

# Interactive XMF: File Format Specification

---

DRAFT 0.9.1a

February 18, 2008

**PUBLIC REVIEW DRAFT**

**NOT FINAL – NOT FOR IMPLEMENTATION**

**Please submit comments at [www.iasig.org/forum](http://www.iasig.org/forum)**

**Published By:**

IASIG: The Interactive Audio Special Interest Group

The MIDI Manufacturers Association

Los Angeles, CA

Interactive XMF Specification

Copyright © 2008 MIDI Manufacturers Association Incorporated

ALL RIGHTS RESERVED. NO PART OF THIS DOCUMENT MAY BE REPRODUCED OR TRANSMITTED IN ANY FORM OR BY ANY MEANS, ELECTRONIC OR MECHANICAL, INCLUDING INFORMATION STORAGE AND RETRIEVAL SYSTEMS, WITHOUT PERMISSION IN WRITING FROM THE MIDI MANUFACTURERS ASSOCIATION.

Printed February, 2008

MMA  
PO Box 3173  
La Habra CA 90632-3173

## About This Document

**\*\*\* READ ME FIRST \*\*\***

### INFORMATION FOR REVIEWERS

The following information is intended to help provide a context for the draft specification. Please read before submitting comments on this draft spec.

#### 1. Instructions for Submissions

Public review comments are being collected via the IASIG's web Forum ([www.iasig.org/forum](http://www.iasig.org/forum)) and via email ([ixmf-comments@iasig.org](mailto:ixmf-comments@iasig.org)) starting February 20, 2008. The iXMF Working Group thanks you for your participation, and will review all comments submitted, however, we cannot guarantee in advance that every comment will receive a detailed reply. Please submit your comments by May 20, 2008.

You must create a Forum account to post threads or replies.

To submit comments on the draft spec:

- First, visit <http://iasig.org/cgi-local/forum/index.pl> and create an account.
- Then go to <http://iasig.org/cgi-local/forum/index.pl?b-ixmf/> and post your comment(s) in new thread(s).

Note: We would prefer you create a separate Forum thread for every issue you wish to comment on, so please include the relevant spec section number in each thread title. If your issue spans multiple spec sections, please choose a specific, descriptive thread title.

To read comments left by others:

- No account is necessary to browse the Forum. Simply visit <http://iasig.org/cgi-local/forum/index.pl?b-ixmf/> and read the threads.

To reply to comments left by others:

- First, visit <http://iasig.org/cgi-local/forum/index.pl> and create an account.
- Then go to <http://iasig.org/cgi-local/forum/index.pl?b-ixmf/> where you can browse the threads, and post your replies where desired.

If your comments are extensive, or you do not feel comfortable making them in public, please send them via email to [ixmf-comments@iasig.org](mailto:ixmf-comments@iasig.org). You can also attach the comments in a PDF, Word, or Text document.

#### 2. Organizations Involved

The IASIG has developed, and continues to maintain, standards important to the interactive audio community, including the I3DL2 spec for interactive 3D audio, and the DLS family of file formats for musical instrument definitions. <http://www.iasig.org>

The MIDI Manufacturers Association (MMA) is IASIG's parent organization, and is the official worldwide source for all English-language MIDI standards. The MMA maintains MIDI 1.0 and develops new MIDI standards to meet evolving industry needs, ranging from musical instruments to show control to mobile phones. <http://www.midi.org>

### 3. History and Motivation

The draft specification was developed by the iXMF Working Group of the IASIG, in response to a requirements document developed by the group. The requirements analysis was driven by the following set of design principles, which serve as a good encapsulation of the motivations behind iXMF:

- Defend against misuse of delivered audio
- Use MIDI and audio together
- Simplify tool development
- Standard formats lead to better tools, so the system should play standard content formats (e.g. MIDI, WAV, AIFF) rather than inventing new ones, and iXMF should be standardized
- Use middleware to transcend least-common-denominator functionality limits
- Provide a data-driven, not code-driven, runtime audio system
- Provide an abstracted interface between coding and audio teams
- Allow a game to request audio services by symbol, not filename
- Give the audio artist control of the audio data by providing an editor tool for bundles of sounds
- Simplify and formalize content management and hand-off procedures
- Allow the audio artist to attach notes to individual sounds

