FIXM Developer’s Manual

Executive Summary: The realm of flight control has evolved extremely rapidly in the previous decades from using localized, uncoordinated systems to implementing national, highly integrated systems. Now the expansion of air travel requires another step: trans-national integration of flight control systems. This effort requires many standardization steps; one of the most important first steps is the common definition of the data that constitute a “flight.” FIXM (Flight Information Exchange Model) provides the models that implement this process. When a majority of flight control systems are able to read and write flight messages in a common FIXM format, they will be able to coordinate the handling of air traffic seamlessly.

This report offers advice and direction to data modellers and programmers who need to extend, modify, or maintain the FIXM logical and physical data models.

30 November 2012

Edition: 1.1

Change History

Version	Date	Author	Reason for change
0.1	26th March 2012	M. Tanino (FAA) H. Lepori (Eurocontrol / SESAR)	Reformatted
0.2	17th April 2012	B. Taylor (Lincoln Lab)	Update
0.3	25th May 2012	B. Taylor (Lincoln Lab)	Updated with review comments
0.4	1 June 2012	B. Taylor (Lincoln Lab)	Updated with review comments
0.9	2 July 2012	B. Taylor (Lincoln Lab)	Release Candidate
1.0	16 Aug 2012	B. Taylor (Lincoln Lab)	V1.0 Release
1.1	30 Nov 2012	B. Taylor (Lincoln Lab)	Updated process, best practices for Version 1.1

Table of Contents

Table of Figures

1 Introduction

The development of the FIXM information architecture proceeds in several phases:

- Capture the knowledge of aviation experts in the FIXM Data Dictionary.
- Represent the contents of the FIXM Data Dictionary as a set of high level conceptual UML diagrams (FICM).
- Use both the FIXM Data Dictionary and the FICM model to derive the FIXM Logical Data Model (FIXM).[1]
- Use the FIXM Logical Model to derive the XML Schemas that define the concrete message structure.
- Steps (1) and (2) are best covered by the FIXM Primer document (Reference 1: FIXM Primer (including FICM conceptual model). This document covers steps (3) and (4) in the list above.

1.1 FIXM Logical Structure

The FIXM information architecture is broken into several layers of representation. In general, each layer is an elaboration of the layers beneath it, and no layer refers to information in an outer layer. As Figure 1 illustrates, the FIXM information model is based on several ISO standards, to assist with compatibility with other data standards active in the flight management regime.

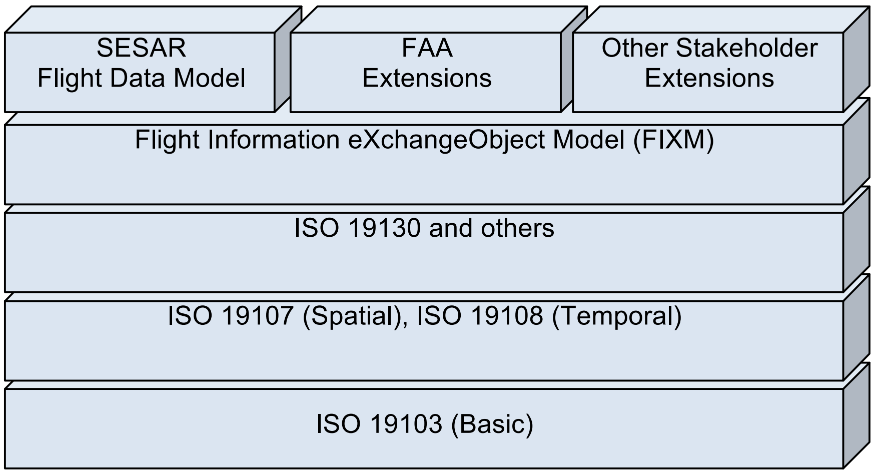
SESAR Flight Data Model
FAA Extensions
Other Stakeholder Extensions
Flight Information eXchangeObject Model (FIXM)
ISO 19130 and others
ISO 19107 (Spatial), ISO 19108 (Temporal)
ISO 19103 (Basic)

Figure 1 - FIXM Layered Information Architecture

1.2 Compatibility with Existing Standards

The FIXM information architecture is designed to be compatible with the following standards:

- Aeronautical Information Exchange Model (AIXM)
- Weather Information Exchange Model (WXXM)

[1] Both FICM and FIXM data models are expressed in Universal Modelling Language (UML) format.

- ISO19107 Geographic Information - Spatial Schema

However, “compatibility” has a particular technical meaning: there must exist a clear algorithm that can convert between FIXM structures and equivalent structures in the other standards without loss of data in either direction. The FIXM data models, but are compatible with both AIXM and WXXM because the most basic level of data representation corresponds to the ISO standards[2].

1.3 A Note on Examples

Many of the sections that follow contain examples of UML data models or XML schema constructs. These examples are chosen to illustrate certain concepts, and do not necessarily represent elements of the XML schemas, nor do they necessarily represent elements planned for later FIXM development.

1.4 FIXM Glossary

A glossary of terms used in FIXM is available at http://www.fixm.aero.

1.5 FIXM Logical Model

A UML model of the FIXM schema structure is available at http://www.fixm.aero

1.6 References

(available at http://www.fixm.aero.)

Reference 1: FIXM Primer (including FICM conceptual model)

Reference 2: FIXM Data Dictionary v1.0

Reference 3 - FIXM XSD Schemas

Reference 4 - Enterprise Architect: www.sparxsystems.com

[2] See **Error! Reference source not found.**: **Error! Reference source not found.** for a list of FIXM abbreviations and terms.

2 FIXM UML Model

The FIXM data architecture is represented in both the logical data model (FIXM) and the XML Schemas (FIXM). The section describes the structure, content, and format of the FIXM logical model. The reader is expected to be minimally familiar with the concepts and notation of UML. It is beyond the scope of this report to provide a UML tutorial, but Section 2.1.1 describes the basic concepts, and the reader is referred to the following for a fuller introduction:
http://www.ibm.com/developerworks/rational/library/769.html

2.1 Schema Package Structure

The FIXM models are arranged in a hierarchy, and it is expected that extension schemas will follow the same pattern. The layers of the hierarchy are:

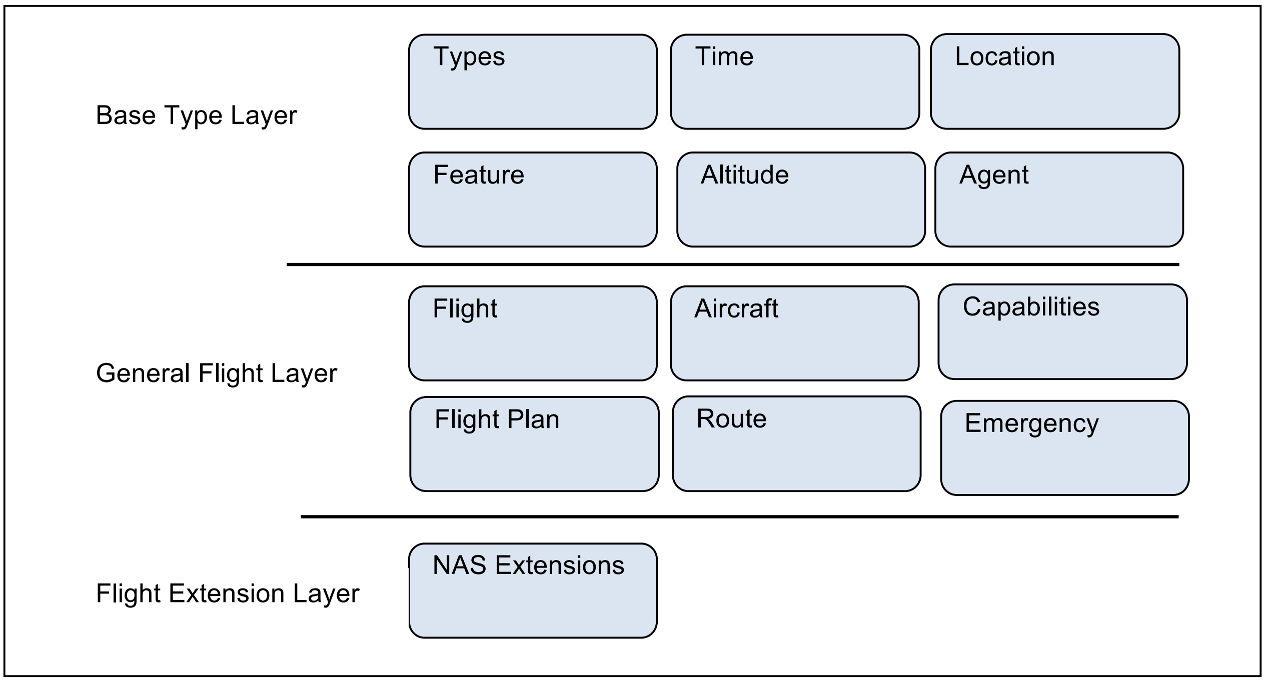

Figure 2 - FIXM Layered Schema Structure

Elements of the General Flight Layer may refer to elements of the Base Type Layer and to each other's elements, and elements of the Flight Extension Layer may refer to elements of the General Flight Layer, the Base Type layer, and other extensions, but references in the other direction are explicitly forbidden.

