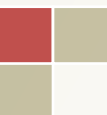


# MARC-HI Everest Framework

## Reference Guide

An in depth guide to the HL7v3 Messaging API created by the MARC-HI project.



Original Author: *Justin Fyfe*

Supplemental Authors: *Duane Bender, Trevor Davis*

Editing/Production: Mark Yendt, Brian Minaji

Proofing: Matthew Ibbotson, Corey Gravelle, Andria Chiaravalle, Jose Aleman, Craig Clark, Stuart Philp, Terence Cook, Brian VanArragon, Paul Brown



© 2009 Mohawk College of Applied Arts Technology  
P.O. Box 2034, Hamilton, Ontario, Canada L8N 3T2



Content for this document has been prepared by the Mohawk  
Applied Research Centre for Health Informatics.

Microsoft®, Microsoft Windows®, Visual Studio™ and the Microsoft .NET Framework are registered trademarks of Microsoft Corporation.

Novell® and SuSE™ are registered trademarks of Novell Corporation.

HL7® is a registered trademark of Health Level 7 Inc.

All other trademarks are the property of their respective owners.

Revision: 11

Publication Date: November 22, 2010

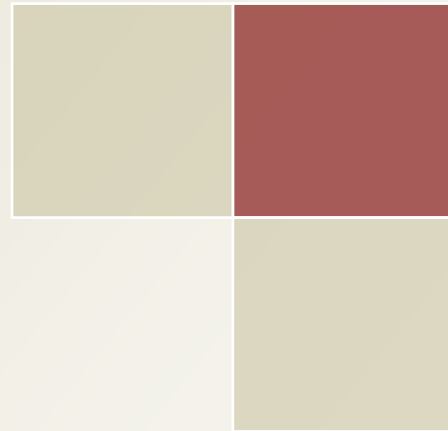
Under no circumstances may you copy, publish, broadcast, reproduce, re-engineer, create derivative works from, post, distribute, use commercially, make or express claims of ownership, authorship or contribution to, or incorporate any part of this document in original or modified form (for example in an abridged version), including, without limitation, its layout, color schemes, content, textual copy, styles, or keywords into any other work of similar nature, document, or any published work, for any reason whatsoever, without the express written consent of Mohawk College of Applied Arts And Technology. All rights are reserved

# Table of Contents

Chapter 1. Introduction .....	4
What is Everest? .....	5
The need for an API .....	5
Overview .....	6
Components of the Framework .....	6
Reading this Guide .....	7
Emblems .....	7
Data Types .....	9
Introduction .....	9
Foundation Classes .....	9
Codified Data .....	13
String / Encapsulated Data .....	15
Quantified Data .....	16
Collections .....	18
RMIM Classes .....	24
Introduction .....	24
Transmission Wrapper .....	25
Control Act Wrapper .....	30
Payload .....	32
Query Parameters .....	33
Abstract Classes .....	34
Introduction .....	37
Formatter Types .....	37
Generic Formatters .....	37
XML Formatters .....	38
Using Formatters .....	38
Graph Aides .....	42
Introduction .....	44
Connector Patterns .....	45
Connecting to Files .....	49
Publishing to a Directory .....	49
Subscribing to a Directory .....	50
Windows Communications Foundation .....	51
Consuming HL7v3 Services .....	52
Hosting HL7v3 Services .....	54
Message Queues .....	61
The General Purpose MIF Renderer .....	63
GPMR Usage .....	63
Optimizing COR Structures .....	66
Extracting API Meta Data .....	71
Structure Attributes .....	72
Property Attributes .....	73
Enumeration Attributes .....	74
Interaction Attributes .....	74
Interaction Response Attributes .....	74
Flavor Attributes .....	74
Flavor Map Attributes .....	75
Marker Attributes .....	75
State Attributes .....	76
State Transition Attributes .....	76

Writing Custom Components.....	76
Creating Formatters.....	76
Creating Connectors.....	78
Contents.....	85
WCF Standalone Service.....	85
BinFormatter Custom Formatter .....	87
MemoryConnector Custom Connector.....	89
Index of Figures .....	101
Index of Examples.....	101

# Chapter 1. Introduction



- *Knowledge pre-requisites*
- *What is the Everest Framework*
- *Overview of the Everest Framework*
- *Framework Components*

This document is a reference guide to accompany the MARC-HI Everest Framework. This document assumes some basic knowledge of the following concepts:

- The Microsoft® .NET Framework, or the Mono Framework
- HL7v3 at a conceptual level. The reader should be familiar with
  - What the RIM (Reference Information Model) is and its role,
  - What an RMIM (Refined Message Information Model) describes and how it relates to the RIM,
  - What an ITS (Implementable Technology Specification) does, and how it relates to the RIM/RMIMs.
- Object oriented programming concepts
- Basic knowledge of the MARC-HI Everest Framework

## What is Everest?

In short, the Everest Framework is designed to ease the creation, formatting, and transmission of HL7v3 structures.

The framework provides a series of consistent, flexible, well documented and powerful libraries, tools and documents that help developers transmit HL7v3 structures.

The HL7v3 API framework is derived directly from MIF (Model Information Format) files published by Canada Health Infoway and corresponds directly to the structures found within the published MIF files. Tools are provided that allow developers to render MIF files into different formats (for more information see "The General Purpose MIF Renderer" in Advanced Topics).

The current version of the Everest Framework is available for the Microsoft .NET framework.

## The need for an API

Application Programmer Interfaces (or APIs) are used in the software development industry to provide an abstract view of a complex model. APIs are known to dramatically improve developer productivity.

HL7v3 provides a rich information model intended for the semantic interoperability of healthcare systems. HL7v3 interfaces can be quite complex as a fully working system should be abstracted away from an ITS, implement the RIM, and communicate on (potentially) a number of different transports. Therefore HL7v3 interfaces are excellent candidates to be abstracted by APIs.

XML Schema files (XSDs) are provided for informative purposes by Canada Health Infoway as part of their standards release process. Software engineers are often tempted to use XML schema files for automated code generation to implement HL7v3 interfaces. This approach does not work well with HL7v3 structures for a number of reasons:

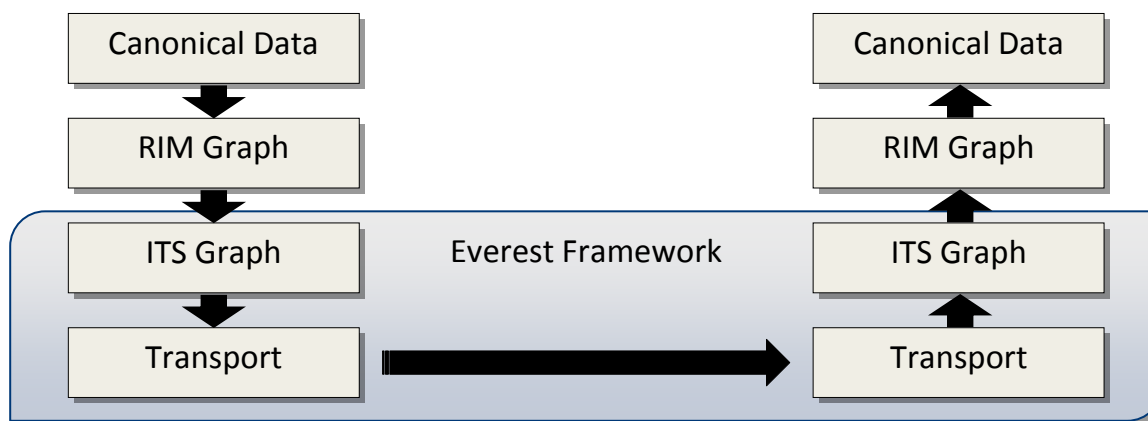
- Important RIM class information and documentation is lost during the translation from MIF to XSD
- Common classes are often not reused across interactions since they cannot be detected by code generation tools

- Automated code generation creates “empty classes” with no executable behavior
- Generating code from XSD (and/or WSDL) tightly couples the application to a single version of an ITS

Although not useful for generating code, these XSD files are useful for validation of XML instances.

## Overview

HL7v3 is designed to be loosely coupled, and a typical HL7v3 system is designed to be implemented using the architecture pictured below (Figure 1)



**Figure 1 - HL7v3 Architecture**

Applications will most likely store their data in an internal canonical format. This format is usually designed to meet the business needs of the application designer. However, it is (usually) impossible to send data in the canonical format directly to a remote system.

In order to communicate, both systems would need to map to a common format, in this case, HL7v3. In the diagram this is called the RIM Graph. The developer is responsible for mapping their canonical data to the RIM classes provided by the API.

After the canonical data is in this RIM format, the developer needs to select an ITS (for example, XML ITS 1.0) and transport mechanism (example: HTTP). The Everest Framework takes care of the serialization and transport processes automatically.

## Components of the Framework

There are four major components within Everest:

1. **Data Types** – The data type classes are a combination of HL7v3 R1 and R2 data types. They provide a rich set of functionality and are the primary components from which RMIM classes are comprised.
2. **RMIM Classes** – RMIM classes are auto-generated using the MIF files supplied by HL7v3, Canada Health Infoway or other organizations. They are a composition of data-types and provide data storage functionality.
3. **Formatters** – Formatters are responsible for translating the in-memory object representation of RMIM classes to a wire format (example: XML).

- 4. Connectors** – Connectors are responsible for transmitting RMIM structures to/from a variety of endpoints.

## Reading this Guide

This guide is formatted to provide a consistent reading experience. As such, you may find the following reading queues helpful

### Emblems



Identifies that a code sample is merely a snippet and that the complete code listing is available at the end of the document.



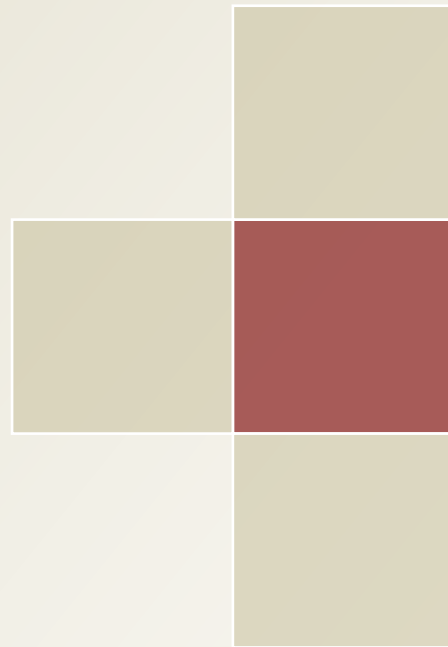
Identifies a feature that is new to the Everest Framework 1.0, or being removed or deprecated for the 1.0 release of the Everest Framework



Read these paragraphs very carefully as they perform some important functionality that should be fully understood



## Chapter 2. Creating Instances



- *The role of Data Types and RMIM Classes*
- *Creating, modifying, converting and validating data types*
- *The roles of the instance wrapper classes*
- *Finding payload information in an instance*

Formatters are used in the chapter for demonstrative purposes. It is not expected that developers will have a complete understanding of formatters by the end of this chapter.

## Data Types

Data types from an HL7v3 standpoint represent the core functionality of the messaging standard. They are used as the building blocks for the larger RMIM structures.

All structures (Data types or RMIM) implement the IGraphable interface. This empty interface is used to mark the class as serializable by a formatter.

## Introduction

The Everest framework implements the HL7v3 data types according to the R2 structures (with R1 data type implementation where no R2 data type could accurately reflect the contents). This design is intentional as it allows developers to produce wire level documents using R1 or R2 representations (More about that in the “Formatters” Chapter).

The data types contain a variety of constructors, methods, and operator overloads that help developers easily interact with the data types. There are five main categories of data types:

- **Foundation** – These are classes that provide a foundation for other data types (examples: PDV, ANY, HXIT, etc...) as well as the fundamental data types that make up HL7v3 (II, etc..)
- **Codified** – These classes are used to transport codified data (examples: CS, CV, CE, etc...)
- **Encapsulated Data** – These data types may be used to encapsulate raw data within an instance (examples: ST, ED, etc...)
- **Quantified Data** – Represents data that can be expressed as a quantity (examples: INT, REAL, QTY, TS, etc...)
- **Collections** – Represents a series, or range of instances. Provides methods to find, sort, and modify the collection (examples: SET, LIST, etc...)

## Foundation Classes

### HXIT (History Item)

The HXIT data type is used to express history information about a particular data type. It tags the time(s) that a particular data type value is valid, as well as the control act event identifier that was responsible for setting the value of the data type.

The HL7v3 specification states that an HXIT data type should be created using the following definition:

```
public class HXIT<T> : T
```

This is not possible in any commonly used language. Therefore, the Mohawk API uses the HXIT class as the basis for all classes.



Data within the HXIT (ControlActRoot/Ext and ValidTimeLow/High) are not rendered by the R1 data formatter and are ignored

## ANY

The ANY data type defines the basic properties for every data type in the Mohawk College API. It contains the definition for a data type's null flavor, update mode, and flavor identification.

### Null Flavor

A null flavor is an exceptional code that hints to the reason as to why a particular data type value is not being passed within an instance. The core data types can have one of the values listed in Table 1.

**Table 1 - Null flavors**

Flavor	Description
Asked Unknown	Information was sought but not found.
Derived	An actual value may exist but it must be derived from other information provided.
Invalid	The value as represented in the instance is not a member of the set of permitted values.
Masked	There is information for this item but it has not been provided due to security, privacy or other reasons.
Not Applicable	The value is known to have no proper value (example: Menstrual cycle for a Male).
Not Asked	The information has not been sought.
Not Available	Information is currently not available, but is expected to be available at a later time.
No Information	No information could be inferred from this excepted value.
Negative Infinity	Negative infinity of numbers.
Positive Infinity	Positive infinity of numbers.
Sufficient Quantity	Specific quantity is not known, but it is known to be greater than zero and is not included because it makes up the bulk of the material.
Other	The actual value is not a member in the value domain of the variable.
Trace	The content is greater than zero, but too small to be quantified.
Un-encoded	The actual value has not yet been encoded within the approved value set.
Unknown	A proper value is applicable, but is not known.

### Update Mode

The update mode dictates how the value of the data type should be treated in an update interaction.

**Table 2 - Update modes**

Update Mode	Description
Add	The item is to be added if it does not currently exist.
Remove	The item is to be removed.
Replace	If the item exists, it is to be replaced.
Add or Replace	If the item exists, it is to be replaced, if it does not exist, it will be added.
No Change	No change is to be made.
Unknown	It is not specified what kind of change this item has or will perform.
Key	This item is part of the identifying information for the object that contains it.

### Flavor

A flavor (or in R1, SpecializationType), modifies the data type so that it behaves differently. For example, imposing a flavor of BUS on an instance identifier (II) automatically sets the "Use" to Business and removes the displayable attribute.

### PDV (Primitive Data Value)

The Primitive Data Value (PDV) data type is a pseudo type created to ease the implementation of classes that wrap a primitive data type.

### II (Instance Identifier)

The Instance Identifier data type is used to uniquely identify objects across domains (roots). In HL7v3 this is the primary data type used to identify anything within a message.

There are many different flavors of the instance identifier data type (listed in Table 3)

**Table 3 - Flavors of II**

Flavor	Restrictions	Description
BUS	<ul style="list-style-type: none"><li>- Use must be "Business"</li><li>- Displayable cannot be set (must be null)</li></ul>	The instance identifier is to be used in a business context only.
PUBLIC	<ul style="list-style-type: none"><li>- Displayable must be true</li></ul>	The instance identifier is globally unique and can be viewed by a data enterer.
TOKEN	<ul style="list-style-type: none"><li>- Displayable cannot be set (must be null)</li><li>- Only root may be set (no extension)</li></ul>	The instance identifier is used as a token, or a locally scoped identifier (example: Message identifier)

The II data type is undergoing some changes with data types R2, the properties marked with <sup>(R2)</sup> are used to identify properties that will only be rendered with R2.

### Root

The root of an instance identifier represents a unique identifier that quarantines the global uniqueness of the instance identifier. This value is usually in the form of an OID (except for the II.TOKEN flavor).

### Extension

The extension represents a character string that is a unique identifier within the scope of the identifier domain.

### IdentifierName



The identifier name property is used as a human readable version of the identifier. This property is only supported in data types R2.

### Displayable

If true, indicates that the identifier is intended for human display and data entry. False indicates that identifier is intended to be used for machine processing.

### Scope

The scope property indicates the scope in which the identifier applies to the object. The possible values are listed in Table 4.

**Table 4 - IdentifierScope values**

Scope	Description
BusinessIdentifier	An identifier associated with the object due to the business practices associated with the object.
ObjectIdentifier	The identifier associated with a particular object.
VersionIdentifier	An identifier references a particular instance of an object.
ViewSpecificIdentifier	An identifier for a particular snapshot of a version of the object.



The scope property is only rendered by the R2 data types formatter and its content is ignored by the R1 formatter.

### Reliability

The reliability property specifies the reliability of the instance identifier. Possible values are listed in Table 5.

**Table 5 - IdentifierReliability values**

Reliability	Description
IssuedBySystem	The identifier was issued by the system responsible for constructing the instance.
VerifiedBySystem	The identifier was not issued by the system responsible for constructing the instance but was verified by it.
UsedBySystem	The identifier was provided to the system that constructed

	the instance but can't be verified.
--	-------------------------------------



The reliability property is only rendered by the R2 data types formatter and its content is ignored by the R1 formatter

### AssigningAuthorityName

The AssigningAuthorityName property identifies the authority responsible for assigning and maintaining the identifier.

#### Use

This property identifies how the instance identifier is intended to be used. Possible values are listed in Table 6.

**Table 6 - IdentifierUse values**

Use	Description
Business	Signifies that the II is intended to be used for business purposes.
Version	Signifies that the II is intended to be used for identifying a version of the particular instance.



The identifier Use property is only rendered by the R1 data types formatter and its value is ignored when using the R2 formatter.

## Codified Data

Codified data is used to communicate data that is restricted to a particular concept domain, value set or any codified data.

Whenever coded data is to be drawn from a structural code domain, the data type (CV,CS, CD, etc...) will be bound to an enumeration. Values from this enumeration can be assigned directly using the equals (=) operator.

If codified data is bound to a code-set, concept domain or value set, the "codeSystem" attribute needn't be set as the formatter will set this on behalf of the developer.

To determine which value set a codified data structure is bound to, or to see if it is bound to any concept domain, check the WIKI documentation (<http://wiki.marc-hi.ca>) for the particular property, or use the code documentation.

### CS (Coded Simple)

The coded simple data type is used to represent codified data in its simplest form where only the code is not predetermined.

The CS data type provides three ways to create an instance as shown in Example 1.

```
// Using a constructor
instance.ResponseModeCode = new CS<ResponseMode>(ResponseMode.Queue);
// Using direct assignment
instance.ResponseModeCode = ResponseMode.Queue;
// Using properties
instance.ResponseModeCode = new CS<ResponseMode>();
instance.ResponseModeCode.Code = ResponseMode.Queue;
```

## Example 1 - Instantiating CS

### CV (Coded Value)

The coded value data type is an example of a flavor that has been mapped into a full class in order to remain compatible with R1 data types. In R2 the CV data type is actually the CD (Concept Descriptor) data type with a flavor of CV.

CV extends the CS data type by adding display name, coding rationale, and original text.

#### Original Text

This value represents the basis as to why the specified code is selected. It is an icon, picture, or text describing what led to the selection of the code.

### CE

The CE data type is another class that is created to remain compatible with R1 data types (in R2 it represents a CD with a flavor of CE). The CE builds on the CV by adding translations for the code.

#### Translation

Translation is a set of concept descriptors that is used to represent the selected code mnemonic in different code or value sets, or in a different version of the selected code system.

### CD (Concept Descriptor)

A CD represents any kind of concept usually by giving a code defined in a code system. A CD builds upon a CE by allowing the assignment of qualifiers.

#### Qualifier

Qualifier is used to qualify or further describe the concept. For example, to describe "LEFT FOOT", the qualifier "LEFT" is applied to the primary code "FOOT".

```
// Declare the Concept Descriptor (Burn of Skin)
CD<String> cd = new CD<String>();
cd.Code = "284196006";
cd.DisplayName = "Burn of Skin";

// Declare Severity as Severe so code becomes Severe Burn of Skin
CR<String> severity = new CR<String>();
severity.Name = new CD<string>();
severity.Value = new CD<string>();
cd.Qualifier = new LIST<CR<string>>();
cd.Qualifier.Add(severity);
severity.Name.Code = "246112005";
severity.Name.CodeSystem = "2.16.840.1.113883.6.96";
severity.Name.CodeSystemName = "SNOMED CT";
severity.Name.DisplayName = "Severity";
severity.Value.Code = "24484000";
severity.Value.CodeSystem = "2.16.840.1.113883.6.96";
severity.Value.CodeSystemName = "SNOMED CT";
```

```
severity.Value.DisplayName = "Severe";

// Add severity to container code
cd.Qualifier = new LIST<CR<String>>();
cd.Qualifier.Add(severity);
```

### Example 2 - Concept Qualifiers

## String / Encapsulated Data

Encapsulated data types provide a mechanism for appending binary, textual or XML data to an instance. Encapsulated data (and its counterpart, String) can be used as a place to enter free-form data.

