



www.clipcode.net

Clipcode's TPL Dataflow Reference Library

Written By Eamon O'Tuathail

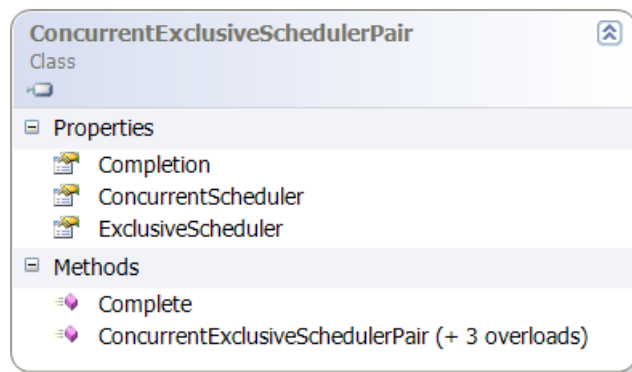
Table of Contents

1: System.Threading.Tasks.....	3
1.1: Overview.....	3
1.2: ConcurrentExclusiveSchedulerPair.....	3
2: System.Threading.Tasks.Dataflow.....	6
2.1: Overview.....	6
2.2: Interface Implementations.....	8
2.3: ActionBlock.....	8
2.4: BatchBlock.....	10
2.5: BatchedJoinBlock(T1, T2, ..).....	12
2.6: BroadcastBlock.....	13
2.7: BufferBlock.....	14
2.8: DataflowBlock.....	15
2.9: DataflowMessageHeader.....	22
2.10: DataflowMessageStatus.....	23
2.11: DataflowOptions.....	24
2.12: ExecutionDataflowOptions.....	25
2.13: GroupingDataflowOptions.....	25
2.14: IDataflowBlock.....	26
2.15: IPropagatorBlock.....	28
2.16: IReceivableSourceBlock.....	28
2.17: ISourceBlock.....	29
2.18: ITargetBlock.....	32
2.19: JoinBlock(T1, T2, ..).....	33
2.20: TransformBlock.....	35
2.21: TransformManyBlock.....	37
2.22: WriteOnceBlock.....	40

1: System.Threading.Tasks

1.1: Overview

The one type that the TPL Dataflow library introduces into the System.Threading.Tasks namespace is:



1.2: ConcurrentExclusiveSchedulerPair

Represents a pair of schedulers - one exclusive and one concurrent - and only one of which is active at any one instance (similar to reader/writer lock).

```
public class ConcurrentExclusiveSchedulerPair {
    // Constructors
    public ConcurrentExclusiveSchedulerPair();
    public ConcurrentExclusiveSchedulerPair(TaskScheduler taskScheduler);
    public ConcurrentExclusiveSchedulerPair(
        TaskScheduler taskScheduler, int maxConcurrencyLevel);
    public ConcurrentExclusiveSchedulerPair(TaskScheduler taskScheduler,
        int maxConcurrencyLevel, int maxItemsPerTask);

    // Task schedulers
    public TaskScheduler ConcurrentScheduler { get; }
    public TaskScheduler ExclusiveScheduler { get; }

    // Property and method for completion
    public Task Completion { get; }
    public void Complete();
}
```

Remarks

This type represents a pair of related task schedulers. At any particular moment, only one of them is active. When the concurrent task scheduler is active, any number of tasks scheduled with it can be running. When the exclusive task scheduler is active, one one task can be running. In this was it operates similar to a reader/writer lock.

1.2.1: ConcurrentExclusiveSchedulerPair()

Default constructor for ConcurrentExclusiveSchedulerPair.

```
public ConcurrentExclusiveSchedulerPair();
```

Remarks

The target scheduler is the default.

1.2.2: ConcurrentExclusiveSchedulerPair(TaskScheduler)

Constructor that sets the target scheduler to be that specified as the parameter.

```
public ConcurrentExclusiveSchedulerPair(TaskScheduler taskScheduler);
```

1.2.3: ConcurrentExclusiveSchedulerPair(TaskScheduler, int)

Constructor that sets the target scheduler and the max concurrency to be as specified in the parameters.

```
public ConcurrentExclusiveSchedulerPair(
    TaskScheduler taskScheduler, int maxConcurrencyLevel);
```

1.2.4: ConcurrentExclusiveSchedulerPair(TaskScheduler, int, int)

Constructor that sets the target scheduler, the max concurrency and the max items per task to be as specified in the parameters.

```
public ConcurrentExclusiveSchedulerPair(TaskScheduler taskScheduler,
    int maxConcurrencyLevel, int maxItemsPerTask);
```

1.2.5: ConcurrencyScheduler

The task scheduler used for concurrent operations.

```
public TaskScheduler ConcurrencyScheduler { get; }
```

1.2.6: ExclusiveScheduler

The task scheduler used for exclusive operations.

```
public TaskScheduler ExclusiveScheduler { get; }
```

1.2.7: Completion

A task that represents completion of the pair.

```
Task Completion { get; }
```

Remarks

This task represents the completion status of the pair of schedulers and is used in determining when they complete.

The schedulers have completed when they stop accepting new tasks.

1.2.8: Complete

Indicates to the pair of schedulers that they should start entering a completed state and from now on not accept more tasks.

```
void Complete();
```

Remarks

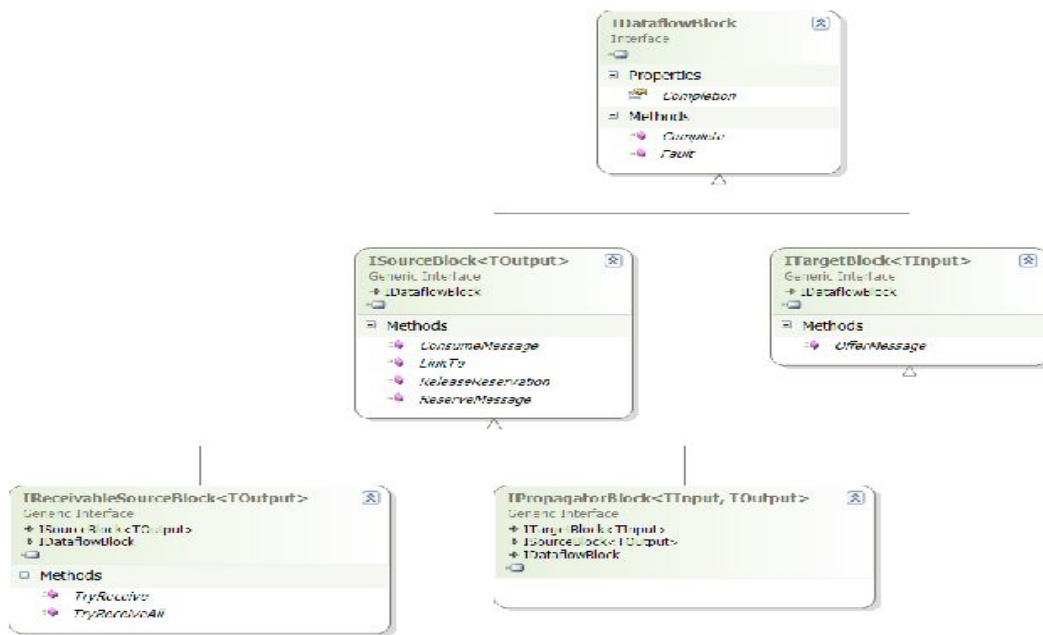
If the schedulers has active tasks, these will be processed before completion. Hence

there may be a timelag between calling this method and the pair actually passing to the completion state. It is important to wait for the task to reach a final state (`Task.Wait()`) before assuming the pair has actually completed.