2.1.1 Main UML Constructs

The FICM Model is expressed using class diagrams of the UML data modelling language: a graphical notation that captures the content and organization of complex data structures. While it is beyond the scope of this report to provide a comprehensive introduction to UML, it is important to introduce the basic concepts and notations. Figure 3 illustrates a typical data model taken from FIXM model.

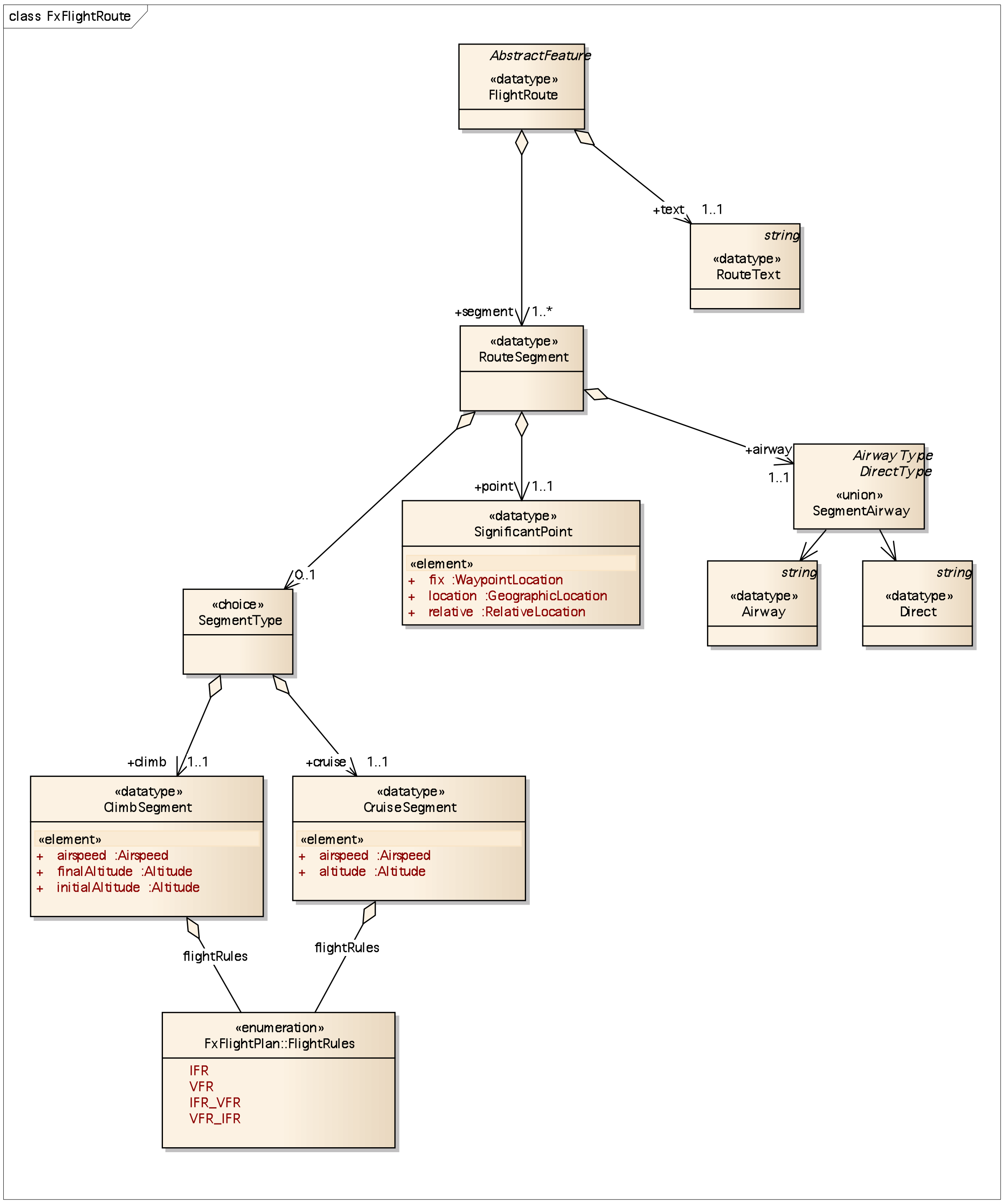

Figure 3 - Sample FIXM UML Class Diagram

2.1.1.1 «choice» (ex: SegmentType)

Choice elements represent exactly one of their contained types, and are often used when two different data types can appear in the same element of a parent type.

2.1.1.2 «union» (ex: SegmentAirway)

Union elements are similar to choice elements in that they can represent any of several element types, but they are usually used for simple types: strings, numerics, etc, where choice elements join more complex structures. However, it is impossible to discover, from an XML message, which of the union alternative types the data represents. If that distinction is important, consider using a choice structure.

2.1.1.3 «enumeration» (ex: FlightRules)

An enumeration is a data item that can only take on a fixed set of legal values. For example, FlightRule can only legally be “VFR”, “IFR”, “IFR/VFR”, “VFR/IFR”. Because some of these enumerations are lengthy, the reader is referred to the FIXM Data Dictionary for the complete list of values.

2.1.1.4 «datatype» (ex: Route)

Datatype elements represent units of information managed by FIXM, and almost always represent items from the FIXM data dictionary. Sometimes a datatype will directly represent a Data Dictionary element, and sometimes it will encapsulate more than one element.

2.1.1.5 Relationships (ex: RouteSegment contains SignificantPoint)

The diamond arrow indicates that RouteSegment contains SignificantPoint, and the multiplicity indicator 0..1 indicates that it optionally contains at most one point. Relationships are tagged with a “multiplicity factor” that indicates how many times they can occur in valid XML.

“1”	occurs exactly once
“0..1”	optionally occurs exactly once
“0...*”	optionally occurs an unlimited number of times
“1..*”	occurs at least once, possibly unlimited number of times

3 FIXM XML Schema (FIXM)

This section describes in detail the structure, content, and construction of the FIXM XSD Schemas (FIXM). The reader is referred to Reference 3 - FIXM XSD Schemas – for availability of the schemas.

3.1 Schema Structure / Organization

The overall FIXM schema is large, so it is divided into several sub-schemas, each related to a specific aspect of the flight information. Despite this division, the schemas are to be considered a unified whole, except as described in Section 4, which describes how to create extension models that can be added to, or subtracted from the FIXM schemas.

The FIXM schemas are arranged in a hierarchy, and it is expected that extension schemas will follow the same pattern. The layers of the hierarchy are:

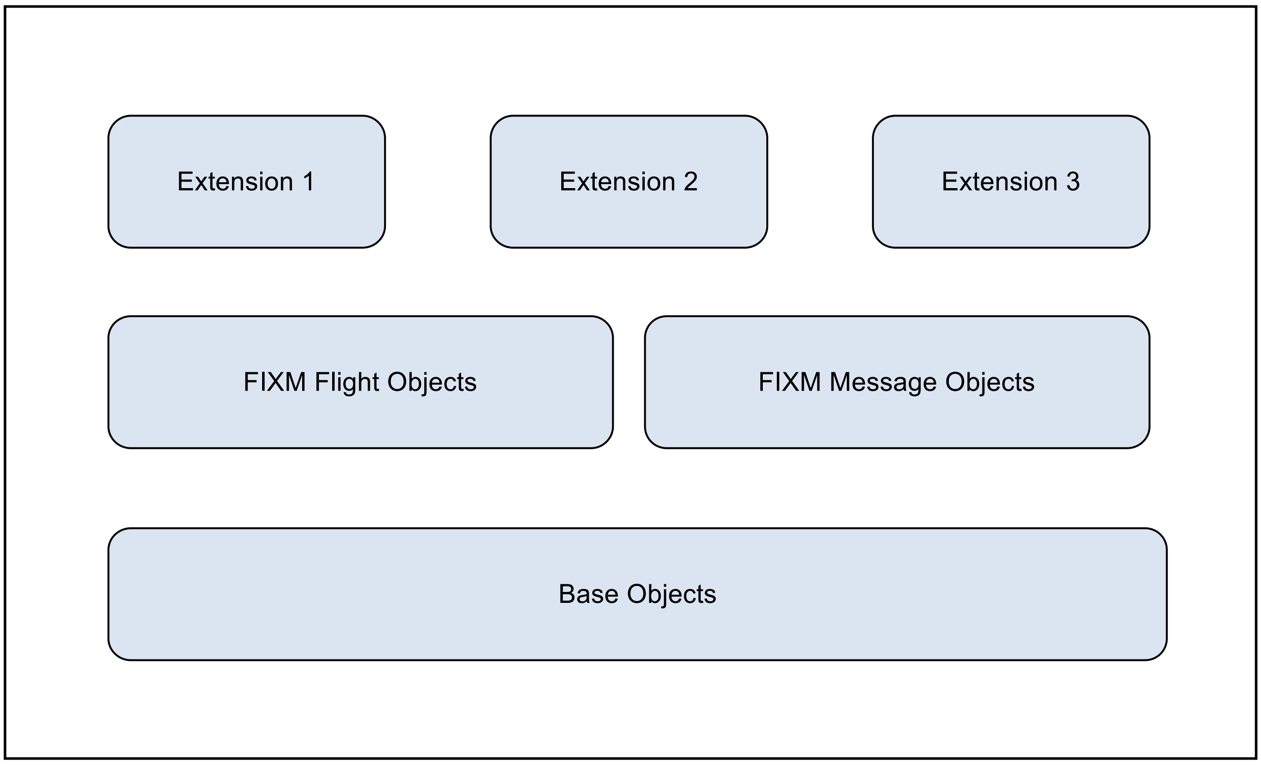

Figure 4 - Hierarchy of FIXM Schemas

The general rule is that a schema may only refer to sibling or children schemas, with no references to upper-level schemas. Thus, for instance, FIXM Schemas refers to the Base Objects, but may not refer either to Extension.

3.2 Namespaces

To keep element names from multiple schemas from interfering within an XSD, they are preceded by a “namespace prefix:”

```
<xsd:element name=”specialFlightType” type=”fx:FlightType”/>
```

Figure 5 - Use of "xsd" name space

In this example, “xsd:” is the prefix of the XML schema language, and “fx:” is the prefix for FIXM schemas. Prefixes are defined in the <schema> statement at the top of the schema, like this:

```
<xsd:schema
   xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:fx="http://www.fixm.aero/fx/1.0"
```

Figure 6 - XSD Name Space Definition

The name spaces defined in FIXM are:

Prefix	URL	Description
xsd	http://www.w3.org/2001/XMLSchema	XML schema objects
base	http://www.fixm.aero/base/1.0	FIXM base objects
fx	http://www.fixm.aero/fx/1.0	FIXM core schemas
msg	http://www.fixm.aero/msg/1.0	FIXM message objects

Table 1 **- FIXM Name Spaces**

3.3 XML Validation

The combined FIXM schemas define precisely what a “well formed XML instance” is. A particular XML instance can be tested for well-formedness by invoking XML validation that compares the prescribed schema structure to the actual XML structure, and reports any discrepancies. Some aspects of validation that are supplied by the XML schemas:

- What element must appear, may appear, or may not appear,
- How many times an element may appear,
- Whether an element can take the “nil” value,
- The allowed data types for the element,
- For numeric elements, the precision, max value, min value,
- For string elements, a pattern that describes allowed strings,
- For string elements, a list of the allowed values (enumerations).

Remember that the content of validated XML may still be semantically incorrect or nonsensical. XML parsers like Xerces support schema validation.

3.4 FIXM Schema Design Principles and Common patterns

The FIXM schemas are written in a particular dialect of XSD that has proven over time to lead to robust, maintainable schemas. A part of that dialect is to use a fixed set of schema patterns: for every kind of problem, solve it with the same structure. This section describes the common FIXM patterns so that the reader can better understand the intent of the schema.

3.4.1 Features and Complex Types

The FIXM schemas provide a special container type called the "AbstractFeatureType" for data that is to be managed as a unit. Many of the complex types inherit from the AbstractFeatureType. The AbstractFeatureType serves as a marker to applications and utilities that the contents of the type are indivisible, and that the object is to be treated as a unit.

```
<xsd:complexType name="ArrivalType">
  <xsd:complexContent>
   <xsd:extension base="base:AbstractFeatureType">
      <xsd:sequence>
         <xsd:element name="arrivalAerodrome" type="fx:AerodromeType"/>
         <xsd:element name="arrivalTimes" type="fx:ArrivalTimesType"/>
         <xsd:element name="arrivalTaxiRoute" type="fx:TaxiRouteType"/>
         <xsd:element name="arrivalTerminal" type="fx:TerminalType"/>
         <xsd:element name="arrivalRunway" type="fx:RunwayNameType"/>
         <xsd:element name="arrivalSpot" type="fx:SpotNameType"/>
         <xsd:element name="arrivalGate" type="fx:GateType"/>
         <xsd:element name="arrivalFix" type="fx:PredefinedFixType"/>
         <xsd:element name="arrivalSpace" type="fx:ParkingSpaceType"/>
         <xsd:element name="arrivalPhase" type="fx:ArrivalPhaseType"/>
         <xsd:element name="alerts" type="fx:FlightAlertType"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
```

Figure 7 - Example Complex Type

3.4.2 Simple Types

Simple types are most commonly used to define a set of enumerations for a value, or to define a string pattern, high and low bounds, or other restrictions on the data. Simple types have the virtue that they produce very compact XML, so they are preferred over complexTypes when the contained data is a single, primitive item.

```
    <xsd:simpleType name="BearingType">
        <xsd:restriction base="xsd:double">
            <xsd:minInclusive value="0.0"/>
            <xsd:maxInclusive value="360.0"/>
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:simpleType name="LatitudeType">
        <xsd:restriction base="xsd:double">
            <xsd:minInclusive value="-90.0"/>
            <xsd:maxInclusive value="90.0"/>
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:simpleType name="LongitudeType">
        <xsd:restriction base="xsd:double">
            <xsd:minInclusive value="0.0"/>
            <xsd:maxInclusive value="360.0"/>
        </xsd:restriction>
    </xsd:simpleType>
```

Figure 8 - Use of Simple Types for Restriction

3.4.3 Inheritance

The structure of the FIXM schema depends heavily on an inheritance tree that propagates types and attributes down from the most general objects to the most concrete. Almost always, the root object of an inheritance tree is an abstract object with a name of the form, "Abstract…" and with the attribute value, "abstract='true'". It is very common for the abstract root object to be used in extension schemas so that any of the extension objects can be substituted in its place.

```
  <xsd:complexType name="AbstractRouteSegmentType" abstract="true">
        <xsd:sequence>
            <xsd:element name="airway" type="fx:SegmentAirwayType"
                         minOccurs="1" maxOccurs="1" nillable="true"/>
            <xsd:element name="point" type="fx:SignificantPointType"
                         minOccurs="1" maxOccurs="1" nillable="true"/>
        </xsd:sequence>
    </xsd:complexType>

   <xsd:complexType name="RouteSegmentType">
        <xsd:annotation>
            <xsd:documentation>
                Represents a concrete "plain" route segment with only
                airway and point elements.
            </xsd:documentation>
        </xsd:annotation>
```

```
      <xsd:complexContent>
         <xsd:extension base="fx:AbstractRouteSegmentType">
            <xsd:sequence>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <xsd:complexType name="CruiseSegmentType">
      <xsd:annotation>
         <xsd:documentation>
            Specifies the altitude, air speed, and flight rules in force
            during a segment.
         </xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="fx:AbstractRouteSegmentType">
            <xsd:sequence>
               <xsd:element name="altitude" type="base:AltitudeType"
                            minOccurs="1" maxOccurs="1" />
               <xsd:element name="airspeed" type="base:AirspeedType"
                            minOccurs="1" maxOccurs="1" />
               <xsd:element name="flightRules" type="fx:FlightRulesType"
                            minOccurs="0" maxOccurs="1" />
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <xsd:complexType name="ClimbSegmentType">
      <xsd:annotation>
         <xsd:documentation>
            Specifies the air speed, and flight rules in force during a
            segment, together with altitude change between entry to the
            segment and exit from the segment.
         </xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="fx:AbstractRouteSegmentType">
            <xsd:sequence>
               <xsd:element name="airspeed" type="base:AirspeedType"
                            minOccurs="1" maxOccurs="1" />
               <xsd:element name="initialAltitude" type="base:AltitudeType"
                            minOccurs="1" maxOccurs="1" />
               <xsd:element name="finalAltitude"
                            type="fx:ClimbSegmentFinalAltitudeType"
                            minOccurs="1" maxOccurs="1" />
               <xsd:element name="flightRules" type="fx:FlightRulesType"
                            minOccurs="0" maxOccurs="1" />
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
```

