
{
    while(true)
    {
        int item = await m_buffer.ReceiveAsync();
        Process(item);
    }
}

// Main
public static void Main()
{
    var p = Task.Factory.StartNew(Producer);
    var c = Consumer();
    Task.WaitAll(p,c);
}
```

Asynchronous Producer/Consumer with a Throttled Producer

```
// Hand-off through a bounded BufferBlock<T>
private static BufferBlock<int> m_buffer = new BufferBlock<int>(
    new DataflowBlockOptions { BoundedCapacity = 10 });

// Producer
private static async void Producer()
{
    while(true)
    {
        await m_buffer.SendAsync(Produce());
    }
}

// Consumer
private static async Task Consumer()
{
    while(true)
    {
        Process(await m_buffer.ReceiveAsync());
    }
}

// Start the Producer and Consumer
private static async Task Run()
{
    await Task.WhenAll(Producer(), Consumer());
}
```

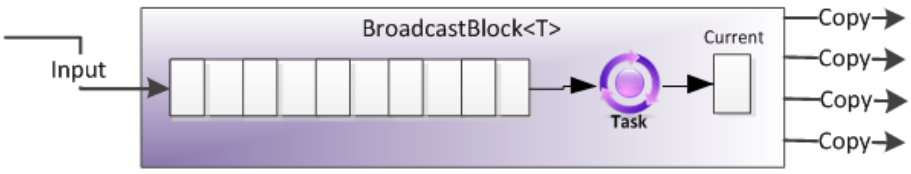
Exposing Ports from an Agent Type

```
public class MyAgent
{
    public MyAgent()
    {
        IncomingMessages = new BufferBlock<int>();
        OutgoingMessages = new BufferBlock<string>();
        Run();
    }

    public ITargetBlock<int>    IncomingMessages { get; private set; }
    public ISourceBlock<string> OutgoingMessages { get; private set; }
```

```
    private async void Run()
    {
        while(true)
        {
            int message = await IncomingMessages.ReceiveAsync();
            OutgoingMessages.Post(message.ToString());
        }
    }
}
…
MyAgent ma = new MyAgent();
ma.IncomingMessages.Post(42);
string result = ma.OutgoingMessages.Receive();
```

BroadcastBlock<T>

Unlike BufferBlock<T>, BroadcastBlock<T>'s mission in life is to enable all targets linked from the block to get a copy of every element published, continually overwriting the "current" value with those propagated to it. When data arrives at the block, all linked targets will be offered a copy of the data, where that copy is defined by a user-provided Func<T,T> cloning function (a function may be omitted for the original instance to be forwarded). Additionally, unlike BufferBlock<T>, BroadcastBlock<T> doesn't hold on to data unnecessarily. After a particular datum has been offered to all targets, that element will be overwritten by whatever piece of data is next in line (as with all dataflow blocks, messages are handled in FIFO order). That element will be offered to all targets, and so on. If no targets are linked from a BroadcastBlock<T>, the data will effectively be dropped. However, BroadcastBlock<T> always maintains the last piece of data it received for anyone that might be interested in consuming it; if target blocks are subsequently linked from a broadcast block, the target will be pushed that last piece of saved data. In this manner, BroadcastBlock<T> may be thought of as an overwrite buffer (in fact, its equivalent in the Visual C++ 2010 asynchronous agents library is called overwrite_buffer<T>).



BroadcastBlock<T> Examples

Save Images to Disk and Display them in the UI

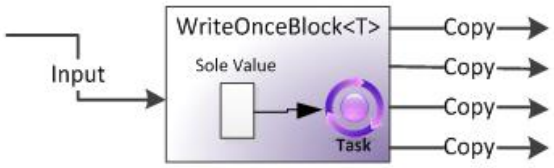
```
var ui = TaskScheduler.FromCurrentSynchronizationContext();
var bb = new BroadcastBlock<ImageData>(i => i);

var saveToDisk = new ActionBlock<ImageData>(item =>
    item.Image.Save(item.Path));

var showInUi = new ActionBlock<ImageData>(item =>
    imagePanel.AddImage(item.Image),
    new DataflowBlockOptions { TaskScheduler=ui });

bb.LinkTo(saveToDisk);
bb.LinkTo(showInUi);
```

Exposing Status from an Agent

```
public class MyAgent
{
    public MyAgent()
    {
        Status = new BroadcastBlock<string>();
        Run();
    }

    public ISourceBlock<string> Status { get; private set; }

    private void Run()
    {
        Status.Post("Starting");
        …
        Status.Post("Doing cool stuff");
        …
        Status.Post("Done");
    }
}
```

WriteOnceBlock<T>

If BufferBlock<T> is the most fundamental block in TPL Dataflow, WriteOnceBlock<T> is the simplest. It stores at most one value, and once that value has been set, it will never be replaced or overwritten. As with BroadcastBlock<T>, all consumers may obtain a copy of the stored value. You can think of WriteOnceBlock<T> in TPL Dataflow as being similar to a readonly member variable in C#, except instead of only being settable in a constructor and then being immutable, it's only settable once and is then immutable. If you're familiar with Visual C++ 2010's asynchronous agents library, WriteOnceBlock<T> is the equivalent of the single_assignment<T> type.



WriteOnceBlock<T> Examples

ReceiveFromAny