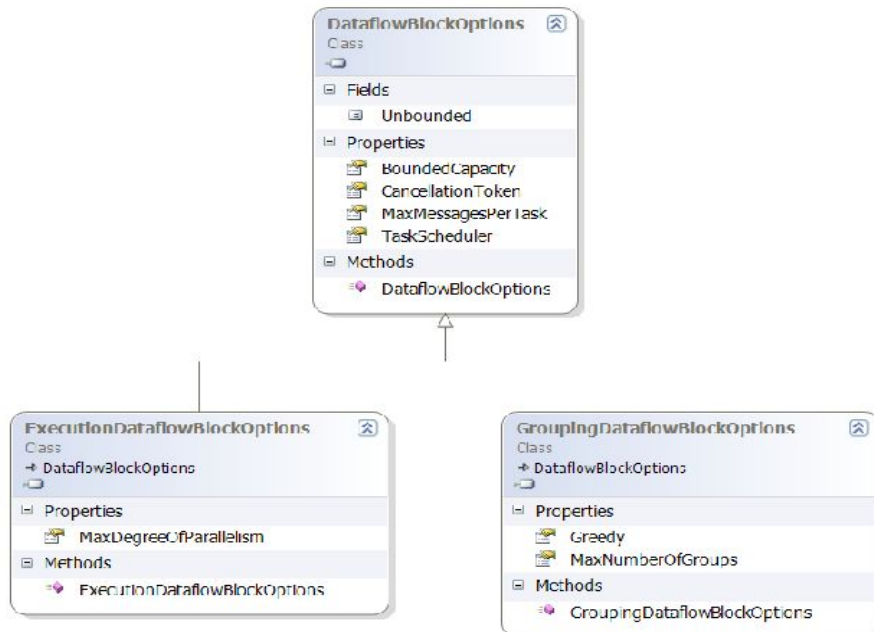
2: System.Threading.Tasks.Dataflow

2.1: Overview

System.Threading.Tasks.Dataflow has the following interfaces:



System.Threading.Tasks.Dataflow has the following classes for options:

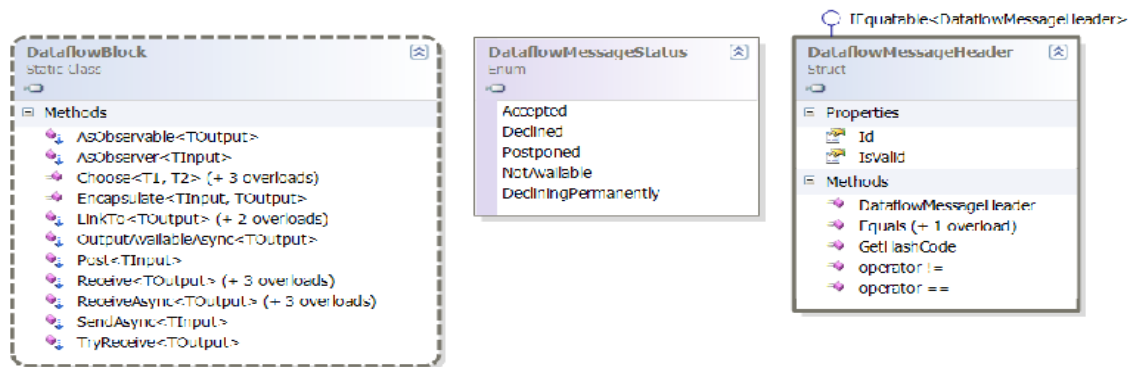


System.Threading.Tasks.Dataflow has the following Block implementations:

The following table summarizes the blocks shown in the screenshots:

Block Name	Key Properties	Key Methods
ActionBlock<TInput>	Completion, InputCount	ActionBlock (+ 3 overloads), Complete
TransformBlock<TInput, TOutput>	Completion, InputCount, OutputCount	Complete, LinkTo, TransformBlock (+ 3 overloads), TryReceive, TryReceiveAll
TransformManyBlock<TInput, TOutput>	Completion, InputCount, OutputCount	Complete, LinkTo, TransformManyBlock (+ 3 overloads), TryReceive, TryReceiveAll
BufferBlock<T>	Completion, Count	BufferBlock (+ 1 overload), Complete, LinkTo, TryReceive, TryReceiveAll
BroadcastBlock<T>	Completion	BroadcastBlock (+ 1 overload), Complete, LinkTo, TryReceive
BatchBlock<T>	BatchSize, Completion, OutputCount	BatchBlock (+ 1 overload), Complete, LinkTo, TriggerBatch, TryReceive, TryReceiveAll
WriteOnceBlock<T>	Completion	Complete, LinkTo, TryReceive, WriteOnceBlock (+ 1 overload)
JoinBlock<T1, T2>	Completion, OutputCount, Target1, Target2	Complete, JoinBlock (+ 1 overload), LinkTo, TryReceive, TryReceiveAll
JoinBlock<T1, T2, T3>	Completion, OutputCount, Target1, Target2, Target3	Complete, JoinBlock (+ 1 overload), LinkTo, TryReceive, TryReceiveAll
BatchedJoinBlock<T1, T2, T3>	BatchSize, Completion, OutputCount, Target1, Target2, Target3	BatchedJoinBlock (+ 1 overload), Complete, LinkTo, TryReceive, TryReceiveAll
BatchedJoinBlock<T1, T2>	BatchSize, Completion, OutputCount, Target1, Target2	BatchedJoinBlock (+ 1 overload), Complete, LinkTo, TryReceive, TryReceiveAll

System.Threading.Tasks.Dataflow has the following helper types:



2.2: Interface Implementations

TPL Dataflow provides a number of interfaces that define the functionality of dataflow blocks. TPL Dataflow also provides a range of common implementations of these.

The following tables lists which block implementations support which interfaces:

	IDataflow Block	ITarget Block	ISource Block	IReceivable SourceBlock	IPropagator Block
ActionBlock	X	X			
BatchBlock	X	X	X	X	X
BatchedJoinBlock	X		X	X	
BroadcastBlock	X	X	X	X	X
BufferBlock	X	X	X	X	X
JoinBlock	X		X	X	
TransformBlock	X	X	X	X	X
TransformManyBlock	X	X	X	X	X
WriteOnceBlock	X	X	X	X	X

2.3: ActionBlock

Performs the action on each message.

```
public sealed class ActionBlock<TInput>
    : ITargetBlock<TInput>, IDataflowBlock {

    public ActionBlock(Action<TInput> action);
    public ActionBlock(Func<TInput, Task> action);
    public ActionBlock(
        Action<TInput> action,
        ExecutionDataflowBlockOptions dataflowBlockOptions);
    public ActionBlock(
        Func<TInput, Task> action,
        ExecutionDataflowBlockOptions dataflowBlockOptions);
```

```
public Task Completion { get; }
public int InputCount { get; }
public void Complete();
}
```

Remarks

This is a target block which performs the action on each message offered via `ITargetBlock.OfferMessage`.

When added to a dataflow network of blocks, it can be considered the edge of the network, since once the message is passed to `ActionBlock`, it leaves the network, as `ActionBlock` does not implement `ISourceBlock`. If one wishes to pass it on to another block after executing the action, then one needs to do this manually within the action, or one needs to use a different block which does implement `ISourceBlock`, such as `TransformBlock`.

The action delegate supplied to the `ActionBlock` constructor can be `Action<TInput>` - in which case the action completes when the delegate returns or `Func<TInput, Task>` - in which case the action completes when the `Task` completes.

`Completion` and `Complete` work as described in `IDataflowBlock`.

`ActionBlock` implements `ITargetBlock.OfferMessage` and `IDataflowBlock.Fault` explicitly, so a cast is required before calling these.

2.3.1: ActionBlock(Action<TInput>)

Creates an `ActionBlock` with the supplied action delegate.

```
public ActionBlock(Action<TInput> action);
```

Parameters

- `action` (`Action<TInput>`) - The delegate to execute for each message

2.3.2: ActionBlock(Func<TInput, Task>)

Creates an `ActionBlock` with the supplied func delegate.

```
public ActionBlock(Func<TInput, Task> action);
```

Parameters

- `action` (`Func<TInput, Task>`) - The delegate to execute for each message

2.3.3: ActionBlock(Action<TInput>, ExecutionDataflowBlockOptions)

Creates an `ActionBlock` with the supplied action delegate using the supplied options.

```
public ActionBlock(
    Action<TInput> action,
    ExecutionDataflowBlockOptions dataflowBlockOptions);
```

Parameters

- action (Action<TInput>) - The delegate to execute for each message
- dataflowBlockOptions (ExecutionDataflowBlockOptions) - Options to configure how the block operates

Remarks

See the coverage of ExecutionDataflowBlockOptions for a detailed discussion of how execution options work.

2.3.4: ActionBlock(Func<TInput, Task>, ExecutionDataflowBlockOptions)

Creates an ActionBlock with the supplied func delegate using the supplied options.

```
public ActionBlock(
    Func<TInput, Task> action,
    ExecutionDataflowBlockOptions dataflowBlockOptions);
```

Parameters

- action (Func<TInput, Task>) - The delegate to execute for each message
- dataflowBlockOptions (ExecutionDataflowBlockOptions) - Options to configure how the block operates

Remarks

See the coverage of ExecutionDataflowBlockOptions for a detailed discussion of how execution options work.

2.3.5: InputCount

The number of messages stored in the input buffer.