Figure 9 - Use of Inheritance

3.4.4 Expressing Alternate Data Types

Some situations require that the schema express any of several alternative values. These situations can be met by inheritance, but it is often difficult to construct the correct inheritance tree. The alternative constructs are the `<choice>` selector and the `<union>` value type. The following example shows how to use both of these to achieve a schema element that can take on a set of value types.

```
<xsd:simpleType name="SegmentAirwayType">
   <xsd:annotation>
       <xsd:documentation>
           Airway description: either name or DCT annotation
        </xsd:documentation>
   </xsd:annotation>
   <xsd:union memberTypes="fx:AirwayType fx:DirectType"/>
</xsd:simpleType>

<xsd:complexType name="SignificantPointType">
   <xsd:choice>
      <xsd:element name="location" type="base:GeographicLocationType"/>
      <xsd:element name="fix" type="base:WaypointLocationType"/>
      <xsd:element name="relative" type="base:RelativeLocationType"/>
   </xsd:choice>
</xsd:complexType>
```

Figure 10 - Defining Multi-Type Elements using Union and Choice

3.4.5 Expressing Repeating Structures

Sometimes it is necessary to represent lists of items, and there are several ways of accomplishing this. If the repeated items are complex types (for example, the segments of a route) then the preferred approach is to use a repeating element:

```
    <xsd:complexType name="FlightRouteType">
        <xsd:annotation>
            <xsd:appinfo source="fx:implements">Route</xsd:appinfo>
            <xsd:documentation>
                The Route object represents a route to be flown by the
                airplane as part of the flight. routes can be represented
                as encoded text strings, but they are composed of a
                sequence of RouteSegments, each of which is an airway of
                some sort with a terminus, and may have other information
            </xsd:documentation>
        </xsd:annotation>
        <xsd:complexContent>
            <xsd:extension base="base:AbstractFeatureType">
                <xsd:sequence>
                    <xsd:element name="text" type="fx:RouteTextType"
                          minOccurs="1" maxOccurs="1" nillable="true"/>
                    <xsd:element name="segment"
                          type="fx:AbstractRouteSegmentType"
                          minOccurs="0" maxOccurs="unbounded"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
```

Figure 11 - Use of Repeating Elements

However, when the repeated element is a simple type like an enumeration code, this repetition structure is not very efficient. So, for lists of simple types, the XSD <list> structure is preferred:

```
    <xsd:complexType name="NavigationCapabilitiesType">
        <xsd:annotation>
            <xsd:documentation>
                The overall navigation and approach aid capabilities.
            </xsd:documentation>
        </xsd:annotation>
        <xsd:complexContent>
            <xsd:extension base="base:AbstractFeatureType">
                <xsd:sequence>
                    <xsd:element name="navigationCodes"
                     type="fx:NavigationCodeListType"
                     minOccurs="0" maxOccurs="1"/>

…
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

   <xsd:simpleType name="NavigationCodeListType">
        <xsd:annotation>
        <xsd:documentation>
            Flat list of ICAO Navigation capability codes.
        </xsd:documentation>
        </xsd:annotation>
      <xsd:list itemType="fx:NavigationCodeType"/>
   </xsd:simpleType>
   <xsd:simpleType name="NavigationCodeType">
        <xsd:annotation>
            <xsd:documentation>
                ICAO Navigation capability code.
            </xsd:documentation>
        </xsd:annotation>
        <xsd:restriction base="xsd:string">
                <xsd:enumeration value="N">
                <xsd:annotation>
                    <xsd:documentation>
                        No NAV/approach aid equipment
                    </xsd:documentation>
                </xsd:annotation>
            </xsd:enumeration>

...
        </xsd:restriction>
    </xsd:simpleType>
```

Figure 12 – Use of XSD <list> Construct

3.4.6 Use of Attributes

In a complex element structure, information can be conveyed in two ways. The first, the <element> structure is the most familiar: it represents the data that makes up the type. The second, the <attribute> structure contains meta-information that describes the type itself.

The FIXM schemas use attributes in situations like the following, where the “code” attribute provides the ICAO standard code for the originator’s address.

```
<xsd:complexType name="OriginatorType">
  <xsd:complexContent>
    <xsd:extension base="base:AbstractAgentType">
      <xsd:attribute name="code" type="fx:AftnAddressType"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="AftnAddressType">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[A-Z]{8}"/>
  </xsd:restriction>
</xsd:simpleType>
```

Figure 13 - Use of Attributes to Describe Meta-Data

Attributes can also be used to signal that an element has or does not have a property. In this case, use of a string attribute with a fixed value is preferred to a boolean attribute, so that the interpretation of the attribute is unambiguous, as in the “airfiled” and “unknown” attributes below.

```
    <xsd:complexType name="AerodromeReferenceType">
        <xsd:annotation>
            <xsd:documentation>
                Aerodromes may be identified by:
                * their ICAO codes ("KDFW")
                * their names ("Dallas Fort Worth")
                * their geographic location (latitude and longitude)
                * "Airfiled" designation ("AFIL")
                * "Unknown" designation ("ZZZZ")

                Notice: the attributes shown below are intended as
                alternatives, not additives. For example, it is a mistake
                to provide both the code and the geographic location, or
                airfiled and unknown. If there is a duplication of
                information, the order of precedence is code, name,
                location, airfiled, unknown.
            </xsd:documentation>
        </xsd:annotation>
        <xsd:sequence>
            <xsd:element name="location" type="base:GeographicLocationType"
                         minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
        <xsd:attribute name="code" type="fx:AerodromeCodeType"
                       use="optional"/>
        <xsd:attribute name="name" type="fx:AerodromeNameType"
                       use="optional"/>
        <xsd:attribute name="airfiled" type="xsd:string" fixed="AFIL"
                       use="optional"/>
        <xsd:attribute name="unknown" type="xsd:string" fixed="ZZZZ"
                       use="optional"/>
    </xsd:complexType>
```

Figure 14 - Use of Fixed Value Attributes

3.4.7 Enumerations

Many of the elements in the FIXM schemas can only take a fixed set of values, usually representing an encoding of data. These fields are called “enumerations” and are explicitly represented in FIXM by a simpleType that restricts the primitive string type. Enumeration lists should contain an OTHER value if there is a possibility that input will contain a value not covered by the enumerations: in this case, OTHER is a signal that a value was provided, but could not be mapped to standard enumeration values.

```
<xsd:simpleType name="FlightRulesType">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="IFR"/>
        <xsd:enumeration value="VFR"/>
        <xsd:enumeration value="IFR_VFR"/>
        <xsd:enumeration value="VFR_IFR"/>
        <xsd:enumeration value="OTHER"/>
    </xsd:restriction>
</xsd:simpleType>
```

Figure 15 - Use of Enumerations for Codes

3.4.8 Lexical Patterns

For character data, the FIXM schemas try, whenever possible, to enforce legal lexical rules. The most obvious rules concern minimum and maximum lengths, which are enforced by the “minLength” and “maxLength” attributes. More challenging are the rules that govern text content. For example, consider these flight identifiers:

AAL283	(legal)
BA929	(legal)
EGF9384	(legal)
AAL1	(legal)
ua128	(not legal)

The pattern of these flight identifiers can be described as “at least one and at most four capital letters followed by at least one and at most four digits”, and is captured in the following regular expression[3]:

```
[A-Z]{1,4}[0-9]{1,4}
```

Regular expressions are used extensively in the description of free text fields to constrain their lexical form, as in the following:

```
<xsd:complexType name="AircraftIdentificationType">
 <xsd:complexContent>
   <xsd:extension base="base:AbstractFeatureType">
     <xsd:sequence>
       <xsd:element name="acid" type="fx:AcidType"/>
       <xsd:element name="callSign" type="fx:CallSignType"/>
       <xsd:element name="beaconCode" type="fx:BeaconCodeType"/>
     </xsd:sequence>
   </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="ComputerIdentificationType">
   <xsd:restriction base="xsd:string">
      <xsd:maxLength value="8"/>
      <xsd:pattern value="[A-Z0-9]{1,8}"/>
   </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="AcidType">
   <xsd:restriction base="xsd:string">
      <xsd:maxLength value="7"/>
      <xsd:pattern value="([A-Z][0-9])|([A-Z][A-Z0-9][A-Z0-9]{0,5})"/>
   </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="BeaconCodeType">
   <xsd:restriction base="xsd:string">
      <xsd:maxLength value="4"/>
      <xsd:pattern value="[0-7]{4}"/>
   </xsd:restriction>
</xsd:simpleType>
```

Figure 16 - Use of Lexical Patterns

[3]. See http://en.wikipedia.org/wiki/Regular_expression for further information on regular expressions.