### ST (String)

The ST data type is used for encapsulating primitive strings. In data types R2 it is possible for a string to contain translations to/from different languages. There are three ways to assign a string value as shown in Example 3.

```
// String Hello! with current language code
ST test = new ST();
test.Value = "Hello!";
// String Hello! with current language code
test = "Hello!";
// String Hello! in a different language
test = new ST("Bonjour!", "fr-ca");
```

### Example 3 - Using the ST class

### ED (Encapsulated Data)

Encapsulated data is used to transport larger portions of text, binary and XML data within an instance. The ED data type is self-aware of its content and will be formatted appropriately for whatever transport is used.

The ED data type can be used to carry any type of information.

```
// Using constructor
ED t = new ED(File.ReadAllBytes(@"test.pdf"), "application/pdf");
```

### Example 4 - Loading a PDF into an ED

### Integrity Check

The ED data type provides an easy mechanism to automatically compute checksums for the data contained within. Computing an integrity check is simple. All the developer needs to do, is set the *IntegrityCheckAlgorithm* property, and set the integrity check to the result of the *ComputeHash* function. If an integrity check algorithm is specified, and no integrity check is assigned, it will be automatically generated when the "IntegrityCheck" value is retrieved.

```
ED edTest = new ED(File.ReadAllBytes("test.xml"), "text/xml");
edTest.Representation = EncapsulatedDataRepresentation.XML;
edTest.IntegrityCheckAlgorithm = EncapsulatedDataIntegrityAlgorithm.SHA1;
// This will be auto-generated if you don't explicitly call it
edTest.IntegrityCheck = edTest.ComputeIntegrityCheck();
Console.WriteLine(Convert.ToBase64String(edTest.IntegrityCheck));
```

### Example 5 - Using integrity checks



When validating an instance of ED, the developer sets the integrity check algorithm, the data and then calls the *ValidateHash* function.

## Reference

The reference property is used to indicate to a recipient that the data is not contained within the instance; however it is located in a publicly available location. This concept is used when the data being referenced is too large to send within the instance (example: Referencing a 30 MB image)

## Thumbnail

The thumbnail property can be used with the reference property to provide an inline "sample" of the data.

```
ED t = new ED((TEL)"http://1.2.12.32/images/big.tif");
t.Thumbnail = new ED(File.ReadAllBytes(@"C:\small.png"), "image/png");
```

### Example 6 - Reference with thumbnail

## Compression

Compressing an ED is similar to generating an integrity check. Example 7 illustrates how to properly compress, assign an integrity check and then check the value is decompressed.

```
// Create instance
ED test = new ED(File.ReadAllBytes("C:\\output.pdf"), "text/xml");
Console.WriteLine("Original: {0}", test.Data.Length);

// Setup compression
test.Compression = EncapsulatedDataCompression.GZ;
test.Data = test.Compress();
Console.WriteLine("Compressed: {0}", test.Data.Length);

// Setup integrity check
test.IntegrityCheckAlgorithm = EncapsulatedDataIntegrityAlgorithm.SHA256;
test.IntegrityCheck = test.ComputeIntegrityCheck();

// Validate
if (!test.ValidateIntegrityCheck())
    Console.WriteLine("Integrity check doesn't match!");
else
{
    test.Data = test.UnCompress();
    Console.WriteLine("UnCompressed: {0}", test.Data.Length);
}
```

### Example 7 - ED with compressed data

## Quantified Data

Quantified data types are used to express a quantity, or quantifiable (measurable) data. This includes boolean and date values. All quantified data types provide an operator overload that allows a developer to treat the data type as a primitive.

## BL (Boolean Value)

A BL, or boolean value can be used to represent either a true, false or null state. This data type provides its own boolean logic that integrates the concept of null flavors. The methods AND, OR, NOT and XOR are available.

## TS (Time Stamp)

The TS data type is used for expressing a unit of time. It encapsulates a native DateTime object, and can be assigned from one.

It is important to note that the value of the TS itself is a string (the representation of the date in string format). The DateValue property of the TS should be used when assigning, and comparing the value.

```
// TS Sample, Time Zone is EDT
TS test = DateTime.Parse("March 1, 2009 12:00:00 AM");
Console.WriteLine(test.ToString()); // output : 20090301000000.0000-0400
test.Flavor = "TS.DATETIME";
Console.WriteLine(test.ToString()); // output : 20090301000000
test.Flavor = "TS.DATE";
Console.WriteLine(test.ToString()); // output : 20090301
```

### Example 8 - Flavors of TS

## DateValuePrecision

The DateValuePrecision property dictates the precision of the DateValue property. A DateTime object within the .NET framework measures its dates/times in "Ticks".

The problem arises from HL7v3's support for less precise dates. For example, an HL7v3 message may specify January 2009, however the .NET DateTime object will understand this as January 1, 2009 0:00:00. In order to compensate for this problem, the TS data type provides a DateValuePrecision property which dictates the actual precision of the date/time.

The available values for the DateValuePrecision property are found in Table 7.

**Table 7 - DateValuePrecision codes**

Value	Example (HL7v3)	Description
Year	2009	Date is precise to the year. This value means anything from Jan 1 2009 to Dec 31 2009.
Month	200901	Date is precise to the month. This value means anything from Jan 1 2009 to Jan 31 2009.
Day	20090101	Date is precise to the day. This value means anything from 0:00:00.0000 Jan 1 2009 to 11:59:59.999 Jan 1 2009.
Hour	2009010115	Date is precise to the hour. This value means anything during 3:00 PM on Jan 1 2009.
Minute	200901011545	Date is precise to the minute. Meaning anything during 3:45 PM on Jan 1 2009.
Second	20090101154522	Date is precise to the second. Meaning anything during 3:45:22 on Jan 1 2009.

Full	20090101154522.0000-0500	DateValue has full precision.
------	--------------------------	-------------------------------

## INT (Integer)

The integer data type represents a 32 bit signed integer value (wraps Int32).

## REAL (Floating Point Number)

The real data type represents a 32 bit floating point value (wraps double)

## MO (Monetary Ordinal)

The monetary amount data type is an extension of the REAL data type and adds a field that allows the developer to specify an ISO 4217 code representing the currency the MO is represented as.

## PQ (Provisioned Quantity)

The PQ, or provisioned quantity data type represents a quantity that is the result of measurement. The value of the PQ is a floating point number, and it is expected the developer will assign a unit drawn from the UCUM(Unified Code for Units of Measure) code set.

The PQ also provides a mechanism for translating the measurement. Example 9 illustrates translating a height measurement in meters to feet.

```
// 2.1 Meters
PQ h = new PQ(2.1f, "m");
// Translates to 6.8897 ft
h.Translation = new SET<PQR>(new PQR(6.8897f, "ft_i",
"2.16.840.1.113883.6.8"), PQR.Comparator);
```

### Example 9 - PQ with metric and imperial measures

## PQR (Provisioned Quantity – Coded Value)

PQR differs from a PQ in that a PQ's "Unit" is selected from the UCUM code set. A PQR may have its unit selected from a code system other than UCUM. PQR extends CD to allow the developer to select any code system they wish for a unit.

## UVP (Universal with Probability)

The UVP data type represents a quantity with a statistical probability assigned to the value. The probability is a decimal number between 0 (none) to 1 (100%)

## Collections

Collections are data types that represent series, lists, intervals and other various collections of other data types.

## SET (Sequence)

The SET data type represents a collection of items in no particular order, where each item in the list is unique. When creating a set, a comparator must be assigned to the set before any items can be added.

### Union

The union function within a set will add all the unique items between two sets into a newly created set. Example 10 illustrates the use of the Union function.

```
// Create sets
SET<II> set1 = new SET<II>(new II("1.1.1.1", "1234"), II.Comparator);
SET<II> set2 = new SET<II>(new II("1.1.1.1", "12345"), II.Comparator);
SET<II> set3 = set1.Union(set2); // set3 has two items
```

#### Example 10 - Union of two sets

### Except

The except function will create a new set containing all the items in set A without the items in set B. Example 11 illustrates the use of the Except function.

```
// Create sets
SET<II> set1 = new SET<II>(new II[]
{
    new II("1.1.1.1", "1"),
    new II("1.1.1.1", "2"),
    new II("1.1.1.1", "3")
}, II.Comparator);
SET<II> set2 = new SET<II>(new II[]
{
    new II("1.1.1.1", "3"),
    new II("1.1.1.1", "4")
}, II.Comparator);
SET<II> set3 = set1.Except(set2); // set3 has two items (1,2)
```

#### Example 11 - Exception of two sets

### Find / Find All

The find and find all functions will return the first (or all) item(s) that match the expression provided. Example 12 illustrates the use of the Find function.

```
// Create set
SET<II> set1 = new SET<II>(new II[]
{
    new II("1.1.1.1", "1"),
    new II("1.1.1.1", "2"),
    new II("1.1.1.1", "3")
}, II.Comparator);

// .NET 3.5 and Mono
II ii2 = set1.Find(ii => ii.Extension == "2" && ii.Root == "1.1.1.1");
// .NET 2.0
ii2 = set1.Find(delegate(II ii)
{
    return ii.Extension == "1" && ii.Root == "1.1.1.1";
});
```

#### Example 12 - Find / Find All

### Intersect

The intersect method will return a new SET instance that contains only the values that appear both in set A and set B. Example 13 illustrates the use of the Intersect function.

```
// Create sets
SET<II> set1 = new SET<II>(new II[]
{
```

```

        new II("1.1.1.1", "1"),
        new II("1.1.1.1", "2"),
        new II("1.1.1.1", "3")
    }, II.Comparator);
SET<II> set2 = new SET<II>(new II[]
{
    new II("1.1.1.1", "3"),
    new II("1.1.1.1", "4")
}, II.Comparator);
SET<II> set3 = (SET<II>)set1.Intersection(set2); // set3 has one item ("3")

```

### Example 13 - Intersecting two sets

## LIST (Collection)

The LIST class represents discrete (but not necessarily unique) items in a sequence. The LIST class behaves much like the .NET Generic List (with the exception of HL7v3 attributes of null flavor, update mode, etc.). The functionality that behaves like the .NET Generic List class will not be illustrated here.

### Sub Sequence

The sub-sequence method retrieves a sub-selection of items from the collection. The example below illustrates the use of the sub-sequence method

```

// Create list
LIST<II> set1 = new LIST<II>(new II[]
{
    new II("1.1.1.1", "1"),
    new II("1.1.1.1", "2"),
    new II("1.1.1.1", "3"),
    new II("1.1.1.1", "3"),
    new II("1.1.1.1", "4")
});
// From set1[1] - set1[2]
LIST<II> set2 = (LIST<II>)set1.SubSequence(1, 2); // contains 2 items
// From set1[2]
set2 = (LIST<II>)set1.SubSequence(2); // set2 contains 3 items

```

### Example 14 - Sub-sequence

## IVL (Interval)

The interval data type represents a range collection. Within an interval, at least two of the following parameters must be populated: Low, Width, or High. An interval also indicates whether the upper or lower bounds are closed (inclusive) or not. An interval may also specify a value that represents the center of the interval.

Example 15 illustrates a time interval from October 1 to November 14 2009.

```

IVL<TS> effectiveTime = new IVL<TS>
(
    DateTime.Parse("October 1, 2009"),
    DateTime.Parse("November 15, 2009")
);
effectiveTime.LowClosed = true;
effectiveTime.HighClosed = true;
effectiveTime.Operator = SetOperator.Inclusive;

```

### Example 15 - Interval from Oct 1 2009 to Nov 15 2009

## PIVL (Periodic Interval)

A periodic interval represents an interval that occurs periodically. The PIVL has two main properties: Phase and Period.

Example 16 illustrates a periodic interval that repeats the interval specified above every year.

```
IVL<TS> effectiveTime = new IVL<TS>
(
    DateTime.Parse("October 1, 2009"),
    DateTime.Parse("November 15, 2009")
);
effectiveTime.Operator = SetOperator.Inclusive;
// Repeat the interval every year
PIVL<TS> pEffectiveTime = new PIVL<TS>(
    effectiveTime,
    new PQ(1.0f, "y")
);
```

### Example 16 - Periodic Interval

## AD (Address)

The address data type is a collection of address parts (ADXP) that make up an address. This makes the AD a powerful data type for representing any commonly used format of an address.

The example below illustrates the construction of an address for 123 Main Street W, Hamilton ON Canada

```
AD test = new AD(
    PostalAddressUse.HomeAddress,
    new ADXP[] {
        new ADXP("123", AddressPartType.BuildingNumber),
        new ADXP("Main", AddressPartType.StreetNameBase),
        new ADXP("Street", AddressPartType.StreetType),
        new ADXP("West", AddressPartType.Direction),
        new ADXP("Hamilton", AddressPartType.City),
        new ADXP("Ontario", AddressPartType.State),
        new ADXP("Canada", AddressPartType.Country),
        new ADXP("L8K 3K4", AddressPartType.PostalCode)
    });
```

### Example 17 - Instantiating the AD data type

The AD data type also contains a method to format the address as a string. Example 18 illustrates this functionality. The applicable formatting strings are outlined in Table 8.

```
Console.WriteLine(test.ToString("{BNR} {STB} {STTYP} {DIR}\r\n{CTY},\r\n{STA}\r\n{CNT}\r\n{ZIP}"));
// Output:
// 123 Main Street West
// Hamilton, Ontario
// Canada
// L8K3K4
```

### Example 18 - Formatting an address

## Table 8 - AD Part types and format strings

<b>Format</b>	<b>Part Type</b>	<b>Description</b>
ZIP	Postal Code	A postal code designating a region defined by the post service
STA	State	A sub-unit of a country
PRE	Precinct	A subsection of a municipality
POB	Post Office Box	A numbered box located in a post station
DEL	Delimiter	Delimiters are printed without framing white space
CTY	City	The name of the city, town, village, etc...
CPA	County	A sub-unit of a state or province
CNT	Country	Country
CEN	Census Tract	A geographic sub-unit delineated for demographic purposes
CAR	Care Of	The name of the party who will take receipt at the specified address
DIR	Direction	The direction (N, W, S, E)
STTYP	Street Type	The designation given to the street (Ave, St, Rd)
STB	Street Name Base	The base name of a roadway, or artery recognized by a municipality
STR	Street Name	The name of the street including the type
BNS	Building Number Suffix	Any alphabetic character, fraction or other text that may appear after the numeric portion of a building number
BNN	Building Number Numeric	The numeric portion of a building number
BNR	Building Number	The number of a building, house or lot alongside the street
SAL	Street Address Line	A full street address line, including number details and the street name and type
DMODID	Delivery Mode ID	Represents the routing information such as a letter carrier route number
DMOD	Delivery Mode	Indicates the type of service offered, method of delivery. Example: A PO box
DINSTQ	Delivery Installation Qualifier	A number, letter or name identifying a delivery location (eg: Station A)
DINSTA	Delivery Installation	The location of the delivery location, usually a town or city

	Area	
DINST	Delivery Installation Type	Indicates the type of delivery installation (facility to which the mail will be delivery prior to final shipping)
DAL	Delivery Address Line	A delivery address line is frequently used instead of breaking out delivery mode, delivery installation
UNIT	Unit Designator	Indicates the type of specific unit contained within a building
UNID	Unit Identifier	The number or name of a specific unit contained within a building
ADL	Additional Locator	This can be a unit designator such as apartment number, suite, etc
AL	Address Line	An address line is for either an additional locator, a delivery address or a street address

## EN (Entity Name)

The entity name data type acts much like the address data type in that it encapsulates a series of entity name parts (ENXP). Example 19 illustrates the instantiation and formatting of the name: James Tiberius Kirk.

```
EN name = new EN(
    EntityNameUse.Legal,
    new ENXP[] {
        new ENXP("James", EntityNamePartType.Given),
        new ENXP("Tiberius", EntityNamePartType.Given),
        new ENXP("Kirk", EntityNamePartType.Family)
    });
Console.WriteLine(name.ToString("{FAM}, {GIV}"));
// output: Kirk, James Tiberius
```

### Example 19 - Instantiating the EN data type

The format strings (and part types) are listed in Table 9 below.

**Table 9 - Name part types and format strings**

Format	Part Type	Description
FAM	Family	The family name that links to the genealogy
GIV	Given	Given name (first, middle or other given names)
SFX	Suffix	A suffix has a strong association to the part immediately before the name part
PFX	Prefix	A prefix has a string association to the immediately following name part
DEL	Delimiter	A delimiter has no meaning other than being literally printed in this name representation



## RMIM Classes

RMIM classes are compositions of base data types that form a specific message type.

### Introduction

The RMIM classes in the Mohawk API are automatically generated from published Model Interchange Format (MIF) files published by various standards organizations. Currently, Everest is distributed with the pre-generated assemblies outlined in Table 10.

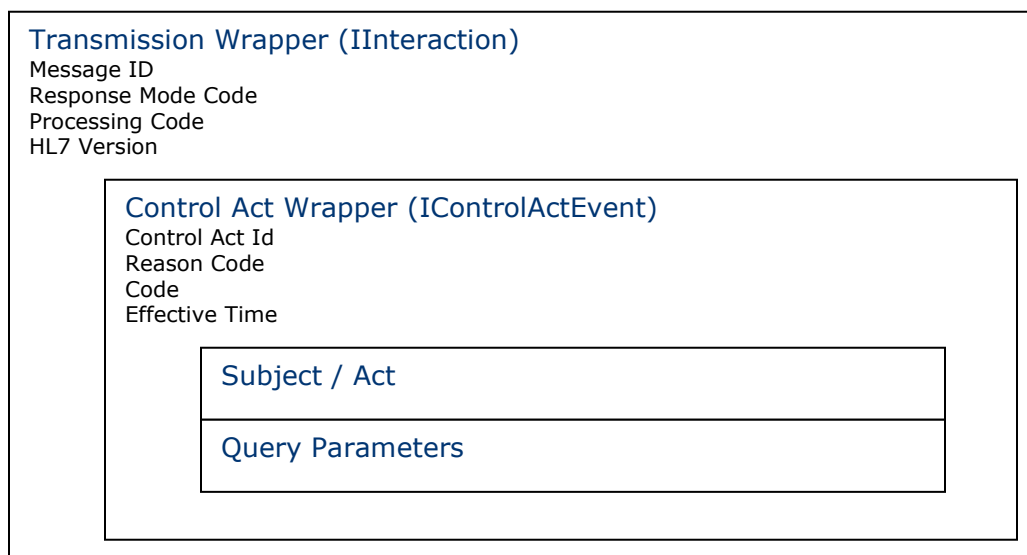
**Table 10 - Bundled RMIM Classes**

Assembly	SDO	Realm	Standard
MARC.Everest.RMIM.CA.R020401	Canada Health Infoway	Canada	MR2009 Delta 1
MARC.Everest.RMIM.CA.R020402		Canada	MR2009 Delta 2
MARC.Everest.RMIM.CA.R020403		Canada	MR2009 Delta 3
MARC.Everest.RMIM.UV.NE2008	HL7 Intl.	Universal	Normative Ed. 2008

Because the RMIM classes are automatically generated, it is possible for developers to add new interactions and message types to the API. This is covered in more detail in the chapter “Advanced Topics”.

This section will illustrate generic concepts of the RMIM classes using message types that are distributed with the Mohawk API. These concepts should apply to all generated classes.

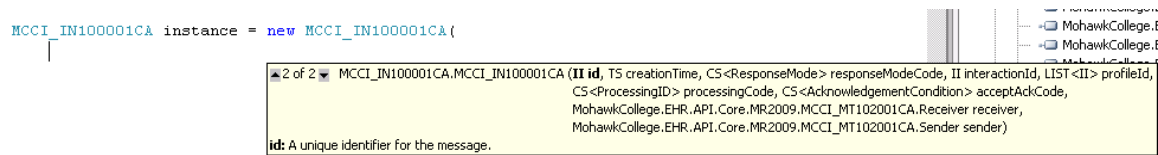
Before an understanding of the message wrappers can be attained, it is important to understand the layout of an HL7v3 message, as shown in Figure 2.



**Figure 2 - Message Layers**

An interaction is made up of a “Transmission Wrapper” which encapsulates a “Control Act” wrapper. The control act wrapper can include a payload (or subject), query parameters or both.

All auto-generated RMIM classes contain detailed documentation that can be used when constructing a new instance (see Figure 3)



**Figure 3 - IntelliSense Help**

In addition to the documentation, RMIM classes are created with up to five different constructors:

1. **Default** – Creates an empty instance.
2. **Mandatory Structural** – Provides all structural mandatory elements as parameters.
3. **Mandatory** – Provides all mandatory elements as parameters.
4. **Required** – Provides the mandatory and required properties as parameters.
5. **Shortcut** – If a node is “empty” or provides no additional information, then a shortcut constructor is provided.