```
public T ReceiveFromAny<T>(params ISourceBlock<T> [] sources)
{
    var wob = new WriteOnceBlock<T>();
    foreach(var source in sources) source.LinkTo(wob, unlinkAfterOne:true);
    return wob.Receive();
}
```

A Dataflow "Const" of T

```
private readonly ISourceBlock<T> c_item = new WriteOnceBlock<T>(i => i);
…
// Code that ensures c_item is initialized
c_item.Post(/* lazily retrieved value */);
…
// Code that relies on c_item always being the same
```

```
T value = await c_item.ReceiveAsync();
```

Splitting a Task's Potential Outputs

```
public static async void SplitIntoBlocks(this Task<T> task,
    out IPropagatorBlock<T> result,
    out IPropagatorBlock<Exception> exception)
{
    result = new WriteOnceBlock<T>(i => i);
    exception = new WriteOnceBlock<Exception>(i => i);

    try { result.Post(await task); }
    catch(Exception exc) { exception.Post(exc); }
}
```

Request/Response

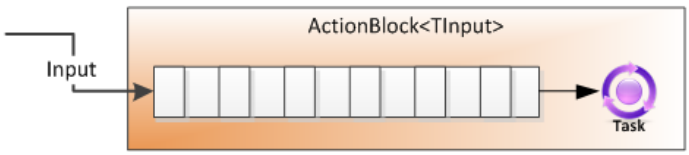
```
// with WriteOnceBlock<T>
var request = …;
var response = new WriteOnceBlock<TResponse>();
target.Post(Tuple.Create(request, response));
var result = await response.ReceiveAsync();

// with TaskCompletionSource<T>
var request = …;
var response = new TaskCompletionSource<TResponse>();
target.Post(Tuple.Create(request, response));
var result = await response.Task;
```

Executor Blocks

ActionBlock<TInput>

ActionBlock<TInput> enables the execution of a delegate to perform some action for each input datum. This delegate can be an Action<TInput>, in which case processing of that element is considered completed when the delegate returns, or it can be a Func<TInput,Task>, in which case processing of that element is considered completed not when the delegate returns but when the returned Task completes. This enables an ActionBlock<TInput> to be used with both synchronous and asynchronous methods.

Data is provided to an ActionBlock<TInput> by posting to it, just as with the previously described buffers. This input data is then buffered by the ActionBlock<TInput> until the block is ready to process it. By default, an ActionBlock<TInput> processes a single message at a time; this is to provide easy to reason about semantics around the sequential ordering of message processing, which will happen in a FIFO order. However, an ActionBlock<TInput> may be optionally configured to process multiple messages concurrently, with control over the degree of parallelism employed. For more information, see the subsequent section of this document on dataflow block options.



ActionBlock<TInput> Examples

Downloading Images Sequentially and Synchronously

```
var downloader = new ActionBlock<string>(url =>
{
    // Download returns byte[]
    byte [] imageData = Download(url);
    Process(imageData);
});

downloader.Post("http://msdn.com/concurrency");
downloader.Post("http://blogs.msdn.com/pfxteam");
```

Downloading Images Sequentially and Asynchronously

```
var downloader = new ActionBlock<string>(async url =>
{
    byte [] imageData = await DownloadAsync(url);
    Process(imageData);
});

downloader.Post("http://msdn.com/concurrency ");
downloader.Post("http://blogs.msdn.com/pfxteam");
```

Throttling Asynchronous Downloads to at most 5 Concurrently

```
var downloader = new ActionBlock<string>(async url =>
{
    byte [] imageData = await DownloadAsync(url);
    Process(imageData);
}, new DataflowBlockOptions { MaxDegreeOfParallelism = 5 });

downloader.Post("http://msdn.com/concurrency ");
downloader.Post("http://blogs.msdn.com/pfxteam");
```

Load Balancing Based on Demand Across N ActionBlocks

```
var dbo = new DataflowBlockOptions { BoundedCapacity = 1 };

var workers = (from i in Enumerable.Range(0, N)
               select new ActionBlock<int>(DoWork, dbo)).ToArray();