3.4.9 Time Elements

All times in FIXM should be expressed in GMT (also called UTC), with hour offset to local time zone if required. Most of the types in the FIXM schema involve time in one way or another; the FIXM base schemas provide a full set of time types:

- SimpleTimeType
  Date/time to millisecond resolution
- DurationType
  Amount of elapsed time since a reference event
- TimeSpanType
  Period of time with a defined beginning and end time.

3.4.10 Free Text

The FIXM schemas manage to capture most of the data about a flight in well-structured types, but inevitably there are data that must be represented as free text: notes, or names, or addresses, etc. For these situations, FIXM provides the FreeTextType in preference to the XSD string type.

The XSD string type should be used as the base type of enumerations and for fixed attribute values (see Figure 14 - Use of Fixed Value Attributes).

3.4.11 Flight Identifiers

Flight data fusion relies on being able to uniquely identify the flight to which data belong, and this is the role of the Globally Unique Flight Identifier (GUFI), normally found in the flight/gufi element. It is beyond the scope of the FIXM standard to specify how an when GUFIs are created, or assigned to flights, but every flight must be identified by its GUFI before it can be placed in a message and broadcast to other applications. GUFI instances have the following format:

```
region.organization.creation.qualifier
```

The fields of the GUFI are described in the following table.

Field	Description	Examples
Region	Geographic region where flight was created	“us”, “eur”
Organization	Agency that created the flight. Agencies are not limited to air traffic control authorities, but may include airlines, commercial flight plan companies, or even individual pilots.	“ZFW” (center) “UAL” (airline) “cfmu” (FIR) “n1945bl” (tail number)
Creation	Date and zulu time at which the GUFI was created, accurate to second resolution.	20120512T174322Z
Qualifier	Extra characters required to ensure complete uniqueness of the first three fields. Usually an auto-incremented integer.	“00837”

Figure 17 - Definition of GUFI Fields

3.4.12 Embedded Type Definition Discouraged

When a complex type logically contains another type, the XSD language provides two ways of writing the structure. The schema may contain a named type for the contained type that is simply referenced in the containing type, or the contained type definition may appear directly in the containing element. In FIXM schemas, the first approach, specifically declaring the contained type, is preferred, and the second, declaring the type in line, is discouraged. The following figures illustrate the two approaches.

```
<xsd:complexType name="FlightType">
  <xsd:complexContent>
    <xsd:sequence>
      <xsd:element name="gufi" type="fx:GlobalUniqueIdentifierType""/>
      <xsd:element name="acid" type="fx:AircraftIdentifierType" />
      <xsd:element name="flightPlan" type="fx:FlightPlanType" />
      <xsd:element name="flightEvent" type="fx:AbstractFlightEventType" />
      <xsd:element name="extension" type="base:AbstractExtensionType""/>
    </xsd:sequence>
  </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="AircraftIdentifierType">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[A-Z]+[A-Z0-9]*"/>
  </xsd:restriction>
</xsd:simpleType>
```

Figure 18 - Explicit Declaration of Contained Type: Encouraged

```
<xsd:complexType name="FlightType">
  <xsd:complexContent>
    <xsd:sequence>
      <xsd:element name="gufi" type="fx:GlobalUniqueIdentifierType""/>
      <xsd:element name="acid"/>
      <xsd:simpleType>
         <xsd:restriction base="xsd:string">
           <xsd:pattern value="[A-Z]+[A-Z0-9]*"/>
        </xsd:restriction>
       </xsd:simpleType>
      <xsd:element name="flightPlan" type="fx:FlightPlanType" />
      <xsd:element name="flightEvent" type="fx:AbstractFlightEventType" />
      <xsd:element name="extension" type="base:AbstractExtensionType""/>
    </xsd:sequence>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 19 - In Line Declaration of Contained Type: Discouraged

3.4.13 Provenance

In some situations, it is important to record where an item of data came from:

- Automated system like TFDM or ERAM
- Human like an ATC or pilot

For this situation, FIXM provides an attribute group named "ProvenanceAttr" that contains two attributes:

- System
  The name of the system that produced the data
- Center
  The ATC center (Flight Information Region) in which the data was produced
- Source
  The source of the data in the message (unstructured text)

To indicate that a type should be tagged with its provenance, add the "ProvenanceAttr" group to its definition. Note that the base type AbstractFeatureType is already tagged with ProvenanceAttr, so if your type extends from AbstractFeatureType specifying this would be redundant. And, since AbstractFeatureType is used for any type that groups elements that should be treated as a unit, there will always be a place to record provenance: on the element itself or on its containing feature.

```
<xsd:complexType name="RouteType">
  <xsd:complexContent>
      <xsd:extension base="base:AbstractFeatureType">
          <xsd:sequence>
             <xsd:element name="text" type="fx:RouteTextType"/>
             <xsd:element name="segment" type="fx:RouteSegmentType"/>
             <xsd:element name="externalRemarks" type="base:FreeTextType"/>
          </xsd:sequence>
        <xsd:attributeGroup ref="base:ProvenanceAttr"/>
      </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 20 - Defining the Provenance of Data

3.5 FIXM XMS Schema Naming Conventions

As the FIXM schema grows and other, related, schemas accrete around it, all of the schemas should bear a family resemblance so that programmers and subject matter experts can easily read and understand the schemas. Part of this standardization is to define a strategy for naming the components of a schema. The rules in this chapter are not meant to limit the schema developer's options, but to standardize the purely mechanical naming conventions so that he or she can concentrate on the more important semantic content.

3.5.1 Limitations

The rules in this chapter pertain only to names of the element, attributes, and other structures of a schema, not to their semantic content. Those topics are also important, but are covered by other chapters of this document. There will always be special situations in which these rules will increase the schema complexity, rather than decrease it as desired. In these situations, relax these rules only as much as needed to achieve a compact, understandable schema. These exceptional situations should be very rare, so if you find many of these situations, review your schema development practice and tools, because it is easy to misunderstand the usage of these rules.

3.5.2 InterCap Naming

The "InterCap" convention is sometimes called "camel case" and is the practice of separating words of a multi-part name by capitalizing the initial character of each word.

- The initial character of the first word appears in lower case for element and attribute names "`aerodromeLocation`.
- The initial character of the first word appears in upper case for type names: “FlightPlanType”.
- When abbreviations and acronyms appear within a name their case must conform to InterCap: "`departureEdct`", not "`departureEDCT`".

3.5.3 Functional Naming

Components of the schema should be named to be suggestive of their function, so that a new reader is likely to understand the purpose of the field "`controllerRunwayAssignment`" not "`controllerAssignment`".

3.5.4 Abbreviations

In general, full words are preferred to abbreviations within names: "`preferredDepartureFix`", not "`prefDepFix`".

- Exception: when the term is a well-recognized industry term and the expanded form would be more confusing than the abbreviation: "`EDCT`" not "`estimatedDepartureClearanceTime`", "`ETD`", not "`estimatedTimeOfDeparture`".
- Exception: when fully expanding the words leads to a name so long it is likely to be more confusing than the abbreviation: not "`AirportConfigApprConditionType`", not "`AirportConfigurationApproachConditionType`".

3.5.5 Schema Names

FIXM Schema names are recorded in the “version” attribute of the <schema> element, followed by the version number of the schema. By convention, the name of the schema begins with the name space prefix assigned to it: in this case, “fx”.

Schemas should be recorded in files with names of the form:

```
        <schema name>.xsd
```

Every schema file begins with the standard XSD header <schema>, and this header should have a “version” attribute containing the name of the schema and its version. This attribute is used by various automation tools to track schema evolution and to measure schema coverage.

```
<xsd:schema
   xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:fx="http://www.fixm.aero/fx/1.0"
   xmlns:base="http://www.fixm.aero/base/1.0"
   targetNamespace="http://www.fixm.aero/fx/1.0"
   elementFormDefault="qualified"
   attributeFormDefault="qualified"
   version="fxFlightPlan:1.0">
```

Figure 21 - Recording Schema Name and Version

3.5.6 Type Names

Simple and complex types that appear at the top level of the schema are referred to simply as "types,"

- Type names appear in InterCap notation with initial capital letter.
- Type names end in the word "`Type`".
- Abstract types (those with attribute `abstract="true"`) begin with the word "`Abstract`".

