# Specification for XMF Meta File Format

**Version 1.00a**

**Revised 8/15/2001**

**Published by:**
The MIDI Manufacturers Association
Los Angeles, CA

## PREFACE

XMF stands for **eXtensible Music Format**.

XMF is a low-overhead meta-file format for bundling collections of data resources (i.e. file images) in one or more formats into a single file. XMF brings stability, structure, ease of handling, transportability, optional data compression, optional data security, and a consistent meta-data framework to resource collections.

Separate Recommended Practice documents define the rules for various Types of XMF files, each intended to address a particular set of purposes. For example, RP 031 defines Type 0 and Type 1 XMF Files, which allow for the bundling of Standard MIDI File (SMF) and Downloadable Sounds (DLS) file images. Additional XMF File Types can be added via the Recommended Practice process at any time, without requiring any changes in the underlying XMF meta-file format.



Printed _____

MMA
PO Box 3173
La Habra CA 90632-3173

# Table of Contents




P.O. Box 3173 • La Habra CA 90632-3173 • Tel (714) 736-9774 • Fax (714) 736-9775 • email mma@midi.org

# 0. Definition of Terms

**extended ASCII** - A text character encoding standard based on the American Standard Code for Information Interchange, as used in other MMA specifications.

**DTD** - Document Type Definition. In XML, a document in XML format which defines a set of XML elements for use in other XML documents.

**GUID** - Globally Unique Identifier, a 16-byte integer uniquely identifying an item. See Chapter 10, "DEC/HP Network Computing Architecture Remote Procedure Call RunTime Extensions Specification Version OSF TX1.0.11", Steven Miller, July 23 1992. (Same definition used in DLS <DLSID> chunk.)

**parser** - Program or device for reading a data format.

**player** - Program or device for playing XMF files.

**RDF** - Resource Description Framework, an XML-based metadata format standard defined by the World Wide Web Consortium (w3c). See http://www.w3c.org/RDF.

**resource** - Block of formatted data corresponding to a file. For example: SMF, DLS, WAV, text, JPG, or manufacturer's non-standard format.

**RMID** - RIFF MIDI file, originally defined in Microsoft(R) Windows(R) Multimedia Programmers Reference, Microsoft Press, 1991, p. 8-31 (now out of print).

**Unicode** - A text character encoding standard that includes all major world scripts in a simple and consistent manner. See http://www.unicode.org.

**URI** - Uniform Resource Indicator, the formal term for a family of names/addresses that refer to resources, in the form of short text strings. 'URL' is an informal term, and is a special case of URI that includes more popular URI schemes such as http:, ftp:, mailto:, etc. See http://www.w3c.org/Addressing.

**VLQ** - Variable Length Quantity, a binary number format with a variable number of bytes. See section 4.1.

**XString** - eXtensible String, a text string format with a leading length integer in VLQ form and no terminating character. See section 4.2.


P.O. Box 3173 • La Habra CA 90632-3173 • Tel (714) 736-9774 • Fax (714) 736-9775 • email mma@midi.org

# 1. Introduction

XMF stands for **eXtensible Music Format**.

XMF is a low-overhead meta-file format for bundling collections of data resources (i.e. file images) in one or more formats into a single file. XMF brings stability, structure, ease of handling, transportability, optional data compression, optional data security, and a consistent meta-data framework to resource collections.

Separate MMA Recommended Practice documents define the rules for various Types of XMF files, each intended to address a particular set of purposes. For example, a separate Recommended Practice document defines Type 0 and Type 1 XMF Files, which allow for the bundling of Standard MIDI File (SMF) and Downloadable Sounds (DLS) file images. Additional XMF File Types can be added via the MMA Recommended Practice process at any time, without requiring any changes in the underlying XMF meta-file format.

**Bundling –**XMF is primarily intended to bundle existing standard music and sound file formats – such as SMF, DLS, and WAV – and not to replace any of them. It is assumed that XMF implementors will have access to existing playback APIs or devices able to render the formats contained in an XMF collection, and will pass the contained resources off to them for rendering, in accordance with Recommended Practice guidelines. An XMF collection may be either a flat list of resources, or hierarchically structured to any depth.

**XMF File Types –** Like XML, the XMF meta-file format is a generalized structure that can be used for many purposes, but is not a directly usable format. Only specific XMF File Types, as defined in separate MMA Recommended Practice documents, can be rendered. In XML terms, an XMF File Type specification would be analogous to a DTD, defining how the container will be used for one purpose or application area.

**Extensibility –** XMF is eXtensible by the content creator and application developer, in four ways:

- Custom resource types can be stored in any XMF file
- Custom meta-data fields are supported, and can be displayed to the end user
- Custom data compression algorithms can be used
- Custom security algorithms can be used

However, the use of custom resource types, compression algorithms, or security algorithms will in most cases diminish file portability. XMF readers are required to ignore all unrecognized resource types, **UnpackerID**s, and **MetaDataItems**, and each XMF File Type Recommended Practice specifies player capabilities and behaviors, so there is no danger of misinterpretation.

**Hierarchically Structured Collections –** XMF uses a hierarchical containment paradigm, expressed in a new low-overhead tree data structure (not RIFF) with simple and consistent parsing rules.

**Scaling Data Structure –** Minimized file size is essential if XMF is to be usable on all playback platforms and networks, including small mobile devices. To that end, the container data structure is scalable. Lightweight and breakaway data structures minimize overhead, and smaller resource collections need fewer bytes to wrap. To conserve space and transmission bandwidth, chunk boundaries are not constrained to word, double-word, or other processor-specific boundaries. Every data item is either of known size or is length-delimited, so it's never necessary to parse through the internals of any item just to move ahead to the next item. There is no upper limit on file size, resource size, or number of resources.

**A Tree of Nodes –** The container hierarchy is expressed as a tree of nodes, starting at the top of the file with a single root node. In the simplest case the **RootNode** holds one single resource – for example, one SMF file. More often the **RootNode** will be a folder containing further nodes – resource nodes and/or further folder nodes, nested to arbitrary depth. In some situations literally hierarchical resource layout will be desirable, and in other situations the hierarchical tree is best used as a small index holding pointers to large resource blocks appearing elsewhere. Both options are available.

**External Resource References –** To facilitate both the sharing of resources among multiple XMF collections and the dynamic publishing of resources on the Internet, any node in the tree can optionally reference a resource in another file, or at an Internet http: address (URI), rather than a data block inside the XMF file.

**Meta-Data –** Each node in the tree (both files and folders) may optionally have any number of independent meta-data items. Each meta-data item can be either a Standard item or a Custom (i.e. application-specific or end-user-defined) item, and the item's contents can be of any length or data type including binary data. Any meta-data item may include multiple content variations, keyed to the playback device's preferred language and country. Meta-data is implemented as a field in the **NodeHeader**; when no meta-data is needed, this field collapses to exactly 1 byte per resource or folder.

**Unpackers –** Each node in the tree (resource or folder) may optionally be individually secured, data-compressed, or otherwise processed with any number of encoders. Each required decoding operation is indicated by a Unpacker selector and the size of the decoded result (for use in memory/disk allocation). Each node can have an independent list of **UnpackerIDs** indicating the sequence of decoding operations required to recover 'clear' playable data (for example: decryption, followed by data decompression, followed by watermark check). These lists appear directly as a field in the **NodeHeader**; when no decoding is needed, the list collapses to exactly 1 byte per resource or folder.

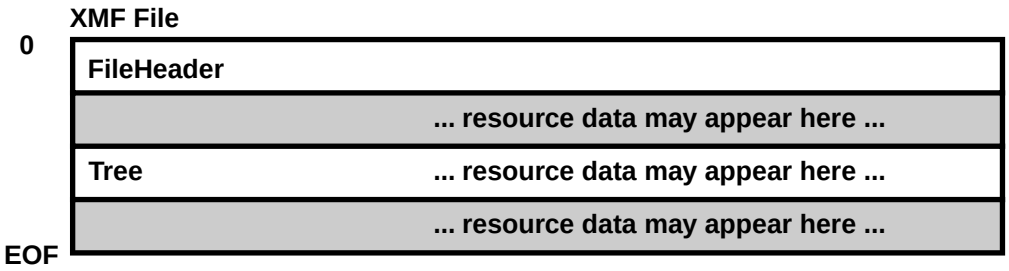
# 2. XMF File Structure

An XMF file begins with exactly one **FileHeader**, and includes exactly one **Tree**.

The **FileHeader** contains the minimum information needed to get a **Tree** parser started (see **2.1. FileHeader Structure**).

The **Tree** describes the resources (SMF file images, DLS file images, WAV file images, etc.) in the collection, expresses the collection's structure, and indicates where to find the resources. Resources may appear either inside the XMF file, or in other files (see **2.2. Tree Structure**).

Resources that appear inside the XMF file may be located either in the space between the **FileHeader** and the **Tree**, within the **Tree**, or in the space between the **Tree** and the end of the file.





P.O. Box 3173 • La Habra CA 90632-3173 • Tel (714) 736-9774 • Fax (714) 736-9775 • email mma@midi.org

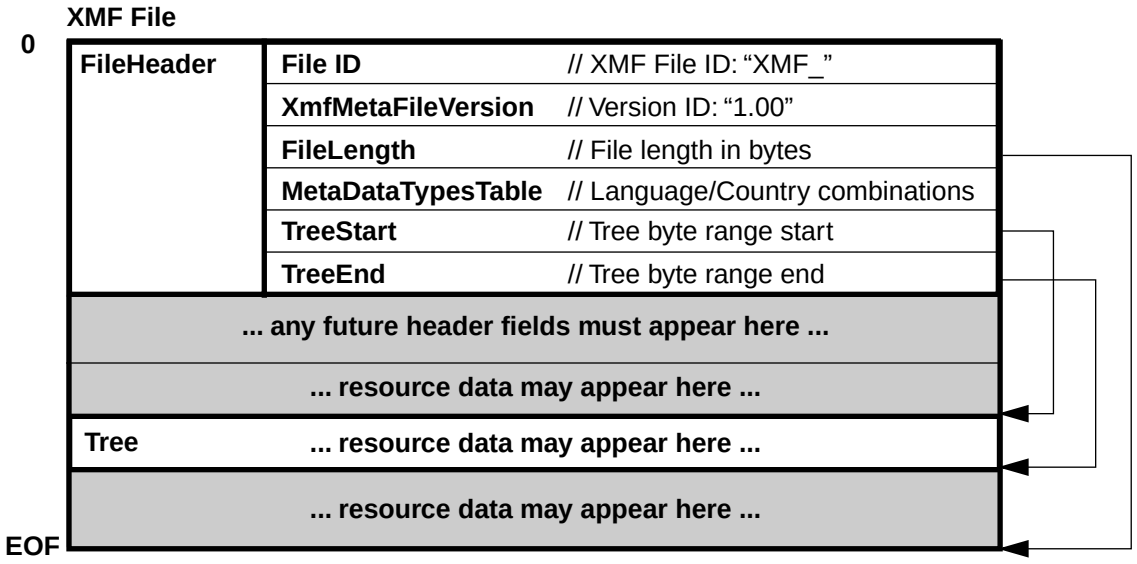
# 2.1. FileHeader Structure

Fields in the **FileHeader** indicate the meta-file spec version, the file's total length, the location of the **Tree**'s start and end, and contain global meta-data format tables.

- The **FileID** field identifies the file as an XMF file. **FileID** is always the 4-byte ASCII sequence: **`XMF_`**
- **XmfMetaFileVersion** is the version of the **XMF Meta-File Format Specification** (this document) to which the file conforms, expressed as a 4-byte ASCII sequence with a decimal point in the second position. This is version 1.00 of the specification, so the byte sequence should be: **`1.00`**
- **FileLength** is the total length of the XMF file, in bytes, in **VLQ** format (see section 4.1). Note that the value contained in this field must reflect the length of this field.

  **Note for XMF file creators:** The value for this field will depend in part on the size of the VLQ used here. Because a change in value may force a change in VLQ size, determining the value and VLQ size may require iteration.
- **MetaDataTypesTable** contains shared information on the languages and countries for which **NodeMetaData** contents are provided, as detailed in the **3. XMF Meta-Data** section of this document.
- **TreeStart** is the offset from the start of the XMF file to the first byte of the **Tree**, in bytes, in **VLQ** format (see section 4.1).
- **TreeEnd** is the offset from the start of the XMF file to the last byte of the **Tree**, in bytes, in **VLQ** format (see section 4.1).

  **Note:** The **FileLength**, **TreeStart**, and **TreeEnd** fields should **not** be used to create space between the **FileHeader** and **Tree**, or between the **Tree** and the end of the file, for storing hidden data. XMF parsers will not find that data, so XMF tools will strip that data when editing an XMF file, and XMF players will not be able to access that data.