### 4. Previous Standards Involved

iXMF depends in part on existing, established standards. The iXMF File Format is based on the XMF container technology which was created and standardized by the MMA. iXMF introduces some new XMF features that will be added to the XMF Meta File Format standard. The iXMF musical timing model is based on the MMA's Standard MIDI File (SMF) format, and SMF is also used as the container for MIDI note contents and for Tempo Maps associated with musical audio files. iXMF files may include audio files in common pre-existing formats, for example WAV, AIFF, or MP3.

### 5. Importance of iXMF Being a Public, Royalty-Free Standard

Because iXMF is a standard, the specification is available to any company or person interested in implementing an iXMF authoring tool, an iXMF player, or any other product supporting the iXMF format. As with most standards, the more iXMF implementations are released, the larger the ecosystem for iXMF content and audio artists who know how to work in iXMF will become. Because iXMF was developed by and will be maintained by a standards organization (IASIG) rather than a single company, it will remain reliably available far into the future, will never be taken private, and a community process will remain in place to evolve the standard in the future, should that become necessary. And because iXMF is royalty free, there is no barrier to adoption of iXMF as a technology component in any platform, and no “tax” on tool and player implementers.

### 6. Support for iXMF in Products

iXMF is not a product, it's a file format standard. To use iXMF, you will need an authoring tool that supports iXMF files, and a player that supports iXMF files. It is up to tool and player companies, not IASIG, to provide products that support iXMF. The IASIG will not be producing any iXMF tools or players. However, we hope many tool and player companies will decide to support iXMF in their products soon. If you have a favorite tool or player in which you would like to see support for iXMF, we encourage you to contact the manufacturer and let them know.

## **7. Many Potential Uses for iXMF**

iXMF has an open and flexible framework, and so may find many uses ranging from an authoring time interchange file format for interactive applications, to a runtime file format that is actually loaded and played directly in a game. Further, iXMF is appropriate for a wide range of platforms, ranging from powerful personal computers, to game consoles, to relatively constrained mobile devices. When dealing with such a wide range, one size of functionality will not fit all. To provide the best fit for each particular use of iXMF technology, it is expected that multiple subtypes (or "profiles") of iXMF will eventually be introduced and standardized.

As a standardized runtime format, iXMF is well suited for many alternative purposes beyond interactive game soundtracks, including but not limited to 'audio skins' for computers or mobile devices, interactive recorded music releases, DJ live performance content, and user interface sonification for applications, and embedded systems including appliances.

## **8. Extensibility**

Game developers are well known for adding new, customized, and unexpected capabilities to existing things. iXMF is carefully designed to facilitate such extensions through the consistent use of defined Extension Areas in all data structures, and an extension mechanism built into its integrated high speed binary scripting language. This allows game engine developers to start with a powerful standard iXMF player and easily add on whatever new or different functionality the creative vision may require.

iXMF files that contain extra custom data may play differently on different players, depending on whether each player can handle the extra data or not. iXMF tools will preserve any extra custom data stored in Extension Areas, and iXMF players will ignore any unrecognized data they may find there. This rule allows for custom tools and standard tools to be used together freely on the same iXMF content, without risk of data loss.

## **9. Interoperability**

iXMF files that conform to a given profile should play with exactly the same behavior on any player that supports that profile (assuming the same levels of system resources are available). Interoperability between different profiles will depend on the details of the particular profiles; for example, a player based only on audio codec X will not be able to play content intended only for audio codec Y. Some profiles may be defined to be upward compatible with other profiles, so that content intended for a very constrained profile will also play correctly on profiles meant for much more capable platforms. An iXMF file can optionally include as metadata a list of the profiles to which it conforms, allowing players to easily decide whether to accept the file, or reject it as unplayable.

iXMF files that do not conform to any particular profile are perfectly legitimate, and encouraged by the specification, however in this case special care must be taken to ensure that the player is the same kind that the content is intended for. Typically this situation arises for a platform-specific flavor of iXMF. To support this use case, and help reduce possible interoperability issues, the IASIG may allow the registration of optional manufacturer-specific profile IDs, in addition ones for the standardized profiles.