Examples: "RouteSegmentType", "AircraftIdentificationType", "AbstractMessageType".

3.5.7 Element and Attribute Names

Elements and attributes are the lowest level components of a schema, and a single item of data, or a structured piece of data defined in a type.

- Element and attribute names appear in InterCap notation with lower case initial letter.
- Element names always take the singular form, even if the maximum occurrence is unbounded.

3.5.8 Enumeration Names

When simple types are used to represent enumeration values, the enumerations should follow the following rules:

- Characters are restricted to upper-case A through Z, digits 0-9, and underscore.
- Underscores are used to separate words in the string
- Abbreviations are encouraged if they keep the words to manageable length

```
<xsd:simpleType name="FlightRulesTypeType">
   <xsd:restriction base="xsd:string">
      <xsd:enumeration value="IFR"/>
      <xsd:enumeration value="VFR"/>
      <xsd:enumeration value="IFR_VFR"/>
      <xsd:enumeration value="VFR_IFR"/>
   </xsd:restriction>
</xsd:simpleType>
```

Figure 22 - Naming Enumeration Values

4 FIXM Schema Evolution

4.1 General Strategy

It is intended that the FIXM schemas be updated at regular intervals, both to address errors and omissions, and to extend their content and coverage. Initial plans are to release major versions approximately yearly, and minor versions as needed during the year. In a situation like this, application developers have a difficult choice: to “freeze” to a given version of the schema and risk obsolescence, or to try continually to keep their applications current with the most recent schema release. While FIXM does not have a general solution to this problem, it does take some steps to make it more manageable:

- Maintaining the mapping between schema elements and the Data Dictionary elements they implement
- Providing “fair warning” when an element will be withdrawn from the schema or significantly modified.
- Providing a window of backward compatibility when an element is withdrawn or modified.

4.2 Deprecating Elements

When it is decided that an element or type should be removed from the FIXM schemas, or significantly modified, that element will be “deprecated” for one minor release before it is removed or modified. Deprecation is a signal to application developers that they need to modify their applications if they wish to use the next version of the FIXM schemas. An element is deprecated by adding the following processing instruction to its scope:

```
<?deprecated version=”…” change=”…” reason=”…” >
```

Where:

- “Version” is the identifier of the version in which the element will be removed or modified.
- “Change” indicates the proposed change to the elements: REMOVE or MODIFY
- “Reason” briefly describes the reason for the impending change

The following example illustrates a proposal to remove an enumeration value from the enumeration FlightTypeType:

```
   <xsd:simpleType name="FlightTypeType">
      <xsd:annotation>
         <xsd:documentation>
            List of possible flight types.
         </xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:string">
         <xsd:enumeration value="MILITARY" />
         <xsd:enumeration value="GENERAL" />
```

```
          <xsd:enumeration value="NON_SCHEDULED" />
          <xsd:enumeration value="SCHEDULED" />
          <xsd:enumeration value="TRAINING">
             <?deprecated version=”2.0” change=”REMOVE”
               reason=”Subsumed under MILITARY code”>
          </xsd:enumeration>
          <xsd:enumeration value="OTHER" />
       </xsd:restriction>
    </xsd:simpleType>
```

Figure 23 – Deprecated Enumeration Value

4.3 Backward Compatibility

New releases of FIXM schemas are guaranteed to be backward compatible only with their immediate prececessors. The following changes may be made without notice or without deprecation, since they will not break applications using the schemas:

- Adding new schemas to the schema set
- Adding new types to a schema
- Adding new elements and attributes to existing types
- Making an existing element optional (ie, minOccurs=”0”)
- Making an existing attribute optional (ie, use=”optional”)
- Adding enumeration values
- Changing string patterns to less restrictive syntax
- Changing value ranges to be less restrictive

Note: Backward compatibility is available from FIXM v2.0 onward. There is no assurance of compatiblity with FIXM 1.x.

5 FIXM Extension Model

The FIXM models and schemas are constructed to support a “core and extension” model of development. The “core” schemas are developed and maintained only by the FIXM data modeling group, currently at MIT Lincoln Laboratory, who are solely responsible for their content and structure. But the FIXM model also supports extension models and schemas that may be developed by the data modeling group or by outside agencies, and used in conjunction with the core. This section discusses the role of extensions, the life cycle of an extension, and relation of extensions to the core and to each other.

5.1 FIXM Core

The FIXM core model and schemas contain the ICAO information about flights: the information that one would expect every flight to share, regardless of its airline, nationality, owner, or any other characteristic. Examples of this universal data:

- Flight identity (GUFI, aircraft Identifier, etc)
- Flight operator (airline, private, government, etc)
- Flight aircraft characteristics
- Flight plan and flight plan updates
- Flight status and location
- Flight delays and cancellations
- Flight emergencies.

Certainly, not every flight expressed in FIXM notation will have all of these attributes: only delayed flights will have delay information, only flights in trouble will have emergency information. But this information is *potentially* applicable to all types of flights, without exception.

The FIXM core started as a small nucleus: simply flight identification and the ICAO 2012 flight plan information; but more flight information is being added, and will be added, to the core as understanding and requirements of flight management increase. The decision of whether a given datum belongs in the core is often complex and may involve consultation with experts or developing prototypes, or other ways of evaluation, but the final decision of whether to include a datum in the core rests with the FIXM data modeling group..

5.2 FIXM Extensions

In contrast to the core data some information applies selectively to some flights but not others. This class of information is modeled in FIXM “extension” models and schemas. There is no physical difference between core and extension schemas, but extensions may be included or excluded from an application, depending on the data that it requires or provides. Some examples of extension data:

- ASDE-X surveillance information is useful only to applications that deal with airport airspace and surface movement, and not to applications that manage flight plans or track flights.

- Surface movement information (runway assignment, taxi routing, etc.) is relevant only to applications that predict and improve airport operations
- NAS-specific information is relevant only to flights operating in the United States, and not to flights operating in Europe.
- Vendor-specific information that supports a particular application or system. [4]
- Data under consideration for inclusion into the FIXM core.

In all of these cases, information may be added to or subtracted from the flight's record based on its characteristics, where it is operating, the characteristics of the airport, or other attributes that may vary from one flight to the next. All of these situations call for the data and schemas to be developed as FIXM extensions.

5.3 Merging Extensions into the FIXM Core

Occasionally, it will be advantageous to merge the model and schemas contained in an extension into the FIXM core and to treat it as universal flight data. Some of these situations:

- The extension represents an experimental model put forth by the FIXM modeling group (see above) that has passed its evaluation.
- The model in the extension is recognized to be so universally applicable (or almost universally) that it should be moved to the core models.
- The model in the extension enriches existing models and schemas in the core, and the enrichments should become universal.

In these cases, the FIXM data modeling group, after consultation with the FIXM sponsors and other FIXM participants, will physically transfer the extension's conceptual and logical models, and the schemas, into the FIXM core models and schemas. In this process, the models and schemas may be adapted to better fit with the structure of the core, and this adaptation may affect some existing applications, so the merging is an activity to be carefully planned and rolled out.

5.4 Removing Elements from the FIXM Core

From time to time, it will be necessary to remove data elements from the FIXM core, either because they are no longer in use, or because they are superseded by later additions. Since some applications will depend on the elements, deletions will happen in a three-step process:

1. Publish intent to delete and solicit comments
2. Mark the element "deprecated" for one cycle of the schemas to give applications time to adjust.
3. Delete the item in the next release.

[4] In this case, there is no requirement that the information be expressed as an extension but, if the vendor so chooses, it becomes subject to the same rules and processes as other extensions.

5.5 Characteristics of Extensions

The size and content of extensions is expected to vary widely from simple value extensions to complete new sub-models, but all well-formed extensions share these characteristics:

- The extension information is represented as a conceptual model, a logical model, and one or more XSD schemas.
- Whenever an extension datum embodies a type expressed in the core, that type is re-used in the extension.
- When an extension datum is a variation of a type expressed in the core, the extension type inherits from the core type, or otherwise extends it, rather than defining a completely new type.
- The extension models and schemas strictly follow the FIXM "best practice" rules described in the FIXM Developer's Guide.
- The extension documentation contains any required addenda to the FIXM documentation required to understand and use the extension.
- The extension has its own XML namespace to prevent name collision with other extensions or the core. If the extension is intended for eventual inclusion in the core, the name space URL is an extension of the FIXM base URL.