## 2.2. Tree Structure

The **Tree** describes the resources in the collection (SMF file images, DLS file images, WAV file images, etc.), expresses the collection's hierarchical structure, and indicates where to find the resources. Resources may be stored directly within the tree, elsewhere in the same XMF file, or in external files, as detailed below under **Reference Types**.

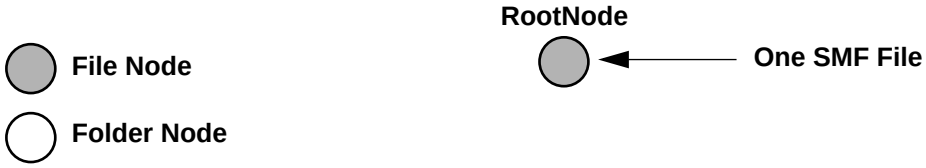
The **Tree** is built using two kinds of **Node**:

- **FolderNodes** – each of which can contain one or more additional nodes, including further **FolderNodes**, and
- **FileNodes** – each of which describes exactly one resource (SMF, DLS, WAV, etc.) and indicates where to find it.

The **Tree** always begins with one logical root, a node called **RootNode**.

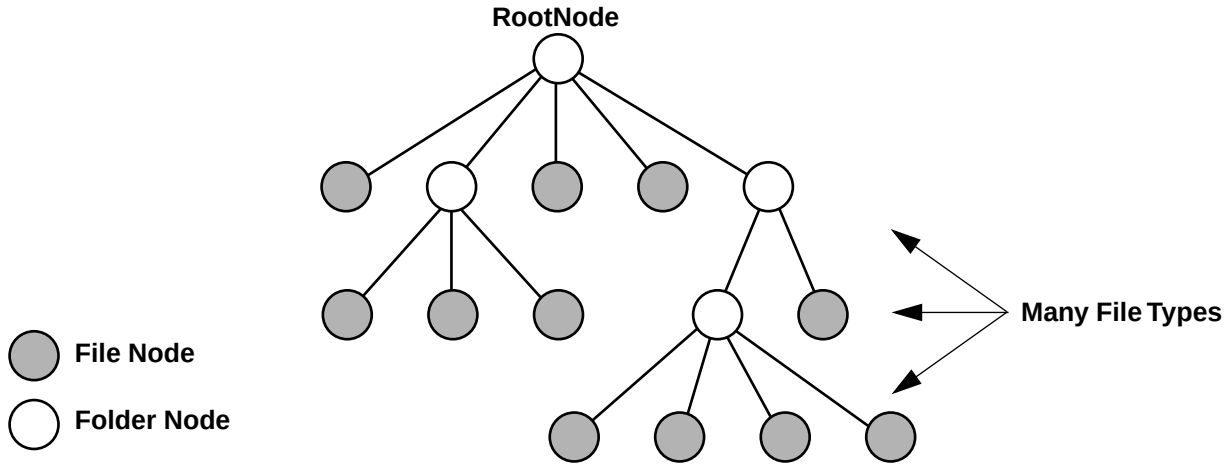
In simple XMF files, the **RootNode** will be a **FileNode** holding exactly one resource.

**Simplest Tree Hierarchy: Single Node**




More often it'll be a **FolderNode** holding several further **FileNode**s, and perhaps additional **FolderNode**s (which may be hierarchically nested to any depth).

**Typical Tree Hierarchy: Many Nodes**

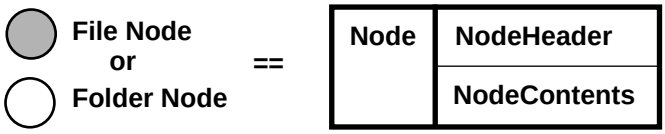


## 2.2.1. Node Structure

Every **Node** in the **Tree** consists of two parts:

- **NodeHeader** and
- **NodeContents**.

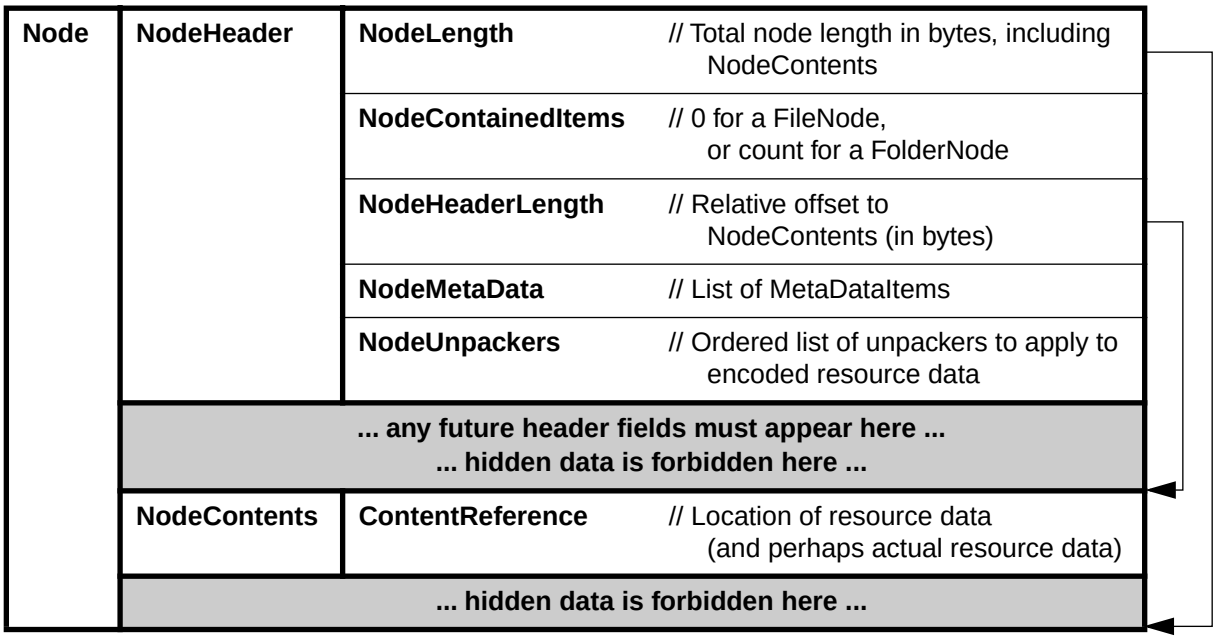
This is true for both **FileNode**s and **FolderNode**s, including the **RootNode**.



## 2.2.1.1. NodeHeader Structure

The **NodeHeader** fields implement the **Tree** structure, and hold information about the resource data blocks that the **Node** represents.

> **Note:** Unlike many resource file formats, in most cases XMF does not require **Nodes** to have names or resource ID numbers. Where needed or desired, this can be achieved via standard meta-data fields.




- **NodeLength** allows the parser to move ahead to the next **Node** at the same level of the **Tree**, including skipping over any **Node** it doesn't recognize or doesn't care about. **VLQ** format (see section 4.1). Note that the value contained in this field must reflect the length of this field.

  > **Note for XMF file creators:** The value for this field will depend in part on the size of the VLQ used here. Because a change in value may force a change in VLQ size, determining the value and VLQ size may require iteration.

  > **Note:** This field should not be used to create space to store hidden data in the gap between the **NodeHeader** and the **NodeContents**. XMF parsers will not find that data, so XMF tools will strip that data when editing an XMF file, and XMF players will not be able to access that data.

- **NodeContainedItems** indicates whether the **Node** is a **FileNode** (resource) or **FolderNode** (container for further **Nodes**). For **FileNode**s, **NodeContainedItems** is 0. For **FolderNodes**, **NodeContainedItems** is the number of directly contained **Nodes** (non-recursive count, i.e. not counting any **Nodes** contained in child **FolderNodes**). **VLQ** format (see section 4.1). Since 0 is used to indicate a **FileNode**, empty **FolderNodes** are not allowed in XMF.

- **NodeHeaderLength** is the relative offset from the start of the **NodeHeader** to the start of the **NodeContents**. This is a hedge to preserve backwards compatibility in future, as it will allow older parsers to skip over any new **NodeHeader** fields that may be added to later versions of the XMF meta-file format specification. **VLQ** format (see section 4.1).

  > **Note:** As a result, all future versions of XMF will have to support the whole initial field set (**NodeLength**, **NodeContainedItems**, **NodeHeaderLength**, **NodeMetaData**, and **NodeUnpackers**), and any additions will have to appear after **NodeUnpackers**.

  > **Note:** This field should **not** be used to create space to store hidden data in the gap between the **NodeHeader** and the **NodeContents**. XMF parsers will not find that data, so XMF tools will strip that data when editing an XMF file, and XMF players will not be able to access that data.

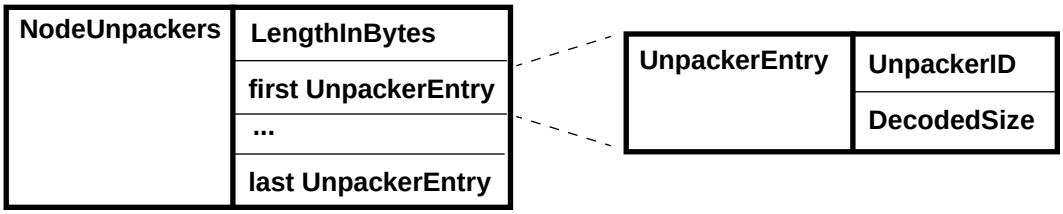
- **NodeMetaData** contains zero or more **MetaDataItems** pertaining to the resource data referenced in the **NodeContents**, as detailed in section **3. XMF Meta-Data**.
- **NodeUnpackers** is an ordered, length-delimited list specifying a sequence of zero or more decoding operations that must be applied to the resource data referenced in the **NodeContents** in order to recover clear, usable (playable) data. Unpackers will typically be used for data compression and intellectual property protection. See section 2.2.1.1.1. for details.

### 2.2.1.1.1. NodeUnpackers Structure

**NodeUnpackers** is an ordered, length-delimited list specifying a sequence of zero or more decoding operations that must be applied to the resource data referenced in the **NodeContents** in order to recover clear, usable (i.e. playable) data. Unpackers will typically be used for data compression and intellectual property protection. **LengthInBytes** is a **VLQ.**

Each decoding operation is expressed as one **UnpackerEntry**, consisting of an **UnpackerID** (see section 5.1) and the size in bytes of the resulting decoded data block. This tells the XMF parser what size buffer to allocate for the decoding operation's output buffer. For format, interpretation, and registration of **UnpackerID**s, see section 5.1. **DecodedSize** is a **VLQ** (see section 4.1).

If **DecodedSize** is 0, determination of the decode buffer size is left to the player. This is intended to support differences in player capabilities, including variable bitrate codecs and variable levels of quality.

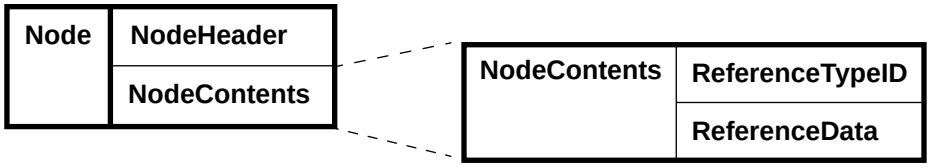



Data compression which is native to the resource file format (such as IMA 4:1 in WAV files) should not be reflected in **NodeUnpackers**, as that processing is handled outside of the XMF parser's scope, in the type-specific playback APIs and devices.

> **Note: FolderNodes** appearing in the **Tree** must not use any unpackers that would render the folder's contained **Nodes** illegible. This rule arises because otherwise we'd need to invent a way to describe how the **Tree** continues on the inside of that 'black box'. If you wish to 'black box' encode a folder, however, it is possible to store it outside the **Tree**, as a detached **Node** (see section 2.2.1.2.1), and point the **FolderNode**'s **ContentReference** at the detached **Node**.

## 2.2.1.2. NodeContents Structure

The **NodeContents** part of a **Node** consists is a structure indicating where to find the described resource. The resource may appear directly within the **NodeContents** structure, elsewhere within the current file, as an external file, or as a resource in another XMF file's collection, as described below.




Every **NodeContents** structure consists of two parts:

- A **ReferenceTypeID** in **VLQ** form indicating an access mechanism. Any **Node** may use any **ReferenceTypeID**; this is true for both **FileNode**s and **FolderNode**.
- A **ReferenceData** field with the data necessary to support that **ReferenceTypeID**. The format of the **ReferenceData** depends on the value of the **ReferenceTypeID**, as section 2.2.1.2.1. shows.

### 2.2.1.2.1. Reference Types

Each **ReferenceTypeID** represents one unique reference type, or way of finding the resource data that the **Node** describes. **ReferenceTypeID**s may only be defined and assigned by the MMA.