## Transmission Wrapper

A transmission wrapper in the API is defined as a class that implements the MARC.Everest.Interfaces.IInteraction interface. There are two major types of transmission wrappers: Initiator (or request) and Responder (or response) transmission wrappers.

Most of the transmission wrappers in the API are identical and contain the fields found in Table 11.

**Table 11 - IInteraction members**

Type / Field	Conformance	Description
TS : Creation Time	Mandatory	Identifies the time that the message was created
CS : Processing Code	Mandatory	Indicates how the recipient should process the message
II : Interaction Id	Mandatory	Indicates the ID of the interaction the sender is attempting to execute
CS : Response Mode	Mandatory	Indicates how the sender is expecting a response
II[] : Profile Id	Mandatory	Indicates the conformance profiles the sender is claiming conformance to
CS : Accept Ack Code	Mandatory	Indicates under what conditions the sender wishes to receive acknowledgements
II : Id	Mandatory	A unique identifier for the message

```
// Create instance
```

```

MCCI_IN100001CA instance = new MCCI_IN100001CA(
    // The Message Identifier
    new II(System.Guid.NewGuid().ToString()),
    // The Creation Time
    DateTime.Now,
    // The Response Mode
    ResponseMode.Immediate,
    // The Interaction Identifier
    MCCI_IN100001CA.GetInteractionId(),
    // The Profile Id
    MCCI_IN100001CA.GetProfileId()
    ,
    // The Processing Mode
    ProcessingID.Production,
    // The Acknowledgement Condition
    AcknowledgementCondition.Always,
    // The intended recipient
    new MARC.Everest.Core.MR2009.MCCI_MT102001CA.Receiver(
        new MARC.Everest.Core.MR2009.MCCI_MT102001CA.Device2(
            new II("1.2.2.2", "1") // id
        )
    ),
    // The sending device node
    new MARC.Everest.Core.MR2009.MCCI_MT102001CA.Sender(
        new MARC.Everest.Core.MR2009.MCCI_MT102001CA.Device1(
            new II("1.2.2.2.2", "123") // id
        )
    )
); // Instance

```

#### Example 20 - Creating transmission wrapper

**Note:** When creating a transmission wrapper, it is important that the developer populate as much data as possible.

#### Id (Message Identifier)

The 'id' field of the transmission wrapper is used to uniquely identify the message instance. The instance identifier used for the message identifier only needs to be populated with the 'root' attribute.

#### CreationTime (Message creation time)

The 'creationTime' property of a transmission wrapper identifies the date and time that the particular message instance was created. For many implementations, the current system date and time will suffice.

#### ResponseModeCode (Response Modality)

The 'responseModeCode' property identifies the manner in which the remote system is expected to respond. This property is bound to the value set "ResponseMode". Possible values are outlined in Table 12.

**Table 12 - Response mode codes**

Name (Enumeration)	Mnemonic (Instance)	Description
Immediate	I	Indicates the action is to be executed immediately. The response of the request

		message is usually the outcome of the entire interaction.
Queued	Q	Indicates the action is to be placed on a queue, to be executed at a later time. The sender has the responsibility to poll the outbound message queue of the receiver to retrieve the response of the request. The receiver is also responsible for correlating the queued response with the request.
Deferred	D	Indicates the action is to be placed on a queue, to be executed at a later time. When the receiver has processed the message, the receiver will actively open a connection to the sender and respond.  <b>Note:</b> The sender must actively listen and wait for a response to the action.

### **InteractionId (Interaction Identifier)**

The 'interactionId' property is used to identify the interaction pattern that is being executed by the sender on the receiver. This value will usually match the name of the interaction type.

### **ProfileId (Conformant with profiles)**

This property is used to identify the conformance profile(s) that this message claims to be conformant with.

### **ProcessingCode (Processing Mode)**

The 'processingMode' code is used to identify the manner in which the sender expects the recipient to process the message.

**Table 13 - Processing codes**

<b>Name (Enumeration)</b>	<b>Mnemonic (Instance)</b>	<b>Description</b>
Production	P	Indicates the message is to be processed as though it were in a production environment
Debug	D	Indicates that the message is to be processed in a debugging modality. This may alter the way that the receiver commits data.
Training	T	Indicates the message is to be processed as though it were in a training mode.

### **AcceptAckCode (Desired acknowledgement)**

This code is used to dictate under which conditions the receiver must send back acknowledgements. On a request, this value is usually set to "Always", on responses the value is set to "Never" (as an acknowledgement does not need to be acknowledged).

**Table 14 - Accept acknowledgment codes**

<b>Name (Enumeration)</b>	<b>Mnemonic (Instance)</b>	<b>Description</b>
Always	AL	Always send back an acknowledgement.
ErrorRejectOnly	ER	Only send back an acknowledgement if the result of processing the interaction was not successful.
Never	NE	Never send an acknowledgement

### **Receiver (Intended Recipient)**

The 'receiver' property identifies the system that is intended to process the request. If the actual recipient is different from the intended recipient, it may be routed to the intended recipient.

### **Sender (Initiating System)**

The 'sender' property is used to define details about the initiating system. Depending on the recipient, this information may be used for

- Access control (example: can this device initiate this action?),
- Routing responses (example: what device do I route the response to?),
- Etc.

## **Responses**

A response interaction is constructed in the same manner that a request is constructed. An acknowledgement is responsible for answering the following questions:

- What is the result of processing?
- What is the message being acknowledged?
- What are the details of processing?

### **TypeCode (Acknowledgement type)**

The type code property is used to express the overall outcome of the processing of the message.

**Table 15 - Acknowledgement types**

<b>Name (Enumeration)</b>	<b>Mnemonic (Instance)</b>	<b>Description</b>
ApplicationAcknowledgement Accept	AA	Receiving application successfully processed the message.
ApplicationAcknowledgement Error	AE	Receiving application found an error in processing the message. Sending error response with additional error detail information.

ApplicationAcknowledgement Reject	AR	Receiving application failed to process message for reason unrelated to content or format. Original message sender must decide on whether to automatically send message again.
AcceptAcknowledgement CommitAccept	CA	Receiving message handling service accepts responsibility for passing message onto receiving application.
AcceptAcknowledgement CommitError	CE	Receiving message handling service cannot accept message for any other reason (e.g. message sequence number, etc.).
AcceptAcknowledgmentCommitReject	CR	Receiving message handling service rejects message if interaction identifier, version or processing mode is incompatible with known receiving application role information.

### TargetMessage (Acknowledged message)

The 'targetMessage' property indicates the message that is being acknowledged. This can be used by the receiver to correlate a request with the acknowledgement.

### AcknowledgementDetail (Processing details)

This property contains a list of details that dictates how the current acknowledgement code has been chosen. This includes conformance validation errors, processing errors, and other errors of a non-clinical relation.

### MessageWaitingNumber (Number of waiting queued messages)

This property indicates the number of messages that the receiver has waiting on its queue on this particular sender.

### MessageWaitingPriorityCode (Priority of waiting messages)

This property indicates the highest priority of message waiting on the queue. This may dictate how the receiver will schedule a queue-poll.

```
MCCI_IN000002CA response = new MCCI_IN000002CA();
response.AcceptAckCode = AcknowledgementCondition.Never;
response.CreationTime = DateTime.Now;
response.Id = new II(Guid.NewGuid());
response.InteractionId = new II("1.2.2.3", "MCCI_IN000002CA");
response.ProcessingCode = ProcessingID.Production;
response.ResponseModeCode = ResponseMode.Immediate;

// Assign recipient
response.Receiver = new MARC.Everest.Core.MR2009.MCCI_MT002200CA.Receiver(
```

```

        new MARC.Everest.Core.MR2009.MCCI_MT002200CA.Device2(
            request.Sender.Device.Id
        )
    );
    response.Sender = new MARC.Everest.Core.MR2009.MCCI_MT002200CA.Sender(
        new MARC.Everest.Core.MR2009.MCCI_MT002200CA.Device1(
            new II(MY_OID, MY_EXT)
        )
    );

    // Create acknowledgement
    response.Acknowledgement = new
    MARC.Everest.Core.MR2009.MCCI_MT002200CA.Acknowledgement(
        AcknowledgementType.AcceptAcknowledgementCommitAccept,
        // Assign the target message
        new MARC.Everest.Core.MR2009.MCCI_MT002200CA.TargetMessage(
            request.Id
        )
    );

```

#### Example 21 - Creating a generic response

## Control Act Wrapper

A control act wrapper (or ControlActEvent) structure is used to identify an action that caused the message to be sent. It is vital to the understanding of what action a message represents.

Control act wrappers implement the interfaces IControlActEvent, IImplementsStatusCode and IIdentifiable. Control act wrappers will implement most (if not all) of the properties in Table 16

**Table 16 - ControlActEvent properties**

Property	Conformance	Description
II : Id	Mandatory	A unique identifier for the control act wrapper assigned by the system in which the event occurred.
CV : Code	Mandatory	Identifies the trigger event that occurred.
CS : StatusCode	Mandatory	Describes the status of the control act wrapper. In many cases this value is fixed to "completed".
IVL<TS> : EffectiveTime	Required	Indicates the time the event should begin and occasionally when it should end.
CV : ReasonCode	Required	Identifies why this specific message interaction is being executed
CE : LanguageCode	Required	Identifies the language used within the message.
?? : RecordTarget*	Mandatory	Identifies the associated patient record for which this message applies.

?? : ResponsibleParty*	Optional	Allows for the trail of responsibility for the action.
?? : Author*	Mandatory	Indicates the person responsible for the event that caused this message.
?? : DataEnterer*	Optional	Indicates the person responsible for entering the query into the system.
?? : Location*	Optional	Indicates the service delivery location where the event occurred.
?? : DataEntryLocation*	Optional	Indicates the location where the record of this event was entered.
?? : PertinentInformation*	Optional	Identifies the authentication data that references information about the people and/or places responsible for the event.
?? : SubjectOf1*	Required	This is the list of clinical and business issues that have been detected and/or managed as part of this event.
?? : SubjectOf2*	Required	Information pertaining to a patient's agreement/acceptance to have their clinical information electronically stored or retrieved.
?? : ComponentOf*	Required	Allows linking of this event to a care composition.
<b>Put Interactions</b>		
T : Subject	Mandatory	Communicates the payload of the put operation.
<b>List Interactions</b>		
T : QueryByParameter	Mandatory	Identifies the query parameters for the list operation.
<b>List Response Interactions</b>		
T[] : Subject	Mandatory	Identifies the results of the query operation.
U : QueryByParameter	Mandatory	The query parameters used to retrieve the data.
?? : QueryAck	Mandatory	The acknowledgement to the query operation.
<b>Response Interactions</b>		
?? : SubjectOf	Required	Replaces the SubjectOf1 property. Identifies the clinical and/or business issues related to the event.

\* - This property may not be included in some control act events, or may be marked under a different property name. "Mandatory" is understood as "Mandatory when present" for these items.



All interactions that have a control act event, payload, or query parameters allow the developer to create each component using a static utility method in the interaction class. This is illustrated in Example 22.

```
COMT_IN100000CA request = new COMT_IN100000CA();

// Object is populated here
request.controlActEvent = COMT_IN100000CA.CreateControlActEvent();
request.controlActEvent.Id = new II(MY_OID, Guid.NewGuid().ToString());

// Continue message population
```

#### Example 22 - Creating control act event

The interaction class contains other “short-cut” methods for creating the payload, and query parameters (if necessary). It is also possible to set the trigger event using the interaction class as illustrated in Example 23.

```
request.controlActEvent = COMT_IN100000CA.CreateControlActEvent();
request.controlActEvent.Id = new II("1.1.1.1", Guid.NewGuid().ToString());
request.controlActEvent.Code = COMT_IN100000CA.GetTriggerEvent();
```

#### Example 23 - Setting trigger event in control act event

## Payload

The payload of a message depends on the message being sent, and what data is required for that clinical operation. It is nearly impossible to cover all payloads that could be sent in a message.

The payload point is often a child-node of the ControlActEvent.Subject property. Finding where to assign the payload can be quite challenging. Here are a few guidelines:

- Look for a property in the control act subject that is not PascalCased but is camelCased.
- Find the documentation for the controlActEvent class and note the generic type argument. The payload point’s property name will match the generic parameter name.

Once the payload point is found, it can be assigned using the static utility methods in the interaction class (or through direct assignment). Example 24 illustrates this.

```
// Continue message population
request.controlActEvent.Subject.act = REPC_IN000076CA.CreateDocument(
    CareSummaryDocumentType.DISCARGEASSESSMENTNOTE,
    "This is my title",
    new MARC.Everest.Core.MR2009.REPC_MT220001CA.Author(
        ParticipationSignature.Signed
    ),
    new MARC.Everest.Core.MR2009.REPC_MT220001CA.Component2(
        "This is the text of my document",
        new MARC.Everest.Core.MR2009.REPC_MT220001CA.Component4(
            MARC.Everest.Core.MR2009.REPC_MT220001CA.DocumentContent.
            CreatePatientCareProvisionEvent(
                ActCareEventType.Emergency,
                new IVL<TS>(DateTime.Now)
            )
        )
    )
);
```

```

    )
  );

```

#### Example 24 - Setting payload in control act event

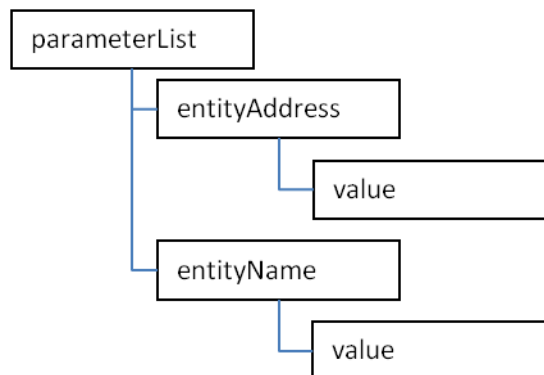
## Query Parameters

Query parameters are used to filter data based on a predetermined number of criteria. The query parameters class have the common attributes outlined in Table 17.

**Table 17 - Query parameter properties**

Property	Conformance	Description
II: QueryId	Mandatory	Uniquely identifies the query in the receiver system. This is used for query continuation
INT: InitialQuantity	Required	Identifies the number of results that the sender wishes to receive in the response to the query.
CS: InitialQuantityCode	Required	Identifies the scope of the InitialQuantity number (ie: number of what to be returned)
CS: ResponseModalityCode	Required	Indicates that the query should be executed on an expedited flow.
?? : parameterList	Mandatory	The list of filter parameters.

The parameter list contains a series of properties that encapsulate the filter value using the "Value" property. This is illustrated in Figure 4.



**Figure 4 - Parameter list filters**

The recommended method of setting these properties is to use the shortcut constructors. An example of this method is illustrated in Example 25.

```

REPC_IN000086CA r = new REPC_IN000086CA();
// Populating a list of filters
r.controlActEvent.QueryByParameter.parameterList.CareCompositionId.Add(
    new MARC.Everest.Core.MR2009.REPC_MT220004CA.CareCompositionId(
        new II("1.1.1.1.1.1", "1234")
    ));

```

```
// Populating a single filter
r.controlActEvent.QueryByParameter.parameterList.RequestingProviderId =
    new MARC.Everest.Core.MR2009.REPC_MT220004CA.RequestingProviderId(
        new II("2.1.4.4.3.34.4", "12345")
    );
```

### Example 25 - Populating query parameter filters

## Abstract Classes

There are many instances of abstract class references within the RMIM structures. These will appear as either a direct reference to an abstract class (example: AuthorPerson), or as a reference to System.Object.

A reference to an abstract class occurs when the RMIM property represents a choice of elements that have a common parent class. For example: A responsible party may be a choice of AssignedEntity (Health Care Provider) or AssignedEntity (Organizational). Because both of these have the same parent class, the property is assigned this class.

On the other hand, if the RMIM property represents a choice where all elements don't have a common parent class, the reference is made to System.Object.

Such properties appear in the documentation as having multiple PropertyAttribute tags (see Figure 5). Using the documentation is recommended if the property has a type of System.Object.

```
C#
[EditorAttribute(typeof(NewInstanceTypeEditor), typeof(UITypeEditor))]
[PropertyAttribute(Name = "personalRelationship", PropertyType = PropertyAttribute.AttributeAttributeType.NonStructural,
    Conformance = PropertyAttribute.AttributeConformanceType.Optional, Type = typeof(PersonalRelationship))]
[PropertyAttribute(Name = "assignedEntity2", PropertyType = PropertyAttribute.AttributeAttributeType.NonStructural,
    Conformance = PropertyAttribute.AttributeConformanceType.Optional, Type = typeof(AssignedEntity))]
[PropertyAttribute(Name = "assignedEntity1", PropertyType = PropertyAttribute.AttributeAttributeType.NonStructural,
    Conformance = PropertyAttribute.AttributeConformanceType.Optional, Type = typeof(AssignedEntity))]
[TypeConverterAttribute(typeof(ExpandableObjectConverter))]
public Object ActingPerson { get; set; }
```

Figure 5 -Documentation of property with choice of values

There are three methods of populating a choice element;

1. The preferred method of setting the choice is to use the SetXXX function on the instance. For example, if one were to populate the Patient property of a record target they could simply just pass in the parameters they know and the correct choice will be selected (as illustrated in Example 26):

```
request.controlActEvent.RecordTarget.SetPatient1(
    new SET<II>(new II("1.1.1.1.1.1", "123"), II.Comparator)
);
```

### Example 26 - Using the set XXX function

2. If the concrete class is known, it can be directly assigned to the property (described in the "Type" parameter of the PropertyAttribute)

```
request.controlActEvent.Author.AuthorPerson =
    new MARC.Everest.Core.MR2009.COCT_MT090108CA.AssignedEntity(
        new SET<II>(new II("1.1.1.1", "123"), II.Comparator),
```

```
HealthCareProviderRoleType.Epidemiologist,  
new MARC.Everest.Core.MR2009.COCT_MT090108CA.Person()  
);
```

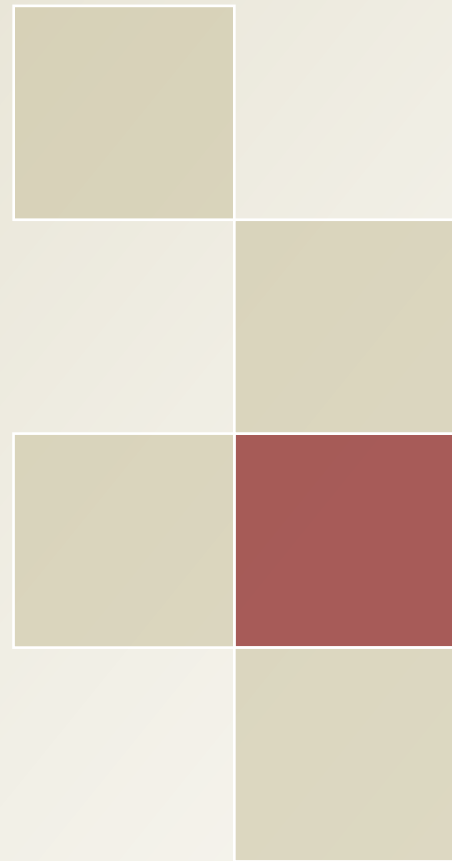
#### **Example 27 - Direct assignment to an abstract property**

3. The final method of assigning an abstract property is to use the static utility function. This involves using the static CreateXXX function on the abstract parent class to create an instance of the choice item.

```
request.controlActEvent.Author.AuthorPerson =  
MARC.Everest.Core.MR2009.PORX_MT060160CA.ChangedBy.CreateAssignedEntity(  
new SET<II>(new II("1.1.1.1", ""), II.Comparator),  
HealthCareProviderRoleType.Epidemiologist,  
new MARC.Everest.Core.MR2009.COCT_MT090108CA.Person()  
);
```

#### **Example 28 - Using static utility function to assign abstract property**

## Chapter 3. Formatting Instances



- *Using formatters to create XML instances*
- *Using Graphing Aides to change data types representations*

This chapter will teach the reader the role of a formatter, how it is used, and its relation to RMIM and Data Type classes. Building on the knowledge of the previous chapter, the developer will be able to:

- Use a formatter to create an XML representation of an RMIM structure in memory, and
- Use a graphing aide to force data types to render as a different revision.

## Introduction

The MARC-HI HL7v3 API is designed to abstract the particular 'wire level format' from the developer. This is done to provide flexibility of formatting at runtime (ie: the developer doesn't need to change code to support a new ITS).

A formatter is responsible for transforming RMIM and data type objects in memory to/from 'wire format'. Formatters may be combined in a variety of ways to attain any desired output rendering.