## **10. Concept of Cues**

The concept of Cues is basic to the iXMF effort. In typical Cue-oriented interactive audio systems, the game requests interactive audio services using named, high-level events, and the system's response to each event is determined primarily by the audio artist. In iXMF these CueIDs are actually numeric inside the iXMF file, but they can be assigned symbolic names in the game source code and content management database; also, Cues can be named through the use of metadata in the iXMF file. The soundtrack's response to a given Cue may be simple, such as playing or halting a single audio file image, or it may be complex, such as controlling a system of several related file images and dynamically manipulating various playback parameters over time.

This kind of functionality is generally considered a requirement for any advanced interactive audio system. To our knowledge, iXMF is the first interactive audio system design with this kind of functionality that is not controlled intellectual property and can be technically implemented on any sufficiently powerful platform, and also the first to be standardized.

## **11. Authoring Tools to Manage Low-Level Details**

The iXMF file format is based on a number of relatively low-level data primitives that must be correctly coordinated to produce the intended interactive soundtrack. These primitives include several sets of numeric IDs, data resources and media files in various formats, and Scripts built out of fairly low-level opcodes. Nobody expects an audio artist to be responsible for setting all these details by hand. It's the job of a good iXMF authoring tool to automatically manage as much of this complexity as possible, so the audio artist doesn't have to. For example, the existence of 'Scripts' in iXMF does not mean that the audio artist has to become a programmer. Instead, good iXMF authoring tools can provide templates for various Cue styles with easy-to use user interfaces, and automatically generate the necessary low-level data and scripts. Low-level scripting would be an option that can be exposed to expert audio artists, or when it's necessary to go beyond the templates the tool provides.

-- END --

## Contents

<b>About This Document .....</b>	<b>1</b>
<b>1. Introduction .....</b>	<b>7</b>
<b>1.1 Technical Overview .....</b>	<b>7</b>
1.1.1 iXMF File Identification .....	8
1.1.2 Playback System Minimum Capability Requirements .....	8
<b>1.2 Handling of Nonstandard Contents .....</b>	<b>9</b>
<b>1.3 Time in iXMF .....</b>	<b>9</b>
1.3.1 Marker Positions in Audio and MIDI Media .....	9
1.3.2 Musical Time Positions .....	9
1.3.3 Musical Tempo .....	9
1.3.4 Algorithmic Descriptions of Musical Time Positions .....	9
1.3.5 High Resolution Linear Time (***RATE TBD***) .....	10
1.3.6 Low Resolution Linear Time (Seconds) .....	10
1.3.7 Sample Frame Offsets in Audio Media .....	10
1.3.8 Tick Offsets in MIDI Media .....	10
<b>2. Terminology .....</b>	<b>11</b>
<b>3. Scripting Language .....</b>	<b>13</b>
<b>3.1 Overview .....</b>	<b>13</b>
3.1.1 Extensibility Mechanism .....	13
<b>3.2 ScriptIDs and Names .....</b>	<b>13</b>
3.2.1 Predefined ScriptIDs and Names .....	13
3.2.2 Special Treatment of CueStart Script .....	14
<b>3.3 iXMF Variables .....</b>	<b>15</b>
3.3.1 Variables Data Format .....	15
3.3.2 Number, Kinds, and Scope of Variables .....	15
<b>3.4 Expressions in Script Statements .....</b>	<b>16</b>
3.4.1 The Expression Stack and Expression Engine .....	16
3.4.2 ExpressionOpcodeIDs .....	18
<b>3.5 Built-In Statements: OpcodeSpace 0x00 .....</b>	<b>19</b>
3.5.1 Statements .....	19
3.5.2 Parameter Formats .....	26
3.5.3 OpcodeID and OperandsLen Values .....	30
3.5.4 Format of “var” Parameters .....	34
<b>3.6 Debugging Statements: OpcodeSpace 0x01 .....</b>	<b>34</b>
3.6.1 Statements .....	35
3.6.2 OpcodeID and OperandsLen Values .....	35