Note that the same collection of resources can be stored in many different ways, depending on which reference types are used for each **Node** – even though the **Tree**'s logical structure remains the same for all such variations – so decisions about **ReferenceType** usage will dramatically affect the overall layout of an XMF file.

See also: **Appendix: Notes on the XMF Meta-File Format**

| ReferenceTypeID | | ReferenceData | |
|---|---|---|---|
| **ID** | **Description** | **Description** | **Format** |
| **1** | **In-Line Resource** | The resource data block | (the actual encoded resource data block) |
| **2** | **In-File Resource** | Offset in bytes from the start of the XMF file to the resource data block | **VLQ** |
| **3** | **In-File Node** | Offset in bytes from the start of the XMF file to a Node whose **ContentReference** leads to the resource data block | **VLQ** |
| **4** | **External Resource File** | URI* of an external resource file** (SMF, WAV, DLS, etc.), e.g.: **file://myOtherFile.wav** | **XString** |
| **5** | **External XMF Resource** | URI* of an external XMF file**, including specific resource name, e.g.: **file://myOtherXmfFile.xmf#myResource** | **XString** |

* The URI stored in the ReferenceData must be coded according to RFC 2396 (US-ASCII with restricted character set). To access characters which cannot be represented in this coding, use the %HH convention to represent the byte values; coding standards for non-European characters in URIs are defined at http://www.w3.org/TR/charmod/.

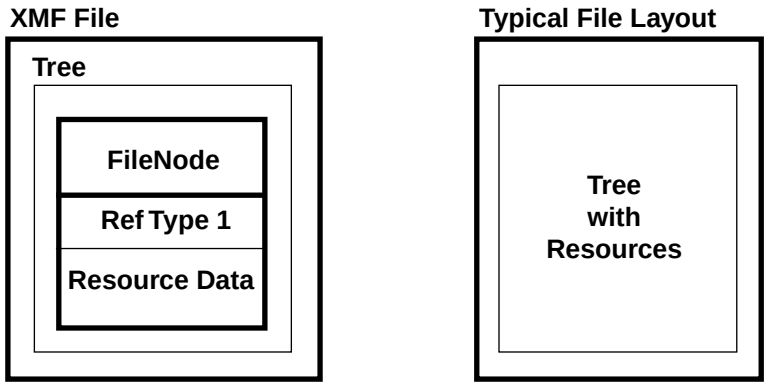
** If the XMF player is displaying a copyright notice for the XMF file, and detects that the external file has a different copyright notice, then the player must display both copyright notices. If the XMF player detects that the external file contains digital rights management (DRM) information, then the XMF player must not use the external file in any way that would violate the DRM usage restrictions.

Note: This RP does not specify the use of any particular DRM technology. Future RPs may define DRM systems for XMF, perhaps using the Unpackers mechanism. Developers interested in preserving interoperability are advised to check the status of this work before introducing their own DRM technologies into XMF.

Each **ReferenceTypeID** is detailed below.

### ReferenceTypeID 1: In-Line Resource

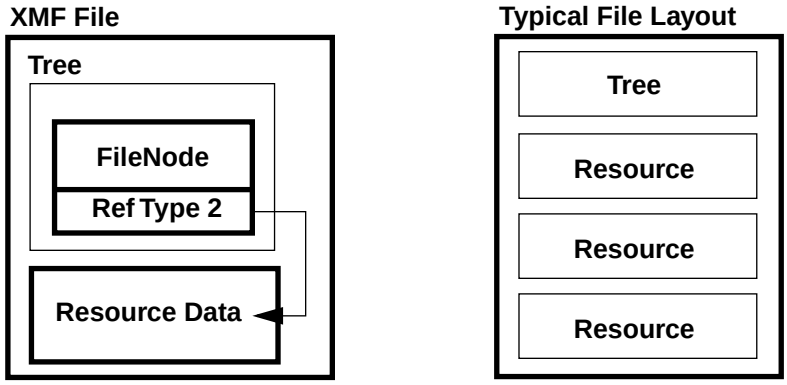
**ReferenceTypeID 1** means the resource data block appears in-line, immediately after the **ReferenceTypeID**, and encoded per the **NodeUnpackers**. In other words, the resource data is stored directly in the **Tree**. Type 1 may be appropriate for small and/or simple files, but is not recommended for large or complicated files as it works against the intended use of the **Tree** as an access accelerator. **FolderNode**s should generally use **ReferenceTypeID** 1, as should XMF files containing only one resource.




**Typical Use Cases:** Simple XMF files, or when an author wishes to pack (encrypt, data-compress, etc.) an entire folder of resources in a single operation.

### ReferenceTypeID 2: In-File Resource

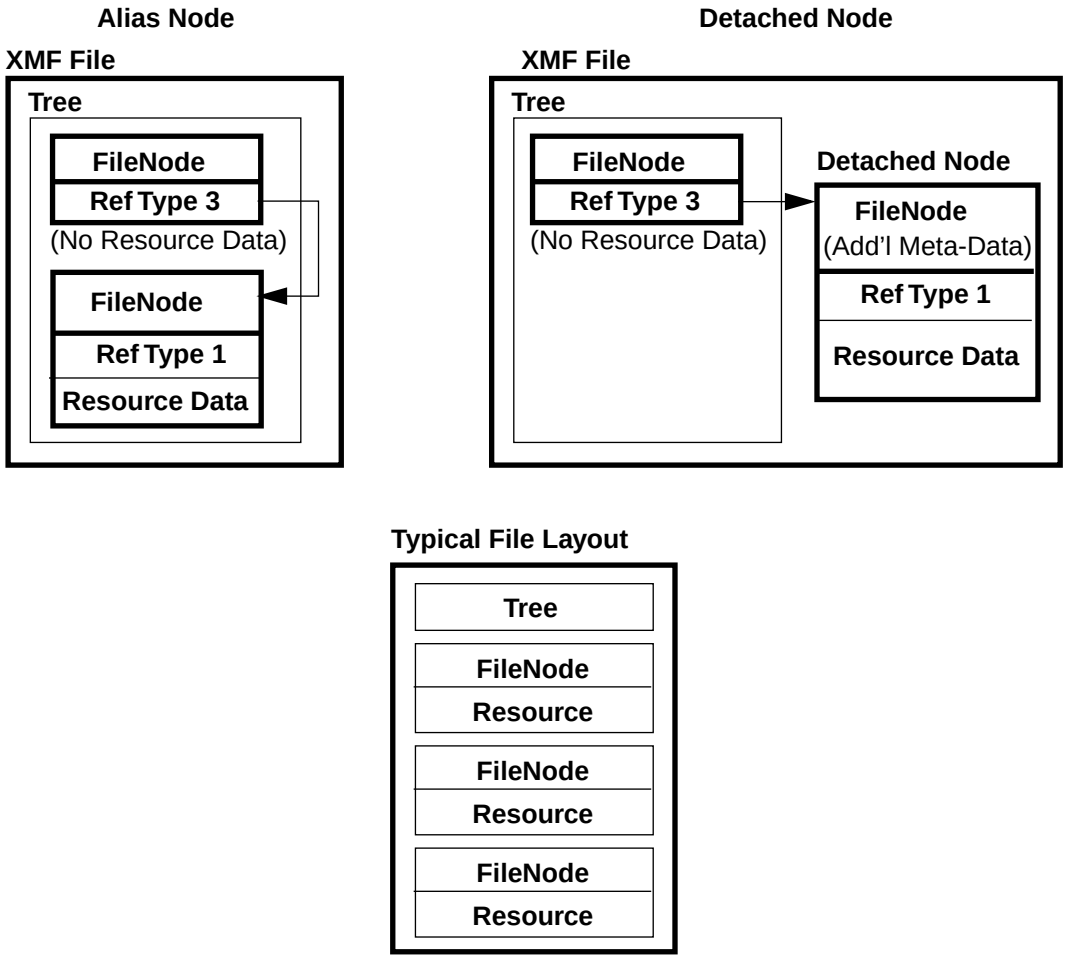
**ReferenceTypeID 2** points to a resource data block appearing elsewhere in the same XMF file, and encoded per **NodeUnpackers**. Typically the target block will be outside the **Tree**; however this can also be used for an alias to a resource appearing in another **Node** in the **Tree**. **ReferenceTypeID** 2 is recommended when the number of resources or the complexity of the **Tree** is high, as it keeps the bulky resources out of the **Tree**, making it easier to load the entire **Tree** in one seek operation.




**Typical Use Cases:** Large resource collections in which in-line resource placement would make Tree navigation cumbersome. **ReferenceTypeID** 2 may also be simpler for XMF generator programs to assemble.

### ReferenceTypeID 3: In-File Node

**ReferenceTypeID 3** points to a further **Node** within the same file. If the target **Node** is also in the main **Tree**, this becomes an alias to another resource. More typically, the target **Node** will be detached from the **Tree**, in order to store additional meta-data in the detached node, without clogging the main **Tree** – again, with the aim of optimizing the **Tree**.




**Typical Use Cases:** Resources needing large amounts of meta-data, or situations in which content developers wish to use a single resource for multiple purposes (such as stand-ins, temporary placeholders, or multiple artists sharing one resource – such as DLS – in their work).

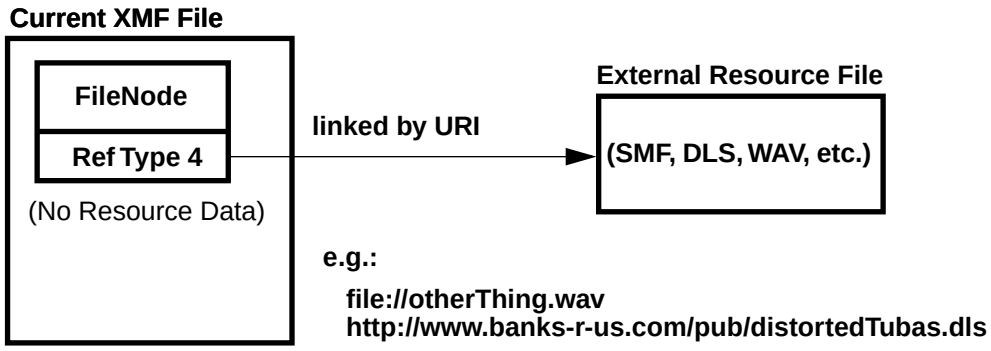
> **Note:** In **Node**s using **ReferenceTypeID** 3, the **NodeUnpacker**s in the referenced **Node** override any **NodeUnpacker**s in the referring node. Any **NodeUnpacker**s indicated in the referring node should be ignored.
>
> **Note:** The target node will not necessarily hold the target resource directly, as it has a **ContentReference** of its own. However, to avoid excessively long or circular seek chains, the target resource data must be found within 4 reference indirections, otherwise the XMF parser must fail and return a 'Too many reference indirections' error.

### ReferenceTypeID 4: External Resource File

**ReferenceTypeID 4** points to external whole resource files (such as SMF, WAV, or DLS files), using the W3C's Uniform Resource Identifier (URI) format. (URI is the formal superset of URL.) URI Schemes **file:** and **http:** should be supported if the playback platform is capable of them; additional schemes may be supported in a future version of this specification. If the external file is encoded with XMF encoders, **NodeUnpackers** should reflect that fact. **ReferenceTypeID** 4 can allow for very small XMF files, can allow external files to be shared by multiple XMF files, can allow rapidly changing resources to be decoupled from the rest of the XMF collection, and can layer uniform meta-data onto resources that don't support it directly.

The format for URIs is defined and described at the following URLs:

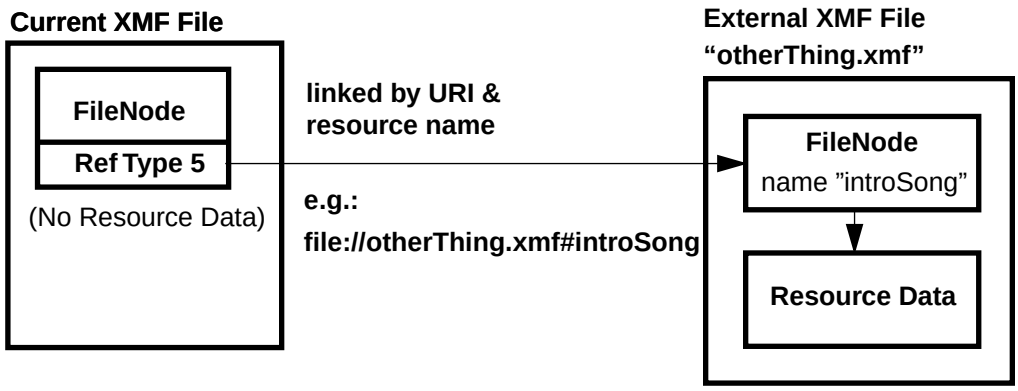
**http://www.ietf.org/rfc/rfc2396.txt**
**http://www.w3.org/Addressing/**




**Typical Use Cases:** This reference type has all the same resource sharing and aliasing benefits as **ReferenceTypeID** 3, with the additional flexibility of allowing the target resource to live outside of the XMF file. This creates many new possibilities because the target file can be updated after the XMF file is distributed. For example, newscasts: a single local XMF file on a computer could link to an audio file on a server that is updated hourly; every time the XMF file is opened, the latest news plays. For interactive content development groups, **ReferenceTypeID** 4's file-level indirection may make management of large media asset collection more flexible; assets subject to frequent change can be kept as external files for easier updating without disturbing the main XMF asset collection.