ISourceBlock<int> dataSource = …;
foreach(var worker in workers) dataSource.LinkTo(workers);
```

Processing on the UI Thread

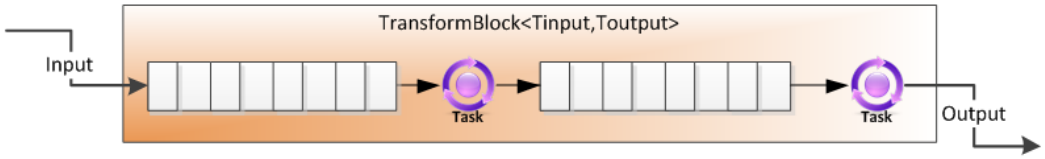
```
var ab = new ActionBlock<Bitmap>(image =>
{
    panel.Add(image);
    txtStatus.Text = "Added image #" + panel.Items.Count;
}, new DataflowBlockOptions {
    TaskScheduler = TaskScheduler.FromCurrentSynchronizationContext() });
```

TransformBlock<TInput,TOutput>

As with ActionBlock<TInput>, TransformBlock<TInput,TOutput> enables the execution of a delegate to perform some action for each input datum; unlike with ActionBlock<TInput>, this processing has an output. This delegate can be a Func<TInput,TOutput>, in which case processing of that element is considered completed when the delegate returns, or it can be a Func<TInput,Task<TOutput>>, in which case processing of that element is considered completed not when the delegate returns but when the returned Task completes. This enables a TransformBlock<TInput,TOutput> to be used with both synchronous and asynchronous methods.

A TransformBlock<TInput,TOutput> buffers both its inputs and its outputs. This output buffer behaves just like a BufferBlock<TOutput>.

As with ActionBlock<TInput>, TransformBlock<TInput,TOutput> defaults to processing one message at a time, maintaining strict FIFO ordering. And as with ActionBlock<TInput>, it may also be configured to process multiple messages concurrently. In doing so, however, it still ensures that output messages are propagated in the same order that they arrived at the block (internally, the block uses a reordering buffer to fix up any out-of-order issues that might arise from processing multiple messages concurrently).



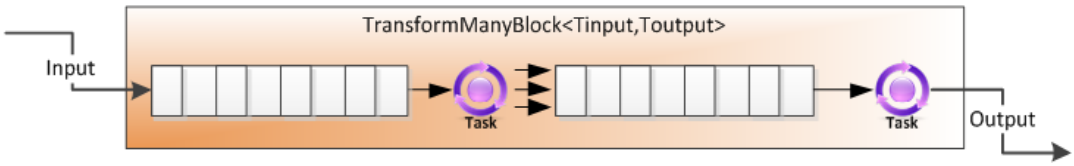
TransformBlock<TInput,TOutput> Examples

A Concurrent Pipeline

```
var compressor = new TransformBlock<byte[],byte[]>(
    input => Compress(input));
var encryptor = new TransformBlock<byte[],byte[]>(
    input => Encrypt(input));
compressor.LinkTo(Encryptor);
```

TransformManyBlock<TInput,TOutput>

TransformManyBlock<TInput,TOutput> is very similar to TransformBlock<TInput,TOutput>. The key difference is that whereas a TransformBlock<TInput,TOutput> produces one and only one output for each input, TransformManyBlock<TInput,TOutput> produces any number (zero or more) outputs for each input. As with ActionBlock<TInput> and TransformBlock<TInput,TOutput>, this processing may be specified using delegates, both for synchronous and asynchronous processing. A Func<TInput,IEnumerable<TOutput>> is used for synchronous, and a Func<TInput, Task<IEnumerable<TOutput>>> is used for asynchronous. As with both ActionBlock<TInput> and TransformBlock<TInput,TOutput>, TransformManyBlock<TInput,TOutput> defaults to sequential processing, but may be configured otherwise.



TransformManyBlock<TInput,TOutput> Examples

Asynchronous Web Crawler

```
var downloader = new TransformManyBlock<string,string>(async url =>
{
    Console.WriteLine("Downloading " + url);
    try { return ParseLinks(await DownloadContents(url)); } catch{}
    return Enumerable.Empty<string>();
});
downloader.LinkTo(downloader);
```

Expanding An Enumerable Into Its Constituent Elements

```
var expanded = new TransformManyBlock<T[],T>(array => array);
```

Filtering by going from 1 to 0 or 1 elements

```
public IPropagatorBlock<T> CreateFilteredBuffer<T>(Predicate<T> filter)
{
    return new TransformManyBlock<T,T>(item =>
        filter(item) ? new [] { item } : Enumerable.Empty<T>());
}
```

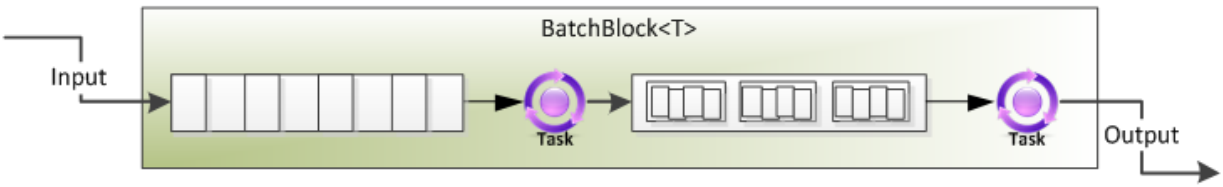
Joining Blocks

BatchBlock<T>

BatchBlock<T> combines N single items into one batch item, represented as an array of elements; in target/source terms, BatchBlock<T> is an ITargetBlock<T> and an ISourceBlock<T[]>. An instance is created with a specific batch size, and the block then creates a batch as soon as it's received that number of elements, asynchronously outputting the batch to the output buffer. If the block is told no more data will arrive while it still has data pending to form a batch, its last batch will contain the remaining input items. Code may also explicitly force a BatchBlock<T> to create a batch from however many items it currently has buffered, even if that's fewer than the batch size.

BatchBlock<T> is capable of executing in both greedy and non-greedy modes. In the default greedy mode, all messages offered to the block from any number of sources are accepted and buffered to be converted into batches. In non-greedy mode, all messages are postponed from sources until enough sources have offered messages to the block to create a batch. Thus, a BatchBlock<T> can be used to receive 1 element from each of N sources, N elements from 1 source, and a myriad of options in between.

BatchBlock<T> is also capable of tracking the number of batches that have been created. When configured as such, after receiving enough data to create a number of batches equal to that specified to its constructor, BatchBlock<T> will decline all further data offered to it.



BatchBlock<T> Examples

Batching Requests into groups of 100 to Submit to a Database