<b>4. Data Structures .....</b>	<b>36</b>
<b>4.1 iXMF File Layout .....</b>	<b>36</b>
4.1.1 Folder Structure.....	36
<b>4.2 CueDefinition Resource Format.....</b>	<b>40</b>
4.2.1 Fields.....	40
4.2.2 MaximumInstanceCount Behavior .....	42
<b>4.3 ChunkDefinition Resource Format.....</b>	<b>43</b>
4.3.1 Fields.....	43
4.3.2 Chunk Synchronization.....	44
4.3.3 Sync Groups.....	45
<b>4.4 ScriptDefinition Resource Format .....</b>	<b>46</b>
4.4.1 Fields.....	46
4.4.2 ScriptInstruction Format .....	46
<b>4.5 PositionRuleDefinition Resource Format.....</b>	<b>48</b>
4.5.1 PositionRuleDefinition Fields.....	48
4.5.2 PositionRuleItem Fields .....	48
<b>4.6 TimePosition Resource Format .....</b>	<b>49</b>
4.6.1 Fields.....	49
<b>4.7 TransitionDefinition Resource Format .....</b>	<b>50</b>
4.7.1 Fields.....	50
4.7.2 SyncTypeID enum .....	50
<b>4.8 MediaTypeDefinition Resource Format .....</b>	<b>52</b>
4.8.1 Fields.....	52
<b>4.9 MetadataTagDefinition Resource Format .....</b>	<b>52</b>
4.9.1 Fields.....	52
<b>4.10 MemorySpaceDefinition Resource Format .....</b>	<b>53</b>
4.10.1 Reserved MemorySpaceIDs.....	53
4.10.2 Fields.....	53
<b>4.11 CallbackDefinition Resource Format .....</b>	<b>53</b>
4.11.1 Fields.....	53
<b>4.12 FadeCurveShapeID Type Format and Value Definitions.....</b>	<b>54</b>
4.12.1 Reserved FadeCurveShapeIDs.....	54
<b>4.13 DspParameterID Type Format and Value Definitions .....</b>	<b>55</b>
4.13.1 Reserved DspParameterIDs .....	55
<b>4.14 VLQ Field Format.....</b>	<b>56</b>
<b>5. References.....</b>	<b>57</b>



# 1. Introduction

Interactive XMF (iXMF) is a file format for interoperable interactive audio content. It assumes that iXMF players will have a certain structure, certain behaviors, and a certain relationship to the host application, which is typically a game. This document specifies the file format in full detail, and characterizes the player in enough detail for it to be implemented in any number of ways.

The authors of this specification fully expected that iXMF will be used by some people and companies as a directly playable run-time content format, and by other people and companies as a development-time content interchange format that is imported into or exported from interactive audio content authoring tools and run-time engines that also read and write other content formats. It is to be expected that when iXMF content is used in non-iXMF players and tools, some of the functionality in the content is likely to be lost.

iXMF uses the XMF Meta File Format [2]; this specification defines iXMF as a new XMF File Type.

This Introduction chapter presents high-level overview material that anyone working with iXMF should read.

## 1.1 Technical Overview

This section describes the technical basics of the iXMF data format and how iXMF players use it to produce adaptive soundtracks for games and other interactive Host programs.

iXMF is based on the XMF Meta File Format [2]. In the XMF Meta File Format, a file can have an internal structure of "folders" and "files", much like a computer file system or the tree structure within a ZIP file.

iXMF is a file format that contains audio content for adaptive audio. Unlike the Standard MIDI File format, for example, iXMF can contain any standard audio file format within the file. For instance, MIDI sequencer data and audio data can both be contained in one iXMF file. iXMF uses the XMF Meta File Format to bundle together related resources that encode high-level semantic entities such as playable MediaFiles, media Chunks, Cues, and Transitions. Also included within the iXMF file are Scripts, which operate on these audio resources to provide real-time adaptive audio.