### ReferenceTypeID 5: External XMF Resource

**ReferenceTypeID 5** points to resources stored in (or referenced by) external XMF files, also via URI (same schemes), but with the resource name indicated using the URI convention (append the filename with a hash [**#**] followed by the desired Node Name). For a type 5 reference to succeed, the target resource must include a Node Name field in its meta-data, and that name must match the referring name. **ReferenceTypeID** 5 has all the advantages of **ReferenceTypeID** 4, but allows the target resource to reside within an XMF file – and as long as the names agree, the link will survive all changes to the target file's structure and other contents.

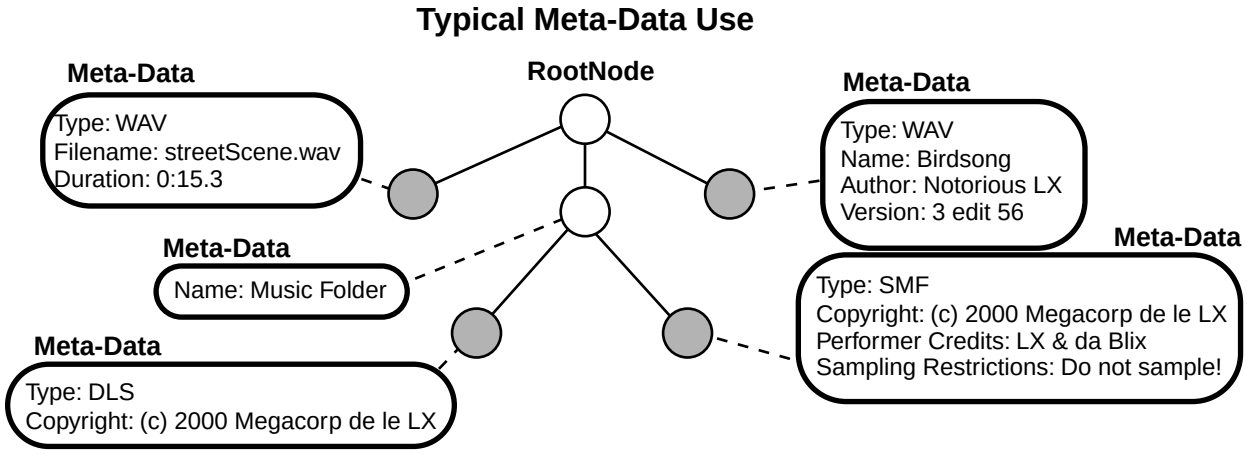



**Typical Use Cases:** This reference type has all the same resource sharing and aliasing benefits as type 4, with the additional flexibility of allowing the target resource to live inside another XMF file.

> **Note:** In **Node**s using **ReferenceTypeID** 5, the **NodeUnpacker**s in the referenced **Node** override any **NodeUnpacker**s in the referring node. Any **NodeUnpacker**s indicated in the referring node should be ignored.
>
> **Note:** The target node will not necessarily hold the target resource directly, as it has a **ContentReference** of its own. However, to avoid excessively long or circular seek chains, the target resource data must be found within 4 reference indirections, otherwise the XMF parser must fail and return a 'Too many XMF indirections' error.

# 3. XMF Meta-Data

Meta-data is ancillary informational or logistical data related to a resource or folder in an XMF file, essentially forming a flexible 'property sheet' for the described item. As an abstract example, a copyright notice might be one **MetaDataItem**, and a performer credit would be a second **MetaDataItem**.




## 3.1. Meta-Data Semantics

XMF can accommodate any number of **MetaDataItem**s for any resource or folder in the collection. Meta-data contents may be either text or binary data, depending on the requirements of the particular field.

The requirements of meta-data standardization, efficiency, and extensibility pull in different directions. To accommodate all three, and still optimize for the most frequently-occurring fields (such as copyright notices and title), XMF supports two categories of meta-data fields: a pre-defined set of Standard Fields for common use, and an open space for arbitrary Custom Fields that end users or applications can freely create and use for non-standard purposes (i.e. application-specific internal data, special user notes, etc.).

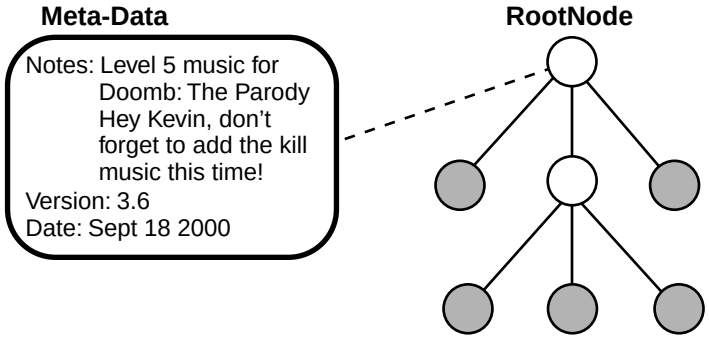
**Support for Internationally Distributed Content –** For each **MetaDataItem** present in a given **Node**, the content creator or distributor can optionally supply multiple alternate versions of the field contents, with each version keyed to one or more Language/Country combinations. This allows an XMF player that knows its geographic location and its user's natural language preference to select and display the appropriate meta-data content version. For example, album liner notes can be provided in English, Japanese Katakana, French, German, Italian, and Spanish, and only the appropriate version will be presented to the user. Text can be encoded in Extended ASCII, Unicode, or compressed Unicode, and can optionally be hidden from the file's end user. Binary meta-data can also be keyed to language and country. However, only the one best content version for each **MetaDataItem** should be presented to the user at any given time.

> **Note:** Future XMF Recommended Practices may specify an XML semantic equivalent to this extensible meta-data mechanism, perhaps using the W3C's Resource Description Format (RDF) – another system that permits multiple (and customized) meta-data etymologies to be used together.

## 3.1.1. Meta-Data Scope Rules

The scope of a **MetaDataItem** depends on where in the Tree it appears.

- Meta-data attached to the **RootNode** is considered to pertain to the XMF file as a whole.




- Meta-data attached to a **FolderNode** is in most cases considered to pertain to all the **FileNode**s and **FolderNode**s it contains, to any nesting depth. If multiple **MetaDataItems** of the same **FieldID** apply to a given **FileNode**, only the most local of those **MetaDataItems** should be used.
- Meta-data attached to a **FileNode** is considered to pertain to that resource only.

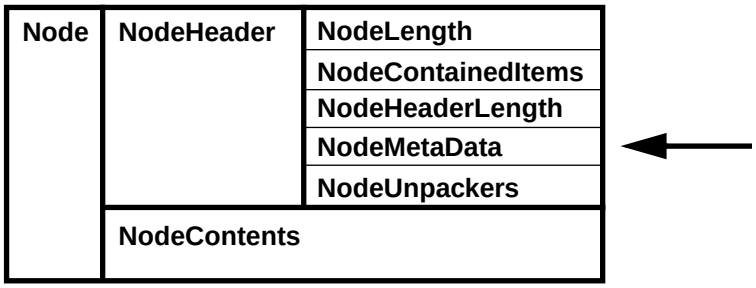
# 3.2. Meta-Data Structures

The meta-data system has two parts:

- Meta-Data information for specific **Nodes** appears in the **NodeMetaData** field of the **NodeHeader** structure, as a list of **MetaDataItem**s, and
- Shared information supporting the multi-country/multi-language meta-data feature appears in the **MetaDataTypesTable** in the **FileHeader** structure.

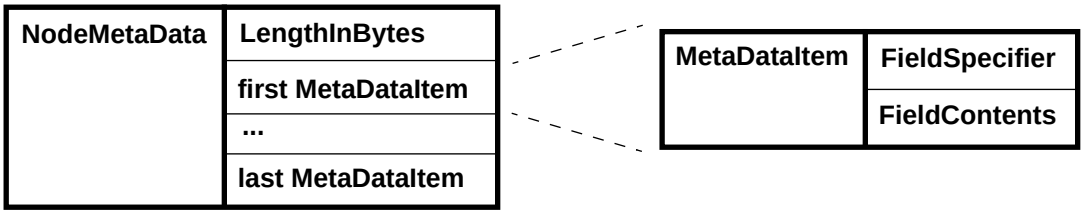
## 3.2.1. NodeMetaData Structure

Recall that meta-data for a **FileNode** or **FolderNode** is stored in the **NodeMetaData** field of the **NodeHeader** structure.




The **NodeMetaData** structure is a length-delimited list of **MetaDataItems**. It may contain any number of **MetaDataItem**s. **LengthInBytes** is a **VLQ** (see section 4.1).

### 3.2.1.1. MetaDataItem Structures




Each **MetaDataItem** consists of two parts, which are further described in the following sections:

- One **FieldSpecifier** structure, and
- One **FieldContents** structure.

### 3.2.1.1.1. FieldSpecifier Structure

in XMF, there are two kinds of meta-data fields:

- Standard fields, for common uses, and
- Custom fields, for special purposes.

Each kind uses a different **FieldSpecifier** format:

- If the first byte of the **FieldSpecifier** is 0, then this **MetaDataItem** contains a Standard meta-data field, and the following **VLQ** holds the **FieldID**. In other words, all Standard **FieldSpecifiers** start with a zero byte, followed by the field number in **VLQ** form. See section 5.2. for the defined Standard **FieldID**s.
- If the first byte of the **FieldSpecifier** is not 0, then this **MetaDataItem** contains a Custom meta data field, and the **FieldSpecifier** is the field's name, in **XString** form.

**FieldSpecifier:**

either **Standard Field Specifier**

**VLQ( 0 ), VLQ( FieldID )**

Example: **`0x00 0x04`**
means the standard
Filename On Disk meta-data field

or **Custom Field Specifier**

**XString( "Your Custom Field Name Goes Here as Text" )**

Example: **`0x16 'This is my Custom Name'`**
means a Custom meta-data item named
'This is my Custom Name'

**Custom Field Specifiers** are formatted as text strings in **XString** form (see section 4.2), and so can use any field specifier at all – the only limitation is that the length cannot be zero. We expect that in most cases users will want custom meta-data to be human readable, and in these cases the desired name in plain text would typically be used directly for the Custom Field Specifier.

> **Example:** If a publisher wanted a Custom field named "Canto Catalog Filename", then the string "Canto Catalog Filename" would appear literally in the XMF file as the Custom Field Specifier, in **XString** form.

**Standard FieldID**s are integers in **VLQ** form. Only the MMA may assign Standard **FieldID**s. This document defines several Standard **FieldID**s (see section 5.2). It is expected that new Standard Fields will be defined in future MMA Recommended Practices. If a parser does not recognize a Standard **FieldID**, it should not attempt to process the **FieldContents** for that **MetaDataItem**.

### 3.2.1.1.2. FieldContents Structure

There are two kinds of meta-data **FieldContents**:

- **Universal Contents**, where just one version of content is present for the **MetaDataItem**. The contents may be either Extended ASCII text or binary data, but not Unicode.
- **International Contents**, where multiple alternate versions of the content are provided, each one keyed to a language and one or more countries. Each alternate may contain either Extended ASCII text, Unicode text, compressed Unicode text, or binary data, and may be either visible to the user or hidden.

**FieldContents** for a Standard meta-data **FieldID** may be Universal, International, or either, as per that **FieldID**'s definition (see section 5.2).

**FieldContents** for a Custom meta-data item may be either Universal or International, on a field-by-field basis.

For Universal contents, the **FieldContents** structure is a **VLQ** of zero (**`0x00`**), then the **LengthInBytes** of the following data in **VLQ** form, a **StringFormatTypeID** in **VLQ** form, and finally the data block. See section 3.2.2.1 for the defined values for a **StringFormatTypeID**.

> Note that the **LengthInBytes** is the number of data bytes plus the size of the **StringFormatTypeID**. As a result, **FieldID**s which require no data, such as flags, can simply use a **LengthInBytes** of 0, and omit the **StringFormatTypeID**. In other words, the **FieldContents** for an empty Universal field is just **`0x00 0x00`**.