```
var batchRequests = new BatchBlock<Request>(batchSize:100);
var sendToDb = new ActionBlock<Request[]>(reqs => SubmitToDatabase(reqs));
batchRequests.LinkTo(sendToDb);
```

Creating a batch once a second

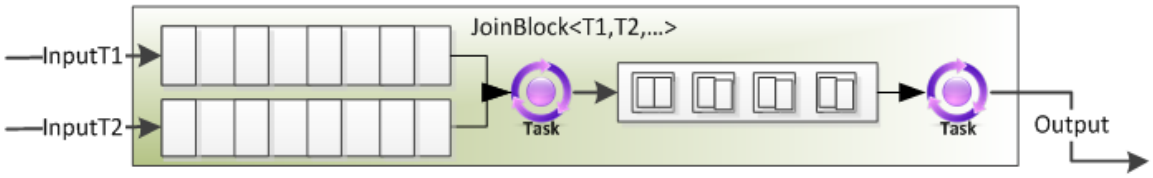
```
var batch = new BatchBlock<T>(batchSize:Int32.MaxValue);
new Timer(delegate { batch.TriggerBatch(); }).Change(1000, 1000);
```

JoinBlock<T1,T2,…>

Like BatchBlock<T>, JoinBlock<T1,T2,…> is able to group data from multiple data sources. In fact, that's JoinBlock<T1,T2,…>'s primary purpose. It itself is not an ITargetBlock<T> but instead exposes as properties targets of its input types. For example, a JoinBlock<string,double,int> will expose three properties of type ITargetBlock<string>, ITargetBlock<double>, and ITargetBlock<int>. The JoinBlock<T1,T2,…> coordinates across all of these inputs to create output tuples that contain one element from each input target. For example, the aforementioned JoinBlock<string,double,int> is itself an ISourceBlock<Tuple<string,double,int>>.

As with BatchBlock<T>, JoinBlock<T1,T2,…> is capable of operating in both greedy and non-greedy mode. In the default greedy mode, all data offered to targets are accepted, even if the other target doesn't have the necessary data with which to form a tuple. In non-greedy mode, the block's targets will postpone data until all targets have been offered the necessary data to create a tuple, at which point the block will engage in a two-phase commit protocol to atomically retrieve all necessary items from the sources. This postponement makes it possible for another entity to consume the data in the meantime so as to allow the overall system to make forward progress.

TPL Dataflow currently provides built-in implementations for two generic arities: JoinBlock<T1,T2> and JoinBlock<T1,T2,T3>.



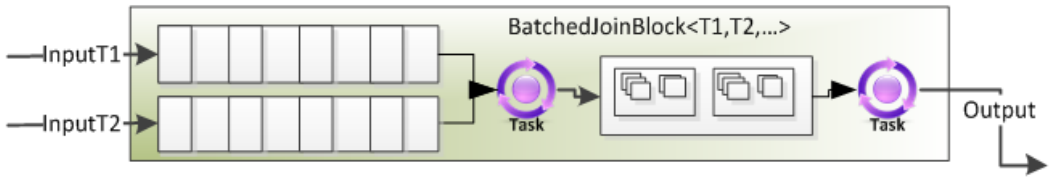
JoinBlock<T1,T2,..> Examples

Processing Requests with a Limited Number of Pooled Objects

```
var throttle = new JoinBlock<ExpensiveObject,Request>();
for(int i=0; i<10; i++)
    requestProcessor.Target1.Post(new ExpensiveObject());
var processor =
    new Transform<Tuple<ExpensiveObject,Request>,ExpensiveObject>(pair =>
{
    var request = pair.Item2;
    var resource = pair.Item1;
    request.ProcessWith(resource);
    return resource;
});
throttle.LinkTo(processor);
processor.LinkTo(throttle.Target1);
```

BatchedJoinBlock<T1,T2,...>

BatchedJoinBlock<T1,T2,...> is in a sense a combination of BatchBlock<T> and JoinBlock<T1,T2,...>. Whereas JoinBlock<t1,T2,...> is used to aggregate one input from each target into a tuple, and BatchBlock<T> is used to aggregate N inputs into a collection, BatchedJoinBlock<T1,T2,...> is used to gather N inputs from across all of the targets into tuples of collections. For example, consider a scatter/gather problem where N operations are launched, some of which may succeed and produce string outputs, and others of which may fail and produce Exceptions. We can create a BatchedJoinBlock<string,Exception> to accept N results, with all of the strings posted to its first target and all of the exceptions posted to its second. When all N results arrive, the BatchedJoinBlock<string,Exception>, which is itself an ISourceBlock<Tuple<IList<string>,IList<Exception>>>, will gather all of the strings into an IList<string>, all of the exceptions into an IList<Exception>, and output both lists together as a single tuple.



BatchedJoinBlock<T1,T2,..> Examples

Scatter/Gather

```
var batchedJoin = new BatchedJoinBlock<string,Exception>(10);
for(int i=0; i<10; i++)
{
    Task.Factory.StartNew(() => {
        try { batchedJoin.Target1.Post(DoWork()); }
        catch(Exception e) { batchJoin.Target2.Post(e); }
```