## Formatter Types

The API defines two types of formatters. Each provides the same functionality but make certain assumptions about the output of the formatter. There are two different interfaces that are implemented by each type of formatter:

- **IStructureFormatter** – This type of formatter makes no assumption about the format of the output. The IStructureFormatter provides graphing and parsing to/from streams.
- **IXmlStructureFormatter** – This type of formatter assumes that the output format will be rendered in some type of XML format. This is used by the WCF connector as the assumption that web services must communicate using XML.
- **ICodeDomStructureFormatter** - This type of formatter uses CodeDOM to create an optimized formatter at runtime. It is possible to get the **GeneratedAssembly** property to save the generated assembly (for use in non JIT environments and to enhance runtime performance)



## Generic Formatters

The IStructureFormatter interface provides a series of methods that can be used to graph/parse to/from generic streams.

### Details

This array contains a list of IResultDetail structures that contain data about the result of the formatting. Any conformance, validation, exception, code lookup errors are included in this array.

### GraphAides

The graph aides collection is used to combine formatters. A formatter is expected to scan its aides before graphing a particular object to/from the wire format.

### GraphObject

The GraphObject method is used to actually graph an RMIM object to the wire level format. The result of this is a 'ResultCode'. Result codes are shown in Table 18.

### Table 18 - Graphing result codes

Code	Description
AcceptedNonConformant	The instance is accepted; however it contains errors that make it non-conformant.
Accepted	The instance is accepted.
Rejected	The instance is rejected because it was not conformant.
Error	An error occurred that prevented the instance from being parsed or graphed.
NotAvailable	The underlying stream is not available so graphing or parsing could not occur.

### Host

When a formatter is a subordinate to another formatter (example: it is a graph aide), this property is populated to the parent.

For example: if formatter A has a graph aide B, then B.Host is set to A.

### HandleStructure

This string is used to dictate what structures a particular formatter can handle. This value will be set to an asterisk ("\*") if the formatter is generic and can handle any structure.

### ParseObject

This method is used to parse an instance from a stream into an RMIM class.

## XML Formatters

The IXmlStructureFormatter interface extends the IStructureFormatter's GraphObject and ParseObject methods to operate on XmlWriter and XmlReader streams.

In addition to this extension, it is possible to pass a type hint to the parse method. This can be used when xml context is lost by subordinate formatters.

## Using Formatters

Using a formatter involves three steps,

- Create a stream,
- Instantiate the formatter, set graph aides and any parameters, and
- Call GraphObject() on the formatter.

Example 29 illustrates the use of a formatter with graph aides to create an XML ITS 1.0 instance with data types R1.

```
// Create the formatter
MARC.Everest.Formatters.XML.ITS1.Formatter its1Formatter = new Formatter();
// Add the R1 graph aide
its1Formatter.GraphAides.Add(typeof(MARC.Everest.Formatters.XML.Datatypes.R1.F
ormatter));
// Graph the object to STDOUT
its1Formatter.GraphObject(Console.OpenStandardOutput(), instance);
```

## Example 29 - Using a formatter

Each formatter may expose different properties that will alter the way that it functions.

### Result Details

When formatting an instance using any of the formatters provided by the Everest Framework, users frequently experience the "null output" problem. This error occurs when the formatter cannot reliably create output due to errors caused during validation.

To retrieve these errors, the user may use the Details property of the formatter as listed in Example 30 - Using formatter result codesExample 30.




```
// Create the XML ITS formatter
Formatter f = new Formatter();
// Add the R1 data types
f.GraphAides.Add(typeof(MARC.Everest.Formatters.XML.Datatypes.R1.Formatter));
// Graph the instance
ResultCode code = f.GraphObject(Console.OpenStandardOutput(), instance);

// Get the result
if (code != ResultCode.Accepted)
    foreach (var dtl in f.Details)
        Console.WriteLine(dtl.Message);
```

## Example 30 - Using formatter result codes

There are several types of result details that can be fired by the graph or parse methods of the formatter, these are outlined in Table 19.

**Table 19 - Result Detail Types**

Type	Description
ResultDetail	General result detail issue
InsufficientRepetitionsResultDetail	Triggered when a collection does not contain the minimum number of repetitions specified in the structure
MandatoryElementMissingResultDetail	Triggered when a mandatory element has a value of <b>null</b> or a <b>nullFlavor</b> is specified.
RequiredElementMissingResultDetail	Triggered when a required element with minimum multiplicity of 1 (ie: populated) is assigned a value of null
VocabularyIssueResultDetail 	Triggered when an issue with vocabulary is detected. Most common cause of this is a CS value where the supplied code is not within the bound code set and a codeSystem hasn't been supplied
NotImplementedElementResultDetail 	Triggered on a parse whereby an element on an incoming message was not understood by the formatter and could not be placed within the specified RMIM structure
FixedValueMisMatchedResultDetail 	Triggered when a value read from a formatter does not match the fixed value in the RMIM class. Since the RMIM class treats



	fixed values as "read only" it is not possible for the formatter to change the RMIM fixed value.
--	--

## XML ITS 1.0 Formatter

The XML ITS 1.0 formatter applies the rules of XML ITS 1.0 against the RMIM object to create an XML instance of that RMIM structure. The XML ITS 1.0 formatter accepts two different parameters that alter the way that instances are generated.

### Settings

The XML ITS 1.0 formatter lets developers modify default behaviors during runtime. There may come a time, for example, when the developer wishes not to include flavor information or may not wish for supplier domains to be automatically included. These settings are enabled or disabled by using the **Settings** property on the XML ITS 1.0 formatter. Example 31 shows an example of enabling only flavor imposing and supplier domain imposing.

```
Formatter fmtr = new Formatter();
fmtr.Settings = SettingsType.AllowFlavorImposing |
    SettingsType.AllowSupplierDomainImposing;
```

#### Example 31 - Using the ITS 1 settings property

The available settings options are listed in Table 20.

**Table 20 - XML ITS 1.0 Settings**

Enumeration Value	Description
AllowFlavorImposing	When set, allows RMIM structures to impose flavors on the generated instance.
AllowUpdateModeImposing	When set, allows RMIM structures to impose default update modes on the generated instance.
AllowSupplierDomainImposing	When set, allows RMIM structures to impose a supplier domain (codeSystem) if one is not specified.

### Validate Conformance

When the validate conformance property is set to false, no validation is performed. While this increases performance, it does not guarantee that generated instances will valid against the conformance statements in the RMIM object.

### Create Required Elements

This property, when set to true, dictates that all populated and required elements will be automatically created when the graph method is called.

```
// Create instance
REPC_IN000076CA request = new REPC_IN000076CA(
    // The Message Identifier
    new II(System.Guid.NewGuid().ToString()),
    // The Creation Time
    DateTime.Now,
    // The Response Mode
    ResponseMode.Immediate,
    // The Message Identifier
```

```

new II("", "REPC_IN000076CA"),
// The Profile Id
new LIST<II>{
    new II[] { new II("R2.04.01") }
},
// The Processing Mode
ProcessingID.Production,
// The Acknowledgement Condition
AcknowledgementCondition.Always,
// The intended recipient
new MARC.Everest.Core.MR2009.MCCI_MT002100CA.Receiver(
    new MARC.Everest.Core.MR2009.MCCI_MT002100CA.Device2(
        new II("1.2.2.2.", "1") // id
    )
),
// The sending device node
new MARC.Everest.Core.MR2009.MCCI_MT002100CA.Sender(
    new MARC.Everest.Core.MR2009.MCCI_MT002100CA.Device1(
        new II("1.2.2.2.2.", "123") // id
    )
)
); // Instance

```

### Example 32 - Creating the reference

#### CreateRequiredElements false

```

<?xml version="1.0" encoding="utf-8"?>
<REPC_IN000076CA ITSVersion="XML_1.0" xmlns="urn:h17-org:v3">
  <id specializationType="TOKEN"
    root="27ab9d9b-9764-4511-b902-0f1969aaaced" />
  <creationTime specializationType="FULLDATETIME"
    value="20090812100440" />
  <responseModeCode code="I" />
  <versionCode code="V3-2008N" />
  <interactionId specializationType="PUBLIC" root=""
    extension="REPC_IN000076CA"
    displayable="true" />
  <profileId root="R2.04.01" />
  <processingCode code="P" />
  <processingModeCode code="T" />
  <acceptAckCode code="AL" />
  <receiver typeCode="RCV">
    <device classCode="DEV" determinerCode="INSTANCE">
      <id specializationType="BUS" root="1.2.2.2."
        extension="1" use="BUS" />
    </device>
  </receiver>
  <sender typeCode="SND">
    <device classCode="DEV" determinerCode="INSTANCE">
      <id specializationType="BUS" root="1.2.2.2.2."
        extension="123" use="BUS" />
    </device>
  </sender>
</REPC_IN000076CA>

```

#### CreateRequiredElements true

```

<?xml version="1.0" encoding="utf-8"?>
<REPC_IN000076CA ITSVersion="XML_1.0" xmlns="urn:h17-org:v3">
  <id specializationType="TOKEN"
root="b5a8bd0b-dbc5-492d-8bc5-dedda5574e6a" />
  <creationTime specializationType="FULLDATETIME"
    value="20090812102106" />
  <responseModeCode code="I" />
  <versionCode code="V3-2008N" />
  <interactionId specializationType="PUBLIC" root=""
    extension="MCCI_IN100001CA" displayable="true" />
  <profileId root="R2.04.01" />
  <processingCode code="P" />
  <processingModeCode code="T" />
  <acceptAckCode code="AL" />
  <receiver typeCode="RCV">
    <device classCode="DEV" determinerCode="INSTANCE">
      <id specializationType="BUS" root="1.2.2.2." extension="1"
        use="BUS" />
    </device>
  </receiver>
  <sender typeCode="SND">
    <device classCode="DEV" determinerCode="INSTANCE">
      <id specializationType="BUS" root="1.2.2.2.2" extension="123"
use="BUS" />
      <name nullFlavor="NI" />
      <desc nullFlavor="NI" />
      <existenceTime nullFlavor="NI" />
      <manufacturerModelName nullFlavor="NI" />
      <softwareName nullFlavor="NI" />
    </device>
  </sender>
  <controlActEvent nullFlavor="NI" />
</REPC_IN000076CA>

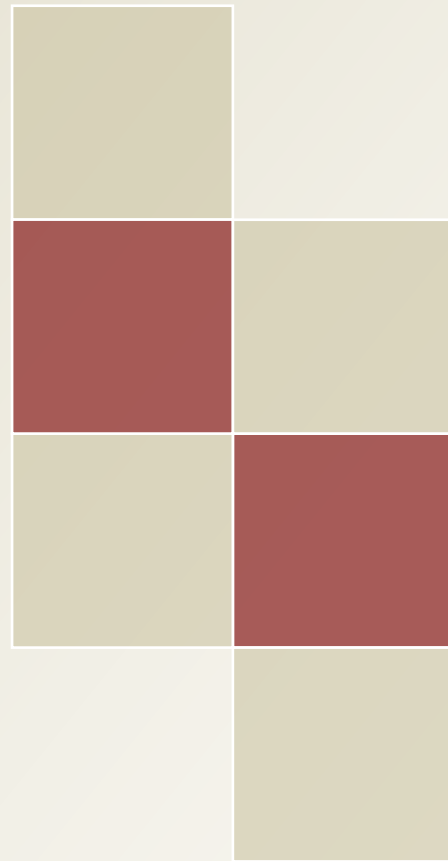
```

## Graph Aides

As stated before, a graph aide is a subordinate formatter whose primary responsibility is to help the primary renderer format message types to the wire level format.

Of the three formatters bundled with the MARC-HI, two are intended to be used solely as aides.

## Chapter 4. Connecting to the World



- *The four patterns of MARC-HI Everest Connectors*
- *Using connectors to publish and subscribe to data*
- *Servicing and soliciting requests using the Windows Communication Foundations framework.*

This chapter covers the sending and retrieving of instances from different sources. The chapter builds upon the concepts learned in the previous chapters to communicate with remote systems. By the end of this chapter, the developer will know:

- The four different patterns for communicating with remote endpoints
- How to use connectors to publish data to a connector
- How to use connectors to subscribe to data from a connector
- How to service requests and solicit requests using the Windows Communication Foundation

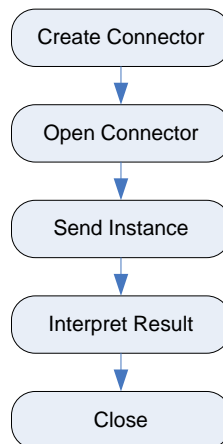
## Introduction

The MARC-HI API provides a means of constructing and formatting instances to a particular ITS. It also provides a series of connectors that can be used to establish a communications channel between the local application and the remote application.

This chapter will investigate the different ways that a connector can be used to transmit and process messages to/from endpoints.

### Sending Data using Connectors

The development pattern for sending data through a connector is common across all types of sending connectors. The following pattern is used



#### Open Connector

The constructor of a connector will always expect a connection string in the format: **"key=value; key=value"**. This connection string is used to dictate the location, authentication and other parameters of the underlying channel. The open method actually validates the connection string data.

#### Send Instance

The "send" method of a connector accepts the IGraphable parameter that represents the data being sent over the channel.

The send instance also returns an ISendResult structure which outlines the details and overall result of the send. These details and the result code itself match those discussed in the "Formatters" section.

## Formatted Connectors

A connector may or may not be formatted, depending on the transport being used. A formatted connector implements the `IFormattedConnector` interface, and allows the developer to assign a formatter instance to the connector.

This formatter instance is used to render any data sent on the connector into the specified format.

### Example: Assigning a formatter to a connector

```
Formatter its1Formatter = new Formatter();
its1Formatter.GraphAides.Add(
    typeof(
        MARC.Everest.Formatters.XML.Datatypes.R1.Formatter
    )
);

MsmqPublishConnector connector = new MsmqPublishConnector();
// Assign the formatter
connector.Formatter = its1Formatter;
```

## Connector Patterns

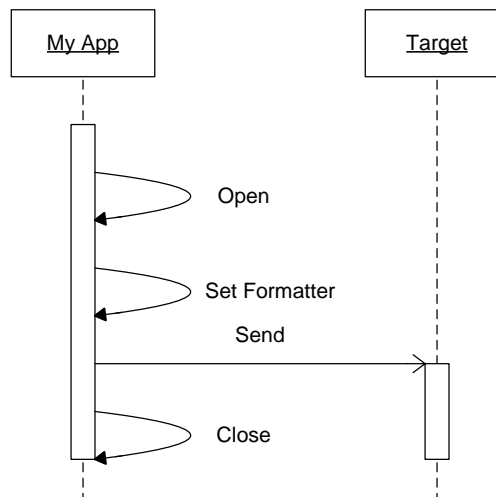
There are four different, unique connection patterns that can be serviced by the MARC-HI API connectors. They are as follows

### Sending

Sending connectors implement the `ISendingConnector` interface. They provide a means of transmitting data in a “fire and forget” manner. The pattern is described as follows:



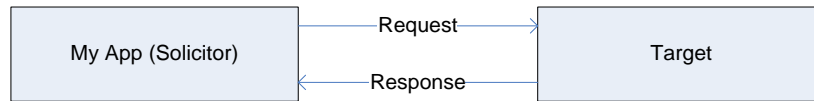
Sending connectors are usually distinguished by the fact they have the word “Publish” in the class name (ie: `FilePublishConnector`, or `MsmqPublishConnector`). The sequence of publishing an instance on a sending connector is as follows:



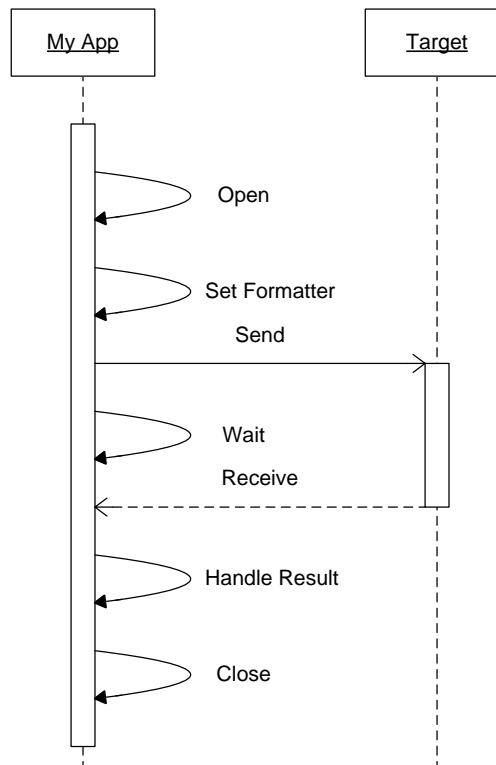
## Send Receive

The connectors that implement the send/receive pattern implement the interface `ISendingReceivingConnector`. They provide a means for transmitting data and correlating a response.

Although the method may be executed asynchronously, the underlying connection to the service is expected to be synchronous (example: HTTP)



The sequence for a request/response pattern is as follows:

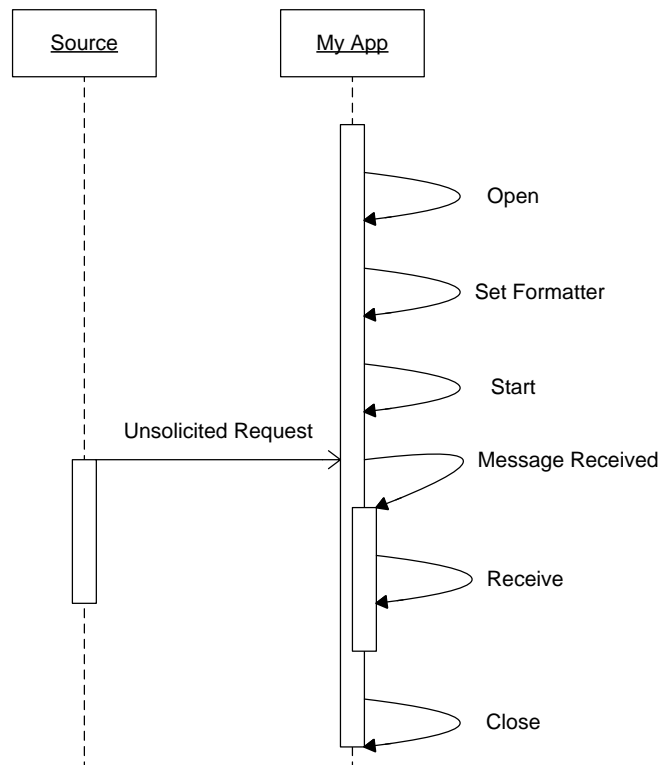


## Listen Wait

The listen/wait connector represents a connector that actively listens for messages through some transport mechanism. There is no expectation that the connector needs to send back a response. Listen/wait connectors implement the `IListenWaitConnector` interface.



The sequence for the listen/wait pattern is as follows:



Every listen/wait connector provides two methods of receiving data. These are blocking and non-blocking (ie: asynchronous).

When receiving in a blocking mode, the application calls the "Receive" method of the connector. All other program functions will be stopped until a message is received.

```
// Create an instance of the connector
FileListenConnector connector = new FileListenConnector();
connector.ConnectionString = @"Directory=C:\temp";
connector.Formatter = its1Formatter;
connector.Open();
connector.Start();
// Wait until a file is available
IReceiveResult result = connector.Receive();
// Process the file
Console.WriteLine(result.Structure.ToString());
// Close connector
connector.Close();
```



### Example 33 - Listen connector receiving in blocking mode

Receiving messages in a non-blocking fashion is a little more complex, however it allows the program to continue a task until a message is received by the connector.

```
// The MessageAvailable delegate will be called when the connector
// has a message available for processing
void connector_MessageAvailable(object Sender,
UnsolicitedDataEventArgs args)
{
    // Receive the message from the sender
    IReceiveResult result = (Sender as FileListenConnector).Receive();
    Console.WriteLine(result.Structure.ToString());
}

void Main()
{
    // Create an instance of the connector
    FileListenConnector connector = new FileListenConnector();
    connector.Formatter = its1Formatter;
    connector.ConnectionString = @"Directory=C:\temp;";
    connector.Open();

    // Assign message available event
    connector.MessageAvailable += new
EventHandler<UnsolicitedDataEventArgs>(
    connector_MessageAvailable
);

    connector.Start(); // Start watching the directory

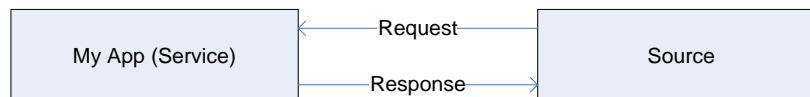
    // Exit condition
    Console.WriteLine("Drop a message in C:\\temp!");
    Console.WriteLine("Press any key to quit...");
    Console.ReadKey();

    // Close connector
    connector.Close();
}
```

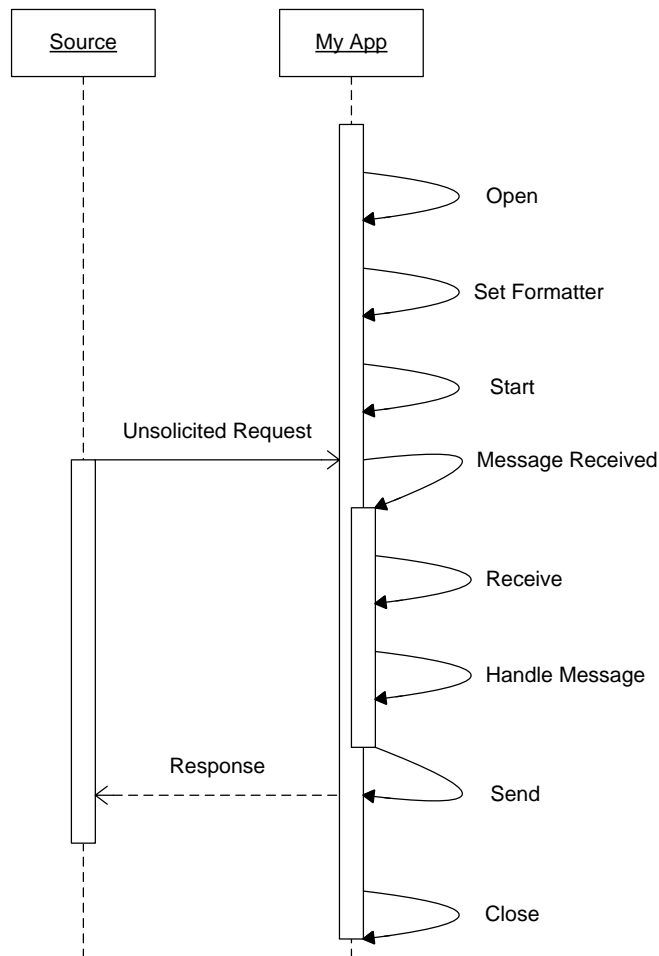
### Example 34 - Listen connector receiving in non-blocking mode

#### Listen Wait Respond

The listen/wait/respond connector represents a service that actively listens for messages, and operates on a transport that expects a response. These connectors implement the *IListenWaitRespondConnector* interface.