2.4: BatchBlock

A block to merge multiple incoming messages into a batched output message.

```
public sealed class BatchBlock<T> : IPropagatorBlock<T, T[]>,
    ITargetBlock<T>, IReceivableSourceBlock<T[]>,
    ISourceBlock<T[]>, IDataflowBlock {
    public BatchBlock(int batchSize);
    public BatchBlock(int batchSize,
        GroupingDataflowBlockOptions dataflowBlockOptions);
    public int BatchSize { get; }
    public Task Completion { get; }
    public int OutputCount { get; }
    public void Complete();
    public IDisposable LinkTo(ITargetBlock<T[]> target,
        bool unlinkAfterOne);
    public void TriggerBatch();
    public bool TryReceive(Predicate<T[]> filter, out T[] item);
    public bool TryReceiveAll(out IList<T[]> items);
}
```

Remarks

This block batches a group of input messages into an output array of messages. How big the group is is specified in the constructor. BatchBlock takes a single type parameter T and from that it knows its input is of type T and its output is of type T[] (an array of zero or more T).

The BatchBlock can also be told to output what it has buffered (even when fewer than the batch size) either by calling TriggerBatch() or Complete().

The grouping options passed to the constructor allows one to specify a maximum number of groups to output and whether the block should operate in greedy mode (in which case it accepts all messages offered to it) or non-greedy mode (in which case it reserves but does not consume all messages until it has enough for a batch, and then consumes them and outputs the batch).

Completion, Complete, LinkTo, TryReceive and TryReceiveAll work as described in IDataflowBlock, ISourceBlock and IReceivableSourceBlock.

BatchBlock implements ITargetBlock, ISourceBlock.[ReserveMessage|ConsumeMessage|ReleaseReservation] and IDataflowBlock.Fault explicitly, so a cast is required before calling these.

2.4.1: BatchBlock(int)

Constructor to create a BatchBlock with the specified batch size.

```
public BatchBlock(int batchSize);
```

Parameters

- batchSize (int) - The size of the batched output

Remarks

The batchSize should be considered a maximum, since smaller batches may be generated, either by calling TriggerBatch() or Complete().

2.4.2: BatchBlock(int, GroupingDataflowBlockOptions)

Constructor to create a BatchBlock with the specified batch size and options.

```
public BatchBlock(int batchSize,
                  GroupingDataflowBlockOptions dataflowBlockOptions);
```

Parameters

- batchSize (int) - The size of the batched output
- dataflowBlockOptions (GroupingDataflowBlockOptions) - options for grouping

Remarks

The batchSize should be consider a maximum, since smaller batched may be generated, either by calling TriggerBatch() or Complete().

The options allows one to state whehter greedy mode should be used and whethere there is a maximum number of batched messages to be output.

2.4.3: BatchSize

The batch size (set via the constructor).

```
public int BatchSize { get; }
```

2.4.4: OutputCount

The number of messages stored in the output buffer.

```
public int OutputCount { get; }
```

2.4.5: TriggerBatch

Forces output of a batched message, even if BatchSize number of input messages has not been received.

```
public void TriggerBatch();
```

2.5: BatchedJoinBlock(T1, T2, ..)

Batches and joins inputs to form a tuple of lists as output.

```
public sealed class BatchedJoinBlock<T1, T2> :
    IReceivableSourceBlock<Tuple<IList<T1>, IList<T2>>>,
    ISourceBlock<Tuple<IList<T1>, IList<T2>>>, IDataflowBlock {
    public BatchedJoinBlock(int batchSize);
    public BatchedJoinBlock(int batchSize,
        GroupingDataflowBlockOptions dataflowBlockOptions);
    public int BatchSize { get; }
    public Task Completion { get; }
    public int OutputCount { get; }
    public ITargetBlock<T1> Target1 { get; }
    public ITargetBlock<T2> Target2 { get; }
    public void Complete();
    public IDisposable LinkTo(ITargetBlock<Tuple<IList<T1>,
        IList<T2>>> target, bool unlinkAfterOne);
    public bool TryReceive(Predicate<Tuple<IList<T1>, IList<T2>>> filter,
        out Tuple<IList<T1>, IList<T2>> item);
    public bool TryReceiveAll(
        out IList<Tuple<IList<T1>, IList<T2>>> items);
}
```

Remarks

Note BatchedJoinBlock does not implement ITargetBlock. Instead, it has ITargetBlock properties for each type it supports. As it receives input via these properties, it creates a tuple, and then creates a batch (i.e. collection) of such tuples, and this is made available as output.

Completion, Complete, LinkTo and TryReceive work as described in IDataflowBlock, ISourceBlock and IReceivableSourceBlock.

BatchedJoinBlock implements ISourceBlock.[ReserveMessage|ConsumeMessage|ReleaseReservation] and IDataflowBlock.Fault explicitly, so a cast is required before calling these.

2.5.1: BatchedJoinBlock()

Constructor to create a JoinBlock with default options.

```
public BatchedJoinBlock(int batchSize);
```

2.5.2: BatchedJoinBlock(GroupingDataflowBlockOptions)

Constructor to create a JoinBlock with the specified options.

```
public batchedJoinBlock(int batchSize,
    GroupingDataflowBlockOptions dataflowBlockOptions);
```

2.5.3: Target1

The TargetBlock for input messages of the first type parameter.

```
public ITargetBlock<T1> Target1 { get; }
```

2.5.4: Target2

The TargetBlock for input messages of the second type parameter.

```
public ITargetBlock<T2> Target2 { get; }
```

2.5.5: BatchSize

The batch size (set via the constructor).

```
public int BatchSize { get; }
```

2.5.6: OutputCount

The number of messages stored in the output buffer.

```
public int OutputCount { get; }
```

2.6: BroadcastBlock

A block which forwards input to each linked target using a cloning function.

```
public sealed class BroadcastBlock<T> : IPropagatorBlock<T, T>,
    ITargetBlock<T>, IReceivableSourceBlock<T>,
    ISourceBlock<T>, IDataflowBlock {
    public BroadcastBlock(Func<T, T> cloningFunction);
    public BroadcastBlock(Func<T, T> cloningFunction,
        DataflowBlockOptions dataflowBlockOptions);
    public Task Completion { get; }
    public void Complete();
    public IDisposable LinkTo(ITargetBlock<T> target,
        bool unlinkAfterOne);
    public bool TryReceive(Predicate<T> filter, out T item);
}
```

Remarks

Each incoming message is broadcast to each linked target. When a new target links to a broadcast block, it also receives the most recently offered message (if any).

Completion, Complete, LinkTo and TryReceive work as described in IDataflowBlock, ISourceBlock and IReceivableSourceBlock.

BroadcastBlock implements ITargetBlock, ISourceBlock.[ReserveMessage|ConsumeMessage|ReleaseReservation] and IDataflowBlock.Fault explicitly, so a cast is required before calling these.

2.6.1: BroadcastBlock(Func<T, T>)

Creates a broadcast block with the specified cloning function.

```
public BroadcastBlock(Func<T, T> cloningFunction);
```

Parameters

- cloningFunction (Func<T, T>) - function to clone the input, called for each target

Remarks

The cloning function parameter may be null, in which can the same reference is passed out to each target. Otherwise, the cloning function is called and the resulting reference is passed out to each target.

2.6.2: BroadcastBlock(Func<T, T>, DataflowBlockOptions)

Creates a broadcast block with the specified cloning function and options.

```
public BroadcastBlock(Func<T, T> cloningFunction,  
    DataflowBlockOptions dataflowBlockOptions);
```

Parameters

- cloningFunction (Func<T, T>) - function to clone the input, called for each target
- dataflowBlockOptions (DataflowBlockOptions) - options to configure how the block should operate

Remarks

The cloning function parameter may be null, in which can the same reference is passed out to each target. Otherwise, the cloning function is called and the resulting reference is passed out to each target.

2.7: BufferBlock

A block which buffers input and makes it available to target.

```
public sealed class BufferBlock<T> : IPropagatorBlock<T, T>,  
    ITargetBlock<T>, IReceivableSourceBlock<T>,  
    ISourceBlock<T>, IDataflowBlock {  
    public BufferBlock();  
    public BufferBlock(DataflowBlockOptions dataflowBlockOptions);
```

```
public Task Completion { get; }
public int Count { get; }
public void Complete();
public IDisposable LinkTo(
    ITargetBlock<T> target, bool unlinkAfterOne);
public bool TryReceive(Predicate<T> filter, out T item);
public bool TryReceiveAll(out IList<T> items);
}
```

Remarks

Each incoming message is buffered to each linked target. When a new target links to a broadcast block, it also receives the most recently offered message (if any).

A BufferBlock maintains a FIFO queue buffer. As messages are received, they are added to the end of the queue. As messages are consumed from the buffer (either manually or by linked targets), they are removed from the head of the queue.