An iXMF file may be described in terms of the following abstractions, all represented as data structures stored inside the file:

- An iXMF file is a collection of any number of named Cues
- A Cue is a collection of Media Chunks, plus some Scripts (rules governing how they are played)
- A Media Chunk is a contiguous region in a playable MediaFile

Within an iXMF file, a root folder contains a subfolder with all the Cue definitions. Cues are events that correspond to predefined actions in the iXMF file. The host requests these events by communicating with the Soundtrack Manager (a middleware layer) to play audio and operate on data. Cues most often represent playable soundtrack elements that produce audio, but also may be used to operate on data that does not produce audio such as to set a variable or execute a callback from the host. The managing of non-audio data is the fundamental means of dynamically adapting audio in the game.

A Media Chunk is a playable media file (MIDI, audio, etc.) or a defined region of a playable media file. You can think of a Media Chunk as a section of a composition. A MediaChunk structure refers to one included playable media file. Any number of MediaChunks can refer to the same playable media file, and their start and end regions can overlap. If a media type (WAV, MIDI, etc.) provides a mechanism for looping and/or branching within a single media chunk, then that mechanism is available within the iXMF system.

Playable audio and MIDI files are grouped into one iXMF subfolder, called MediaFiles. This subfolder includes all playable media files needed to present the intended soundtrack in the product.

To produce adaptable audio, Scripting functions are used to control how the Media Chunks are played and how they perform operations in response to host Cue requests. The Scripts are created by the audio artist during development of the audio, and are included in the iXMF file.

In an iXMF file, there are many variables that can be optionally set or omitted, at any of three levels of scope: the whole-file scope (highest level) is called Global, there is a Cue scope that is visible to all Cues of the same type, and per-Cue-instance scope (lowest level) called Local. A variable setting at a higher scope will be visible to all lower levels; but a property setting at a lower level will not be visible to any higher levels. This approach allows defaults to be roughed-in globally, and then fine-tuned at an increasingly local level until done.

A special built-in mechanism is provided for Cues mainly consisting of chains of Chunks, supporting Transitions with precise overlap, crossfade, and synchronization options. To use this mechanism, a Cue must include a CueStart Script with a SetNextChunk() statement. This will cause the IXE to start the first Chunk in the chain. For non-interactive chains, simply set each Chunk's DefaultNextChunk field to point to the next Chunk in the desired chain. Or, for a Cue that makes adaptive, runtime decisions about the order of Chunks, a content creator would provide appropriately written Scripts for that Cue which would examine the values of iXMF variables and then use the SetNextChunk() statement to pick the desired Chunk to play next.

#### 1.1.1 iXMF File Identification

An iXMF File is internally identified by XMF File Type [2], and externally identified by filename extension, and optionally by MIME type and Mac OS File Type:

<b>XMF File Type</b>	4
<b>Filename Extension</b>	.ixmf
<b>Mac OS File Type</b>	'iXMF'
<b>MIME Media Type</b>	'audio/ixmf' for audio-only content

#### 1.1.2 Playback System Minimum Capability Requirements

The specification of minimum capability requirements for playback systems supporting the iXMF file format (for example number of simultaneous Cues or simultaneously playing Chunks) is beyond the scope of this document. The scope of this document is limited to describing the file format common to all iXMF files. The authors of this specification intend that any playback system minimum capability requirements be addressed in separate documents, such as profiles, defining the specific sets of functionality and/or levels of performance needed for particular iXMF application areas; for example: advanced game consoles, personal computers, media player appliances, and mobile devices of various complexity levels. A metadata mechanism is provided for encoding within an iXMF file which profile(s) the file conforms to.

The runtime API of the IXE is also outside of the scope of this document. However, in all cases, if an attempt to load any given iXMF file, or to make any particular runtime use of loaded iXMF data, would cause the IXE to exceed its capabilities, it is recommended that the IXE not take the requested action and return an appropriate and descriptive indication of the error condition. Such errors should in most cases be avoidable if careful planning and targeting of the intended playback system are done.