For International contents, the **FieldContents** structure is a list of **ContentVersion** structures, prefixed with both a **NumberOfVersions** count and a **LengthInBytes** (both in **VLQ** format). Each **ContentVersion** structure consists of two parts: a **MetaDataType** specifier in **VLQ** form, indicating a string format/language/country combination (see below); and the length-delimited **VersionData**. A zero length indicates no **VersionData**.

**FieldContents:**

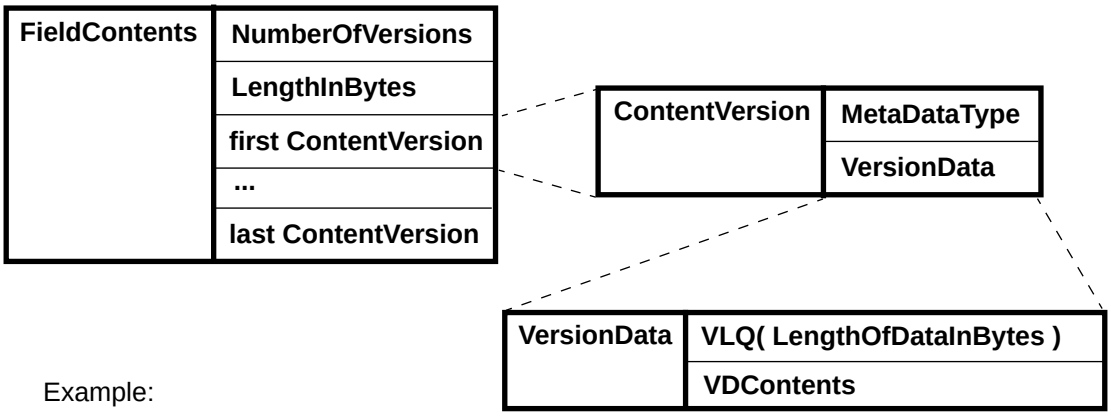
either **Universal Contents** without Country/Language version selection

**VLQ( 0 ), VLQ( LengthInBytes ), VLQ( StringFormatTypeID ), UniversalData**

Example:

```
// FieldContents --------------------------------------
 0x00         // Initial zero indicates UniversalContents
 0x81 0x01    // 129 bytes follow
 0x07         // Binary data, hidden
 <128 bytes of binary data>
```

or **International Contents** with Country/Language version selection




Example:

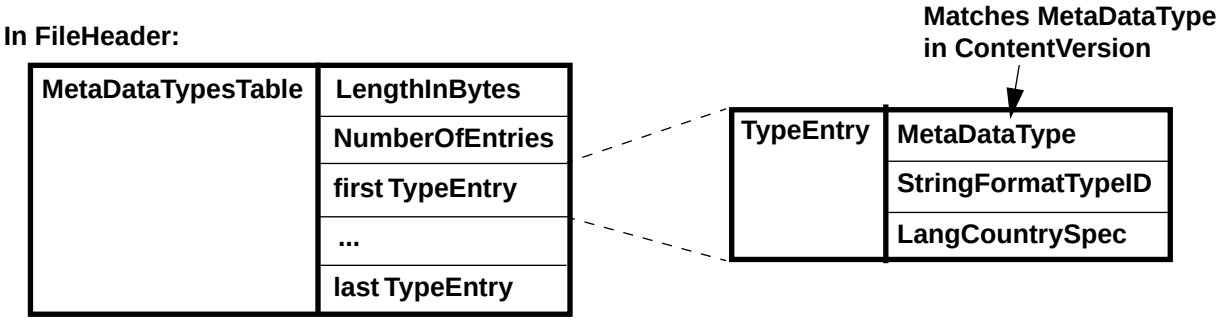
```
// FieldContents --------------------------------------

0x03    // NumberOfVersions: 3 versions
        // -- Initial non-zero indicates InternationalContents
0x30     // 48 bytes follow

// First ContentVersion -----------------
0x03                    // For MetaDataType 3,
0x0C 'Hello, world'     // VersionData is this XString

// Second ContentVersion ----------------
0x01                     // For MetaDataType 1,
0x12 'Bonjour, la France' // VersionData is this XString

// Third ContentVersion -----------------
0x02                    // For MetaDataType 2,
0x0F 'Bonjour, Quebec'   // VersionData is this XString
```

## 3.2.2. MetaDataTypesTable Format

To avoid repeating a Language/Country and **StringFormatTypeID** for every **ContentVersion** present in the XMF file, all unique language/country/format combinations are stored in the **FileHeader** in a list called the **MetaDataTypesTable**. The **MetaDataTypesTable** is delimited both by length (to allow parsers to skip over the table) and by count (for easier detection of out-of-range **MetaDataType** values).

To obtain a **ContentVersion**'s intended **LangCountrySpec** and **StringFormatTypeID**, a parser would search the **MetaDataTypesTable** for a **TypeEntry** for the **MetaDataType** used in the **ContentVersion**. **MetaDataType** values must start at 1 and go up to **NumberOfEntries**, but are not required to appear in the **MetaDataTypes** table in any particular order.

**LengthInBytes**, **NumberOfEntries**, and **StringFormatTypeID** are all **VLQ**s. **LangCountrySpec** is in **XString** form.




Example:

```
0x16    // LengthInBytes is 22
0x03    // NumberOfEntries

// First TypeEntry
0x01            // MetaDataType 1
0x00            // StringFormatID 0 is visible ASCII
0x05 'fr-fr'    // LangCountrySpec says French / France

// Second TypeEntry
0x03            // MetaDataType 3 (order is not required)
0x00            // StringFormatID 0 is visible ASCII
0x02 'en'       // LangCountrySpec says English / all countries

// Third TypeEntry
0x02            // MetaDataType 2
0x00            // StringFormatID 0 is visible ASCII
0x05 'fr-ca'    // LangCountrySpec says French / Canada
```

## 3.2.2.1. StringFormatTypeID Definitions

**StringFormatTypeID**s are integers in **VLQ** form, and may only be defined and assigned by the MMA. This document defines eight **StringFormatTypeID**s:

**0 –** Extended ASCII, visible to the user
**1 –** Extended ASCII, hidden from the user
**2 –** Unicode, visible to the user
**3 –** Unicode, hidden from the user
**4 –** Compressed Unicode, visible to the user
**5 –** Compressed Unicode, hidden from the user
**6 –** Binary data, visible to the user
**7 –** Binary data, hidden from the user

**Note:** In this coding, the low bit of the **StringFormatTypeID** integer acts as a visible/hidden flag. Any **StringFormatTypeID**s assigned in future should continue this pattern.

**Unicode Compression** is as defined in the Unicode Consortium document **Unicode Technical Report #6**. This compression scheme involves both a moving 7-bit window across the 16-bit Unicode character space, and a fixed 7-bit window into Roman character space. This allows full access to international characters at string sizes roughly comparable to ASCII.

Please refer to:

**http://www.unicode.org/unicode/reports/tr6/**

**Unicode Format –** Unicode contents for **StringFormatTypeID**s 2 and 3 must conform to version 2.0 of the Unicode standard, in UTF-16 coding. It is not required to start the Unicode contents with a Byte Order Mark (0xFEFF), because the byte order of the entire XMF file may be determined from the first two bytes of the **FileHeader** structure (section 2.1). When importing text content into Unicode from other text encodings, developers should be careful to observe correct transcoding, based on the ISO/IEC 10646 character set.

## 3.2.2.2. LangCountrySpec Format and Selection Behavior

**LangCountrySpec** syntax and registry are as defined in the Internet Engineering Task Force's HTTP standard (RFC 2068 at section 3.10), as derived from RFC 1766, ISO 639 (language codes) and ISO 3166 (country codes).

For example, the following are valid **LangCountrySpec**s:

English for all countries: **`en`**

English for the US: **`en-us`**

English for Canada: **`en-ca`**

French for Canada: **`fr-ca`**

English for the US and Canada only: **`en-us,ca`**

An XMF playback program or device should make an effort to determine its location and its user's natural language at the time the XMF file is first opened, and select the appropriate **ContentVersion** for every meta-data item it accesses in the XMF file. When multiple **ContentVersion**s are available for a **MetaDataItem**, only the one best match should be displayed, never multiple **ContentVersion**s at once for the same meta-data item. The system's location and country must not be changed during parsing of a given XMF file, to ensure synchronization of all such selections for that file.

Please refer to:

**http://ietf.org/rfc/rfc1766.txt**
**http://ietf.org/rfc/rfc2068.txt**
**http://www.oasis-open.org/cover/iso639a.html**
**http://www.din.de/gremien/nas/nabd/iso3166ma/codlstp1/en_listp1.html**

# 4. Atomic Structures

The **VLQ** and **XString** structures are the basis of most data fields in XMF files.

## 4.1. VLQ Format

To save space and avoid all maximum size limitations, most XMF integers are represented via the Variable Length Quantity number format (per "Standard MIDI Files 1.0", p. 2). Basically, a **VLQ** is an integer that's however many bytes long it needs to be to hold the given value. The last byte is signalled by a clear high bit, and all other bytes have the high bit set. This leaves 7 bits of usable data per byte, so turning a **VLQ** into a binary integer means an iterative process of shifting an accumulator left 7 bits, masking the next byte's high bit off, and adding. One-byte **VLQ**s are considered 'last byte,' and must have a high bit of zero.

For example:

```
0       =       0x00    =       0x00
64      =       0x40    =       0x40
127     =       0x7f    =       0x7f
128     =       0x80    =       0x81 0x00
8192    =       0x2000=         0xC0 0x00
16383   =       0x3fff=         0xff 0x7f
2097151=        0x1fffff=       0xff 0xff 0x7f
```

| | | hi bit | |
|---|---|---|---|
| **VLQ** | **first Byte** | **1** | **7 data bits** |
| | **...** | **1** | **7 data bits** |
| | **last Byte** | **0** | **7 data bits** |

## 4.2. XString Format

The **XString** format used in several places is a length-delimited ASCII text string, where the length is encoded in a leading **VLQ** rather than a fixed-size integer. This is similar to the Pascal String of olde, but allows the string to have any length, and allows a parser to skip over any **XString** it isn't interested in, without having to scan for a terminating character.

| | |
|---|---|
| **XString** | **VLQ( LengthOfTextInBytes )** |
| | **Text Data** |

# 5. Managed ID Formats and Assignment

This section describes the format, management, and assignment of the number spaces for **UnpackerID**s, Standard Meta-Data **FieldID**s, and **ResourceFormatID**s.

## 5.1. UnpackerID Format and Assignment

In a **NodeUnpackers** list in a **NodeHeader**, an **UnpackerID** selects a decoding algorithm to apply to the encoded resource data referenced in the **NodeContents**. Every **UnpackerID** is prefixed by an **UnpackerTypeID** that picks one of four number spaces, each of which uses a different format and mechanism to prevent assignment collisions, as detailed in the following four subsections.

### 5.1.1. Standard UnpackerIDs

**Standard UnpackerID**s are prefixed with **UnpackerTypeID 0** (**`0x00`**).

Standard **UnpackerID**s are reserved for unpacker types that all XMF parsers are required to implement, and may only be defined by the MMA. There will probably be very few Standard **UnpackerID**s, so in practice they'll all fit into one byte for the foreseeable future.

A **Standard UnpackerID** consist of one integer in **VLQ** form.

Standard **UnpackerID**s may only be defined by the MMA.

| UnpackerID | Unpacker Name | Algorithm Reference |
|---|---|---|
| **0** | No Unpacker | (No operation) |

**Example:**

```
0x00 // Standard UnpackerID follows
0x00 // No Unpacker
```

**Note:** Developers who wish to support XMF to the fullest degree are advised to check the current status of the XMF Files unpackers work.

## 5.1.2. MMA Manufacturer UnpackerIDs

**MMA Manufacturer UnpackerID**s are prefixed with **UnpackerTypeID 1** (**0x01**).

MMA Manufacturer **UnpackerID**s are reserved for manufacturer-specific unpacker types. Each MMA Manufacturer is free to create **UnpackerID**s based on their own MMA Manufacturer ID, subject to the following format restrictions. Manufacturers are free to either publish their **UnpackerID**s or keep them proprietary.

MMA Manufacturer **UnpackerID**s contain two parts: first the company's assigned MMA Manufacturer ID in ordinary 1-byte or 3-byte form, then a **VLQ** containing the company's internal unpacker ID.

Note that all MMA Manufacturer IDs are either one byte long or three bytes long, as indicated by the first byte (**0x00** means a three-byte ID).