5.6 Modeling the Extension Data using UML

Building any FIXM extension should begin with modelling the data in UML. The FIXM schemas were developed using Enterprise Architect 9.0.908 from Sparx Systems (Reference 4). If you are using that tool or an equivalent tool, you can import the FIXM UML model, and use the existing model structures to build your extension model.

In any case, follow the explicit UML rules described in Section 2: FIXM UML Model so that the extension model blends well with the existing FIXM models. See Reference 1: FIXM Primer (including FICM conceptual model) for an introduction to the FIXM models. Try to avoid:

- Duplicating data types
- Referring to data in other extensions
- Defining data types that redefine existing data structures.

5.7 Implementing the Extension Schemas

FIXM Extensions are ultimately expressed in one or more XSD files. This section describes the rules for creating schemas that work well with the rest of the FIXM schemas.

5.7.1 Follow the FIXM Schema Conventions

Tne extension XSD file should follow all the explicit and implicit conventions described in Section 3: FIXM XML Schema. To summarize:

- Follow the FIXM SCHEMAS naming conventions
- Choose a namespace URL for your schemas and make them available through a catalog at that URL.
- Divide your logical schema into physical schemas of manageable size, each of which addresses a logically distinct regime.

5.7.2 Using the FIXM Base Types

The FIXM base objects (namespace “base:”) are carefully crafted to support flight data at the most basic level. Study them carefully and use them whenever possible, because your schemas will interoperate with existing FIXM flight schemas if they share basic types.

- Most of your complex types should extend the base:AbstractFeatureType if the contained information should always be treated as an atomic unit.
- If you need to express a 2D location, you will almost certainly find a suitable type in the baseLocation schema.
- If you need to represent counts or quantities of something, search the types in the baseTypes schema for the appropriate measurement type, and use it. If you need to derive your own measurement type, follow the pattern of the FIXM base quantities.
- If your object contains free text field (that is, a string that does not adhere to a specified format) it should use the FreeTextType defined in the baseTypes schema, either directly or by extending it.
- If you need to specify a time instant, a time span, a time duration, or anything else having to do with time, consult the FIXM baseTime schema for a suitable type.
- If you need to create your own time type, try to extend an existing FIXM base type. If there is no suitable FIXM base type to extend, follow the pattern set by the FIXM time types and base your type on the xsd:dateTime basic XSD type.
- If you have a situation that truly requires a new basic data type, contact the FIXM support team to arrange for it to be integrated with the FIXM base objects.

5.7.3 Using the FIXM Schema Types

Similarly, the more you can re-use the FIXM schema types, the better your extension will integrate with the rest of the FIXM schemas, messages, and applications. It is impossible to know what you will need from the FIXM objects, but here are some important schemas to read:

- All the base schemas
  The base schemas contain the most primitive data types in FIXM.
- FxFlight
  The root of all information known about a flight. One way or another, your extension information should be accessible from this object.

- FxFlightPlan
  All information about the times and routes taken by the flight at various stages from planning to flight.
- FxAircraft
  Anything that is known about the actual airframe is reachable from this type. If your extensions concern the airframe itself, you should probably extend this type.
- fxAerodrome
  "Aerodrome" is the designation of an installation capable of handling flight arrivals and departures. If your extension needs to mention an airport by code or name, you should be using AerodromeReference or another object in this schema.

5.7.4 Follow the FIXM Explicit and Implicit Conventions

Before starting a FIXM extension, read the explicit FIXM conventions in the following chapters and adhere to them. But just as important, read the FIXM base and flight schemas and look at their "style." The more closely you can follow that style, the more easily your extension will interoperate with the FIXM flight schemas.

5.7.5 Extend the AbstractExtension Object

This is probably the easiest of the methods to implement. In the baseExtension schema is a type, AbstractExtensionType, derived from AbstractFeatureType. This type is provided specifically to support extension schemas, because the FIXM Flight type contains an unbounded list names "extension" of AbstractExtensionType objects (see Figure 24). This means that, if your root types extend from AbstractExtensionType, you can add them to the flight's "extensions" list and they will be carried along with the flight, to be marshaled into XML and unmarshaled out of XML.

```
<xsd:complexType name="FlightType">
  <xsd:complexContent>
     <xsd:extension base="base:AbstractFeatureType">
        <xsd:sequence>
          <xsd:element name="gufi" type="fx:GufiType" />
                   . . .
          <xsd:element name="extension"
                       type="base:AbstractExtensionType"
                       maxOccurs="unbounded"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
```

Figure 24 - The FlightType Extension List

5.7.6 Extend One or More FIXM Types

If your extension adds information to a FIXM flight type (for example, adding a new transponder code) then you can create a new object that extends the appropriate FIXM object and use it wherever in the FIXM flight types that its parent is used.

```
<xsd:element name="ExtendedIdentification"
             substitutionGroup="fx:AircraftIdentification"
             type="fx:ExtendedtIdentificationType">
</xsd:element>

<xsd:complexType name="ExtendedIdentificationType">
   <xsd:complexContent>
      <xsd:extension base="fx:AircraftIdentificationType">
         <xsd:sequence>
            <xsd:element name="modeC" type="ModeCType"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="ModeCType">
   <xsd:restriction base="xsd:string">
      <xsd:maxLength value="6"/>
      <xsd:pattern value="F[0-9A-F]{5}"/>
   </xsd:restriction>
</xsd:simpleType>
```

Figure 25 - Extending FIXM Objects

5.8 FIXM XSD Extension Techniques

The XML Schema Language provides many mechanisms useful for extending and modifying schemas, but only the techniques described in this section are sanctioned for FIXM extension schemas.

5.8.1 Adding a New Extension Name Space

Each extension should be given its own name space, declared in the "targetNameSpace" attribute of the <schema> element. FIXM extension name spaces are always of the form:

```
http://www.fixm.aero/fixm/ext/<extension name>
```

It is customary, but not required, to limit extension names to five characters or fewer. The following figure illustrates the definition and use of an extension namespace for the NAS extension.

```
<xsd:schema
   xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:fx="http://www.fixm.aero/fx/1.0"
   xmlns:base="http://www.fixm.aero/base/1.0"
   xmlns:nas="http://www.fixm.aero/ext/nas/1.0"
   targetNamespace="http://www.fixm.aero/ext/nas/1.0"
   elementFormDefault="qualified"
   attributeFormDefault="qualified"
   version="nas:1.0">
```

Figure 26 - Adding Extension Name Space

5.8.2 Adding New Extension Types

To define a type known only within the scope of this extension, simply declare it in the normal way using <complexType> or <simpleType>. Since the new type does not modify any core elements, no special techniques are required.

5.8.3 Obscuring Core Types

It may become advisable to “hide” a type defined in the core schemas from referencing within the extension schemas. There is no reliable way to achieve this, and it is not sanctioned as an extension technique.

5.8.4 Adding New Elements to Types

To add extra data elements to a complex type defined in the core, use XSD extension to define a new, derived type within the extension schema. Applications that use the extension can refer either to the base elements defined in the core, or to the new elements in the derived type:

```
<xsd:complexType name="NasSegmentType">
  <xsd:complexContent>
    <xsd:extension base="fx:SegmentType">
       <xsd:sequence>
        <xsd:element name="plannedDelay" type="base:DurationType"/>
        </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 27 - Adding an Element to a Core Type

5.8.5 Obscuring Elements from Types

There is no reliable mechanism for making core elements non-referenceable from the extension schema, and this is not a sanctioned extension technique.

5.8.6 Changing Data Type of Elements

The data type of core elements may be changed in the extension, but only if the new data type is an extension of the original data type. This is accomplished by introducing an extension type as shown in section 5.8.4, and and renaming the element in the extension type. In the following example, the “altitude” element replaces the corresponding element in the core SegmentFlightInfoType, but redefines it to be a NAS altitude format.

```
    <xsd:complexType name="NasSegmentFlightInfoType">
        <xsd:complexContent>
            <xsd:extension base="fx:SegmentFlightInfoType">
                <xsd:sequence>
                  <xsd:element name="altitude" type="nas:NasAltitudeType"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
```

Figure 28 - Changing Element Data Type

5.8.7 Changing Cardinality of Elements

There is no reliable mechanism to change the “minOccurs” or “maxOccurs” or “nillable” attributes of a core element, and this is not a sanctioned extension technique

5.8.8 Changing the Pattern of a String Type

To redefine the allowed pattern for a core SimpleType, extend the core type into an extension SimpleType and provide the new pattern in the “pattern” attribute, then use the extension type in place of the core type:

```
    <xsd:simpleType name="NasRouteTextType">
        <xsd:restriction base="fx:RouteTextType">
            <xsd:pattern value="[A-Z0-9\. \+/]+"/>
        </xsd:restriction>
    </xsd:simpleType>
```

Figure 29 - Changing a String Type Pattern

5.8.9 Adding Enumeration Values from a String Type

To extend a core enumeration string type by adding new enumeration values, create an extension type containing the new values and use the XSD <union> element to form a new type containing both the core and the extension enumeration values. In the following example, the NasAltitudeExtensionType contains the NAS-specific altitude type enumerations, and the NasAltitudeTypeType contains the combined enumerations for the core and the NAS extension.