## 1.2 Handling of Nonstandard Contents

Because iXMF files are generalized containers, applications and devices that read iXMF files must expect that there may be additional contents beyond those specified in this document, and not fail when such unrecognized contents are encountered. Unrecognized contents may include XMF MetaDataItems not mentioned in the iXMF standard, nonstandard resource formats, including but not limited to media file formats, and ExtensionAreas within standard iXMF resources. Unrecognized contents may also include undefined values in standard identifier enumerations such as ExpressionOpcodeIDs, OpcodeSpaceIDs, Opcode IDs, narrowingRuleIDs, sortTypeIDs, pickTypeIDs, weightingCurveIDs, fadeCurveShapeIDs, dspParameterIDs, mediaHandlingTypeIDs, transportStateIDs, narrowingInclusionRuleIDs, fadeDirectionIDs, muteOrUnmuteIDs, ItemType in a PositionRule, PositionType in a TimePosition, and syncTypeIDs,

While it may be acceptable for an iXMF validator tool to flag and automatically remove items that are not supported for a particular iXMF player profile, iXMF authoring tools should in general preserve all unrecognized contents, as it may represent vendor-specific or player-specific extension data, which may be critical from a creative perspective. No tool should ever attempt to modify any unrecognized data component in an iXMF file.

## 1.3 Time in iXMF

An important focus of iXMF is the precise placement of events in time. Several kinds of time must be accommodated.

### 1.3.1 Marker Positions in Audio and MIDI Media

A marker's time is wholly defined by its time position within the media file. In iXMF, markers are referred to only by name, i.e. the short ASCII string they contain. If a Script with the same name is present within a Cue, or in a SharedScripts folder in the Cue's namespace, it will be executed whenever playback passes through that marker. By including an InvokeCallback() statement in the called Script, the audio artist can cause the IXE to make a callback to the game or other host application when playback passes through a marker.

### 1.3.2 Musical Time Positions

In iXMF, musical time is based on the Standard MIDI File mechanism of Ticks, plus Beats and Bars if time signature information has been set. A specific musical time may always be defined by a Tick number offset from the start of the musical piece. If a time signature has been set, then a specific musical time may also be defined by the combination of a Bar number, a Beat number within the bar, and a Tick number within the Beat.

### 1.3.3 Musical Tempo

Note that musical time is different from linear time in seconds in that it is elastic, based on the tempo at which the music is played. The same musical time position in Ticks (or Bars, Beats, and Ticks) will occur sooner when the music is played quickly, and later when the music is played slowly. Tempo is generally expressed in Beats per Minute.

### 1.3.4 Algorithmic Descriptions of Musical Time Positions

For some functions iXMF uses an algorithmic description of musical time in Ticks, called a PositionRule. A PositionRule is made up of one or more PositionRuleItems, each of which can express either one specific musical time, or a repeating interval. As a result, a PositionRule can express a single musical time position, or an infinite number of musical time positions at a simple repeating interval, or any combination of any number of either.

### **1.3.5 High Resolution Linear Time (\*\*RATE TBD\*\*)**

The IXE maintains a high resolution linear time clock in **\*\*RATE TBD\*\***. Several Script statements take high resolution linear time Duration parameters, and the GetTime() statement returns the current IXE high resolution clock time.

### **1.3.6 Low Resolution Linear Time (Seconds)**

In a few places iXMF uses rough time estimates in terms of seconds. These are represented as integers in whole seconds.

### **1.3.7 Sample Frame Offsets in Audio Media**

The start and end boundaries of every audio Chunk within its MediaFile are set in terms of sample frame offsets relative to the start of the file, by means of the StartOffset and EndOffset fields in the ChunkDefinition resource. These offsets are integers of variable length.

### **1.3.8 Tick Offsets in MIDI Media**

The start and end boundaries of every MIDI Chunk within its MediaFile are set in terms of Tick offsets from the start of the file, by means of the StartOffset and EndOffset fields in the ChunkDefinition resource. These offsets are integers of variable length.