Completion, Complete, LinkTo and TryReceive and TryReceiveAll work as described in IDataflowBlock, ISourceBlock and IReceivableSourceBlock.

BufferBlock implements ITargetBlock, ISourceBlock.[ReserveMessage|ConsumeMessage|ReleaseReservation] and IDataflowBlock.Fault explicitly, so a cast is required before calling it.

2.7.1: BufferBlock()

Creates a buffer block.

```
public BufferBlock();
```

2.7.2: BufferBlock(DataflowBlockOptions)

Creates a buffer block with the specified options.

```
public BufferBlock(DataflowBlockOptions dataflowBlockOptions);
```

Parameters

- dataflowBlockOptions (DataflowBlockOptions) - options to configure how the block should operate

2.7.3: Count

The number of messages stored in the buffer.

```
public int Count { get; }
```

2.8: DataflowBlock

DataflowBlock is a static class contains a number of useful extension methods and other helper methods that work with dataflow blocks.

```
public static class DataflowBlock {

    public static IObservable<TOutput> AsObservable<TOutput>(
        this ISourceBlock<TOutput> source);
    public static IObservable<TInput> AsObserver<TInput>(
        this ITargetBlock<TInput> target);

    public static Task<int> Choose<T1, T2>(
        ISourceBlock<T1> source1, Action<T1> action1,
        ISourceBlock<T2> source2, Action<T2> action2);
    public static Task<int> Choose<T1, T2>(
        ISourceBlock<T1> source1, Action<T1> action1,
        ISourceBlock<T2> source2, Action<T2> action2,
        DataflowBlockOptions dataflowBlockOptions);
    public static Task<int> Choose<T1, T2, T3>(
        ISourceBlock<T1> source1, Action<T1> action1,
        ISourceBlock<T2> source2, Action<T2> action2,
        ISourceBlock<T3> source3, Action<T3> action3);
    public static Task<int> Choose<T1, T2, T3>(
        ISourceBlock<T1> source1, Action<T1> action1,
        ISourceBlock<T2> source2, Action<T2> action2,
        ISourceBlock<T3> source3, Action<T3> action3,
        DataflowBlockOptions dataflowBlockOptions);

    public static IPropagatorBlock<TInput, TOutput>
        Encapsulate<TInput, TOutput>(ITargetBlock<TInput> target,
            ISourceBlock<TOutput> source);
    public static IDisposable LinkTo<TOutput>(
        this ISourceBlock<TOutput> source,
        ITargetBlock<TOutput> target);
    public static IDisposable LinkTo<TOutput>(
        this ISourceBlock<TOutput> source,
        ITargetBlock<TOutput> target,
        Predicate<TOutput> predicate);
    public static IDisposable LinkTo<TOutput>(
        this ISourceBlock<TOutput> source,
        ITargetBlock<TOutput> target,
        Predicate<TOutput> predicate,
        bool discardsMessages);

    public static Task<bool> OutputAvailableAsync<TOutput>(
        this ISourceBlock<TOutput> source);

    public static bool Post<TInput>(this ITargetBlock<TInput> target,
        TInput item);

    public static TOutput Receive<TOutput>(
        this ISourceBlock<TOutput> source);
    public static TOutput Receive<TOutput>(
        this ISourceBlock<TOutput> source,
        CancellationToken cancellationToken);
    public static TOutput Receive<TOutput>(
        this ISourceBlock<TOutput> source,
        TimeSpan timeout);
    public static TOutput Receive<TOutput>(
        this ISourceBlock<TOutput> source,
        TimeSpan timeout,
        CancellationToken cancellationToken);
    public static Task<TOutput> ReceiveAsync<TOutput>(
        this ISourceBlock<TOutput> source);
```

```

public static Task<TOutput> ReceiveAsync<TOutput>(
    this ISourceBlock<TOutput> source,
    CancellationToken cancellationToken);
public static Task<TOutput> ReceiveAsync<TOutput>(
    this ISourceBlock<TOutput> source,
    TimeSpan timeout);
public static Task<TOutput> ReceiveAsync<TOutput>(
    this ISourceBlock<TOutput> source,
    TimeSpan timeout,
    CancellationToken cancellationToken);

public static Task<bool> SendAsync<TInput>(
    this ITargetBlock<TInput> target,
    TInput item);

public static bool TryReceive<TOutput>(
    this IReceivableSourceBlock<TOutput> source,
    out TOutput item);
}

```

Remarks

These static methods can be divided into a number of categories: exchanging messages with a block, interop with Rx (Reactive extensions), Choose, LinkTo and Encapsulate.

2.8.1: AsObservable

Returns an IObservable wrapper for an ISourceBlock.

```

public static IObservable<TOutput> AsObservable<TOutput>(
    this ISourceBlock<TOutput> source);

```

Parameters

- source (this ISourceBlock<TOutput>) - The ISourceBlock to be wrapped

Return Value

- IObservable<TOutput> - The observable which wraps the ISourceBlock

Remarks

The returned IObservable complies with the Rx contract and can be use dto integrate datablow blocks and Rx stream processing.

It is noted that there is no method for going in the other direction (if the input were an IObservable, it would return an ISourceBlock).

2.8.2: AsObserver

Returns an IObserver wrapper for an ITargetBlock.

```

public static IObserver<TInput> AsObserver<TInput>(
    this ITargetBlock<TInput> target);

```

Parameters

- target (this ITargetBlock<TInput>) - The ITargetBlock to be wrapped

Return Value

- `IObserver<TInput>` - The observer which wraps the `ITargetBlock`

Remarks

It is noted that there is no method for going in the other direction (if the input were an `IObserver`, it would return an `ITargetBlock`).

2.8.3: Choose

Monitor a multiple of sources and calls the relevant action for the source which provides a message first.

```
public static Task<int> Choose<T1, T2>(
    ISourceBlock<T1> source1, Action<T1> action1,
    ISourceBlock<T2> source2, Action<T2> action2);
public static Task<int> Choose<T1, T2>(
    ISourceBlock<T1> source1, Action<T1> action1,
    ISourceBlock<T2> source2, Action<T2> action2,
    DataflowBlockOptions dataflowBlockOptions);
public static Task<int> Choose<T1, T2, T3>(
    ISourceBlock<T1> source1, Action<T1> action1,
    ISourceBlock<T2> source2, Action<T2> action2,
    ISourceBlock<T3> source3, Action<T3> action3);
public static Task<int> Choose<T1, T2, T3>(
    ISourceBlock<T1> source1, Action<T1> action1,
    ISourceBlock<T2> source2, Action<T2> action2,
    ISourceBlock<T3> source3, Action<T3> action3,
    DataflowBlockOptions dataflowBlockOptions);
```

Parameters

- `sourceX` (`ISourceBlock<>`) - a source to monitor for an incoming message
- `actionX` (`Action<>`) - the action to execute if the equivalent source provides the message first
- `dataflowBlockOptions` (`DataflowBlockOptions`) - options to configure the block

Return Value

- `Task<int>` - A task whose result is the 0-based index of the successful source

2.8.4: Encapsulate

Creates a propagator whose interface implementations come from different blocks.

```
public static IPropagatorBlock<TInput, TOutput>
    Encapsulate<TInput, TOutput>(ITargetBlock<TInput> target,
    ISourceBlock<TOutput> source);
```

Parameters

- target (ITargetBlock<TInput>) - The target to be encapsulated
- source (ISourceBlock<TOutput>) - The source to be encapsulated

Return Value

- IPropagatorBlock<TInput, TOutput> - The result of the encapsulation

Remarks

ITargetBlock implementation is provided by the target parameter and whose ISourceBlock and IDataflowBlock implementatins are provided by the source parameter.

2.8.5: LinkTo

Links a source to a target, possibly using a predicate.

```
public static IDisposable LinkTo<TOutput>(
    this ISourceBlock<TOutput> source,
    ITargetBlock<TOutput> target);
public static IDisposable LinkTo<TOutput>(
    this ISourceBlock<TOutput> source,
    ITargetBlock<TOutput> target,
    Predicate<TOutput> predicate);
public static IDisposable LinkTo<TOutput>(
    this ISourceBlock<TOutput> source,
    ITargetBlock<TOutput> target,
    Predicate<TOutput> predicate,
    bool discardsMessages);
```

Parameters

- source (this ISourceBlock<TOutput> - the source block
- target (ITargetBlock<TOutput>) - the target block
- predicate (Predicate<TOutput>) - the

discardMessages (bool) - whether messages which fail the predicate are to be discarded (true) or delined (false)

Return Value

- IDisposable - Call Dispose() to unlink

2.8.6: OutputAvailableAsync

Returns a task with a bool result which completes when either output is available (true) or will never become available (false).

```
public static Task<bool> OutputAvailableAsync<TOutput>(
    this ISourceBlock<TOutput> source);
```

Parameters

- source (this ISourceBlock<TOutput>) - The source in whose output

available one is interested

Return Value

- Task<bool> - The task which can be awaited

2.8.7: Post

Post is shorthand for calling ITargetBlock.OfferMessage with a null source block and message header id set to 1.

```
public static bool Post<TInput>(this ITargetBlock<TInput> target,
                                TInput item);
```

Parameters

target (this ITargetBlock<TInput>) - the target to which the message is to be offered

- item (TInput) - The message to offer to the target

Return Value

- bool - whether the message was accepted by the target

Remarks

It is important to note that tasks may spin up in processing the message, so one should not assume processing is complete once Post returns.