The sequence for the listen/wait/respond pattern is as follows



## Connecting to Files

The first example of a connector that will be covered will be the file connector. This connector allows an application to publish and subscribe to directories on the user's computer.

## Publishing to a Directory

Publishing to a directory requires the use of the FilePublishConnector class in the assembly "MARC.Everest.Connectors.File".

**Table 21 - File Publish connector connection string**

Parameter	Value(s)	Description
Directory	The directory in which publish	Indicates where files should be published
File	The name of the file to publish	Indicates the fixed name of the file to publish. Can't be combined with "naming"

Naming	guid	Using guides		Indicates how files should be named if a fixed name is not present. Useful if publishing multiple instances over the lifetime of one connector.
	id	Using the ID node		

```
// Create the formatter instance for the FilePublishConnector
// implements IFormattedConnector
Formatter its1Formatter = new Formatter();
its1Formatter.GraphAides.Add(
    typeof(
        MARC.Everest.Formatters.XML.Datatypes.R1.Formatter
    )
);

// Create an instance of the connector
FilePublishConnector connector = new
    FilePublishConnector(@"Directory=C:\temp;File=output.xml");
connector.Formatter = its1Formatter;
connector.Open();
ISendResult result = connector.Send(instance);

// Close connector
connector.Close();
```

#### Example 35 - Publishing a file (output.xml) to the directory C:\temp

## Subscribing to a Directory

Directory subscription is used when an application wishes to “listen” to a directory on the user’s machine for new instances.

**Table 22 - File Listen connector connection string**

Parameter	Value(s)	Description
Directory	Name of the directory to watch	Indicates which directory the connector should listen to.
Pattern	Any wildcard	Indicates the pattern that files must match before they are processed
ProcessExisting	Boolean value (true/false)	If true, all existing files in the directory will be processed when start() is called. Default is false.
KeepFiles	Boolean value (true/false)	If true, files that are processed will not be removed from the file system. Default is false.

```
void connector_MessageAvailable(object sender,
UnsolicitedDataEventArgs args)
{
    // Start the receive operation (starts on another thread)
    IAsyncResult iar = (sender as
```

```

        FileListenConnector).BeginReceive(null, null);
        iar.AsyncWaitHandle.WaitOne(); // Wait until complete
        IReceiveResult result = (sender as
            FileListenConnector).EndReceive(iar);

        // Write the result
        Console.WriteLine(result.Structure.ToString());
    }

    void Main()
    {
        Assembly.Load(new AssemblyName("MARC.Everest.Core.MR2009"));

        Formatter its1Formatter = new Formatter();
        its1Formatter.GraphAides.Add(
            typeof(
                MARC.Everest.Formatters.XML.Datatypes.R1.Formatter
            )
        );

        // Create an instance of the connector
        FileListenConnector connector = new
            FileListenConnector(@"Directory=C:\temp;KeepFiles=true");
        connector.Formatter = its1Formatter;
        connector.Open();

        // Assign message available event
        connector.MessageAvailable += new
            EventHandler<UnsolicitedDataEventArgs>(
                connector_MessageAvailable);

        connector.Start(); // Start watching the directory

        // Exit condition
        Console.Write("Press any key to quit...");
        Console.ReadKey();

        // Close connector
        connector.Close();
    }
}

```

**Example 36 - Subscribing to the directory C:\temp**

## Windows Communications Foundation

The Windows Communication Foundation (WCF) connector provides a gateway to use the API's serialization functionality with the WCF framework. The WCF connector may be used with any of the following bi-directional WCF bindings:

- **basicHttpBinding** – Used for SOAP 1.0 connections.
- **wsHttpBinding** – Used for SOAP 1.1/1.2 connections. Provides message and transport level encryption, reliable messaging, etc...
- **netTcpBinding** – A binary transport for use over TCP sockets.
- **namedPipeBinding** – A binary transport that may be used to establish IPC (or named pipes).

The WCF connector may be used in one of three modes:

- **Client** – In this mode, the WcfClientConnector is used to communicate with a remote endpoint.
- **Standalone Service** – In this mode, the WcfServerConnector is used to host an in-process service (example: A windows service)
- **IIS Service** – In this mode, IIS is used to host the service (example: A web service)

## Consuming HL7v3 Services

Consuming an HL7v3 service requires some knowledge of the WCF framework. The first stage of connecting to a service using the connector requires editing the web of app config files.

### Step 1: Add configuration section handler

The first configuration section will define a handler for the WCF connector configuration section. This is required whenever using the WCF connector.

```
<configSections>
  <section
    type="MARC.Everest.Connectors.WCF.Configuration.ConfigurationSection,
MARC.Everest.Connectors.WCF, Version=1.0.0.0"
    name="marc.everest.connectors.wcf"/>
</configSections>
```

### Step 2: Add an endpoint configuration

WCF requires a configuration section for the endpoint configuration. In this sample, we'll be using wsHttpBinding to connect to a local service listening on port 8000.

```
<client>
  <endpoint binding="wsHttpBinding" address="http://localhost:8000"
name="ApplicationClient"
contract="MARC.Everest.Connectors.WCF.Core.IConnectorContract"
bindingConfiguration="sampleBindingConfig">
  </endpoint>
</client>
```

### Step 3: Add a binding configuration

The binding configuration is used to configure specific parameters related to a particular binding. The sample uses the "sampleBindingConfig" binding configuration. It is important that any binding's maxReceivedMessageSize be greater than 12 kb.

In this example, the binding configuration will use message encryption and WS-RM.

```
<bindings>
  <wsHttpBinding>
    <binding name="sampleBindingConfig"
maxReceivedMessageSize="12000000">
      <reliableSession enabled="true"/>
      <security mode="Message"/>
    </binding>
  </wsHttpBinding>
</bindings>
```

```
</wsHttpBinding>
</bindings>
```

#### Step 4: Specify WCF connector soap actions and formatter

Because the WCF connector uses SOAP, each structure type must supply its own soap action. The "MARC.Everest.Connectors.wcf" section is used to express this information.

```
<marc.everest.connectors.wcf
  formatter="MARC.Everest.Formatters.XML.ITS1.Formatter,
  MARC.Everest.Formatters.XML.ITS1"
  aide="MARC.Everest.Formatters.XML.Datatypes.R1.Formatter,
  MARC.Everest.Formatters.XML.Datatypes.R1">
<action
  type="MARC.Everest.Core.Interactions.MCCI_IN100001CA"
  action="urn:h17-org:v3:MCCI_IN100001CA_I"/>
</marc.everest.connectors.wcf>
```

This configuration section accepts a series of "Action" elements; each action element describes the desired soap action to use for each of the message types it supports.

The inclusion of the formatter attribute is required on all MARC.Everest.Connectors.wcf sections (the aide is optional). This is used to describe the default formatter. The developer may override this formatter in their code by setting the "Formatter" property.

Once the client configuration is complete, the developer may now start to work on the code.

#### Step 5: Code the application

Coding the application requires the use of the WcfClientConnector. When the connector is opened, it must be given the name of the endpoint with which the connector will be communicating

The connection string parameters for the WcfClientConnector are listed in Table 23.

**Table 23 - WCF Client connector connection string**

Parameter	Value(s)	Description
EndpointName	Name of an endpoint	The name of the endpoint in the application configuration file that the connector should communicate with.

```
// Create the connector
WcfClientConnector conn = new WcfClientConnector();
conn.ConnectionString = "EndpointName=ApplicationClient";

// Establish the channel
conn.Open();

// Send the instance
IAsyncResult iaSendResult = conn.BeginSend(instance, null, null);
Console.WriteLine("Requesting...");

iaSendResult.AsyncWaitHandle.WaitOne();
```

```

ISendResult sndResult = conn.EndSend(iaSendResult);

// Check that the result has come back
if (
    sndResult.Code != ResultCode.Accepted &&
    sndResult.Code != ResultCode.AcceptedNonConformant)
    Console.WriteLine("Request Failed!");
else
{
    // Receive result
    IAsyncResult iaReceiveResult = conn.BeginReceive(
        sndResult, null, null);

    Console.WriteLine("Parsing Result...");

    // Wait for parser to complete
    iaReceiveResult.AsyncWaitHandle.WaitOne();

    Console.WriteLine("Ok.");

    IReceiveResult rcvResult = conn.EndReceive(iaReceiveResult);

    // Was the received result non-conformant
    if (
        rcvResult.Code != ResultCode.Accepted &&
        rcvResult.Code != ResultCode.AcceptedNonConformant
    )
        Console.WriteLine("Response non-conformant!");
    else
        Console.WriteLine(rcvResult.Structure.ToString());
}

// Close the channel
conn.Close();

```

#### Example 37 - Sending an instance using WCF

## Hosting HL7v3 Services

Hosting an HL7v3 service using WCF can be done in one of two ways (standalone or IIS). Each method has a different set of constraints and considerations that must be addressed before choosing which method to use.

**Table 24 - WCF Server Connector decision matrix**

Concept	Standalone	IIS Hosted
Hosting	Windows Service	IIS
Load Balancing	Manually created	IIS Load Balancing
Messaging	Receive all messages received at endpoint	Forwards messages to specific "handlers" based on content
Configuration	App Config & Code	Web Config only
Formatter Assignment	1 Formatter, infinite graph	1 Formatter, 1 Graph Aide

	aides	
Bindings	basicHttpBinding, wsHttpBinding, netTcpBinding, namedPipeBinding	basicHttpBinding, wsHttpBinding
Ports	Any port, any URI	IIS port at deployed URI

## Standalone Mode

Hosting an application in standalone mode requires the developer to handle the management of the service and service state. In standalone mode, the developer writes a windows service which actively listens on the specified channel.

The first step to creating a standalone service is to setup the app.config file

```
<system.serviceModel>
  <services>
    <service name="ApplicationService">
      <endpoint bindingNamespace="http://temp/"
        bindingConfiguration="TestBinding"
        name="TestEndpoint" address="http://localhost:8000"
        contract="MARC.Everest.Connectors.WCF.Core.IConnectorContract"
        binding="wsHttpBinding"/>
    </service>
  </services>
...
<bindings>
  <wsHttpBinding>
    <binding name="TestBinding" maxReceivedMessageSize="12000000">
      <reliableSession enabled="true"/>
      <security mode="Message"/>
    </binding>
  </wsHttpBinding>
</bindings>
```

### Example 38 - Sample app.config for hosted service

After the app.config file has been setup properly, it is time to create the service application. The following example implements a console application that returns an MCCI\_IN000002CA structure back to any caller.

The actual "main" function will contain the logic that will setup the connector and start listening on the endpoint specified (The complete code sample can be found in appendix A)

```
// Main function
public static void Main(string[] args)
{
  Assembly.Load(new AssemblyName("MARC.Everest.Core.MR2009"));
  // Instantiate the connector
  WcfServerConnector connector = new WcfServerConnector();
  // Assign the formatter
  connector.Formatter = new
MARC.Everest.Formatter.XML.ITS1.Formatter();
  connector.Formatters.GraphAides.Add(
    typeof(MARC.Everest.Formatters.XML.Datatypes.R1.Formatter)
```



```
);
```

Next, the service must subscribe to the `MessageAvailable` event. This event is fired when the underlying channel receives a message from a remote endpoint (ie: it does not mean the message has been parsed or validated, just that a message is available to be consumed)

```
// Subscribe to the message available event
connector.MessageAvailable += new
EventHandler<UnsolicitedDataEventArgs>(
    connector_MessageAvailable
);
```

Finally, the console application will start the listener, and wait until a key is pressed (the exit condition)

```
// Open and start the connector
connector.ConnectionString = "ServiceName=ApplicationService";
connector.Open();
connector.Start();

Console.WriteLine("Server is started, press any key to close...");
Console.ReadKey();

// Teardown
connector.Stop();
connector.Close();
} // End main function
```

The connection string value for the `WcfServerConnector` is listed in Table 25.

**Table 25- WCF Server connector connection string**

Parameter	Value(s)	Description
ServiceName	Name of a WCF service	The name of the WCF service to bind to

The `connector_MessageAvailable` method needs to be coded. The `MessageAvailable` event is an event handler that passes an `UnsolicitedDataEventArgs` parameter. The sender of the event is the connector that is notifying the application of the new data.

```
// Delegate handles the message available event
static void connector_MessageAvailable(object sender,
    UnsolicitedDataEventArgs e)
{
    // Cast connector
    WcfServerConnector connector = sender as WcfServerConnector;
```

Because the sender is a `WcfServerConnector`, it is best practice to cast it to that type and use the cast variable throughout the function. Because the notification is that of a message available, the application must receive the message (de-serialize and validate).

```
// Receive the structure
WcfReceiveResult rcvResult = connector.Receive() as WcfReceiveResult;
```

The WCF server connector supports bi-directional communications, it is important that the application sends a message back to the caller. In this scenario, the application will send a generic application acknowledgment with some validation errors.

An application acknowledgement must contain the identifier of the request message that it is acknowledging. Since there will be quite a bit of detail in the acknowledgement, it is best practice to create an instance of an acknowledgement structure.

```
// Prepare acknowledgement structure
Acknowledgement acknowledgement = new Acknowledgement();

// Assign the correlation
acknowledgement.TargetMessage = new TargetMessage(
    (rcvResult.Structure as IIdentifiable).Id
);
```

The connector.Receive() function returns an instance of WcfReceiveResult. This class contains some data about the details of de-serialization. The developer can use these details to determine if the application can process the message. In the following code sample, this data is used to determine which ack type to respond with.

```
// Determine the deserialization outcome
if (rcvResult.Code != ResultCode.Accepted
&& rcvResult.Code != ResultCode.AcceptedNonConformant)
    // There were problems parsing the request message
    acknowledgement.TypeCode =
    AcknowledgementType.AcceptAcknowledgementCommitError;
else
    // Message is all good
    acknowledgement.TypeCode =
    AcknowledgementType.AcceptAcknowledgementCommitAccept;
```

The additional data in the WcfReceiveResult class can be used to append details about problems de-serializing the message.

```
// Append all details
foreach (IResultDetail dtl in rcvResult.Details)
{
    AcknowledgementDetail detail = new AcknowledgementDetail(
        AcknowledgementDetailType.Information,
        AcknowledgementDetailCode.SyntaxError
    );
    detail.Text = dtl.Message;
    acknowledgement.AcknowledgementDetail.Add(detail);
}
```

This structure is appended to the response message.

```
// Create a response
MCCI_IN000002CA response = new MCCI_IN000002CA(
    new II(Guid.NewGuid().ToString()),
    DateTime.Now,
```

```

        ResponseMode.Immediate,
        new II("", "MCCI_IN000002CA"),
        new LIST<II>(),
        ProcessingID.Production,
        AcknowledgementCondition.Never,
        new MARC.Everest.Core.MR2009.MCCI_MT002200CA.Receiver(
            new MARC.Everest.Core.MR2009.MCCI_MT002200CA.Device2(
                new II()
            )
        ),
        new MARC.Everest.Core.MR2009.MCCI_MT002200CA.Sender(
            new MARC.Everest.Core.MR2009.MCCI_MT002200CA.Device1(
                new II(MY_OID)
            )
        ),
        acknowledgement
    );

```

Finally, the application needs to send a response back the remote endpoint. In order to do this, the application must correlate the response with the request.

```

        // Send the result
        WcfSendResult sndResult = connector.Send(response, rcvResult);
    } // end connector_MessageAvailable

```

As with the WcfReceiveResult, the WcfSendResult class returned by the send operation contains details about the serialization of the response.

When a response message is not serialized properly, the WcfServerConnector will notify a caller by raising the InvalidResponse event. In this event, it is possible for the application developer to provide an alternative message (ie: correct the issue).

```

connector.InvalidResponse += new EventHandler<MessageEventArgs>(
connector_InvalidResponse
);
static void connector_InvalidResponse(object sender,
MessageEventArgs e)
{
    if (e.Code != ResultCode.NotAvailable)
        e.Alternate = new MCCI_IN000002CA(); // Construct alternative

    // if e.Alternate is not specified a soap fault is thrown
} // end connector_InvalidResponse

```

### Example 39 - Invalid response handler

If the event is not handled, or an "alternative" message is not specified, a soap fault will be sent to the remote endpoint.

## Hosting in IIS

The WCF Server connector may also be hosted within an IIS environment. The first step to hosting in IIS is to register the ConnectorService as the service handler. This is done by editing the Service.svc file:

```

<%@ ServiceHost Language="C#"
    Service="MARC.Everest.Connectors.WCF.Core.ConnectorService,

```

```
MARC.Everest.Connectors.WCF, Version=1.0.0.0"%>
```

The web configuration file also needs to be updated. The configuration section "marc.everest.connectors.wcf" is registered first.

```
<configSections>
  <section
type="MARC.Everest.Connectors.WCF.Configuration.ConfigurationSection,
  MARC.Everest.Connectors.WCF, Version=1.0.0.0"
  name="marc.everest.connectors.wcf"/>
</configSections>
```

How the services element in the System.ServiceModel configuration is populated depends on the transport being used. The only limitation is that the name of the service must match **MARC.Everest.Connectors.WCF.Core.ConnectorService** and the contract must be set to

**MARC.Everest.Connectors.WCF.Core.IConnectorContract.**

```
<services>
  <service behaviorConfiguration="Test"
name="MARC.Everest.Connectors.WCF.Core.ConnectorService">
    <endpoint bindingNamespace=http://marc.mohawkcollege.ca/hi
bindingConfiguration="TestBinding" name="Testndpoint"
contract="MARC.Everest.Connectors.WCF.Core.IConnectorContract"
binding="basicHttpBinding"/>
  </service>
</services>
```

Finally, the marc.everest.connectors.wcf configuration section is used to instruct the connector how to process inbound messages. This includes setting a formatter, graph aide.

```
<marc.everest.connectors.wcf
  formatter="MARC.Everest.Formatters.XML.ITS1.Formatter,
  MARC.Everest.Formatters.XML.ITS1"
  aide="MARC.Everest.Formatters.XML.Datatypes.R1.Formatter,
  MARC.Everest.Formatters.XML.Datatypes.R1">
  <messageHandler classifier="/*"
type="TestHostedWcfService.Test.Test, TestHostedWcfService"/>
</marc.everest.connectors.wcf>
```

The developer must also set one or more messageHandler elements. These elements are used by the connector to classify incoming messages based on their content and route the message to an appropriate message receiver.

Classification is performed on the entire soap message, it is therefore possible to classify on WS-Addressing or soap:action elements in the soap header. In Example 40, any message with a body root node of REPC\_IN000076CA is routed to the REPC\_IN000076CA handler, and anything else is routed to the class "Test".

```
<messageHandler classifier="/*[local-name() = 'Body']/*[local-name() =
'REPC_IN000076CA']"
type="TestHostedWcfService.Test.REPC_IN000076CA,
TestHostedWcfService"/>
<messageHandler classifier="/*" type="TestHostedWcfService.Test.Test,
```

```
TestHostedWcfService"/>
```

#### Example 40 - Classification rules

Once the web application is configured, the developer may now implement the message handler classes. This is done by implementing the `MARC.Everest.Connectors.WCF.Core.IMessageReceiver` interface (see Example 41)

```
public class Test : IMessageReceiver
{
    #region IMessageReceiver Members

    public MARC.Everest.Interfaces.IGraphable MessageReceived(
MARC.Everest.Interfaces.IGraphable received,
MARC.Everest.Connectors.ResultCode graphResult,
MARC.Everest.Connectors.IResultDetail[] graphDetails
    )
    {
        return received;
    }

    #endregion
}
```

#### Example 41 - IMessageReceiver implementation

The "MessageReceived" method is called when the WCF connector receives a new message from the transport mechanism. The message is parsed, validated and passed to the method. The three parameters of the "MessageReceived" method are:

##### **received**

This parameter represents a `IGraphable` object that is to be processed.