```
    });
}
var results = await batchedJoin.ReceiveAsync();
foreach(string s in results.Item1) Console.WriteLine(s);
foreach(Exception e in results.Item2) Console.WriteLine(e);
```

Beyond the Basics

Configuration Options

The built-in dataflow blocks are configurable, with a wealth of control provided over how and where blocks perform their work. Here are some key knobs available to the developer, all of which are exposed through the DataflowBlockOptions class and its derived types (ExecutionDataflowBlockOptions and GroupingDataflowBlockOptions), instances of which may be provided to blocks at construction time.

TaskScheduler

Blocks schedule tasks to perform their underlying work, whether that work is running a user-provided delegate for each piece of data received, propagating data from a source to a target, or retrieving previously offered data from a source. This work all runs on a System.Threading.Tasks.TaskScheduler, the core representation for an execution resource in TPL. By default, dataflow blocks schedule work to TaskScheduler.Default, which targets the internal workings of the .NET ThreadPool. Developers may override this on per-block bases by providing the block with an instance of the abstract TaskScheduler class; the block will use that TaskScheduler for all scheduled work.

The .NET Framework 4 included two built-in schedulers, the default for targeting the ThreadPool, and a TaskScheduler that targets a SynchronizationContext. This latter scheduler may be used to target a dataflow block to run all of its work on the UI thread, enabling easy marshaling of dataflows through the UI for relevant needs. Additionally, System.Threading.Tasks.Dataflow.dll includes a ConcurrentExclusiveSchedulerPair type, which exposes a “concurrent” TaskScheduler and an “exclusive” TaskScheduler. This pair of schedulers cooperates to ensure that any number of tasks scheduled to the concurrent scheduler may run concurrently as long as no exclusive tasks are executing; as soon as an exlusive task is executing, however, no other tasks, concurrent or exclusive, may be executing. In effect, ConcurrentExclusiveSchedulerPair provides an asynchronous reader/writer lock that schedules at the Task level, and dataflow blocks may target one of these schedulers.

Further, the TaskScheduler abstraction is extensible, with much potential functionality available for end-users to implement. The Parallel Extensions Extras project available for download from http://code.msdn.microsoft.com/ParExtSamples includes close to a dozen TaskScheduler implementations.

MaxDegreeOfParallelism

By default, an individual dataflow block processes only one message at a time, queueing all not-yet-processed messages so that they may be processed when the currently processing message is completed. This ensures behavior that’s easily reasoned about by the developer, with ordering guarantees maintained. However, there are cases where a developer wants to process messages as fast

as possible, and is ok with the relaxation that multiple messages be processed in parallel. To achieve this, the MaxDegreeOfParallelism option is available. It defaults to 1, meaning only one thing may happen in a block at a time. If set to a value higher than 1, that number of messages may be processed concurrently by the block. If set to DataflowBlockOptions.Unbounded (-1), any number of messages may be processed concurrently, with the maximum automatically managed by the underlying scheduler targeted by the dataflow block.

Note that MaxDegreeOfParallelism is a maximum, not a requirement. Due to either the functional semantics of a given dataflow block or due to available system resources, a block may choose to execute with a lower degree of parallelism than specified. A block will never execute with a higher degree of parallelism. Also note that MaxDegreeOfParallelism applies to an individual block, so even if set to 1, multiple dataflow blocks may be processing in parallel. A TaskScheduler targeted by multiple blocks may be used to enforce policy across multiple blocks. For example, if two dataflow blocks are each configured to target the exclusive scheduler of a single ConcurrentExclusiveSchedulerPair instance, all work across both blocks will be serialized. Similarly, if both blocks are configured to target the concurrent scheduler of a single ConcurrentExclusiveSchedulerPair, and that pair has been configured with a maximum concurrency level, all work from both blocks will be limited in total to that number of concurrent operations.

MaxMessagesPerTask

TPL Dataflow is focused on both efficiency and control. Where there are necessary trade-offs between the two, the system strives to provide a quality default but also enable the developer to customize behavior according to a particular situation. One such example is the trade-off between performance and fairness. By default, dataflow blocks try to minimize the number of task objects that are necessary to process all of their data. This provides for very efficient execution; as long as a block has data available to be processed, that block's tasks will remain to process the available data, only retiring when no more data is available (until data is available again, at which point more tasks will be spun up). However, this can lead to problems of fairness. If the system is currently saturated processing data from a given set of blocks, and then data arrives at other blocks, those latter blocks will either need to wait for the first blocks to finish processing before they're able to begin, or alternatively risk oversubscribing the system. This may or may not be the correct behavior for a given situation. To address this, the MaxMessagesPerTask option exists. It defaults to DataflowBlockOptions.Unbounded (-1), meaning that there is no maximum. However, if set to a positive number, that number will represent the maximum number of messages a given block may use a single task to process. Once that limit is reached, the block must retire the task and replace it with a replica to continue processing. These replicas are treated fairly with regards to all other tasks scheduled to the scheduler, allowing blocks to achieve a modicum of fairness between them. In the extreme, if MaxMessagesPerTask is set to 1, a single task will be used per message, achieving ultimate fairness at the potential expense of more tasks than may otherwise have been necessary.