2.8.8: Receive

Receive a message from an ISourceBlock synchronously.

```
public static TOutput Receive<TOutput>(
    this ISourceBlock<TOutput> source);
public static TOutput Receive<TOutput>(
    this ISourceBlock<TOutput> source,
    CancellationToken cancellationToken);
public static TOutput Receive<TOutput>(
    this ISourceBlock<TOutput> source,
    TimeSpan timeout);
public static TOutput Receive<TOutput>(
    this ISourceBlock<TOutput> source,
    TimeSpan timeout,
    CancellationToken cancellationToken);
```

Parameters

- source (this ISourceBlock) - The source of the message
- cancellationToken (CancellationToken) - The CancellationToken which allows the asynchronous receive to be canceled
- timeout (TimeSpan) - a timeout for waiting for receive to complete

Return Value

- Output - The received message

2.8.9: ReceiveAsync

Receive a message from an `ISourceBlock` asynchronously.

```
public static Task<TOutput> ReceiveAsync<TOutput>(
    this ISourceBlock<TOutput> source);
public static Task<TOutput> ReceiveAsync<TOutput>(
    this ISourceBlock<TOutput> source,
    CancellationToken cancellationToken);
public static Task<TOutput> ReceiveAsync<TOutput>(
    this ISourceBlock<TOutput> source,
    TimeSpan timeout);
public static Task<TOutput> ReceiveAsync<TOutput>(
    this ISourceBlock<TOutput> source,
    TimeSpan timeout,
    CancellationToken cancellationToken);
```

Parameters

- source (this `ISourceBlock`) - The source of the message
- cancellationToken (`CancellationToken`) - The `CancellationToken` which allows the asynchronous receive to be canceled
- timeout (`TimeSpan`) - a timeout for waiting for receive to complete

Return Value

- `Task<Output>` - The task that completes when the message has been received (or canceled)

2.8.10: SendAsync

Offers a message to a target allowing for reservation.

```
public static Task<bool> SendAsync<TInput>(
    this ITargetBlock<TInput> target,
    TInput item);
```

Parameters

- target (this `ITargetBlock<TInput>`) - The target
- item (`TInput`) - The input message

Return Value

- `Task<bool>` - A task that completes when the asynchronous send has completed

2.8.11: TryReceive

Synchronously receive (if available) a message from the source (using no filter).

```
public static bool TryReceive<TOutput>(
    this IReceivableSourceBlock<TOutput> source, out TOutput item);
```

Parameters

- source (this IReceivableSourceBlock) - the source to use
- item (out TOutput) - the returned message

Return Value

- bool - If a message has been received

Remarks

This extension method is a helper which calls `IReceivableSourceBlock.TryReceive` with a null filter.

2.9: DataflowMessageHeader

Passed between a source and a target to identify a particular message.

```
public struct DataflowMessageHeader : IEquatable<DataflowMessageHeader> {
    public DataflowMessageHeader(long id);
    public static bool operator !=(
        DataflowMessageHeader left, DataflowMessageHeader right);
    public static bool operator ==(
        DataflowMessageHeader left, DataflowMessageHeader right);
    public long Id { get; }
    public bool IsValid { get; }
    public bool Equals(DataflowMessageHeader other);
    public override bool Equals(object obj);
    public override int GetHashCode();
}
```

Remarks

Note that `DataflowMessageHeader` is not part of the message itself, rather it is a separate instance that is passed separately in the various APIs. This means the message type does not have to derive from it or include it as a field/property.

`DataflowMessageHeader` is used as a parameter in `ITargetBlock.OfferMessage` and in `ISourceBlock.[ConsumeMessage|ReserveMessage|ReleaseReservation]`.

It identifies the message via a long ID in the source from which it comes from. Hence the combination of id and source must be unique, but the same id may be used with different sources at the same time and a particular target may be offered messages with the same ID from different sources at the same time.

2.9.1: DataflowMessageHeader(long id)

Creates a new `DataflowMessageHeader` with the specified Id.

```
public DataflowMessageHeader(long id);
```

Parameters

- id (long) - The id for the message header (must not be 0)

Remarks

Calls to this constructor with a parameter of 0 results in an `ArgumentException` being raised with the error message: "To construct a `DataflowMessageHeader` instance, either pass a non-zero ID or use the parameterless constructor."

As a struct, a Default Value is also available, which is a default constructor that sets `Id` to 0. This shows up in Intellisense, but its use is not recommended, as a header with `Id=0` is invalid, and since the struct is immutable, there is no way of changing the `Id` after creation.

2.9.2: Id

The `Id` for the message.

```
public long Id { get; }
```

2.9.3: IsValid

Whether the message header is valid (whether `Id != 0`).

```
public bool IsValid { get; }
```

2.10: DataflowMessageStatus

Status of a message as it flows between source and target.

```
public enum DataflowMessageStatus
{
    Accepted = 0,
    Declined = 1,
    Postponed = 2,
    NotAvailable = 3,
    DecliningPermanently = 4,
}
```

Values

It has the following values:

- Accepted - The target has accepted the message (target owns the message)
- Declined - The target has declined the message. (source still owns the message)
- Postponed - The target has postponed the message (source still owns the message)
- NotAvailable - The target tried accepting the message, but it was not available (neither target nor source owns message - source probably previously offered it to a different target, which accepted)
- DecliningPermanently - The target declined this message and will do likewise with all subsequent messages

(source owns message)

Remarks

This enum is the return value for `ITargetBlock.OfferMessage`. See the remarks for that methods to learn more about what `DataflowMessageStatus` represents.

2.11: DataflowOptions

Options to influence how a dataflow block should operate.

```
public class DataflowBlockOptions {
    public const int Unbounded = -1;
    public DataflowBlockOptions();
    public int BoundedCapacity { get; set; }
    public CancellationToken CancellationToken { get; set; }
    public int MaxMessagesPerTask { get; set; }
    public TaskScheduler TaskScheduler { get; set; }
}
```

Remarks

For each dataflow block, there is a constructor that take an options parameter and one that does not (the latter simply uses a default set of options).

There are three options types in TPL Dataflow: `DataflowBlockOptions` (used as the options parameter for `BufferBlock`, `BroadcastBlock`, `WriteOnceBlock`; `ExecutionDataflowBlockOptions` for executing blocks (`ActionBlock`, `TransformBlock` and `TransformManyBlock`) and `GroupingDataflowBlockOptions` for grouping blocks (`JoinBlock` and `BatchedJoinBlock`). Both `ExecutionDataflowBlockOptions` and `GroupingDataflowBlockOptions` derive from `DataflowBlockOptions`, so each block can be passed the options described here.

2.11.1: DataflowBlockOptions()

Creates a `DataflowBlockOptions` instance with default values.

```
public DataflowBlockOptions();
```

2.11.2: BoundedCapacity

A maximum capacity for the number of messages a block may be using at any one time.

```
public int BoundedCapacity { get; set; }
```

2.11.3: CancellationToken

The `CancellationToken` which can be used to request cancellation.

```
public CancellationToken CancellationToken { get; set; }
```

2.11.4: MaxMessagesPerTask

The maximum number of messages a block can process in a task.

```
public int MaxMessagesPerTask { get; set; }
```

Remarks

defaults to Unbounded (-1).

2.11.5: TaskScheduler

A property exposing the task scheduler.

```
public TaskScheduler TaskScheduler { get; set; }
```

Remarks

The default is to use the ThreadPool.

It should be noted that blocks schedule tasks even when accepting messages (e.g. via a post), so the sequencing might not as expected for developers used to synchronous execution (when a `DataflowBlock.Post` extension method or an implementation of `ITargetBlock.OfferMessage` returns, that might not guarantee that the message has been processed by the target).