**Examples:**

```
// Example 1: 3-byte Mfr ID
0x01 // MMA Mfr UnpackerID follows
0x00 0x7c 0x7f // Electric Kazoo MMA Mfr. ID (3-byte since hibyte = 0)
0x0C // Electric Kazoo's Unpacker type 12

// Example 2: 1-byte Mfr ID
0x01 // MMA Mfr UnpackerID follows
0x7c // KazooTronics MMA Mfr. ID (1-byte since hibyte not zero)
0x81 0x01 // KazooTronics's Unpacker type 129
```

## 5.1.3. Registered UnpackerIDs

**Registered UnpackerID**s are prefixed with **UnpackerTypeID 2** (**0x02**).

Registered **UnpackerID**s are reserved for custom or proprietary unpacker types, but are not linked to MMA Manufacturer IDs, and may only be defined and assigned by the MMA. This allows non-MMA manufacturers access to eXtensibility without incurring the higher byte count of Non-Registered **UnpackerID**s.

A registered **UnpackerID** consists of one integer in **VLQ** form.

To obtain a Registered **UnpackerID**, or for the MMA's registration policy, see **http://www.midi.org**. For the current list of Registered **UnpackerID**s, see **http://www.midi.org**.

**Example:**

```
0x02 // Registered UnpackerID follows
0x47 // Registered Unpacker type 71
     // (must be listed on MMA website to be valid)
```

## 5.1.4. Non-Registered UnpackerIDs

**Non-Registered UnpackerID**s are prefixed with **UnpackerTypeID 3** (**`0x03`**).

Non-Registered **UnpackerID**s are reserved to allow non-MMA manufacturers to use arbitrary private compression and data protection mechanisms without MMA registration. To avoid collisions, a Non-Registered **UnpackerID**s must be generated as a Globally Unique Identifier (GUID).

A Non-Registered **UnpackerID** consists of one GUID in **VLQ** form.

While GUIDs are defined to be 16 bytes long, typically length will increase to 19 bytes when converted to **VLQ** form. However, in the conversion to **VLQ** any contiguous runs of high-order zero bits will be truncated at 7-bit boundaries, perhaps producing a **VLQ** shorter than 19 bytes.

**Example:**

```
0x03 // Non-Registered UnpackerID follows
<16-byte GUID in VLQ form appears here>
```

## 5.2. Meta-Data Standard FieldID Format and Assignment

Standard **FieldID**s for XMF Meta-Data are integers in **VLQ** format. This document defines seven Standard **FieldID**s for logistical and informational purposes. Only the MMA may assign Standard **FieldID**s. Note that up to 127 Standard **FieldID**s can be defined without exceeding a 1-byte **VLQ**.


P.O. Box 3173 • La Habra CA 90632-3173 • Tel (714) 736-9774 • Fax (714) 736-9775 • email mma@midi.org

## 5.2.1. Standard FieldID Assignments

**Important:** Future MMA Recommended Practices for XMF are expected to define further Standard **FieldID**s, addressing informational meta-data (i.e. authoring and commerce information such as copyright notices and credits) and rights management, with guidelines for cross-coding to and from major relevant standards.

The following table contains the Standard FieldID assignments.

**Note:** In defining new Standard meta-data fields, every **FieldID** must be defined as requiring either Universal or International **FieldContents**, or allowing the use of either.

| FieldID | Field Name and Notes | Valid for Node Types | Contents Format |
|---|---|---|---|
| **0** | **XMF File Type**<br>E.g. XMF Type 0, or XMF Type 1, etc., including RevisionID.<br>Values to follow in later RPs. | File or Folder, but only allowed in the **RootNode** | Universal, Binary (hidden or visible)<br>See section 5.2.2 for **FieldContents** format. |
| **1** | **Node Name**<br>Name must be unique in the XMF file. | File or Folder | Universal, Extended ASCII (hidden or visible) |
| **2** | **Node ID Number**<br>Optional, for parser convenience. | File or Folder | Universal Binary (hidden or visible)<br>**VLQ** |
| **3** | **Resource Format**<br>E.g. SMF, DLS, WAV, etc.<br>Values to follow in later RPs. | File | Universal, Binary (hidden or visible)<br>See section 5.3 for **FieldContents** format. |
| **4** | **Filename on Disk**<br>Excluding filename extension. | File | Universal, Extended ASCII (hidden or visible) |
| **5** | **Filename Extension on Disk**<br>For OS's other than Mac OS. | File | Universal, Extended ASCII (hidden or visible)<br>Including dot, e.g. "**`.dls`**". |
| **6** | **Mac OS File Type and Creator**<br>Formatted as 8 ASCII characters, e.g. "**TYPECR8R**" | File | Universal, Extended ASCII (hidden or visible) |
| **7** | **MIME Type**<br>(e.g. "**`audio/wav`**") | File | Universal, Extended ASCII (hidden or visible) |
| **8** | **Title**<br>Cross-coding guidelines expected to follow in a later RP. | File or Folder | Universal, Extended ASCII (hidden or visible),<br>or International |
| **9** | **Copyright Notice**<br>Cross-coding guidelines expected to follow in a later RP. | File or Folder | Universal, Extended ASCII (hidden or visible)<br>or International |
| **10** | **Comment**<br>Cross-coding guidelines expected to follow in a later RP. | File or Folder | Universal, Extended ASCII (hidden or visible)<br>or International |

# 5.2.2. Field Contents Interpretation Notes

## FieldID 0: XMF File Type

The **XMF File Type** meta-data field identifies the Type to which the XMF file conforms, for example XMF Type 0 or XMF Type 1, and the specification Revision level within that Type. This field is only valid in the **RootNode**; if the reading parser encounters it elsewhere, it must not process it as a file type indicator.

Note that this is distinct from the **XmfMetaFileVersion** appearing in the **FileHeader**, which indicates the version of the **XMF Meta-File Format Specification** being used.

The **FieldContents** of an XMF File Type **MetaDataItem** is a **FileTypeID** followed by a **RevisionID**, both in **VLQ** form. **FileTypeID**s are integers corresponding directly to XMF File Types (i.e. 0x07 means XMF Type 7), and may only be assigned by the MMA, typically through separate Recommended Practice documents (see for example **Type 0 and Type 1 XMF Files**). **RevisionID**s are integers corresponding directly to revision numbers of the given XMF File Type (i.e. 0x00 means initial release, 0x03 means third revision), and may only be assigned by the MMA.

# 5.3. ResourceFormatID Format and Assignment

Every **FileNode** must include in its **NodeMetaData** one **MetaDataItem** indicating the resource's data format (e.g. SMF, DLS, WAV, etc.), using the **ResourceFormat** Standard meta-data field (Standard meta-data **FieldID** 3). Every **ResourceFormatID** is prefixed by a **FormatTypeID** that picks one of four number spaces, each of which uses a different format and mechanism to prevent assignment collisions, as detailed in the following four subsections. The **FieldContents** for a **ResourceFormat MetaDataItem** depends on the **FormatTypeID**. This closely parallels the **UnpackerID** scheme.

## 5.3.1. Standard ResourceFormatIDs

**Standard ResourceFormatID**s are prefixed with **FormatTypeID 0** (**0x00**), followed by the **ResourceFormatID** in VLQ form.

Standard **ResourceFormatID**s are reserved for standard data formats such as SMF, DLS, WAV, etc. that all XMF parsers for a given XMF File Type are required to implement. There will probably be very few Standard **ResourceFormatID**s, so in practice they'll all fit into one byte for the foreseeable future.

| ResourceFormatID | Format Name | Defined in RP for XMF File Type |
|---|---|---|
| **0** | Standard MIDI File (SMF), Type 0 | XMF Type 0 and 1 |
| **1** | SMF, Type 1 | XMF Type 1 |
| **2** | Downloadable Sounds (DLS), Level 1 | XMF Type 0 and 1 |
| **3** | DLS, Level 2 | XMF Type 0 and 1 |
| **4** | DLS, Level 2.1 | XMF Type 0 and 1 |

**Note:** Standard **ResourceFormatID**s may only be defined by the MMA, and will typically be added via the Recommended Practice documents that define new XMF File Types. Developers who wish to support XMF to the fullest degree are advised to check the current status of the XMF Files work.

**Example:**

```
// MetaDataItem ------------------------------------

// FieldSpecifier:
0x00 // VLQ( 0 ): Standard meta-data FieldID follows
0x03 // VLQ( 3 ): FieldID 3: This is a Resource Format MetaDataItem

// FieldContents:
0x00 // VLQ( 0 ): Universal contents follows
0x02 // VLQ( 3 ): LengthInBytes 3: 3 bytes follow
0x06 // VLQ( 6 ): Visible Binary data follows
// UniversalData starts here
0x00 // VLQ( 0 ): FormatTypeID 0: Standard ResourceFormatID follows
0x01 // VLQ( 1 ): ResourceFormatID 1: Resource is in SMF Type 1 format
```

## 5.3.2. MMA Manufacturer ResourceFormatIDs

**MMA Manufacturer ResourceFormatID**s are prefixed with **FormatTypeID 1** (**0x01**).

MMA Manufacturer **ResourceFormatID**s are reserved for manufacturer-specific resource types. Each MMA Manufacturer is free to create **ResourceFormatID**s based on their own MMA Manufacturer ID, subject to the following format restrictions. Manufacturers are free to either publish their **ResourceFormatID**s or keep them proprietary.

A MMA Manufacturer **ResourceFormatID** contains two parts: first the company's assigned MMA Manufacturer ID in ordinary 1-byte or 3-byte form, then a **VLQ** containing the company's internal format ID.

Note that all MMA Manufacturer IDs are either one byte long or three bytes long, as indicated by the first byte (**0x00** means a three-byte ID).

**Example:**

```
// MetaDataItem ------------------------------------

// FieldSpecifier:
0x00 // VLQ( 0 ): Standard meta-data FieldID follows
0x03 // VLQ( 3 ): FieldID 3: This is a Resource Format MetaDataItem

// FieldContents:
0x00 // VLQ( 0 ): Universal contents follows
0x06 // VLQ( 6 ): LengthInBytes 6: 6 bytes follow
0x06 // VLQ( 6 ): Visible Binary data follow
// UniversalData starts here
0x01 // VLQ( 1 ): FormatTypeID 1: MMA Mfr ResourceFormatID follows
0x00 0x7c 0x7f // Electric Kazoo MMA Mfr. ID
0x0A // VLQ( 10 ): Resource is in Electric Kazoo's format type 10
```

## 5.3.3. Registered ResourceFormatIDs

**Registered ResourceFormatID**s are prefixed with **FormatTypeID 2** (**0x02**).

Registered **ResourceFormatID**s are reserved for custom or proprietary resource formats, but are not linked to MMA Manufacturer IDs, and may only be defined and assigned by the MMA. This allows non-MMA manufacturers access to eXtensibility without incurring the higher byte count of Non-Registered **ResourceFormatID**s.

A Registered **ResourceFormatID** consists of one integer in **VLQ** form.

To obtain a Registered **ResourceFormatID**s, or for the MMA's registration policy, see **http://www.midi.org**. For the current list of Registered **ResourceFormatID**s, see **http://www.midi.org**.

**Example:**

```
// MetaDataItem ------------------------------------

// FieldSpecifier:
0x00 // VLQ( 0 ): Standard meta-data FieldID follows
0x03 // VLQ( 3 ): FieldID 3: This is a Resource Format MetaDataItem

// FieldContents:
0x00 // VLQ( 0 ): Universal contents follows
0x04 // VLQ( 4 ): LengthInBytes 4: 4 bytes follow
0x06 // VLQ( 6 ): Visible Binary data follow
// UniversalData starts here
0x02  // VLQ( 2 ): FormatTypeID 2: Registered ResourceFormatID follows
0x81 0x46 // VLQ( 197 ): Resource is in Registered format number 197
          // (must be listed on MMA website to be valid)
```

## 5.3.4. Non-Registered ResourceFormatIDs

**Non-Registered ResourceFormatID**s are prefixed with **FormatTypeID 3** (**0x03**).

Non-Registered **ResourceFormatID**s are reserved to allow non-MMA manufacturers to use arbitrary private resource formats without MMA registration. To avoid collisions, a Non-Registered **ResourceFormatID**s must be generated as a Globally Unique Identifier (GUID).

A Non-Registered **ResourceFormatID** consists of one GUID in **VLQ** form.

While GUIDs are defined to be 16 bytes long, typically length will increase to 19 bytes when converted to **VLQ** form. However, in the conversion to **VLQ** any contiguous runs of high-order zero bits will be truncated at 7-bit boundaries, perhaps producing a **VLQ** shorter than 19 bytes.