```
<xsd:simpleType name="NasAltitudeExtensionType">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="ABOVE"/>
        <xsd:enumeration value="BLOCK"/>
        <xsd:enumeration value="VFR"/>
        <xsd:enumeration value="VFR_PLUS"/>
        <xsd:enumeration value="VFR_ON_TOP"/>
        <xsd:enumeration value="VFR_ON_TOP_PLUS"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="NasAltitudeTypeType">
    <xsd:union memberTypes="base:AltitudeTypeType
                            nas:NasAltitudeExtensionType"/>
</xsd:simpleType>
```

Figure 30 - Extending an Enumeration Type

5.8.10 Removing Enumeration Values from a String Type

There is no reliable mechanism for obscuring an enumeration value defined in the core, and this is not a sanctioned extension technique.

5.8.11 A Word About the XSD <redefine> Element

The XSD language provides an element type called <redefine> that is intended to modify many attributes of inherited types, including cardinality, nillable status, enumerations, min and max values. So, the <redefine> element could be used to implement the restrictions described in sections 5.8.5, 5.8.6, 5.8.7, 5.8.10. However, the <redefine> element is notriously difficult to use correctly, and produces unreliable results in many run time XML systems, including XML Beans and JAXB.

For these reasons, the <redefine> element is not a sanctioned extension technique for FIXM.

5.9 Simple Extension Example

The following figure illustrates a small extension schema that unites the examples shown in the preceding sections. Specifically, it illustrates

- Separate name space (“nas”)
- Importing “base” and “fx” schemas for reference
- Use of extension to create new types local to the extension
- Use of extension to add new elements or redefine elements

```
 <xsd:schema
   xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:fx="http://www.fixm.aero/fx/1.0"
   xmlns:base="http://www.fixm.aero/base/1.0"
   xmlns:nas="http://www.fixm.aero/ext/nas/1.0"
   targetNamespace="http://www.fixm.aero/ext/nas/1.0"
   elementFormDefault="qualified"
   attributeFormDefault="qualified"
   version="0.9">

    <xsd:import namespace=”http://www.fixm.aero/base/1.0”
          schemaLocation="../../base/base.xsd"/>
    <xsd:import namespace="http://www.fixm.aero/fx/1.0"
          schemaLocation="../../fx/fx.xsd"/>
    <xsd:include schemaLocation="./nasAltitudes.xsd"/>

    <xsd:complexType name="NasRouteType">
        <xsd:complexContent>
            <xsd:extension base="fx:RouteType">
                <xsd:sequence>
                    <xsd:element name="text" type="nas:NasRouteTextType"/>
                </xsd:sequence>
                <xsd:attribute name="format" type="fx:RouteFormatType"
                     use="required" fixed="NAS"/>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

    <xsd:simpleType name="NasRouteTextType">
        <xsd:restriction base="fx:RouteTextType">
            <xsd:pattern value="[A-Z0-9\. \+/]+"/>
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:complexType name="NasSegmentFlightInfoType">
        <xsd:complexContent>
            <xsd:extension base="fx:SegmentFlightInfoType">
                <xsd:sequence>
                  <xsd:element name="altitude" type="nas:NasAltitudeType"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>

</xsd:schema>
```

Figure 31 - Example FIXM ExtensionSample

6 Application Development with FIXM

6.1 Intended Audience

The information in this section is useful to anyone who needs to implement software utilizing the FIXM schemas and is therefore geared toward software developers. In particular, it will be useful to those writing applications that need to create FIXM flight data, read and interpret FIXM flight data, or modify FIXM flight data. The discussions below assume that the reader is familiar to some degree with:

- General XML notation and conventions
- XML Schema Definition (XSD) notation and conventions
- Object-oriented programming experience
- Standard Java coding practices[5]
- Apache XMLBeans XML binding **or**
- DOM XML management packages **or**
- SAX XML parsing packages **or**
- JAXB XML binding

This guide presents a number of ways to interact with the FIXM schemas programmatically, but it is not intended to be a comprehensive software cookbook. The reader should absorb the logic behind the examples and apply that to his or her own situation.

6.2 Examples of FIXM Usage

The developer has a number of options for programmatically interacting with the FIXM schemas. We will detail three separate approaches:

- DOM parsers
  DOM (Document Object Model) systems such as DOM4J or Apache Xerxes 2 implement an in-memory representation of an XML document, and provide translation between the XML text representation and the memory model. They also provide methods for creating and modifying the in-memory document components.
- Apache XMLBeans Object Binding
  Schema binding mechanisms such as Apache XMLBeans use the XSD schemas to generate Java access objects that facilitate access to the XML structures through normal Java operations. Every FIXM release will contain a JAR file with the XMLBeans bindings of the most recent schemas.
- JAXB Object Binding
  JAXB is an XML object library based on Enterprise Java standard objects,

[5] Although the examples shown in this document are written in Java, equivalent concepts and software exist for C and C++.

rather than Apache objects. It is neither better nor worse than XMLBeans, just an alternative. Every FIXM release will contain a JAR file with the JAXB bindings of the most recent schemas.

The examples in the following sections are brief, but complete, Java programs that all accomplish the same task: extracting the GUFI field from a FIXM Flight object. They are available, along with all required libraries and test data at http://www.fixm.aero.

6.3 DOM Parsers

DOM parsers implement a convention for interacting with objects in XML documents. They operate on the document as a whole and allow arbitrary navigation and modification of the XML tree. We will discuss interacting with FIXM via Apache's DOM4J parser library, Xerces2, specifically.

Information on how to use the Xerces2 DOM parser for reading and manipulating schemas can be found on the Xerces2 DOM FAQ page.

```
package aero.fixm.examples;

import java.io.File;
import java.io.IOException;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

public class DomExample {

public static void main(String[] args) {
      DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
      DocumentBuilder db;
      File example = new File("example.xml");
      try {
            db = dbf.newDocumentBuilder();
            Document doc = db.parse( example );
            NodeList nl = doc.getElementsByTagName("fx:Gufi");
            if ( nl != null && nl.getLength() > 0 ) {
                  Node gufiNode = nl.item( 0 );
                  System.out.println(gufiNode.getTextContent());
            }
            else {
                  System.out.println("No GUFI:");
            }
      } catch (Exception e) {
            e.printStackTrace(System.err);
      }
   }
}
```

6.4 XML Schema Bindings (Apache XML Beans)

XML schema bindings take a different approach to XML access than both DOM and SAX by providing mappings between XML and object-oriented classes. The bindings software uses the XML schema to compile interfaces and classes that can be used to read and manipulate XML data using both getters and setters.

```
package aero.fixm.examples;



import java.io.File;
import aero.fixm.xmlbeans.fx.FlightDocument;
import aero.fixm.xmlbeans.fx.FlightType;

public class XmlBeansExample {

  public String getGUFI( File fpXml ) {

  String gufi;
    try {
      FlightDocumentType fd = FlightDocument.Factory.parse( System.in );
      FlightType fp = fd.getFlight();
      gufi = fp.getGufi();
    } catch (Exception e ) {
      System.err.println( e.getMessage() );
    }

    return gufi;
  }
}
```

6.5 XML Schema Bindings (JAXB)

JAXB is the standard Java/XML binding technology supplied with the JavaX package set.

```
package aero.fixm.examples;

import java.io.File;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Unmarshaller;

import aero.fixm.jaxb.fx.FlightType;

public class JaxbExample {

  private static final String JAXB_CONTEXT_FX = "aero.fixm.jaxb.fx";

  public String getGUFI( File flightXml ) {
    JAXBContext jc;
    try {
      jc = JAXBContext.newInstance( JAXB_CONTEXT_FX );
      Unmarshaller u = jc.createUnmarshaller();
      FlightType fp = (FlightType) u.unmarshal( flightXml );

      return fp.getGufi();
    } catch ( JAXBException e ) {
      System.err.println( e.getMessage() );

    }
    return null;
  }
}
```