2.12: ExecutionDataflowOptions

Additional options for executing blocks.

```
public class ExecutionDataflowBlockOptions : DataflowBlockOptions {  
    public ExecutionDataflowBlockOptions();  
    public int MaxDegreeOfParallelism { get; set; }  
}
```

2.12.1: ExecutionDataflowOptions

Constructor to create a new `ExecutionDataflowBlockOptions` instance with default values.

```
public ExecutionDataflowBlockOptions();
```

2.12.2: MaxDegreeOfParallelism

Property stating how many messages a block should process in parallel.

```
public int MaxDegreeOfParallelism { get; set; }
```

Remarks

The default is 1.

2.13: GroupingDataflowOptions

Additional options for grouping blocks.

```
public class GroupingDataflowBlockOptions : DataflowBlockOptions {  
    public GroupingDataflowBlockOptions();  
    public bool Greedy { get; set; }  
    public long MaxNumberOfGroups { get; set; }  
}
```

2.13.1: GroupingDataflowBlockOptions

Constructor for GroupingDataflowBlockOptions.

```
public GroupingDataflowBlockOptions();
```

2.13.2: Greedy

Greedy means whether a block immediately accepts all messages offered to it, even if it cannot process them straight away.

```
public bool Greedy { get; set; }
```

Remarks

The default is true.

Some blocks need multiple inputs before they can output. So the question arises what happens when they are offered some inputs but not enough to output. When greedy is set to true, they accept all incoming blocks immediately. When it is false, they postpone offered messages, until they have enough postponed messages in order to output a message.

2.13.3: MaxNumberOfGroups

The maximum number of groups this block should output.

```
public long MaxNumberOfGroups { get; set; }
```

Remarks

After the maximum has been output, the block declines all subsequent incoming messages.

2.14: IDataflowBlock

The base interface for all dataflow blocks.

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

2.14.1: Completion

A task that represents completion of the block.

```
Task Completion { get; }
```

Remarks

This task represents the completion status of the dataflow block and is used to determine when and how the block transitions to a final state.

A block is said to have completed when it stops processing messages for good. After this point, message input (including postponed messages) and message output will no longer occur.

The Completion task has three potential final states – RanToCompletion when it finishes correctly processes all the block's messages (e.g. call `IDataflowBlock.Complete`), Faulted when a fault has occurred (e.g. an exception occurs in code processing a message, or from outside, by calling `IDataflowBlock.Fault`) or Canceled when cancellation has occurred (a cancellation token can be set using `DataflowBlockOptions.CancellationToken`). When it is not in a final state (i.e. when it is processing messages), the Completion task is in the `WaitingForActivation` state.

2.14.2: Complete

Indicates to the block that it should start entering a completed state.

```
void Complete();
```

Remarks

If the dataflow block has buffered messages, these will be processed before completion. Hence there may be a timelag between calling this method and the task actually passing to the `RanToCompletion` state. It is important to wait for the task to reach a final state (`Task.Wait()`) before assuming the task has actually completed.

2.14.3: Fault

Indicates to the block that it has completed in a faulted state.

```
void Fault(Exception exception);
```

Parameters

- `exception (Exception)` - The exception that caused the fault.

Remarks

The `Task.Status` property for the Completion task is set to `Faulted`.

The `Task.Exception` property for the Completion task is set to an `AggregateException` containing the supplied exception,

All buffered messages are lost.

Note that the concrete block implementations in TPL Dataflow (e.g. `ActionBlock`, `BufferBlock`, etc.) all implement `IDataflowBlock.Fault` explicitly, which means an instance of one of these needs to be cast to `IDataflowBlock` in order to call the `Fault` method.

It is important to wait for the task to reach a final state (`Task.Wait()`) before assuming the task has actually faulted.

2.15: IPropagatorBlock

A propagator block is both a source and a target.

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

Remarks

No need methods are added.

2.16: IReceivableSourceBlock

A source block from which messages can be received without linking and without having to have a target block.

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

Remarks

The methods on ISourceBlock are to be called either to LinkTo a target, or from a target to which a message has been offered (by a linked source). In contrast, IReceivableSourceBlock can be used to receive messages from a source without linking and its methods do not require an ITargetBlock parameter.

2.16.1: TryReceive

Synchronously receive (if available) a message from the source (that complies with the filter, if specified).

```
bool TryReceive(Predicate<TOutput> filter, out TOutput item);
```

Parameters

- filter (Predicate<TOutput>) - a filter that, if non-null, the message must match in order for it to be returned
- item (out TOutput) - the returned message

Return Value

- bool - If a message has been received

Remarks

Note that System.Threading.Tasks.Dataflow.DataflowBlock static class has an extension method called TryReceive which in addition to the extension type (IReceivableSourceBlock<TOutput>) has a single extra parameter, out TOutput item. It can be used to call TryReceive with a null filter.

2.16.2: TryReceiveAll

Synchronously receive all messages that the source has available.

```
bool TryReceiveAll(out IList<TOutput> items);
```

Parameters

- items (out IList<TOutput>) - list of items

Return Value

- bool - if at least one message was returned (if items.Count > 0)

Remarks

Receives all the messages that the source has available to be consumed.

2.17: ISourceBlock

Represents a dataflow block that is the source of messages.

```
public interface ISourceBlock<out TOutput> : IDataflowBlock {
    TOutput ConsumeMessage(DataflowMessageHeader messageHeader,
        ITargetBlock<TOutput> target, out bool messageConsumed);
    IDisposable LinkTo(ITargetBlock<TOutput> target,
        bool unlinkAfterOne);
    void ReleaseReservation(DataflowMessageHeader messageHeader,
        ITargetBlock<TOutput> target);
    bool ReserveMessage(DataflowMessageHeader messageHeader,
        ITargetBlock<TOutput> target);
}
```

Remarks

It is best to consider the methods in two groups - one is used to set up the dataflow network, and consists of just one method, LinkTo. The other group - comprising of ReserveMessage, ConsumeMessage and ReleaseReservation, (analogous to prepare/commit/rollback in transactions). Each of these require a parameter of a DataflowMessageHeader and an ITargetBlock, with ConsumeMessage also requiring an out messageConsumed boolean parameter. An important point to note is that these should only be called by linked targets for message headers which have been offered to the target (e.g. they are usually called in the target's implementation of OfferMessage).

2.17.1: ConsumeMessage

Consumes a message that has been offered to a target.

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

Parameters

- messageHeader (DataflowMessageHeader) - header for a message which

has already been offered to the target

- target (ITargetBlock<TOutput>) - the originator of the call
- messageConsumed (out bool) - whether the message has been consumed

Return Value

- TOutput - the message

Remarks

A target calls this method of a source when the source calls the target's OfferMessage with the ConsumeToAccept bool set to true. A source may have some substantial effort required in order to pass a message to a target (e.g. cloning a broadcast message), and hence only wants to do it if the target really will accept it. Hence by setting consumeToAccept to true, the target implementation of OfferMessage can first decide if it wishes to accept the message, and if so, call ISourceBlock.ConsumeMessage, but critically, if not, then the substantial effort is not expended.

If the target calls ConsumeMessage, then its return value is the message, not that passed in as the message parameter to OfferMessage.

If messageConsumed is set to false, then ITargetBlock.OfferMessage returns with a status of Declined. If messageConsumed is set to true and the return value is not null, then ITargetBlock.OfferMessage returns a status of Accepted. If messageConsumed is set to true and the return value is null, then ITargetBlock.OfferMessage returns a status of NotAccepted. This is important, since it means null is not a valid message. (However, if one calls ITargetBlock.OfferMessage with a null value for the message, it is valid). In general, a null message does not make sense.

2.17.2: LinkTo

Links a target to a source.

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

Parameters

- target (ITargetBlock<TOutput>) - the target to link to
- unlinkAfterOne (bool) - should the target be unlinked after the first message (usually set to false)

Return Value

- IDisposable - used to unlink from target

Remarks

TPL Dataflow provides functionality to build up a network of dataflow blocks. `LinkTo` is the key API used to connect blocks. Calling `Dispose()` on the returned `IDisposable` can be used to disconnect blocks. If a block is buffering, and `LinkTo` is called later, then the source will offer the buffered messages to the just linked target.

2.17.3: ReleaseReservation

Releases a previously reserved message.

```
void ReleaseReservation(DataflowMessageHeader messageHeader,  
    ITargetBlock<TOutput> target);
```

Parameters

- `messageHeader` (`DataflowMessageHeader`) - header for a message which has already been offered to the target
- `target` (`ITargetBlock<TOutput>`) - the originator of the call

Remarks

Releases a previously successfully reserved message (where `true` was returned to an earlier call to `ReserveMessage`). The target needs to keep a record of messages it reserves (the `DataflowMessageHeader` and `ISourceBlock`), so that it can either consume the message or release it.