MaxNumberOfGroups

The grouping blocks are capable of tracking how many groups they've produced, and automatically complete themselves (declining further offered messages) after that number of groups has been

generated. By default, the number of groups is DataflowBlockOptions.Unbounded (-1), but it may be explicitly set to a value greater than one.

CancellationToken

Cancellation is an integral part of any parallel system. To aid with incorporating cancellation into applications, .NET 4 saw the introduction of CancellationToken, a type that represents a cancellation request. Tasks, parallel loops, and coordination primitives may accept cancellation tokens and monitor them for cancellation requests; TPL Dataflow supports the same with dataflow blocks. When a dataflow block is constructed, it may be provided with a CancellationToken. This token is monitored during the dataflow block's lifetime. If a cancellation request arrives prior to the block's completion, the block will cease operation as politely and quickly as possible. In doing so, it will allow currently executing functions to finish but will prevent additional work from starting, and will then complete the dataflow block even if the block had data available to be processed. Data will be purged from the block, the block will drop all connections to sources and targets, and the block will transition to a Canceled state (such that its CompletionTask will be in the Canceled state, unless in the meantime an exception occurred in the block's processing, in which case it'll be in the Faulted state). In this fashion, CancellationToken provides an efficient means by which an entire network of dataflow block processing may be canceled via a single call to the token's associated CancellationTokenSource's Cancel method.

Greedy

By default, target blocks are greedy and want all data offered to them. In the case of JoinBlock<T1,T2,…>, this means accepting data even if the corresponding data with which to join isn't yet available. In the case of BatchBlock<T>, this means accepting data and buffering it even if it hasn't yet been offered all of the data necessary to complete a batch.

This greedy behavior can be beneficial for performance, but it can also have its downsides. Consider a single BufferBlock<T1> linked to two different JoinBlock<T1,T2> instances. The buffer will offer data it receives to the first join block, which will always accept and buffer the data; as a result, the second join may never be satisfied, unless it has another source or input supplying instances of T1. And even if it does have such sources, they may not be as forthcoming with data, such that the join has to wait longer to be satisfied. To account for this, join blocks can be configured to be non-greedy. When in non-greedy mode, join blocks postpone all offered messages, thereby leaving the offered data in the source block (and allowing the source block to in turn offer the data to others). Only when the join block has been offered all of the data it would need to satisfy a join does it then go back to the relevant sources to reserve and consume the data. This is done using a two-phase commit protocol in order to ensure that data is only consumed if all of the data will be consumed.

BoundedCapacity

The majority of the dataflow blocks included in System.Threading.Tasks.Dataflow.dll support the specification of a bounded capacity. This is the limit on the number of items the block may be storing and have in flight at any one time. By default, this value is initialized to DataflowBlockOptions.Unbounded (-1), meaning that there is no limit. However, a developer may

explicitly specify an upper bound. If a block is already at its capacity when an additional message is offered to it, that message will be postponed.

This bounded support drives several significant scenarios. First and foremost, bounding is useful in a dataflow network to avoid unbounded memory growth. This can be very important for reliability reasons if there's a possibility that producers could end up generating data much faster than the consumers could process it. As an example of this, consider a scenario where image processing is being done on images flying off of a web cam. If the frame rate is too fast for the processing to keep up with, the desired behavior is that frames are dropped so that, while choppy, the current image reflects the present time rather than images from the camera from minutes ago. Logically, this could be achieved by linking a BroadcastBlock<Frame> to an ActionBlock<Frame>. However, an unbounded action block will itself buffer all data offered to it while that data awaits processing, which means that the broadcast block won't adequately perform its intended duty, which is to drop frames that aren't needed. Instead, the ActionBlock<Frame> may be configurded with a bounded capacity of one, such that it will postpone offered messages while it currently has a message buffered or being processed. Only when it is done its current processing will it then go back to the source and ask for the previously offered and postponed message. In the case of BroadcastBlock<Frame>, the broadcast block will drop frames when new ones arrive, so the ActionBlock<Frame> will always be able to pick up the latest frame rather than having to sort through leagues of buffered ones. This will not only help with ensuring the latest frame is a current one, it will also help to alleviate the memory pressure that results from having to store all of that data unnecessarily.