##### **graphResult**

This parameter represents the outcome of the de-serialization. The values `Accepted` and `AcceptedNonConformant` mean that the received object is populated. Any other value indicates that no processing could be performed on the message.

##### **graphDetails**

This parameter represents a set of result details that outline any problems or detected issues with the incoming message itself.

Example 42 illustrates how to use the data in the "MessageReceived" parameters to classify incoming messages and hand them off to a processing function.

```
public class Test : IMessageReceiver
{
    #region IMessageReceiver Members

    public MARC.Everest.Interfaces.IGraphable MessageReceived(
MARC.Everest.Interfaces.IGraphable received,
MARC.Everest.Connectors.ResultCode graphResult,
MARC.Everest.Connectors.IResultDetail[] graphDetails)
    {
    }
```

```

{
    // Not valid
    if (graphResult != ResultCode.Accepted ||
graphResult != ResultCode.AcceptedNonConformant)
        throw new InvalidOperationException("Message invalid!");
    // Put discharge care summary
    else if (received is REPC_IN000076CA)
        return putDischargeSummary(received);
    // List discharge care summary
    else if (received is REPC_IN000086CA)
        return listDischargeSummary(received);
    else // Couldn't understand
        return genericResponse(received);
}

```

**Example 42 - Message classification in IMessageReceiver**

## Message Queues

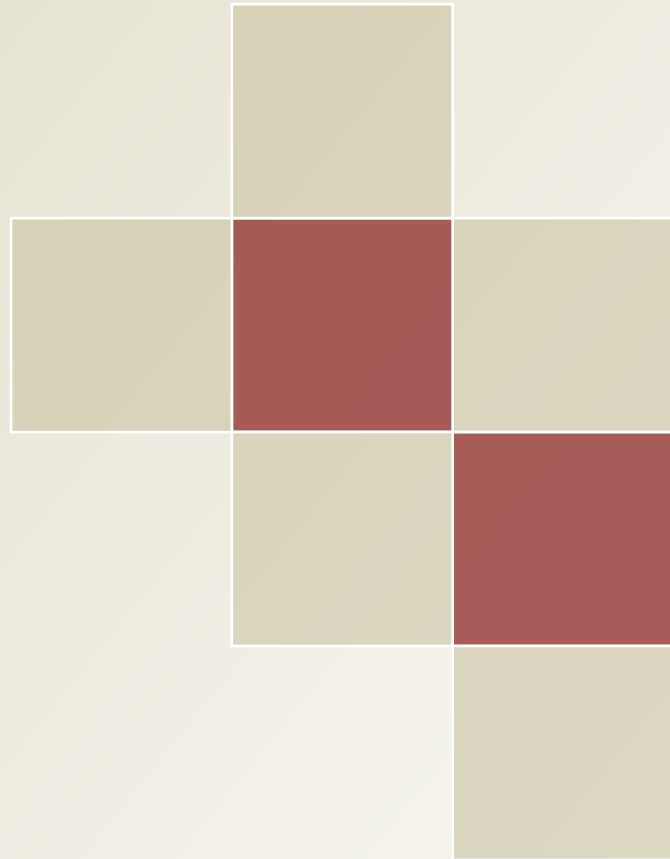
The message queue connector allows a developer to publish and subscribe to Microsoft Message Queues. The MSMQ connector works like all other pub/sub connectors (example: the file connector).

The connection string parameters for both the MsmqPublishConnector and MsmqListenConnector are listed in Table 26.

**Table 26 - MSMQ connector connection string**

Parameter	Value(s)	Description
Queue	The name of the queue ({server}\{queue})	Identifies the queue the connector should connect to.
Exclusive	True or false	If true, the connector will lock the queue for exclusive access.
Cache	True or false	If true, the connector will cache the queue messages.

## Chapter 5. Advanced Topics



- *How to generate a DLL from MIFv2 files*
- *How to write a custom formatters and connectors*
- *How to extract formatter meta data from a generated DLL*

## The General Purpose MIF Renderer

GPMR (General Purpose MIF Renderer) is a console utility that can be used to generate a variety of different output from MIFv2 files. The following different rendering outputs are being supported:

- HTML (and Deki-Wiki extended HTML)
- .NET Assemblies (and C# source code project)
- Java Libraries (and Java source code)

GPMR's rendering components do not read MIF structures. Instead, the MIF structures are abstracted in a format called COR (Common Object Representation). COR contains a small set of data structures that represent MIF structures in a more traditional OOP fashion (associations are removed, inheritance is corrected, etc...)

## GPMR Usage

Being a console utility, GPMR's behavior is modified through the use of command line switches. These switches can be passed in one of two ways:

- **Long-Hand Notation:** Where the full name of the parameter is prefixed with two dashes (example: --verbosity=X)
- **Short-Hand Notation:** Where a single character, prefixed with a single dash, is used to represent a parameter (example: -v X)

In addition to this, GPMR's parameters are broken into different categories based on what part of the rendering they modify.

### Core Parameters

Verbosity	
-v <i>N</i>   --verbosity= <i>N</i>	Controls the verbosity of the output. This is a 16 bit-mask. Output is controlled by adding the values below. (Example: Fatal and Errors is -v <b>3</b> ).  1 – Fatal 2 – Errors 4 – Warnings 8 – Debug 16 - Information
-d   --debug	Shortcut to -v 15. Only debug, warning error and fatal messages are displayed.
-e   --errors	Shortcut to -v 3. Only fatal and error messages are displayed.
-c   --chatty	All messages are displayed.
-q   --quiet	No messages are displayed.
Help	



<code>--version</code>	Shows the version of all installed GPMR components and quits.
<code>-?   --help</code>	Shows detailed usage instructions and quits.
<b>Input</b>	
<code>-s <i>filePattern</i>   --source=<i>filePattern</i></code>	Add a source MIF file (or wildcard) to the rendering list.
<b>Output</b>	
<code>-o <i>directory</i>   --output=<i>directory</i></code>	Set the target of the primary renderer.
<code>--extension=<i>assembly</i></code>	Adds a new extension to the GPMR pipeline.
<code>-r <i>renderId</i>   --renderer=<i>renderId</i></code>	Activates a renderer.
<b>Debugging</b>	
<code>--pipe-sniffer=<i>outputFile</i></code>	Dumps the contents of the rendering pipeline to <i>outputFile</i> after each pipeline state change.

## DEKI / HTML Renderer

<code>--deki-url=<i>url</i></code>	The root URL of the MindTouch Deki-Wiki server on which to publish articles. Url should be the root of the deki where API calls can also be made. Example: <a href="http://142.222.45.23">http://142.222.45.23</a>
<code>--deki-path=<i>articlePath</i></code>	The path within the wiki to publish articles to. Example: <code>/sample/articles/mif</code>
<code>--deki-user=<i>login</i></code>	The user account to publish articles under.
<code>--deki-password=<i>password</i></code>	The password of the user specified in the <code>--deki-user</code> . If this is not specified, the user will be prompted for it when appropriate.
<code>--deki-htmlpath=<i>localPath</i></code>	Indicates a local path that HTML files should be saved to. This can be combined with the <code>--deki-nopub=true</code> parameter to generate local only documentation.
<code>--deki-template=<i>templateName</i></code>	The name of the template to use.
<code>--deki-nopub=<i>false true</i></code>	If set to true, no articles are published to the deki-wiki. Only local files are saved in the <code>--deki-htmlpath</code> location.

## RMIM Based API C# Renderer

<code>--rimbapi-root-class=<i>className</i></code>	Name of the root class to use for an inheritance structure. The default is <code>RIM.InfrastructureRoot</code> .
<code>--rimbapi-target-ns=<i>namespaceName</i></code>	Sets the target namespace for the generated files. The default is 'output'.
<code>--rimbapi-compile=<i>true false</i></code>	If true, performs a compile on the output project.
<code>--rimbapi-dllonly=<i>true false</i></code>	If true, only the DLL is produced. All source code files are removed.
<code>--rimbapi-license=<i>bsd mit lgpl</i></code>	Sets the license that should be pre-pended to all generated files.
<code>--rimbapi-org=<i>organizationName</i></code>	Sets the name of the organization to place on copyrights.
<code>--rimbapi-gen-vocab=<i>true false</i></code>	If true, generates structural vocabulary as enumerations. If false, coded values are represented as strings. Default is false.
<code>--rimbapi-gen-rim=<i>true false</i>*</code>	If true, generates classes in the RIM namespace. Default is false.
<code>--rimbapi-oid-profileid=<i>oid</i></code>	Changes the default OID used for the root of ProfileId returned by GetProfileId (default is 2.16.840.1.113883.2.20.2)
<code>--rimbapi-oid-interactionid=<i>oid</i></code>	Changes the default OID used by the GetInteractionId function (default is 2.16.840.1.113883.1.18)
<code>--rimbapi-oid-triggerevent=<i>oid</i></code>	Changes the default OID used by the GetTriggerEvent function (default is 2.16.840.1.113883.1.18)
<code>--rimbapi-profileid=<i>profileid</i></code>	Enables the GetProfileId function, and sets its return value to the specified profile

## XSD Based Renderer (Experimental)

<code>--xsd-target-ns=<i>namespace-uri</i></code>	Specifies the target namespace uri of the generated XSDs.
<code>--xsd-xslt=<i>true false</i></code>	If true, generates a series of XSL files that transform between the generated XSDs and any collapsed representation.
<code>--xsd-gen-vocab=<i>true false</i></code>	If true, generates a voc.xsd based on the vocabulary structures in the MIF files.
<code>--xsd-version=<i>ITS_XXX</i></code>	Sets the fixed string of ITS_Version in the XSD.

<code>--xsd-datatypes-xsd=<i>datatypes.xsd</i></code>	Specifies the XSD to look for data-types.
<code>--xsd-rim-xslt=<i>true false</i></code>	If true, generates a series of XSL files that translate instances to/from a RIM based ITS representation.

\* - Enabling this feature may cause compilation to fail as all data types are not yet implemented.

## Examples

Generates DEKI WIKI documentation and publishes to server 192.168.0.100 at location /MyMifDocumentation using account fyfej

```
gpmr -r DEKI "C:\mifs\*.mif" --deki-url=http://192.168.0.100
--deki-path=/MyMifDocumentation --deki-user=fyfej
```

Generates HTML documentation and place into C:\html

```
gpmr -r DEKI "C:\mifs\*.mif" --deki-nopub=true --deki-htmlpath=C:\html
```

Generates a Visual Studio project in C:\cs

```
gpmr -r RIMBA_CS "C:\mifs\*.mif" -o "C:\cs"
```

Generates a Visual Studio project in C:\CS with namespace "test.api", organization "My Organization", with BSD license, all vocabulary, and all RIM classes

```
gpmr -r RIMBA_CS "C:\mifs\*.mif" -o "C:\cs" --rimbapi-target-ns=test.api
--rimbapi-gen-rim=true --rimbapi-gen-vocab=true
--rimbapi-license=bsd --rimbapi-org="My Organization"
```

Generates a DLL in C:\CS with namespace test.api and all vocabulary

```
gpmr -r RIMBA_CS "C:\mifs\*.mif" -o "C:\cs" --rimbapi-target-ns=test.api
--rimbapi-gen-vocab=true --rimbapi-license=bsd
--rimbapi-org="My Organization"
--rimbapi-compile=true --rimbapi-dllonly=true
```

## Optimizing COR Structures

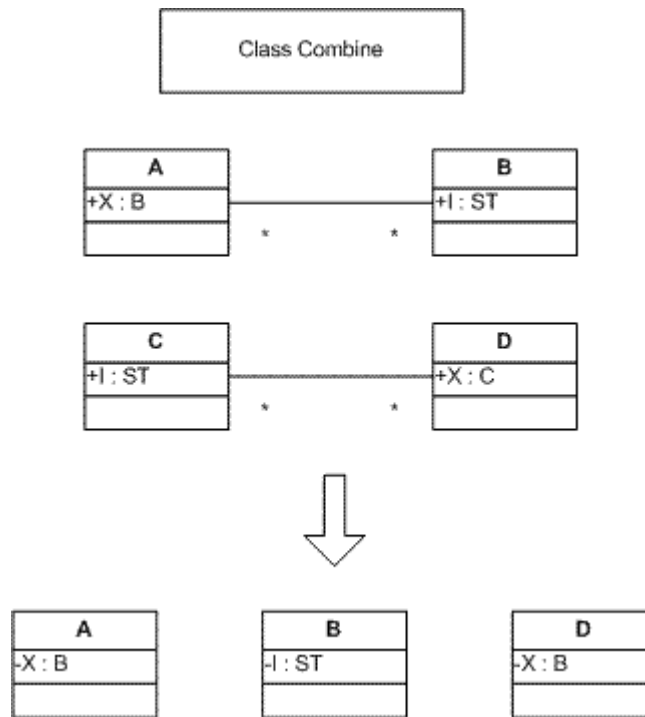
GPMR version 0.8.x and later come with a special pipeline trigger called the COR optimizer trigger. This trigger, when activated, will apply a series of algorithms to optimize the COR structures within the GPMR rendering pipeline.

There are four different methods that the COR Collapser will optimize the COR structures. While it may look like a destructive process, the collapse keeps a log of optimizations performed on a particular property so that any collapsing can be "inflated" at a later time.

## Combining

In the combine algorithm, if two classes share identical content (illustrated in Figure 6 with class **B** and **C**), then the combining algorithm merges the two classes and updates and reference.

This optimization provides a lighter weight COR structure.

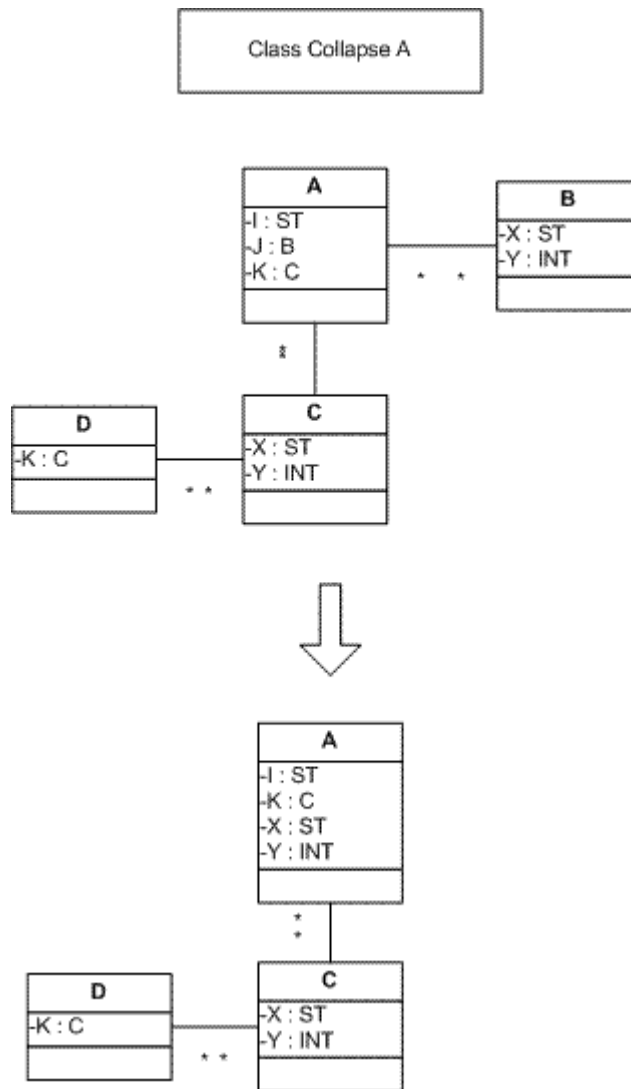


**Figure 6 - Combining algorithm**

### **Collapsing (Method A)**

This algorithm will identify classes that can be “rolled” into their parents. In this algorithm, if a level of nesting (illustrated in Figure 7 as class B to A) is not necessary and carries no additional information it is rolled into class A (creating a class AB).

Although this is not correct according to the pure model, it reduces the complexity of class A by getting rid of class B. It is important to note that this algorithm is only applied if class B is of a one-to-one relation and does not act as a relation to another complex structure (illustrated as class C in Figure 7)

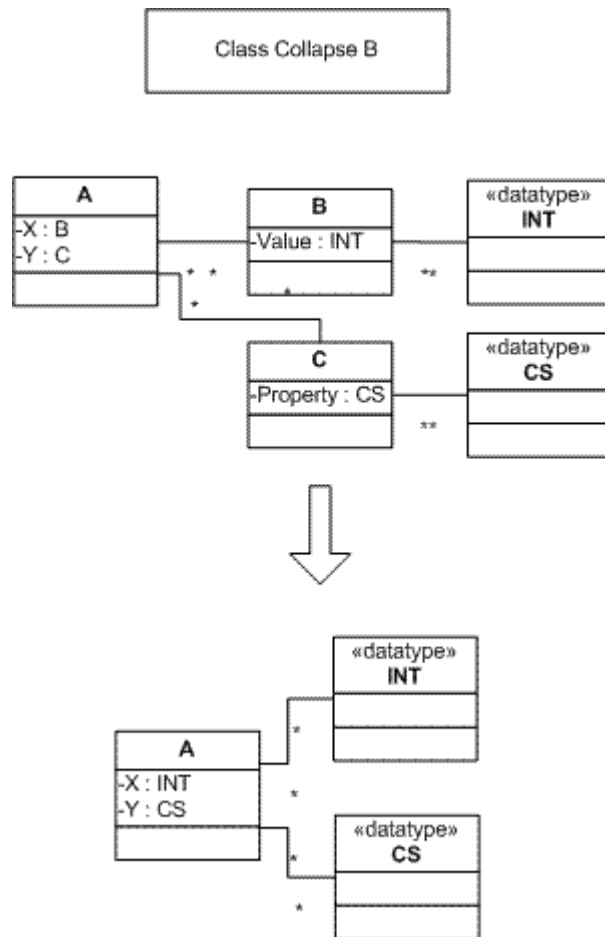


**Figure 7 - Collapsing method A**

### Collapsing (Method B)

The second collapsing method will reduce complexity by getting rid of nesting that provides no functional purpose (in Figure 8 this is represented by class **B** and **C**).

This algorithm is applied quite a lot when dealing with query parameters. For example, if the element query parameter "administrativeGender" (a code) is represented in the model as *administrativeGender.Value.Code* the collapsing algorithm will get rid of the *.value* nesting and will translate the data-type to the parent *administrativeGender*.



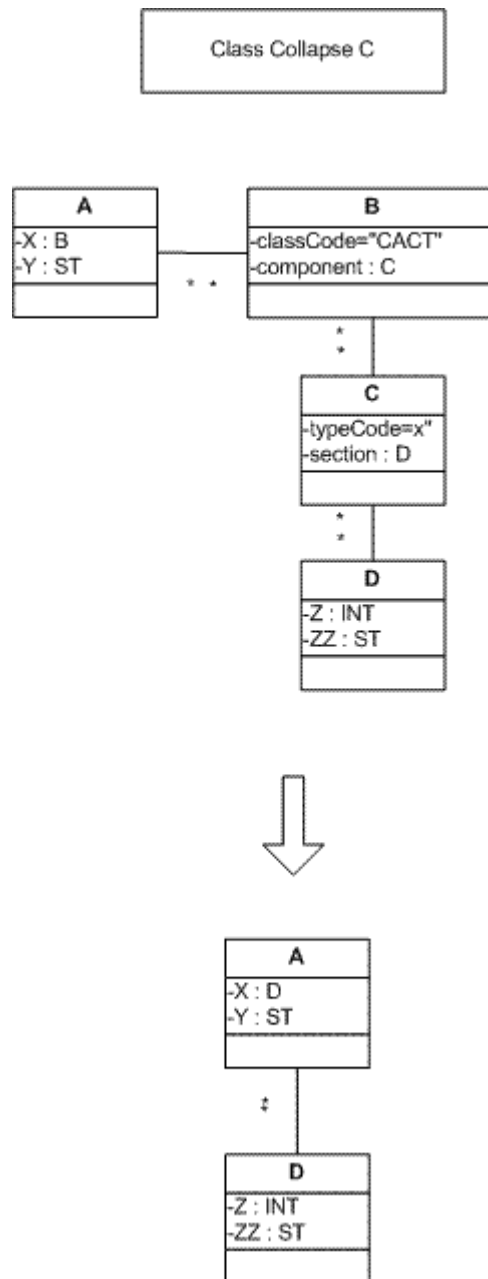
**Figure 8 - Collapsing method B**

### Collapsing (Method C)

The third, and final method of collapsing, requires an "ignore list". The COR collapser will remove deep-nesting by determining if the nested class "adds value". This is done by collecting the count of properties within the class in question and removing "unnecessary elements" (fixed elements, or elements that add no value at this level such as "nullFlavor").