2.17.4: ReserveMessage

Reserves a message for later consumption.

```
bool ReserveMessage(DataflowMessageHeader messageHeader,  
    ITargetBlock<TOutput> target);
```

Parameters

- `messageHeader` (`DataflowMessageHeader`) - header for a message which has already been offered to the target
- `target` (`ITargetBlock<TOutput>`) - the originator of the call

Return Value

- `bool` - whether the message has been reserved

Remarks

This method should only be called by a target which previously has been offered the message (e.g. from inside its implementation of `ITargetBlock.OfferMessage`).

If the return value is `true`, then the target needs, at some later point, to either call `ConsumeMessage` or `ReleaseReservation`. Some sources have a bounded

capacity, and reserved messages remain the property of the source and are stored in its buffer. When the buffer is full, the source cannot accept more messages. Hence, a target should be careful to keep track of messages it reserves.

2.18: ITargetBlock

Interface representing a dataflow block to which messages may be offered.

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

Remarks

A dataflow block that can receive messages should implement this interface. It adds one method, OfferMessage, to the IDataflowBlock interface and this can be called directly or via a call to the DataflowBlock.Post extension method.

The supplied concrete implementations of ITargetBlock are ActionBlock, BatchBlock, BroadcastBlock, BufferBlock, TransformBlock, TransformManyBlock and writeOnceBlock.

2.18.1: OfferMessage

Offers a message to a target.

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

Parameters

- messageHeader (DataflowMessageHeader) - header for this message
- messageValue (TInput) - the actual message
- source (ISourceBlock<TInput>) - from where the message came
- consumeToAccept (bool) - whether a call to source.ConsumeMessage is required in order to accept message

Return Value

- DataflowMessageStatus - status of offered message (see DataflowMessageStatus for discussion of values)

Remarks

Note that `ITargetBlock.OfferMessage` takes a message header, the actual message value, the source (`ISourceBlock`) and a Boolean `consumeToAccept`.

In the typical case when a source directly (i.e. `consumeToAccept` set to false) offers a message to a target and it accepts, then the result will be `Accepted`. In this case, the source may be null, as it is not used.

If a source offers a message to a target with `consumeToAccept` set to true, then it means it requires the target to call the source's `ConsumeMessage` method to retrieve the message. In this case, source must be non-null (otherwise an `ArgumentExpcetion` is raised for `consumeToAccept` with the error message set to "The argument must be false if no source from which to consume is specified).

If the target is no longer accepting messages (e.g. `IDataflowBlock.Complete` has been called for it), then calls to `OfferMessage` will return `DecliningPermanently`.

`ITargetBlock.OfferMessage` will return a status of declined when it does not want to, or cannot, accept the specific message offered. Some source blocks need to carry out work to provide a message to a target (for example, a broadcast block needs to clone a message). Some targets will not accept a message, and so as not to have to carry out the work prior to being sure the target will accept, the source can call `OfferMessage` with a `consumeToAccept` value set to true, which means the target must call

`ISourceBlock.ConsumeMessage` in order to accept the message. When the target does this, and `ConsumeMessage` actually returns the message (its `messageConsumed` Boolean is set to true), then `OfferMessage` will return with a status of `Accepted`. However, when `ConsumeMessage` does not return the message (`messageConsumed` set to false), then `ITargetBlock.OfferMessage` will return a status of declined.

If `ITargetBlock.OfferMessage` is called with `consumeToAccept` set to true, the target calls `ISourceBlock.ConsumeMessage` with the message header, and if this cannot make available the message, then its out `messageConsumed` boolean parameter will be set to false, and in turn the result of `ITargetBlock.OfferMessage` will be set to `NotAvailable`.

2.19: JoinBlock(T1, T2, ..)

Joins messages received via public `ITargetBlock` interfaces into output tuples.

```
public sealed class JoinBlock<T1, T2>
    : IReceivableSourceBlock<Tuple<T1, T2>>,
      ISourceBlock<Tuple<T1, T2>>, IDataflowBlock {
    public JoinBlock();
```

```

public JoinBlock(GroupingDataflowBlockOptions dataflowBlockOptions);
public Task Completion { get; }
public ITargetBlock<T1> Target1 { get; }
public ITargetBlock<T2> Target2 { get; }
public void Complete();
public IDisposable LinkTo(ITargetBlock<Tuple<T1, T2>> target,
    bool unlinkAfterOne);
public bool TryReceive(Predicate<Tuple<T1, T2>> filter,
    out Tuple<T1, T2> item);
public bool TryReceiveAll(out IList<Tuple<T1, T2>> items);
}

```

Remarks

A `JoinBlock` does not itself implement `ITargetBlock`. Instead, it exposes `ITargetBlock` properties and uses these to accept incoming messages. It then joins them together and offers them as a tuple as output.

Two variants of `JoinBlock` are offered, one that takes two type parameters and exposed two `ITargetBlocks`, and another that takes three type parameters and exposes three target blocks.

Whereas `BatchBlock` offers a single type parameter, and so its output is an array of that type, `JoinBlock` has multiple, potentially different, type parameters, and so offers a tuple. Also see `BatchedJoin`.

`Completion`, `Complete`, `LinkTo`, `TryReceive` and `TryReceiveAll` work as described in `IDataflowBlock`, `ISourceBlock` and `IReceivableSourceBlock`.

`JoinBlock` implements `ISourceBlock`.`[ReserveMessage|ConsumeMessage|ReleaseReservation]` and `IDataflowBlock`.`Fault` explicitly, so a cast is required before calling these.

2.19.1: JoinBlock()

Constructor to create a `JoinBlock` with default options.

```
public JoinBlock();
```

2.19.2: JoinBlock(GroupingDataflowBlockOptions)

Constructor to create a `JoinBlock` with the specified options.

```
public JoinBlock(GroupingDataflowBlockOptions dataflowBlockOptions);
```

2.19.3: Target1

The `TargetBlock` for input messages of the first type parameter.

```
public ITargetBlock<T1> Target1 { get; }
```

2.19.4: Target2

The `TargetBlock` for input messages of the second type parameter.

```
public ITargetBlock<T2> Target2 { get; }
```

2.20: TransformBlock

A block with an input buffer and an output buffer that executes the supplied function (which produces one output for one input) in between.

```
public sealed class TransformBlock<TInput, TOutput> :
    IPropagatorBlock<TInput, TOutput>, ITargetBlock<TInput>,
    IReceivableSourceBlock<TOutput>, ISourceBlock<TOutput>, IDataflowBlock{
    public TransformBlock(Func<TInput, Task<TOutput>> transform);
    public TransformBlock(Func<TInput, TOutput> transform);
    public TransformBlock(Func<TInput, Task<TOutput>> transform,
        ExecutionDataflowBlockOptions dataflowBlockOptions);
    public TransformBlock(Func<TInput, TOutput> transform,
        ExecutionDataflowBlockOptions dataflowBlockOptions);

    public Task Completion { get; }
    public int InputCount { get; }
    public int OutputCount { get; }
    public void Complete();
    public IDisposable LinkTo(ITargetBlock<TOutput> target,
        bool unlinkAfterOne);
    public bool TryReceive(Predicate<TOutput> filter, out TOutput item);
    public bool TryReceiveAll(out IList<TOutput> items);
}
```

Remarks

TransformBlock takes two type parameters, so the input type can be different from the output type.

TransformBlock is a one-to-one transformation, where one input passed to the transform function results in one output (in contrast to TransformManyBlock, where one input results in zero or more outputs; also in contrast to ActionBlock, which has no output).

The transform function passed to the constructor can either be Func<TInput, TOutput> or Func<TInput, Task<TOutput>>, the difference being that the function is considered complete with the former when it returns and with the latter when the task completes executing.

TransformBlock uses FIFO ordering, regardless of whether one or multiple messages are processed concurrently (using DataflowBlockOptions).

Completion, Complete, LinkTo, TryReceive and TryReceiveAll work as described in IDataflowBlock, ISourceBlock and IReceivableSourceBlock.

TransformBlock implements ITargetBlock, ISourceBlock.[ReserveMessage|ConsumeMessage|ReleaseReservation] and IDataflowBlock.Fault explicitly, so a cast is required before calling these.

2.20.1: TransformBlock(Func<TInput, Task<TOutput>>)

Creates a TransformBlock instance with the transform function and default options.

```
public TransformBlock(Func<TInput, Task<TOutput>> transform);
```

Parameters

- transform (Func<TInput, Task<TOutput>>) - the transform function (may not be null)

Remarks

The transform function completes when Task<TOutput> completes.