Finally, consider an attempt at parallelization by creating multiple TransformBlock<TInput,TOutput> instances, all of which are linked from a single BufferBlock<TOutput>. A BufferBlock<TOutput> offers data to targets in the order those targets were registered. If all of the transform blocks are configured to be unbounded (the default), all of the buffer's data will be offered to the first transform block, which will greedily accept and buffer all of it. This will lead to poor parallelization as all of the other transforms sit idly by. Instead, all of the transforms may be configured to have a small bounded capacity. As a result, they will postpone offered messages while full, and only go back to the source to consume a message when the transform block is available to do processing. In this way, fine-grained load balancing is achieved automatically across all of the target transforms, with each grabbing the next piece of data automatically when the block is ready for more.

Static Methods

Beyond the functionality exposed directly from the ISourceBlock<TOutput> and ITargetBlock<TInput> interfaces, there is a set of common operations developers often employ when working with dataflow blocks. Such additional operations are exposed as methods from the DataflowBlock class, built on top of the functionality exposed by the interfaces, and include:

- Choose
    - Choose allows multiple sources to be specified along with an action delegate for each. It will atomically accept one and only one message across all of the sources, executing the associated action delegate with the received data.

- Encapsulate
    - Creates a propagator block out of a target block and a source block.
- LinkTo
    - The ISourceBlock<TOutput> interface provides a LinkTo method for connecting the source to a target, but extension methods are provided for additional control over linking. Such functionality includes being able to add a filter predicate to a link in order to control what data is propagated across what links and what the behavior should be if the predicate is not met (e.g. should such a message simply be dropped, should such a message be declined and offered to other targets, etc.)
- OutputAvailableAsync
    - An extension method that asynchronously informs a consumer when data is available on a source block or when no more data will be available. Data is not removed from the source.
- Post
    - An extension method that asynchronously posts to the target block. It returns immediately whether the data could be accepted or not, and it does not allow for the target to consume the message at a later time.
- SendAsync
    - An extension method that asynchronously sends to target blocks while supporting buffering. A Post operation on a target is asynchronous, but if a target wants to postpone the offered data, there is nowhere for the data to be buffered and the target must instead be forced to decline. SendAsync enables asynchronous posting of the data with buffering, such that if a target postpones, it will later be able to retrieve the postponed data from the temporary buffer used for this one asynchronously posted message.
- Receive
    - An extension method that aynchronously receives data from a source, blocking until data is available or the block informs the operation that no data will be available. The blocking may be timed out or canceled through parameters passed to the Receive method.
- ReceiveAsync
    - The same as Receive, except asynchronously. Rather than returning instances of TOutput, ReceiveAsync returns instances of Task<TOutput>.
- AsObservable
    - An extension method that creates an IObservable<T> for an ISourceBlock<T>. Any IObserver<T> instances subscribed to the IObservable<T> will be broadcast data removed from the source block.
- AsObserver
    - An extension method that creates an IObserver<T> for an ITargetBlock<T>. OnNext calls on the observer will result in the data being sent to the target, OnError calls will result in

the exception faulting the target, and OnCompleted calls will result in Complete called on the target.

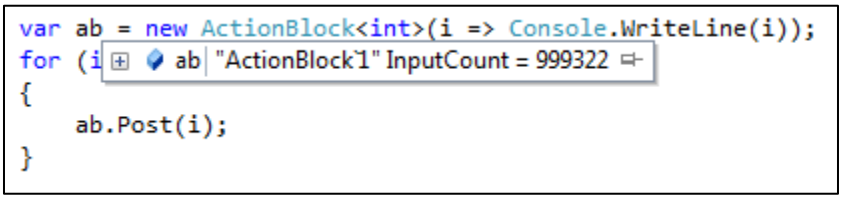
There are a plethora of interesting extension methods that could be built on top of the ISourceBlock<TOutput> and ITargetBlock<TInput> interfaces. The methods exposed from DataflowBlock represent a common subset and are those expected to be most useful to developers building solutions that incorporate dataflow blocks.

Debugging

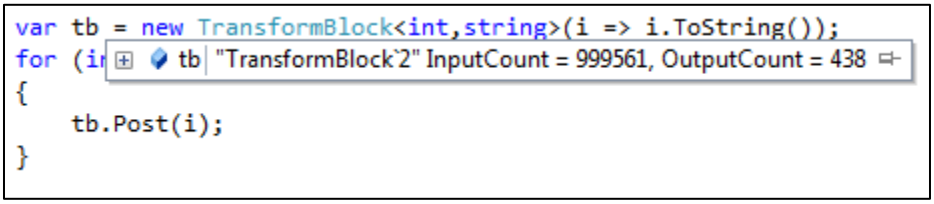
Several facilities are provided to help debug applications that utilize TPL Dataflow.

Debugger Display Attributes

All relevant types in TPL Dataflow are attributed with DebuggerDisplayAttribute in order to provide relevant information at a glance in the debugger. For example, ActionBlock<TInput> will display the number of messages buffered and waiting to be processed:

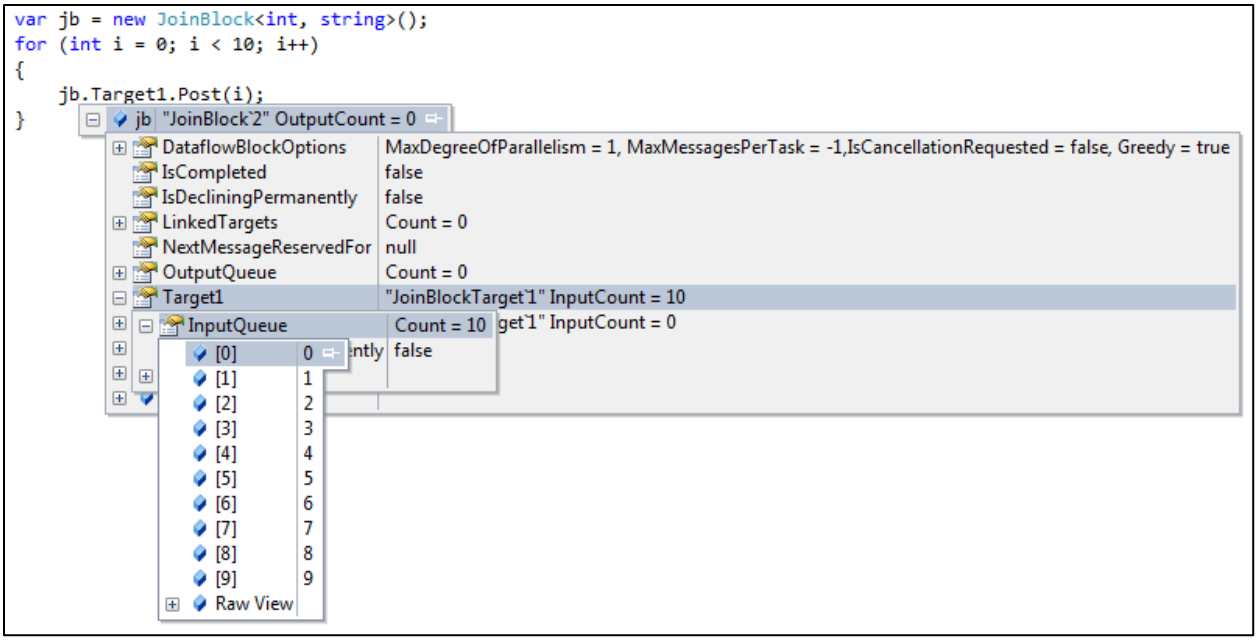


And TransformBlock<TInput,TOutput> will display the number of messages buffered both prior to being processed and after having been processed:



Debugger Type Proxies

All relevant types are fitted with debugger type proxies to elevate relevant information to the developer using dataflow blocks. This provides an easy mechanism for a developer to drill in and quickly understand the state of a dataflow block, what data it has buffered, if it's currently processing, and the like. For example, JoinBlock<T1,T2,…> provides a display like the following when its tooltip is expanded:



Developing Custom Dataflow Blocks

System.Threading.Tasks.Dataflow.dll includes the handful of blocks previously described, but the system becomes more powerful when custom dataflow blocks are developed by 3rd parties. There are a variety of ways to provide such functionality, ranging from implementing the interfaces from scratch to building blocks that layer functionality on top of existing ones.

Consider the need to have a dataflow block that generates sliding windows. As an example, if we use a window size of 3 and send in the data { 0, 1, 2, 3, 4, 5 }, this block will map from int to int[] and will output the arrays { 0, 1, 2}, {1, 2, 3}, {2, 3, 4}, and {3, 4, 5}.

There are a variety of ways we could build such a block. One straightforward approach is to utilize an ActionBlock<T> as the target half of the block, and a BufferBlock<T[]> as the source half. We can then utilize the DataflowBlockExtensions' Encapsulate method to create a propagator block from the target and source blocks:

```
private static IPropagatorBlock<T,T[]> CreateSlidingWindow(int windowSize)
{
    var queue = new Queue<T>();
    var source = new BufferBlock<T[]>();
    var target = new ActionBlock<T>(item =>
    {
        queue.Enqueue(item);
        if (queue.Count >  windowSize) queue.Dequeue();
        if (queue.Count == windowSize) source.Post(queue.ToArray());
    });
    return DataflowBlockExtensions.Encapsulate(target, source);
}
```

Taking this a step further, instead of using Encapsulate, we could build a new type ourselves that implements IPropagatorBlock<T,T[]> and that includes as fields the source and target to which it delegates:

```
public class SlidingWindowBlock<T> : IPropagatorBlock<T,T[]> {
    private readonly int m_windowSize;
    private readonly ITargetBlock<T> m_target;
    private readonly ISourceBlock<T> m_source;

    public SlidingWindowBlock(int windowSize) {
        var queue  = new Queue<T>();
        var source = new BufferBlock<T[]>();
        var target = new ActionBlock<T>(item =>
        {
            queue.Enqueue(item);
            if (queue.Count >  m_windowSize) queue.Dequeue();
            if (queue.Count == m_windowSize) source.Post(queue.ToArray());
        });
        m_windowSize = windowSize;
        m_target = target;
        m_source = source;
    }

    ...
}
```

Such an implementation affords the ability to do additional work before and after delegating to the built-in and contained blocks. Either implementation also affords the ability to connect up the blocks in other arbitrary ways. For example, with the sliding window example, consider what happens if less data is posted than the window size. In that case, the ActionBlock<T> will complete its processing without having every dumped the queue into the buffer block. To mitigate that, we can utilize the ActionBlock<T>'s Completion Task to do work when the ActionBlock<T> completes, in this case copying the queue into the buffer block if it contains fewer than the needed number of elements. We can also take this opportunity to inform the BufferBlock<T[]> that it won't be receiving any more input, such that it can also complete when all of its data is consumed:

```
target.Completion.ContinueWith(t =>
{
    if (queue.Count > 0 && queue.Count < windowSize) source.Post(queue.ToArray());
    if (t.IsFaulted) source.Fault(t.Exception);
    else source.Complete();
});
```

As was done for all of the blocks built into System.Threading.Tasks.Dataflow.dll, it's also possible simply to implement the ITargetBlock<TInput> and/or ISourceBlock<TOutput> interfaces directly, providing whatever functionality is deemed important for the custom block's needs. By correctly implementing one or more of these interfaces, custom blocks will integrate into a dataflow network just like the built-in blocks, and all extension methods (such as those defined on DataflowBlockExtensions) will also "just work".