If the counter value is 1 (ie: the class has only one "useful" element) and that element provides a link to another complex type (Method B is used for simple types) then the complex type reference is promoted to the parent level. This algorithm continues until a complex structure that adds value (counter > 1) is found, or a simple type is found.

This algorithm is illustrated in Figure 9. Class **B** and **C** add no real value to the structure, and are removed.



**Figure 9 - Collapsing method C**

### Using Collapsing

Activating the collapsing algorithms is done by appending the following command-line switches to the GPMR command prompt

<code>--optimize=true false</code>	If true, then the COR optimization trigger will execute (activates optimizing trigger)
<code>--collapse=true false</code>	Enables or disables the collapsing algorithms in the COR optimization trigger

<code>--combine=<i>true false</i></code>	Enables or disables the combining algorithms in the COR optimization trigger
<code>--collapse-ignore-fixed=<i>true false</i></code>	If true, the collapsing algorithm will treat fixed value properties as "adding no value"
<code>--collapse-useless-name=<i>name</i></code>	This parameter can be used multiple times. It is used by the collapsing algorithm to determine a "best name" for collapsed structures. Examples of useless names are "component", "section"
<code>--collapse-important-name=<i>name</i></code>	This parameter is used multiple times to create a list of important words. These are used to preserve context. Examples of important names are "subject", "componentOf"
<code>--collapse-adv-naming=<i>true false</i></code>	Enables or disables the use of advanced naming algorithm. This is used by the collapsing algorithm to determine the "best" name for a collapsed element.
<code>--collapse-ignore=<i>name</i></code>	Used multiple times to build a list of named elements that can be ignored when counting "useful" entries. This item must be used if collapsing is used. A good example of what to put in here is "nullFlavor"

## Extracting API Meta Data

The code generated from GPMR has lots of meta-data appended to it. This meta-data can be used for anything ranging from formatting, to validation and response creation.

There are 10 attributes that can be assigned to RMIM structures and properties, they are listed in Table 27.

**Table 27 - Everest meta-data attributes**

Attribute	Assigned To	Description
StructureAttribute	Class	Specifies the type, name and other data related to an RMIM or data-type structure.
PropertyAttribute	Property	Specifies the conformance, name, occurrence, type and other properties related to an RMIM property.
EnumerationAttribute	Field	Specifies a wire-representation (code mnemonic) for an enumerated value.



InteractionAttribute	Class	Identifies that a structure is an interaction, and specifies its trigger event.
InteractionResponseAttribute	Class	Specifies the possible responses to an interaction and the trigger events those response interactions send.
FlavorAttribute	Method	Identifies a method as a validation function for a particular flavor.
FlavorMapAttribute	Class	Specifies a flavor->type map on a class. For example: CD.CV maps to CV.
MarkerAttribute	Property	Marks a particular field as having special meaning to an HL7 system.
StateAttribute	Property	Specifies a value that can be assigned to a property. This is used as a state validation.
StateTransitionAttribute	Property	Specifies a valid transition of the value for a property. This is used for validating state transition.

## Structure Attributes

The StructureAttribute identifies a class as being an RMIM structure. It carries data related to the name, type and code system identifier found in the MIF to C#. The valid structure types are listed in Table 28.

**Table 28 - StructureAttribute types**

Structure Type	Description
MessageType	Identifies the class as an RMIM message type class. This class has no entry point and should be used as a component of an interaction.
DataType	Identifies the class as a data-type class. This means that class contains special validation functionality.
Interaction	Identifies the class as an interaction. Interactions may be used as an entry point for serialization and are expected to have an InteractionAttribute attached.
ValueSet	Identifies an enumeration as a value set. Each literal in the enumeration is expected to have an EnumerationAttribute appended.

**Note:** When the structure attribute is used to identify an enumeration as an RMIM structure, it will carry the name, and code domain.

## Property Attributes

Property attributes are used to identify a property within an RMIM or data-type structure as “serializable”. If a property does not have a PropertyAttribute attached to it, it should not be serialized.

Property attributes carry information such as wire format name, conformance, min/max occurs, default update mode, imposed flavors, and other data.

### Impose Flavor

The ImposeFlavorId parameter of the property attribute is used to enforce a specific type of flavor on the property. For example, if a property is of type II and the ImposeFlavorId is set to BUS, then the value of the property must meet the criteria for an II.BUS flavor.

### Supplier Domain

The SupplierDomain parameter of the property attribute is used to identify a code realm that values can be selected from. When the supplier domain is set on a codified data type, the code-system should be set to the supplier domain.

This attribute can also be used in automating form generation and code validation.

### Type

The property attribute can be used to identify conditional serialization. The type property is used to attain this goal. The type parameter is a qualifier that dictates when the current property attribute data should be applied to the class property.

Example 43 illustrates this concept. When the value of MyProperty is set to an instance of “FooType” the first PropertyAttribute data is applied. When MyProperty is an instance of BarType, then the “bar” PropertyAttribute data is applied.

```
[Property (Name = “foo”, Type = typeof(FooType))]  
[Property (Name = “bar”, Type = typeof(BarType))]  
public Object MyProperty { get; set; }
```

### Example 43 - Conditional property formatting

### Property Type

The property type parameter of PropertyAttribute is used to identify the type of property. This data can be used based on the rules of the particular ITS. The possible values for PropertyType are listed in Table 29.

**Table 29 - Property types**

Property Type	Description
Structural	The property is structural in nature, and carries some sort meaning related to the containing class (in XML this would be rendered as an Attribute).
NonStructural	The property is non-structural, meaning it is an association with another class (in XML this would be rendered as an Element).

### Conformance

The conformance parameter of the PropertyAttribute class is used to identify the conformance rules related to the property. Possible values are outlined in Table 30.

**Table 30 - Conformance values**

Conformance	Description
Optional	Support for the property is optional. Implementers must note if they support the concept or not.
Populated	The property must be assigned (ie: must not be null) but can be assigned a null flavor.
Required	The property may be assigned if data is available.
Mandatory	The property must be assigned, and no null flavors are permitted.

## Enumeration Attributes

The enumeration attribute specifies a particular enumeration literal's code mnemonic. This represents a wire level representation of the literal, and allows the literal itself to be different than the mnemonic.

To convert to/from wire format, the utility class `MARC.Everest.Connectors.Util` can be used.

## Interaction Attributes

The interaction attribute is used to identify additional information related to an interaction structure. Interaction attributes can be found on structures that have a type of "Interaction".

The interaction attribute carries the code mnemonic for the default trigger event for a particular interaction.

## Interaction Response Attributes

The interaction response attributes specify the possible responses for a particular interaction. They are appended to RMIM structures with a type of "Interaction".

The name parameter of the interaction response attribute will specify the name parameter of the structure attribute of the structure which represents the response.

## Flavor Attributes

The flavor attribute is used to identify a function that can be used to validate a certain data-type flavor. The function may also modify the instance of the data-type it is validating to make it valid (if it is not valid already).

The method this attribute is attached to must have the following characteristics:

- It must be scoped as `STATIC`
- It must have a return type of `bool`
- It must accept a parameter that exactly matches the encapsulating type

Example 44 illustrates a method that validates the flavor `FOO.BAR`.

```
[Structure(Name = "FOO")]
public class foo<T>
{
    [Property(Name = "value")]
    public T Value { get; set; }
```

```

[Flavor(Name = "BAR")]
public static bool bar(foo<T> instance)
{
    return instance.Value != null;
}
}

```

**Example 44 - Flavor attribute usage**

## Flavor Map Attributes

The flavor map attribute is used primarily in the data-types classes to facilitate R1/R2 compatibility. The flavor map is used when an R1 data-type has been transferred to an R2 flavor of a data type (CV is an example, in R2 it is CD.CV).

The flavor map attribute is applied to the class that the flavor is being moved to. Example 45 illustrates the use of the flavor map attribute on the CD data type.

```

[Serializable]
[Structure(Name = "CD", StructureType = StructureAttributeType.DataType)]
[FlavorMapAttribute(FlavorId = "CV", Implementor = typeof(CV))]
[FlavorMapAttribute(FlavorId = "CE", Implementor = typeof(CE))]
public class CD<T> : CE<T>, IConceptDescriptor
{

```

**Example 45 - Flavor map usage**

## Marker Attributes

Marker attributes are used to signal that a property has a special meaning within an HL7v3 message. This can be used to identify status code, mood code, or other structural elements of an HL7v3 system.

The possible markers are listed in Table 31

**Table 31 - Marker attribute types**

Marker Type	Description
Flavor	The property is used to get or set the flavor of the instance.
NullFlavor	The property is used to signify a reason as to why the instance has no data.
UpdateMode	The property is used to express the mode in which the instance should be used to modify records.
ContextConduction	The property is used to signal the type of context conduction used in the instance.
ClassCode	The property is used to express the HL7v3 class code of the current instance.
TypeCode	The property is used to express the HL7v3 type code of the current instance.
DeterminerCode	The property is used to get/set the instance's determiner code.
Data	The property represents the internal data of the instance. In XML this is interpreted as the inner text.

## State Attributes

The state attribute is used to identify possible values that a property can have. This is very useful when implementing the HL7v3 state machines.

## State Transition Attributes

The state transition attribute specifies the possible transitions a property's value may take.

## Writing Custom Components

The MARC-HI Everest Framework is very flexible and supports third party extensions to the framework itself. This is done by implementing the various interfaces provided in the core framework assembly to perform additional tasks.

Using the framework, a developer could write a new connector to a proprietary transport, or a new formatter (Example: Writing an EDI over MLLP formatter and connector).

Custom components are not supported by MARC-HI, and developers should use third party extensions at their own risk.

## Creating Formatters

Creating a formatter can be a very tedious job depending on the amount of meta-data used to format the message. As stated in the "Using Formatters" section, all formatters must implement the `IStructureFormatter` interface.

In this section, a new `BinFormatter` class is created that uses the native .NET `BinaryFormatter` class to graph and parse objects to/from streams.

First, the new class is created that implements the `IStructureFormatter`.

```
/// <summary>
/// Formats structures to binary
/// </summary>
public class BinFormatter : IStructureFormatter
{
    /// <summary>
    /// Create a new instance of BinFormatter
    /// </summary>
    public BinFormatter() { this.GraphAides = new List<Type>(); }
```

Next, the `GraphAides` property must be declared. It is recommended that a private copy of a list of `IStructureFormatter` is used to keep instances of the types in `GraphAides`.

```
// A local list of instantiated graph aides
private List<IStructureFormatter> graphAides;

/// <summary>
/// Aides to this formatter
/// </summary>
public List<Type> GraphAides
{
    get;
```

```

        set;
    }

```

The details, handle structure and host properties need to be created. This class only uses the Details property.

```

/// <summary>
/// Tell any caller this formatter handles any structure
/// </summary>
public List<string> HandleStructure
{
    get { return new List<string>() { "*" }; }
}

/// <summary>
/// Details of the operation
/// </summary>
public IResultDetail[] Details
{
    get;
    private set;
}

/// <summary>
/// Get or set the host formatter
/// </summary>
public IStructureFormatter Host
{
    get;
    set;
}

```

The formatter must implement a clone method. This is used by connectors when multi-threading requests. Each thread will receive its own clone of the formatter. When cloning the object, it is important to reset the details and local copy of graph aides instances to null.

```

/// <summary>
/// Clone this formatter
/// </summary>
public IStructureFormatter Clone()
{
    BinFormatter bf = this.MemberwiseClone(); // Clone the object
    bf.graphAides.Clear(); // don't share graph aide instances
    bf.Details = null; // clear details
    // Now instantiate a new set of graph aides
    foreach (Type t in bf.GraphAides)
        bf.GraphAides.Add(t.Assembly.CreateInstance(t.FullName));
    return bf;
}

```

The final step to creating a formatter is writing the GraphObject and ParseObject. These functions may get very complicated and implementing "formatlets" is a recommended practice (see the source code for the ITS1 XML Formatter)

```

/// <summary>
/// Graph the object
/// </summary>
/// <param name="s">The stream to graph to</param>
/// <param name="o">The IGraphable object to graph</param>
/// <returns>The result of the operation</returns>
public ResultCode GraphObject(Stream s, IGraphable o)
{
    try
    {
        BinaryFormatter fmt = new BinaryFormatter();
        fmt.Serialize(s, o);
        return ResultCode.Accepted;
    }
    catch (Exception e)
    {
        Details = new IResultDetail[] {
            new ResultDetail(ResultDetailType.Error, e.Message, e)
        };
        return ResultCode.Error;
    }
}

/// <summary>
/// Parse object from <paramref name="s"/>
/// </summary>
/// <param name="s">The stream to parse the object from</param>
/// <returns>The parsed object</returns>
public IGraphable ParseObject(System.IO.Stream s)
{
    try
    {
        BinaryFormatter fmt = new BinaryFormatter();
        return (IGraphable)fmt.Deserialize(s);
    }
    catch (Exception e)
    {
        Details = new IResultDetail[] {
            new ResultDetail(ResultDetailType.Error, e.Message, e)
        };
        return null;
    }
}

```

## Creating Connectors

Creating a connector requires a high attention to detail and quite a bit of analysis work. Connectors must operate in synchronous and asynchronous modes, and must operate in a variety of environments which may not be known at design time.

Before creating a connector, the developer must answer these questions

- What interaction pattern does this connector implement (send, receive, send/receive)?
- Is this connector sending XML or some other formatted data?

- Can this connector operate in a listen/wait state?

After answering these questions, development of the connector may begin. The following interfaces must be implemented to attain the attributes outlined in Table 32

**Table 32 - Connector interfaces**

Interface	Attributes
IConnector	Base of all connector implementations.
IFormattedConnector	Connector that handles formatted data on the wire.
ISendingConnector	Connector that publishes data.
IReceivingConnector	Connector that receives data.
IListenWaitConnector	Connector that subscribes or actively listens for data.
IListenWaitRespondConnector	Connector that listens for data and provides an immediate response
ISendReceiveConnector	Connector that solicits a response from a remote endpoint.

Once the developer has determined the appropriate set of interfaces to implement, the development of the connector can commence.

In this section, a new MemoryConnector will be created to send instances to memory streams. The MemoryConnector is a formatted publishing connector.

The first step is to define an ISendResult implementation that can be used by our memory connector. Example 46 illustrates a simple implementation.

```

/// <summary>
/// Implementation of ISendResult
/// </summary>
public class SendResult : ISendResult
{
    #region ISendResult Members

    /// <summary>
    /// The result of the code
    /// </summary>
    public ResultCode Code { get; internal set; }

    /// <summary>
    /// The details of the result
    /// </summary>
    public IResultDetail[] Details { get; internal set; }

    #endregion
}

```

**Example 46 - SendResult implementation**



After this implementation of ISendResult is completed, the development of the connector can commence. The class declaration illustrates the functionality this connector will support (Formatted, Sending connector).

```
public class MemoryConnector :  
    IConnector,  
    IFormattedConnector,  
    ISendingConnector  
{
```

Although the constructor restrictions are not defined in the IConnector interface, it is recommended that developers implement the two constructors: a default constructor and a connection string constructor.

```
/// <summary>  
/// Create a new instance of the memory connector  
/// </summary>  
public MemoryConnector() { }  
/// <summary>  
/// Create a new instance of the memory connector with the  
/// specified connectionString  
/// </summary>  
/// <param name="connectionString">The connection string</param>  
public MemoryConnector(string connectionString)  
{  
    this.ConnectionString = connectionString;  
}
```

After these constructors are completed, the implementation of the IConnector interface follows. The IConnector methods in this sample are empty (as memory is connectionless). In a connection oriented environment, these methods would be used to setup the connection string, and manage the connection to the underlying transport.

```
/// <summary>  
/// Close the connection  
/// </summary>  
public void Close()  
{  
    // No need to close memory  
}  
  
/// <summary>  
/// Get or set the connection string  
/// </summary>  
public string ConnectionString { get; set; }  
  
/// <summary>  
/// Returns true if the connection is open  
/// </summary>  
public bool IsOpen()  
{  
    return true; // No need to check if memory is open  
}
```

```

/// <summary>
/// Open the connector
/// </summary>
public void Open()
{
    // No need to open the memory stream
}

```

The `IFormattedConnector` interface is a simple implementation. Its “Formatter” property is used to specify the “wire” format to use when sending instances.

```

/// <summary>
/// Get or set the formatter this connector is using
/// </summary>
public IStructureFormatter Formatter { get; set; }

```

The `ISendingConnector` members are where the majority of processing is completed. Because the `ISendingConnector` supports asynchronous and synchronous calls to the send method, it is recommended that developers implement a “worker bee” class.

This class performs the actual formatting and sending duties and can communicate these results back to a caller. This allows for code-reuse among both methods of calling the send method.

```

/// <summary>
/// Good practice to have a worker class to do the
/// actual work
/// </summary>
private class Worker
{
    /// <summary>
    /// The formatter to use
    /// </summary>
    public IStructureFormatter Formatter { get; set; }

    /// <summary>
    /// The result of the worker
    /// </summary>
    public SendResult Result { get; set; }

    /// <summary>
    /// The callback of the worker
    /// </summary>
    public event WaitCallback Completed;

    /// <summary>
    /// Perform the work
    /// </summary>
    /// <param name="state">A user state. In our case, the thing
    /// to be sent</param>
    public void Work(object state)
    {
        try
        {

```

```

        MemoryStream ms = new MemoryStream();
        Result = new SendResult();

        // Format and set result details
        Result.Code = Formatter.GraphObject(ms, (IGraphable)state);
        Result.Details = Formatter.Details;
    }
    catch (Exception e)
    {
        // Append the error
        Result.Code = ResultCode.Error;
        Result.Details = new IResultDetail[] {
            new ResultDetail(ResultDetailType.Error, e.Message, e)
        };
    }

    // Callback
    if (Completed != null) Completed(this);
}
}

```

The asynchronous send methods “Begin/End Send” require the MemoryConnector class to keep track of IAsyncResult and the SendResult classes created during sending. This is done in our sample implementation with a dictionary.

```

/// <summary>
/// The results
/// </summary>
private Dictionary<IAsyncResult, SendResult> results = new
Dictionary<IAsyncResult, SendResult>();

```

The implementation of the BeginSend method allows a caller to execute a send operation in asynchronous mode. The MARC-HI Everest Framework follows the standard convention of performing an asynchronous operation.

```

/// <summary>
/// Asynchronous send operation
/// </summary>
/// <param name="data">The data being sent</param>
/// <param name="callback">A callback for when the method is complete</param>
/// <param name="state">A user state</param>
/// <returns>An IAsyncResult representing the operation</returns>
public IAsyncResult BeginSend(
    IGraphable data,
    AsyncCallback callback,
    object state)
{
    // Good practice to check if the connector is open
    if (!IsOpen())
        throw new ConnectorException(
            ConnectorException.MSG_INVALID_STATE,
            ConnectorException.ReasonType.NotOpen, null);
}

```

The begin send method will create a new instance of our “worker bee” class and will clone the MemoryConnector’s current formatter. This is done to ensure that formatters being executed on multiple threads are kept isolated.

```
// Create a new worker
Worker w = new Worker();

// Always clone the current formatter
w.Formatter = Formatter.Clone();
```

The async result is created using the MARC-HI Everest “SendResultAsyncResult” class in the core assembly. A callback is also established with the worker instance. When the worker is completed its operation, this delegate will be called and the result of the work will be placed in the results dictionary.

```
// Create the async result
IAsyncResult result = new SendResultAsyncResult(state, new
AutoResetEvent(false));

// Set the callback
w.Completed += new WaitCallback(delegate(object sender)
{
    Worker sWorker = sender as Worker;

    // Set the result in the dictionary
    lock (this.results)
        this.results.Add(result, sWorker.Result);

    // Notify any listeners
    (result.AsyncWaitHandle as AutoResetEvent).Set();
    if (callback != null) callback(result);
});
```

Finally, the worker’s “Work” method is queued in the thread pool and the instance to be graphed is passed.

```
// Execute
ThreadPool.QueueUserWorkItem(w.Work, data);

return result;
}
```

The end send method of the connector is used by a caller to retrieve the result of the send operation.

```
/// <summary>
/// Complete the asynchronous send operation
/// </summary>
/// <param name="asyncResult">The reference to retrieve the result for</param>
/// <returns>The result of the send</returns>
public ISendResult EndSend(IAsyncResult asyncResult)
{
    SendResult result = null;
    if (results.TryGetValue(asyncResult, out result)) // There is a response
    {
```

```

        lock (this.results) // Remove the result
            this.results.Remove(asyncResult);

        return result; // return the result
    }
    else // Result is not available
        return null;
}

```

Finally, the developer must create an implementation of the synchronous send method. Because the work is completed by the “worker bee” class, this method is relatively straightforward.

```

/// <summary>
/// Synchronous send operation
/// </summary>
/// <param name="data">The data to send</param>
/// <returns>The result of the send operation</returns>
public ISendResult Send(MARC.Everest.Interfaces.IGraphable data)
{
    // Good practice to check if the connector is open
    if (!IsOpen())
        throw new ConnectorException(
            ConnectorException.MSG_INVALID_STATE,
            ConnectorException.ReasonType.NotOpen, null);

    // Create a new worker
    Worker w = new Worker();

    // Always clone the current formatter
    w.Formatter = Formatter.Clone();

    // Perform the work
    w.Work(data);

    return w.Result;
}

```

Thus completes an implementation of a MemoryConnector. The complete source for this sample can be found in Appendix A, or in the samples included with this package.