2.20.2: TransformBlock(Func<TInput, TOutput>)

Creates a TransformBlock instance with the transform function and default options.

```
public TransformBlock(Func<TInput, TOutput> transform);
```

Parameters

- transform (Func<TInput, TOutput>) - the transform function (may not be null)

2.20.3: TransformBlock(Func<TInput, Task<TOutput>>, DataflowBlockOptions)

Create a TransformBlock instance with the transform function and custom options.

```
public TransformBlock(Func<TInput, Task<TOutput>> transform,  
    ExecutionDataflowBlockOptions dataflowBlockOptions);
```

Parameters

- transform (Func<TInput, Task<TOutput>>) - the transform function (may not be null)
- dataflowBlockOptions (DataflowBlockOptions) - options for this block

Remarks

The transform function completes when Task<TOutput> completes.

See the coverage of DataflowBlockOptions for a detailed discussion of how options work.

2.20.4: TransformBlock(Func<TInput, TOutput>, DataflowBlockOptions)

Create a TransformBlock instance with the transform function and custom options.

```
public TransformBlock(Func<TInput, TOutput> transform,  
    ExecutionDataflowBlockOptions dataflowBlockOptions);
```

Parameters

- transform (Func<TInput, Task<TOutput>>) - the transform function (may not be null)
- dataflowBlockOptions (DataflowBlockOptions) - options for this block

Remarks

The transform function completes when Task<TOutput> completes.

See the coverage of DataflowBlockOptions for a detailed discussion of how options work.

2.20.5: InputCount

The number of messages stored in the input buffer.

```
public int InputCount { get; }
```

2.20.6: OutputCount

The number of messages stored in the output buffer.

```
public int OutputCount { get; }
```

2.21: TransformManyBlock

A block with an input buffer and an output buffer that executes the supplied function (which produces zero or more outputs for each input) in between.

```
public sealed class TransformManyBlock<TInput, TOutput> :
    IPropagatorBlock<TInput, TOutput>, ITargetBlock<TInput>,
    IReceivableSourceBlock<TOutput>, ISourceBlock<TOutput>, IDataflowBlock {
    public TransformManyBlock(Func<TInput,
        IEnumerable<TOutput>> transform);
    public TransformManyBlock(Func<TInput,
        Task<IEnumerable<TOutput>>> transform);
    public TransformManyBlock(Func<TInput,
        IEnumerable<TOutput>> transform,
        ExecutionDataflowBlockOptions dataflowBlockOptions);
    public TransformManyBlock(Func<TInput,
        Task<IEnumerable<TOutput>>> transform,
        ExecutionDataflowBlockOptions dataflowBlockOptions);

    public Task Completion { get; }
    public int InputCount { get; }
    public int OutputCount { get; }
    public void Complete();
    public IDisposable LinkTo(ITargetBlock<TOutput> target,
        bool unlinkAfterOne);
    public bool TryReceive(Predicate<TOutput> filter, out TOutput item);
    public bool TryReceiveAll(out IList<TOutput> items);
}
```

Remarks

TransformManyBlock takes two type parameters, so the input type can be

different from the output type.

TransformManyBlock is a one-to-zero-or-more transformation, where one input passed to the transform function results in zero or more outputs (in contrast to TransformBlock, where one input results in one output; also in contrast to ActionBlock, which has no output).

The transform function passed to the constructor can either be `Func<TInput, IEnumerable<TOutput>>` or `Func<TInput, Task<IEnumerable<TOutput>>>`, the difference being that the function is considered complete with the former when it returns and with the latter when the task completes executing.

Completion, Complete, LinkTo, TryReceive and TryReceiveAll work as described in IDataflowBlock, ISourceBlock and IReceivableSourceBlock.

TransformManyBlock implements ITargetBlock, ISourceBlock.[ReserveMessage|ConsumeMessage|ReleaseReservation] and IDataflowBlock.Fault explicitly, so a cast is required before calling these.

2.21.1: TransformManyBlock(Func<TInput, IEnumerable<TOutput>>)

Creates a TransformManyBlock instance with the transform function and default options.

```
public TransformManyBlock(Func<TInput,
    IEnumerable<TOutput>> transform);
```

Parameters

- transform (Func<TInput, IEnumerable<TOutput>>) - the transform function (may not be null)

2.21.2: TransformManyBlock(Func<TInput, Task<IEnumerable<TOutput>>>)

Creates a TransformBlock instance with the transform function and default options.

```
public TransformManyBlock(Func<TInput,
    Task<IEnumerable<TOutput>>> transform);
```

Parameters

- transform (Func<TInput, Task<TOutput>>) - the transform function (may not be null)

Remarks

The transform function completes when Task<IEnumerable<TOutput>> completes.

2.21.3: TransformManyBlock(Func<TInput, IEnumerable<TOutput>>, DataflowBlockOptions)

Create a TransformBlock instance with the transform function and custom

options.

```
public TransformManyBlock(Func<TInput,
    IEnumerable<TOutput>> transform,
    ExecutionDataflowBlockOptions dataflowBlockOptions);
```

Parameters

- transform (Func<TInput, Task<IEnumerable<TOutput>>>) - the transform function (may not be null)
- dataflowBlockOptions (DataflowBlockOptions) - options for this block

Remarks

The transform function completes when Task<IEnumerable<TOutput>> completes.

See the coverage of DataflowBlockOptions for a detailed discussion of how options work.

2.21.4: TransformManyBlock(Func<TInput, Task<IEnumerable<TOutput>>>, DataflowBlockOptions)

Create a TransformManyBlock instance with the transform function and custom options.

```
public TransformBlock(Func<TInput, Task<TOutput>> transform,
    ExecutionDataflowBlockOptions dataflowBlockOptions);
```

Parameters

- transform (Func<TInput, Task<IEnumerable<TOutput>>>) - the transform function (may not be null)
- dataflowBlockOptions (DataflowBlockOptions) - options for this block

Remarks

The transform function completes when Task<IEnumerable<TOutput>> completes.

See the coverage of DataflowBlockOptions for a detailed discussion of how options work.

2.21.5: InputCount

The number of messages stored in the input buffer.

```
public int InputCount { get; }
```

2.21.6: OutputCount

The number of messages stored in the output buffer.

```
public int OutputCount { get; }
```

2.22: WriteOnceBlock

A block containing a single message which may be written to just once and consumed any number of times.

```
public sealed class WriteOnceBlock<T> : IPropagatorBlock<T, T>,
ITargetBlock<T>, IReceivableSourceBlock<T>, ISourceBlock<T>,
IDataflowBlock {
    public WriteOnceBlock(Func<T, T> cloningFunction);
    public WriteOnceBlock(Func<T, T> cloningFunction,
        DataflowBlockOptions dataflowBlockOptions);
    public Task Completion { get; }
    public void Complete();
    public IDisposable LinkTo(ITargetBlock<T> target,
        bool unlinkAfterOne);
    public bool TryReceive(Predicate<T> filter, out T item);
}
```

Remarks

A block that stores the first message written to it, and afterwards only ever offers that message. Developers are used to concepts such as singleton, consts and readonly variables. WriteOnceBlock brings these ideas to TPL Dataflow. The assignment may happen only once (DataflowMessageStatus.Accepted is returned from ITargetBlock<T>.OfferMessage for first usage) and subsequent assignments are declined (DataflowMessageStatus.DecliningPermanently is returned from ITargetBlock<T>.OfferMessage).

Completion, Complete, LinkTo and TryReceive work as described in IDataflowBlock, ISourceBlock and IReceivableSourceBlock.

WriteOnceBlock implements ITargetBlock, ISourceBlock.[ReserveMessage|ConsumeMessage|ReleaseReservation], IDataflowBlock.Fault and IReceivableSourceBlock.TryReceiveAll explicitly, so a cast is required before calling these.

2.22.1: WriteOnceBlock(Func<T,T>)

Creates a WriteOnceBlock instance with the cloning function and default options.

```
public WriteOnceBlock(Func<T, T> cloningFunction);
```

Parameters

- cloningFunction (Func<T, T>) - the cloning function (may be null)

Remarks

This block returns the initial reference or a clone depending on whether null or a cloning function in passed in to the constructor.

2.22.2: WriteOnceBlock(Func<T,T>, DataflowBlockOptions)

Create a WriteOnceBlock instance with the cloning function and custom options.

```
public WriteOnceBlock(Func<T, T> cloningFunction,  
    DataflowBlockOptions dataflowBlockOptions);
```

Parameters

- cloningFunction (Func<T, T>) - the cloning function (may be null)
- dataflowBlockOptions (DataflowBlockOptions) - options for this block

Remarks

This block returns the initial reference or a clone depending on whether null or a cloning function is passed in to the constructor.

See the coverage of DataflowBlockOptions for a detailed discussion of how options work.