**Example:**

```
// MetaDataItem ------------------------------------

// FieldSpecifier:
0x00 // VLQ( 0 ): Standard meta-data FieldID follows
0x03 // VLQ( 3 ): FieldID 3: This is a Resource Format MetaDataItem

// FieldContents:
0x00 // VLQ( 0 ): Universal contents follows
0x15 // VLQ( 21 ): LengthInBytes 21: 21 bytes follow
0x06 // VLQ( 6 ): Visible Binary data follow
// UniversalData starts here
0x03  // VLQ( 3 ): FormatTypeID 3: Registered ResourceFormatID follows
<19 bytes appear here> // GUID in VLQ form
```

# Appendix: Notes on the XMF Meta-File Format

**Topics:**



# Purpose of XMF

In recent years the need for a standard, universal file-bundling format for music and sound data has become clear. This is especially true in interactive multimedia and Internet entertainment delivery where the audio producer's output is often not just a single file, but rather a collection of related files for each title, scene, or setting. In many cases more than one file format is needed – for example, games usually use both SMF and digital audio files.

Further, some media file formats tend by their nature to create dependencies between files. Notably, when the DLS file format is used for content-specific samples (such as sung lyrics), there is an implied link between the DLS file and the SMF file that uses it.

Attempting to use unbundled collections of files poses a number of difficult problems for the content producer, the content user, and the tool developer:

- Hard to keep the collection together
- Hard to move the collection from one computer to another
- Hard to move the collection to different operating systems (file types, .zip vs. sit, etc.)
- Hard to ensure version synch among files in the collection
- Hard to keep other files out of the collection
- Each file type has its own meta-data conventions (SMF, DLS, WAV, etc.)
- No standard way to add meta-data notations to the entire collection
- Hard to express how the collection should be used – what does 'play' or 'load' mean for the collection?
- No consistent cross-platform data-compression standard for a collection, so files tend to be large
- No consistent cross-platform information security (intellectual property protection) standard for a collection, so business models are limited
- Relationship between the music/sound content creator and the programmer/user is complicated by all the above ambiguities

XMF addresses all of these problems, bringing important benefits to all interested parties:

- **For users of music and sound media** XMF makes working with music and sound material easier by reducing the collection to a single file. Expectations of playback behavior for each area of application are made clear through a series of separate Recommended Practices. If desired, the differences between media types can be made invisible – it doesn't matter whether the XMF file contains an SMF file or a WAV file, it loads and plays just the same in either case.

- **For producers of music and sound media** XMF provides increased confidence in the collection's integrity by guarding against misuse in a variety of ways, as well as the increased convenience of delivering just a single file and being able to attach meta-data to the collection as a whole. The addition of data security features makes new forms of business possible – for example, think of telephone networks handling per-use micro-payments for cell phone ring tones, or DLS banks that need to be unlocked with a purchased key.
- **For developers of playback software and applications** XMF adds (in separate Recommended Practices) unambiguous interpretation rules to collections of music and sound materials, builds on top of the existing media-specific playback APIs (SMF, DLS, WAV, etc.), and is a cross-platform standard. Any needed custom resource types (for example: softsynth patch banks, configuration setups, etc.) can stored in XMF files, and will be ignored by other playback software. In future, new resource types could be adopted by the MMA for 'playlists' of SMF phrases or samples, enabling new XMF applications ranging from cross-platform remixers to a 'universal session file' format.
- **For developers of authoring software** XMF provides a gateway to cross-platform and interactive playback environments, including those beyond computers.

# Advice for XMF Readers

If your program is asked to play an XMF file, then you'll have some XMF parser code sitting on top of playback APIs that support SMF, DLS, WAV, and/or other resource types. XMF's scope does not extend into the internals of those resource types or their playback API's – only resource access, and the setup of the virtual playback studio.

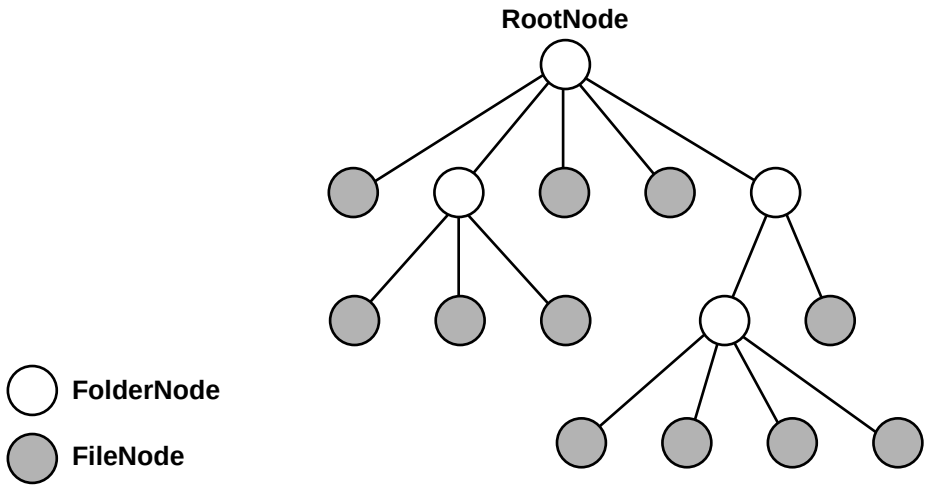
Your XMF reader has to know two things:

**How do I know what resources are in, and/or referred to by, the XMF file?**
**How do I know what I'm supposed to do with those resources?**

## How do I know what resources are in, and/or referred to by, the XMF file?

XMF files are logically structured as hierarchical **Tree**s, very much like a computer file system with 'files' (resources) located in 'folders' that may be nested to any depth.

**Example Tree Hierarchy**



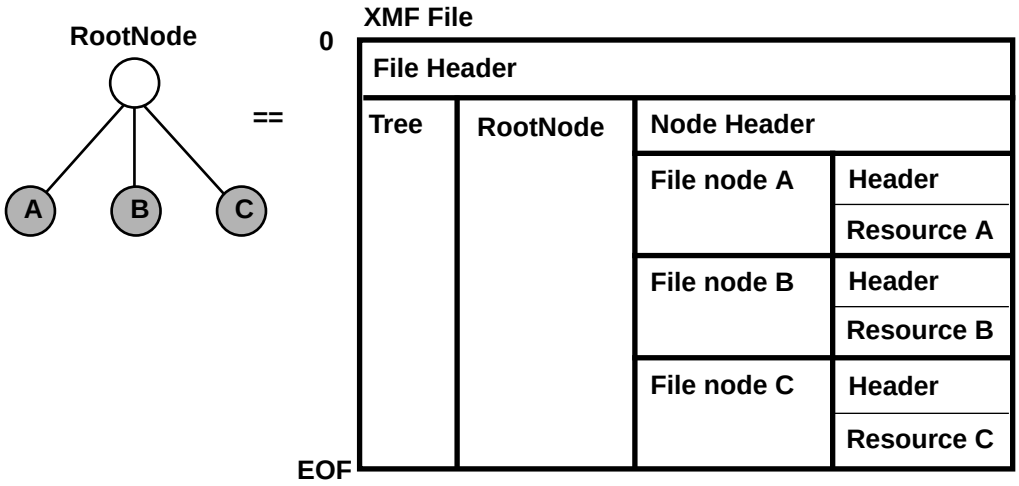
As with file systems, the structure and depth of the **Tree** is arbitrary and under the control of the end user. Therefore, your XMF reader code must be able to handle any arbitrary **Tree** a user may invent, and must actually scan the **Tree** in order to determine what resources are present (or referred to) in a given XMF file.

## Resource Storage and File Layout

You should be aware that the resources described by a given logical **Tree** may be stored in many places, depending on the resource Reference Type used in each **FileNode**, and this can produce many different storage profiles. Resources may be stored directly within the **Tree**, elsewhere in the same XMF file, or in external resource files (SMF, WAV, DLS, etc.) or external XMF files.

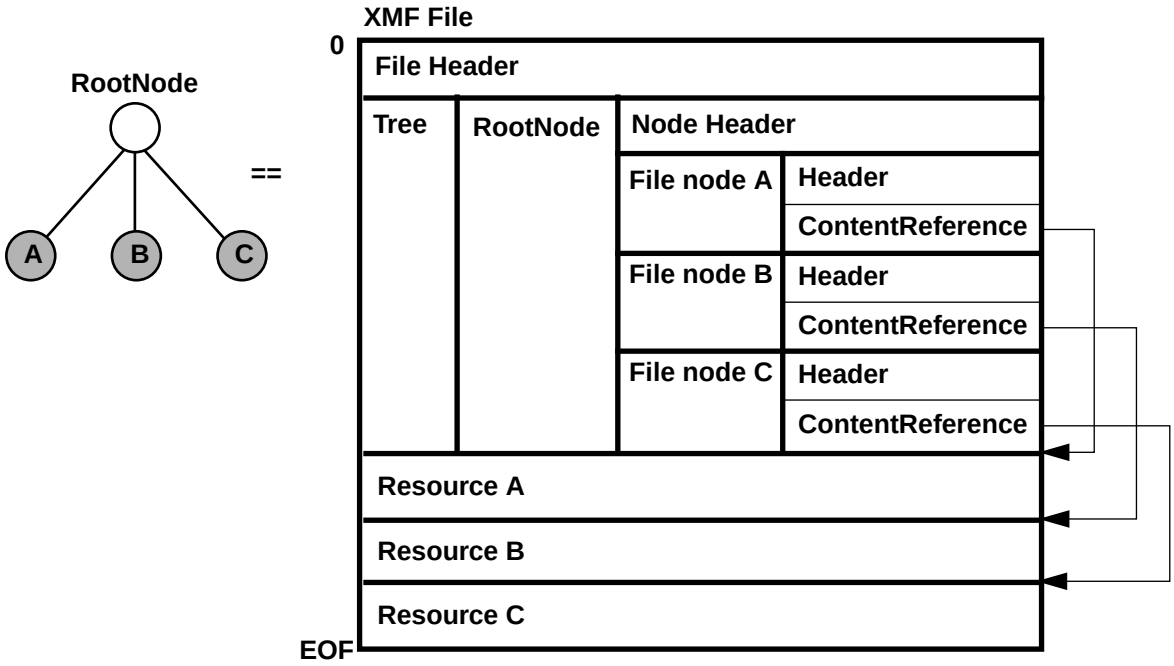
Storing the resources directly in the **Tree** produces hierarchical containment:

### Hierarchical Storage Example



The resources can also be stored elsewhere in the file, producing a flat storage profile and a much smaller **Tree**:

### Flat Storage Example

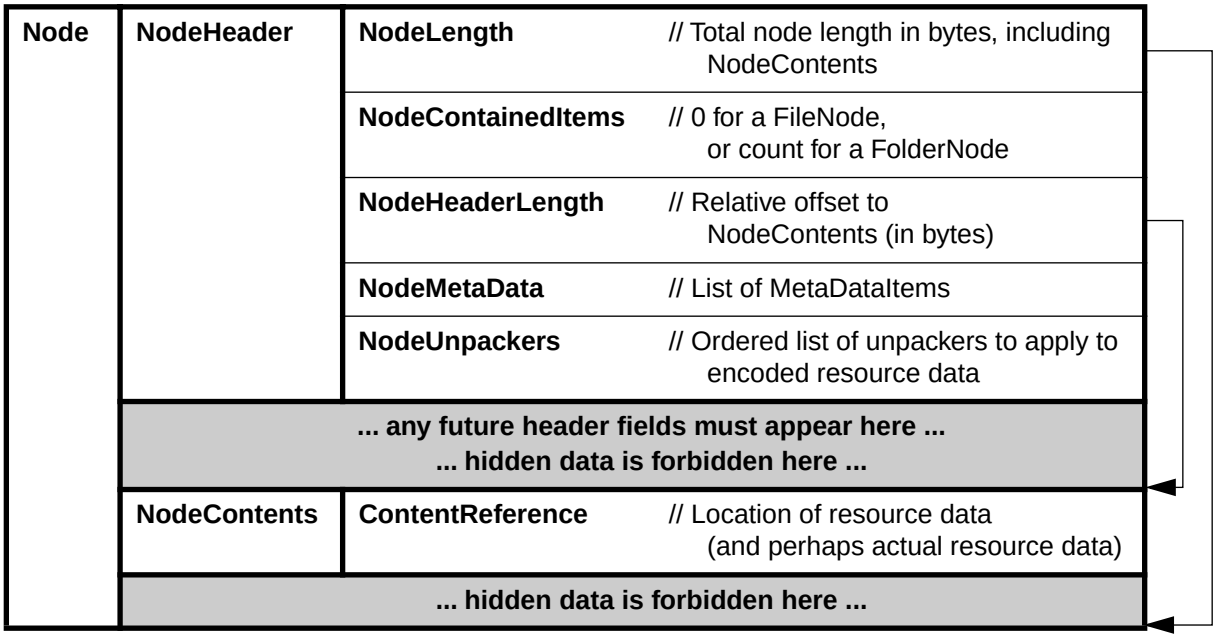



Many other storage profiles are also available, including XMF files that are only collections of references to external files (see **2.2.1.2.1. Reference Types** in the **XMF Meta-File Format Specification**).