# Appendix A – Source Code

This appendix contains the complete source code reference to any of the larger code samples in the developer's guide.

## Contents

WCF Standalone Service .....	85
BinFormatter Custom Formatter .....	87
MemoryConnector Custom Connector .....	89

## WCF Standalone Service

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using MARC.Everest.Connectors.WCF;
using MARC.Everest.Connectors;
using MARC.Everest.Core.MR2009.Interactions;
using MARC.Everest.DataTypes;
using MARC.Everest.Core.MR2009.Vocabulary;
using MARC.Everest.Core.MR2009.MCCI_MT002200CA;
using MARC.Everest.Interfaces;

namespace Test
{
    public class ServerTest
    {
        // Main function
        public static void Main(string[] args)
        {
            Assembly.Load(new AssemblyName("MARC.Everest.Core.MR2009"));

            // Instantiate the connector
            WcfServerConnector connector = new WcfServerConnector();
            // Assign the formatter
            connector.Formatter = new
                MARC.Everest.Formatters.XML.ITS1.Formatter();
            connector.Formatter.GraphAides.Add(
                typeof(
                    MARC.Everest.Formatters.XML.Datatypes.R1.Formatter
                )
            );

            // Subscribe to the message available event
            connector.MessageAvailable += new
                EventHandler<UnsolicitedDataEventArgs>(
                    connector_MessageAvailable);

            // Open and start the connector
            connector.ConnectionString = "ServiceName=ApplicationService";
            connector.Open();

            connector.Start();
        }
    }
}
```

```

        Console.WriteLine("Server started, press any key to close...");
        Console.ReadKey();

        // Teardown
        connector.Stop();
        connector.Close();
    }

    // Delegate handles the message available event
    static void connector_MessageAvailable(object sender,
        UnsolicitedDataEventArgs e)
    {
        // Cast connector
        WcfServerConnector connector = sender as WcfServerConnector;
        // Receive the structure
        WcfReceiveResult rcvResult = (WcfReceiveResult)connector.Receive();

        // Prepare acknowledgement structure
        Acknowledgement acknowledgement = new Acknowledgement();

        // Assign the correlation
        acknowledgement.TargetMessage = new TargetMessage(
            (rcvResult.Structure as IIdentifiable).Id
        );

        // Determine the deserialization outcome
        if (rcvResult.Code != ResultCode.Accepted &&
            rcvResult.Code != ResultCode.AcceptedNonConformant)
        {
            // There were problems parsing the request message
            acknowledgement.TypeCode =
                AcknowledgementType.AcceptAcknowledgementCommitError;
        }
        else
        {
            // Message is all good
            acknowledgement.TypeCode =
                AcknowledgementType.AcceptAcknowledgementCommitAccept;
        }

        // Append all details
        foreach (IResultDetail dtl in rcvResult.Details)
        {
            AcknowledgementDetail detail = new AcknowledgementDetail(
                AcknowledgementDetailType.Information,
                AcknowledgementDetailCode.SyntaxError
            );
            detail.Text = dtl.Message;
            acknowledgement.AcknowledgementDetail.Add(detail);
        }

        // Create a response
        MCCI_IN000002CA response = new MCCI_IN000002CA(
            new II(Guid.NewGuid().ToString()),
            DateTime.Now,
            ResponseMode.Immediate,
            MCCI_IN000002CA.GetInteractionId(),

```

```

        MCCI_IN000002CA.GetProfileId(),
        ProcessingID.Production,
        AcknowledgementCondition.Never,
        new
            MARC.Everest.Core.MR2009.MCCI_MT002200CA.Receiver(
                new
                    MARC.Everest.Core.MR2009.MCCI_MT002200CA.Device2(
                        new II()
                    )
                ),
            new
                MARC.Everest.Core.MR2009.MCCI_MT002200CA.Sender(
                    new
                        MARC.Everest.Core.MR2009.MCCI_MT002200CA.Device1(
                            new II("1.2.3.4")
                        )
                    ),
                acknowledgement
            );

        // Send the result
        connector.Send(response, rcvResult);
    }
}
}

```

## BinFormatter Custom Formatter

```

using System;
using System.Collections.Generic;
using System.Text;
using MARC.Everest.Connectors;
using System.Runtime.Serialization.Formatters.Binary;

namespace Test
{
    /// <summary>
    /// Formats structures to binary
    /// </summary>
    public class BinFormatter : IStructureFormatter
    {
        /// <summary>
        /// Create a new instance of BinFormatter
        /// </summary>
        public BinFormatter() { this.GraphAides = new List<Type>(); }

        #region IStructureFormatter Members

        /// <summary>
        /// Clone this formatter
        /// </summary>
        public object Clone()
        {
            BinFormatter bf = (BinFormatter)this.MemberwiseClone(); // Clone

```



```

        bf.graphAides.Clear(); // don't share graph aide instances
        bf.Details = null; // clear details
        // Now instantiate a new set of graph aides
        foreach (Type t in bf.GraphAides)
            bf.GraphAides.Add(Type.GetType(t.FullName));
        return bf;
    }

    // A local list of instantiated graph aides
    private List<IStructureFormatter> graphAides;

    /// <summary>
    /// Aides to this formatter
    /// </summary>
    public List<Type> GraphAides
    {
        get;
        set;
    }

    /// <summary>
    /// Tell any caller this formatter handles any structure
    /// </summary>
    public List<string> HandleStructure
    {
        get { return new List<string>() { "*" }; }
    }

    /// <summary>
    /// Details of the operation
    /// </summary>
    public IResultDetail[] Details
    {
        get;
        private set;
    }

    /// <summary>
    /// Get or set the host formatter
    /// </summary>
    public IStructureFormatter Host
    {
        get;
        set;
    }

    /// <summary>
    /// Graph the object
    /// </summary>
    /// <param name="s">The stream to graph to</param>
    /// <param name="o">The IGraphable object to graph</param>
    /// <returns>The result of the operation</returns>
    public ResultCode GraphObject(Stream s, IGraphable o)
    {
        try

```

```

        {
            BinaryFormatter fmt = new BinaryFormatter();
            fmt.Serialize(s, o);
            return ResultCode.Accepted;
        }
        catch (Exception e)
        {
            Details = new IResultDetail[] {
                new ResultDetail(ResultDetailType.Error, e.Message, e)
            };
            return ResultCode.Error;
        }
    }

    /// <summary>
    /// Parse object from <paramref name="s"/>
    /// </summary>
    /// <param name="s">The stream to parse the object from</param>
    /// <returns>The parsed object</returns>
    public IGraphable ParseObject(Stream s)
    {
        try
        {
            BinaryFormatter fmt = new BinaryFormatter();
            return (IGraphable)fmt.Deserialize(s);
        }
        catch (Exception e)
        {
            Details = new IResultDetail[] {
                new ResultDetail(ResultDetailType.Error, e.Message, e)
            };
            return null;
        }
    }

    #endregion
}
}

```

## MemoryConnector Custom Connector

### SendResult.cs

```

/// <summary>
/// Implementation of ISendResult
/// </summary>
public class SendResult : ISendResult
{
    #region ISendResult Members

    /// <summary>
    /// The result of the code
    /// </summary>
    public ResultCode Code { get; internal set; }
}

```

```

    /// <summary>
    /// The details of the result
    /// </summary>
    public IResultDetail[] Details { get; internal set; }

    #endregion
}

```

## MemoryConnector.cs

```

public class MemoryConnector :
    IConnector,
    IFormattedConnector,
    ISendingConnector
{
    #region Constructors

    /// <summary>
    /// Create a new instance of the memory connector
    /// </summary>
    public MemoryConnector() { }

    /// <summary>
    /// Create a new instance of the memory connector with the
    /// specified connectionString
    /// </summary>
    /// <param name="connectionString">The connection string</param>
    public MemoryConnector(string connectionString)
    {
        this.ConnectionString = connectionString;
    }

    #endregion

    #region IConnector Members

    /// <summary>
    /// Close the connection
    /// </summary>
    public void Close()
    {
        // No need to close memory
    }

    /// <summary>
    /// Get or set the connection string
    /// </summary>
    public string ConnectionString { get; set; }

    /// <summary>
    /// Returns true if the connection is open
    /// </summary>
    public bool IsOpen()
    {
        return true; // No need to check if memory is open
    }

}

```

```

    /// <summary>
    /// Open the connector
    /// </summary>
    public void Open()
    {
        // No need to open the memory stream
    }

#endregion

#region IFormattedConnector Members

    /// <summary>
    /// Get or set the formatter this connector is using
    /// </summary>
    public IStructureFormatter Formatter { get; set; }

#endregion

    /// <summary>
    /// Good practice to have a worker class to do the
    /// actual work
    /// </summary>
    private class Worker
    {
        /// <summary>
        /// The formatter to use
        /// </summary>
        public IStructureFormatter Formatter { get; set; }

        /// <summary>
        /// The result of the worker
        /// </summary>
        public SendResult Result { get; set; }

        /// <summary>
        /// The callback of the worker
        /// </summary>
        public event WaitCallback Completed;

        /// <summary>
        /// Perform the work
        /// </summary>
        /// <param name="state">A user state. In our case, the thing
        /// to be sent</param>
        public void Work(object state)
        {
            try
            {
                MemoryStream ms = new MemoryStream();
                Result = new SendResult();

                // Format and set result details
                Result.Code = Formatter.GraphObject(ms, (IGraphable)state);
            }
            catch { }
        }
    }

```

```

        Result.Details = Formatter.Details;
    }
    catch (Exception e)
    {
        // Append the error
        Result.Code = ResultCode.Error;
        Result.Details = new IResultDetail[] {
            new ResultDetail(ResultDetailType.Error, e.Message, e)
        };
    }

    // Callback
    if (Completed != null) Completed(this);
}

#region ISendingConnector Members

/// <summary>
/// The results
/// </summary>
private Dictionary<IAsyncResult, SendResult> results =
    new Dictionary<IAsyncResult, SendResult>();

/// <summary>
/// Asynchronous send operation
/// </summary>
/// <param name="data">The data being sent</param>
/// <param name="callback">A callback to execute </param>
/// <param name="state">A user state</param>
/// <returns>An IAsyncResult representing the operation</returns>
public IAsyncResult BeginSend(IGraphable data, AsyncCallback callback,
    object state)
{
    // Good practice to check if the connector is open
    if (!IsOpen())
        throw new ConnectorException(
            ConnectorException.MSG_INVALID_STATE,
            ConnectorException.ReasonType.NotOpen, null);

    // Create a new worker
    Worker w = new Worker();

    // Always clone the current formatter
    w.Formatter = Formatter.Clone() as IStructureFormatter;

    // Create the async result
    IAsyncResult result = new SendResultAsyncResult(state, new
        AutoResetEvent(false));

    // Set the callback
    w.Completed += new WaitCallback(delegate(object sender)
    {
        Worker sWorker = sender as Worker;
    });
}

```

```

        // Set the result in the dictionary
        lock (this.results)
            this.results.Add(result, sWorker.Result);

        // Notify any listeners
        (result.AsyncWaitHandle as AutoResetEvent).Set();
        if (callback != null) callback(result);
    });

    // Execute
    ThreadPool.QueueUserWorkItem(w.Work, data);

    return result;
}

/// <summary>
/// Complete the asynchronous send operation
/// </summary>
/// <param name="asyncResult">The reference of the result </param>
/// <returns>The result of the send</returns>
public ISendResult EndSend(IAsyncResult asyncResult)
{
    SendResult result = null;
    if (results.TryGetValue(asyncResult, out result))
    {
        lock (this.results) // Remove the result
            this.results.Remove(asyncResult);

        return result; // return the result
    }
    else // Result is not available
        return null;
}

/// <summary>
/// Synchronous send operation
/// </summary>
/// <param name="data">The data to send</param>
/// <returns>The result of the send operation</returns>
public ISendResult Send(IGraphable data)
{
    // Good practice to check if the connector is open
    if (!IsOpen())
        throw new ConnectorException(
            ConnectorException.MSG_INVALID_STATE,
            ConnectorException.ReasonType.NotOpen, null);

    // Create a new worker
    Worker w = new Worker();

    // Always clone the current formatter
    w.Formatter = Formatter.Clone();

    // Perform the work

```

```
        w.Work(data);  
        return w.Result;  
    }  
#endregion  
}
```

## Appendix B – Type Operators

As stated in the “Data Types” section of this guide, the data types provide operator overloads. The following chart is a reference of all data type operators that are overloaded.

Type	Operator	Result	Example
<b>ADXP</b>	ADXP = string	Cast from string (Type set to AddressLine)	ADXP adxpInstance = “123 Main Street”;
	string = ADXP	Cast to string	String street = adxpInstance;
<b>BL</b>	BL = string	Cast from string	BL blInstance = “true”;
	bool = BL	Cast to Boolean	if(blInstance == true)
	BL = bool	Cast from Boolean	BL blInstance = false;
	bool? = BL	Cast to nullable Boolean	bool? B = (bool?)blInstance;
	BL = bool?	Cast from nullable Boolean	BL blInstance = (BL)B;
<b>CD</b>	CD = string	Code set to value of string	CD cdInstance = “AR”; (same as : cdInstance.Code = “AR”)
	string = CD	String set to value of code	String code = cdInstance; (same as : code = cdInstance.Code)
<b>CD&lt;T&gt;</b>	T = CD<T>	T set to value of code	ResponseMode m = cdResponseMode;
	CD<T> = T	Code set to value of T	CD<ResponseMode> cdInstance = ResponseMode.Immediate;
	CD<T> = CD	CD upgraded to CD<T>	CD<ResponseMode> cdInstance = new CD(“I”);
<b>CE&lt;T&gt;</b>	T = CE<T>	T set to value of Code	String mode = ceInstance; (same as : mode = ceInstance.Code)
	CE<T> = T	Code set to value of T	CE<String> mode = “123”;
<b>CS</b>	CS = string	Code set to value of string	CS csInstance = “P”; (same as : csInstance.Code = “P”)
	string = CS	String set to value of Code	String value = csInstance; (same as: value = csInstance.Code)



<b>CS&lt;T&gt;</b>	CS<T> = CS	CS upgraded to CS<T>	CS<ResponseMode> rm = new CS("AL");
	CS<T> = string	Code = literal of string	CS<ResponseMode> rm = "Always";
	CS<T> = T	Code set to value of T	CS<ResponseMode> rm = ResponseMode.Always;
	T = CS<T>	T set to value of Code	ResponseMode mode = rm;
<b>CV</b>	CV = string	Code set to value of string	CV cvInstance = "RCP-3454"; (same as: cvInstance.Code = "")
	string = CV	String set to value of code	String code = cvInstance; (same as : code = cvInstance.Code)
<b>CV&lt;T&gt;</b>	CV<T> = CV	CV upgraded to CV<T>	CV<ReasonCode> code = new CV("RSN");
	CV<T> = T	Code set to value of T	CV<ReasonCode> cvInstance = ReasonCode.Reason;
	T = CV<T>	T set to value of Code	ReasonCode code = cvInstance;
<b>ED</b>	string = ED	String set to contents of ED	String content = new ED("Hello");
	byte[] = ED	Byte[] set to contents of ED	Byte[] content = new ED("Hello");
	ED = string	Contents of ED set to string (mediaType = "text/plain", representation = TXT)	ED edInstance = "Hello";
<b>ENXP</b>	String = ENXP	Cast to string	String namePart = enxpInstance;
<b>INT</b>	INT = Int32	Cast from Int32	INT value = 1;
	Int32 = INT	Cast to Int32	int myInteger = new INT(23);
	INT = string	Cast from String (Parse)	INT value = "23";
<b>LIST&lt;T&gt;</b>	LIST<T> = List<T>	Cast from System.Collections.Generic.List	LIST<String> myList = new List<String>(new String[] { "Bob", "Jim", "Joe" });
<b>PQ</b>	double = PQ	Cast to double	Double value = new PQ(23.32f, "ft_i");
	PQ = double	Cast from double	PQ value = 23.32f; value.Unit = "ft_i";
<b>REAL</b>	Double = REAL	Cast to double	Double d = new REAL(2.34f);
	REAL = double	Cast from double	REAL r = 2.34f;

<b>RTO&lt;S,T&gt;</b>	Double = RTO<S,T>	Cast to double (floating point representation of ratio)	Double f = new RTO<INT,INT>(1,2); // Result = 0.5f
<b>ST</b>	ST = string	Cast from string	ST stringValue = "String Value!";
	String = ST	Cast to string	String str = stInstance;
<b>TEL</b>	TEL = string	Cast from string	TEL email = "mailto:test@google.com";
	String = TEL	Cast to string	String emailAddress = telInstance;
<b>TN</b>	TN = string	Cast from string	TN trivialName = "General Hospital";
<b>TS</b>	TS = DateTime	Cast from DateTime	TS now = DateTime.Now
	DateTime = TS	DateTime from TS	DateTime whenWasIt = new TS(DateTime.Now);

# Appendix C - Licensing

The MARC-HI Everest Framework is licensed under the following terms:

**Apache License**  
**Version 2.0, January 2004**  
<http://www.apache.org/licenses/>

## TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

### 1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not

limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

**2. Grant of Copyright License.** Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

**3. Grant of Patent License.** Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

**4. Redistribution.** You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

- a. You must give any other recipients of the Work or Derivative Works a copy of this License; and
- b. You must cause any modified files to carry prominent notices stating that You changed the files; and
- c. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
- d. If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

**5. Submission of Contributions.** Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

**6. Trademarks.** This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

**7. Disclaimer of Warranty.** Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

**8. Limitation of Liability.** In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

**9. Accepting Warranty or Additional Liability.** While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

## Appendix D – Indices

### Index of Figures

- HL7v3 Architecture.....	6
- Message Layers .....	24
- IntelliSense Help.....	25
- Parameter list filters .....	33
-Documentation of property with choice of values .....	34
- Combining algorithm .....	67
- Collapsing method A.....	68
- Collapsing method B.....	69
- Collapsing method C.....	70

### Index of Examples

- Instantiating CS.....	14
- Concept Qualifiers .....	15
- Using the ST class.....	15
- Loading a PDF into an ED .....	15
- Using integrity checks.....	15
- Reference with thumbnail.....	16
- ED with compressed data .....	16
- Flavors of TS .....	17
- PQ with metric and imperial measures .....	18
- Union of two sets .....	19
- Exception of two sets .....	19
- Find / Find All .....	19
- Intersecting two sets .....	20
- Sub-sequence.....	20
- Interval from Oct 1 2009 to Nov 15 2009.....	20
- Periodic Interval .....	21
- Instantiating the AD data type.....	21
- Formatting an address.....	21
- Instantiating the EN data type .....	23
- Creating transmission wrapper .....	26
- Creating a generic response .....	30
- Creating control act event .....	32
- Setting trigger event in control act event .....	32
- Setting payload in control act event .....	33
- Populating query parameter filters .....	34
- Using the set XXX function .....	34
- Direct assignment to an abstract property.....	35
- Using static utility function to assign abstract property .....	35
- Using a formatter.....	39
- Using formatter result codes .....	39
- Using the ITS 1 settings property.....	40
- Creating the reference .....	41
- Listen connector receiving in blocking mode .....	48
- Listen connector receiving in non-blocking mode.....	48

- Publishing a file (output.xml) to the directory C:\temp .....	50
- Subscribing to the directory C:\temp .....	51
- Sending an instance using WCF .....	54
- Sample app.config for hosted service .....	55
- Invalid response handler .....	58
- Classification rules .....	60
- IMessageReceiver implementation .....	60
- Message classification in IMessageReceiver .....	61
- Conditional property formatting .....	73
- Flavor attribute usage .....	75
- Flavor map usage .....	75
- SendResult implementation .....	79



## ON THE WEB:

- Appendix X – Message Structure Detail Documentation
- MARC-HI Wiki with much more development guidance information

Available at <http://everest.marc-hi.ca>

**CATEGORY:** C#

**COVERS:** HL7v3, MARC|HI Everest Framework

**USER LEVEL:** Intermediate - Advanced