## Parsing the Tree

Each file or folder in the **Tree** is represented as a **Node**, and all **Node**s use the same data structure. You start your parse at the **Tree**'s **RootNode**, whose location is indicated in the **FileHeader**'s **TreeStart** field.

The structure for all **Node**s is:




- **NodeLength** is a **VLQ** described below under **Navigating Nodes**.

  **Note:** This field should **not** be used to store hidden data in the gap after the end of the **NodeContents**. XMF parsers will not find that data, so XMF tools will strip that data when editing an XMF file, and XMF players will not be able to access that data.

- **NodeContainedItems** is a **VLQ** that indicates whether the **Node** is a **FileNode** (resource) or **FolderNode** (container for further **Nodes**), and for **FolderNodes** indicates the number of directly contained **Nodes**. If a **FolderNode**, then the **NodeContents** describes catenated **Nodes** for the contained files and/or folders. Since 0 is used to indicate a **FileNode**, empty **FolderNodes** are not allowed in XMF.
- **NodeHeaderLength** is a **VLQ** that takes you to the **NodeContents** part. This is redundant for XMF version 1.00, but preserves backwards compatibility if MMA adds more fields in future.

  **Note:** This field should **not** be used to store hidden data in the gap between the **NodeHeader** and the **NodeContents**. XMF parsers will not find that data, so XMF tools will strip that data when editing an XMF file, and XMF players will not be able to access that data.

- **NodeMetaData** contains meta-data for the resource. (See **3. XMF Meta-Data** section of the **XMF Meta-File Format Specification** for format).
- **NodeUnpackers** is a list of the decoding operations you have to apply to the stored resource data block to get back clear data that you can play (see **2.2.1.1. NodeHeader Structure** section of the **XMF Meta-File Format Specification**). Unpackers are usually data decompressors, or decrypters or other info-security operations.
- **NodeContents** is a structure indicating where the resource data lives.

Note that you won't have to actually go get the resource data for every **Node** – just the ones you know you need.

When you do need a resource, a **NodeContents** can use any of five different Reference Types, each describing a different way to get from the **Node** to the resource data it describes:

- **1:** Immediate, in-line resource data
- **2:** Pointer to raw resource data elsewhere in the file
- **3:** Pointer to a resource **Node** elsewhere in file (with its own **NodeContents** structure)
- **4:** Reference to an external whole file (i.e. SMF, DLS, WAV)
- **5:** Reference to a named resource in an external XMF file.

See section **2.2.1.2. NodeContents Structure** for parsing details.

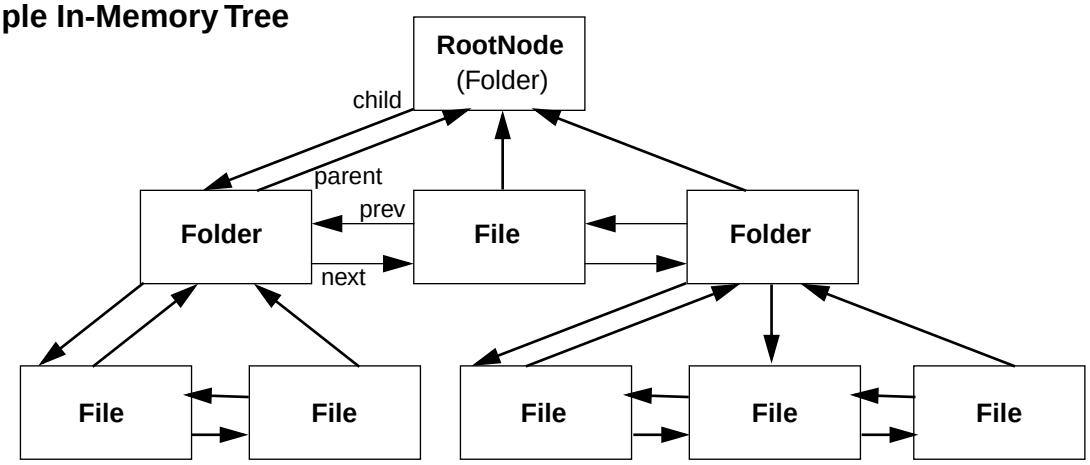
## Navigating Nodes

Navigation among **Nodes** is handled via the **NodeLength** field. It indicates the **Node** structure's length in bytes (including **NodeContents**), so you can use it to jump ahead to the next **Node** at the same level of the **Tree**.

For a **FolderNode**, **NodeLength** also tells you when you've read the last contained item – just compare your read pointer against the parent folder's expected ending offset (or decrement the **NodeContainedItems**).

Note that **NodeLength** only facilitates forward motion through the tree. If you anticipate the need to move around the **Tree** more flexibly, then the first time you parse the Tree you may want to build an in-memory tree of your own as you go – with **`next`**, **`prev`**, **`parent`**, and **`child`** pointers to accelerate subsequent random & backwards access.

**Example In-Memory Tree**




## How do I know what I'm supposed to do with those resources?

The XMF meta-file format per se has no required player behaviors. However, each XMF File Type may specify a different set of rules for how players, converters, and other tools and applications should handle the resources in an XMF file. These rules are defined in a separate Recommended Practice document defining each XMF File Type. A **MetaDataItem** in the **RootNode** indicates the XMF File Type of the file you're parsing, and the Type should also be indicated via filename extension, MIME type, and/or Mac OS File Type (as appropriate for the context). Each player, tool, or application must keep track of which XMF File Types it knows how to handle, and should exercise caution and discretion when opening an unrecognized XMF File Type.

## Resource Relationships

While an XMF file may be a simple bag of independently useful resources, some XMF use contexts will involve relationships between resources – for example, a game scene may need several SMFs played in a certain order, or an SMF may need a specific DLS bank to play correctly. The XMF meta-file structure as presently conceived does not directly express such relationships (other than via folder groupings), instead relying on meta-data, new resource types (included as part of the same XMF collection), and Recommended Practice documents defining particular XMF file Types, to express them.

Resource relationships fall into two major categories, further explained below:

**Resource Groupings**
**Resource Dependencies**

## Resource Groupings

Music and sound authors can use the folder hierarchy to arrange resources into arbitrary groups – by project, by media type, or any other desired criteria. Custom meta-data could also be used to create groupings via tagging, as could naming conventions using the standard meta-data **Node Name** field, or the **Filename on Disk** field. Although any XMF user could set up their own rules for what to do with those groupings, XMF as presently conceived does not include any universal rules governing them; that kind of interpretation is left to negotiations between the people and tools who create XMF files and the people and tools that use XMF files. Some specific standard behaviors may be defined in the XMF Recommended Practice documents that define specific XMF file Types – for example, **Autostart** and **Preload** are defined in the Recommended Practice document for XMF Type 0 and Type 1.

This leaves the door open for any number of application-specific grouping and interpretation conventions – i.e. a game publisher might invent one way of doing things, and a phone network might invent another that better fits its requirements, but the commonly readable resources from either source's XMF files could always be interchanged, and a flexible XMF authoring tool could create both kinds of files.

## Resource Dependencies

XMF as presently conceived does not include any generalized mechanism for directly representing resource dependencies. Where necessary, specific kinds of dependence can be represented by inventing new resource types and including such resources in XMF files. For example, some SMFs will rely on patch banks for custom synth types, and this dependency is expected to be represented in a new **BankMap** resource type. We expect to be able to handle any other important resource dependencies that come to light in future in a similar way.

## What do I do if I can't do what's expected?

Parsers should anticipate the following error conditions, and report them to the user if appropriate in the context of the intended user experience:

– Too many XMF indirections
– Too many reference indirections
– Can't access required resource
– Can't **Preload** requested resource (XMF Type 0 and 1)
– Can't **Autostart** requested resource (XMF Type 0 and 1)
– Can't parse XMF file
– External **`file:`** access not supported
– External **`http:`** access not supported
– Unrecognized Standard meta-data **FieldID**
– Unrecognized Standard **UnpackerID**
– Unrecognized Registered **UnpackerID**
– No unpacker for requested **UnpackerID**
– Unrecognized Standard **ResourceFormatID**
– Unrecognized Registered **ResourceFormatID**
– No handler for requested **ResourceFormatID**

# Advice for XMF Writers

This section contains suggestions for creating XMF files.

## How to Build an XMF File

The use of **VLQ**s and the hierarchical nature of the **Tree** means that XMF files are most easily assembled in hierarchical order, starting from the most deeply nested point and working up the tree to the **RootNode**.

The suggested procedure is:

1. Collect all resources (SMF, WAV, DLS, etc.) you want to bundle into the XMF file, as files and/or data blocks in memory.
2. Collect the locations (URI's, paths) of all external resources you want to reference from the XMF file.
3. Collect all the meta-data you want to apply to each bundled and referenced resource.
4. Apply all desired encoders to each resource, recording the unencoded size and **UnpackerID** for each operation, and the order of operations.
5. Plan your bundled resource storage.

   You have a great deal of discretion in terms of how to put the resources together. The simplest approach is to store all resources directly in the **Tree**. However if you want to optimize the **Tree** you may want to store the resources elsewhere in the file, use aliases, and/or use detached **Nodes** to store each item's meta-data and unpackers outside of the **Tree**. See **Resource Storage and File Layout**.
6. Create the **Node** structure (**NodeHeader** and **ContentReference** structures) for every bundled and referenced resource. This is where meta-data and **UnpackerID**s are associated with the resources.
7. Create the **Tree** structure by catenating all the **Nodes** and stitching them together with their **NodeLength** fields. You should start at the deepest point in the hierarchy and work up the **Tree** to the **RootNode**, for three reasons:

   A) Outer chunk lengths depend on inner chunk lengths;

   B) The size of each **VLQ** field depends on the number it must hold; and

   C) A **FolderNode**'s contained **Nodes** appear in-line.
8. If you're storing resources outside the **Tree** (Reference Type 2), or using detached **Nodes** (Reference Type 3), catenate those blocks before or after the **Tree**.
9. Create the **FileHeader** structure and catenate it to the start of the file.

   This completes the XMF file.

## Suggested Practices

- If your XMF file contains more than one resource, keep the **Tree** lean by storing resources outside of the **Tree** (via ReferenceType 2), and place the **Tree** immediately after the **FileHeader**. In most cases this allows the **Tree** to be retrieved in a single seek or **http:** fetch, and it can then be used as an accelerator for accessing individual resources.

- If you use detached **Node**s (ReferenceType 3), don't use the same meta-data **FieldSpecifier**s in both the detached **Node** and **Node** in the **Tree**. Although XMF readers are required to use both (to avoid data loss), such duplication could be confusing to end users.
- If you use detached **Node**s (ReferenceType 3), don't duplicate the **NodeUnpackers** list in both the detached **Node** and the **Node** in the **Tree**; or, if you must, be very sure the two lists agree. Although XMF readers are required to use the detached version, such duplication could be confusing to system designers, and could cause unnecessary unpacker searches.


P.O. Box 3173 • La Habra CA 90632-3173 • Tel (714) 736-9774 • Fax (714) 736-9775 • email mma@midi.org

# Appendix: XMF Working Group Membership

| | |
|---|---|
| David Cadmus | Alesis |
| Chris Grigg | Beatnik (Chair and Editor) |
| Steve O'Connell | Bitheadz |
| David Sumich | Bitheadz |
| Ron Kuper | Cakewalk |
| Scott Ruda | E-Mu |
| Jim Wright | IBM Research |
| Rick Cohen | Kurzweil/YCRDI |
| Byron Jacquot | Kurzweil/YCRDI |
| Rob Rampley | Line 6 (Co-Chair and TSB Liaison) |
| Bill Moline | Live Update |
| James Van Buskirk | NemeSys Music |
| Matti Hamalainen | Nokia |
| Marcus Zetterquist | Propellerheads |
| Mike D'Amore | Sonic Foundry |
| Duane Ford | Staccato Systems |
| Mario Ewald | Steinberg |
| Wolfgang Kundrus | Steinberg |
| Michael J. Bundschuh | Sun Microsystems |
| Mark Phillips | US Robotics |
| Athan Billias | Yamaha (AMEI Liaison) |
| Mike Overlin | Yamaha |

# Beatnik XMF Core Team

Chris Muir, Steve Hales, Andrew Rostaing, Doug Scott, Tim Maroney, Chris Van Rensburg, Mark Deggeler, Tom Lichtenberg, Chris Grigg.