## 2. Terminology

**Adapter Layer (AL)** – Portable implementations of iXMF Engine code can be made possible by defining an Adapter Layer (AL) interface for sending a small set of simple audio commands to the platform's Native Playback API's. Porting an iXMF implementation to any given platform could be accomplished by implementing the AL for that platform. This interface could also send information back to the Host via Callback Functions and shared variables. AL concepts are detailed in Appendix D: Adapter Layer.

**Auditioning Tool** – A software program that, under user control, interfaces with its own copy of the iXMF Engine in the same manner as the game (or other Host application) will interface with its copy of the iXMF Engine, in order to test the playback and adaptive responses of the iXMF content. An Auditioning Tool may in some cases run simultaneously with the Editor and/or the Host, allowing for real-time editing.

**Callback Function** – A Host executable function registered by name with the iXMF Engine, and invoked by Cues via either a Script statement, or a marker placed in the playable media.

**Callback Script** – A Cue-level Script invoked when a Native Media Player encounters a marker during playback of its Chunk.

**Channel** – For a stereo audio file, the Left and Right audio signals are considered separate channels. For a 5.1 multitrack audio file, the Left Front, Center Front, Right Front, Left Rear, Right Rear, and Low Frequency audio signals are all considered separate channels.

**Chunk** – A playable media file or file image, or a defined contiguous region of a playable media file or file image. In other words, limiting it to audio and MIDI content: an audio file or file image, a contiguous time region within an audio file or file image, a Standard MIDI File or file image, or a contiguous time region within a Standard MIDI File or file image.

**Cue** – A numbered (and optionally named) event that the Host signals to the iXMF Engine (IXE). The IXE responds with a corresponding, predefined action that was designed by the audio artist. A Cue can furnish any combination of operations that the IXE is capable of performing. For interactive applications, Cues will usually directly represent playable soundtrack elements; however, it is also possible to create both Cues that do much more than that, and Cues that don't directly result in any audible change in the soundtrack. For example, a Cue can merely set a variable and then end, or it can load (or otherwise prepare) a piece of media, or it can invoke a Host Callback Function.

**Editor** – A software program that allows the audio artist to create and manipulate the contents of iXMF Files.

**Host** – The game, Editor, other authoring application, or Auditioning Tool in which the iXMF Engine is running.

**IXE** – See **iXMF Engine**.

**iXMF** – Interactive XMF, the open standard interoperable interactive audio file format that this document defines.

**iXMF Engine (IXE)** – A software middleware layer that reads iXMF Files and provides games (or other Host applications) the same advanced interactive audio services functionality on any platform, with optional platform-specific extensibility mechanisms. An IXE receives requests for Cues from its Host, and handles them by supplying any number of Native Media Players with sound media content stored in iXMF Files, and coordinating the players' operation according to instructions prepared by an audio artist. These instructions are included in the same iXMF File as the playable sound media.

**iXMF File** – A file conforming to the iXMF format.

**Native Media Player** – A mechanism provided by a Native Playback API for playing one Chunk, for example an audio file player or Standard MIDI File player. If an Adapter Layer is used, the Native Media Player is directly driven by the Adapter Layer.

**Native Playback API** – Platform-specific media playback software (typically just audio and MIDI) driven by the Adapter Layer. It may be either part of the platform OS or provided by a third party.

**Script** – A simple program created by the audio artist, for the purpose of controlling the behavior (including adaptive response) of a Cue. Scripts are invoked by name, and a Cue may have any number of Scripts. Scripts are stored in the iXMF File in binary (opcode) form.

**Track** – Depending on the media type, a Chunk may have multiple, individually selectable or controllable Tracks intended to play in parallel. Transport controls, such as play and stop, affect all of a Chunk's Tracks, so sync among the Tracks is always maintained. For example, an audio file or file image may have multiple channels, each of which may be individually muted, faded, or processed with DSP. Another example is a Standard MIDI File in format 1, which may have any number of parallel Tracks of timed MIDI events and meta-events.

**Transition** – The introduction of a new Chunk into a running soundtrack. A Transition has several properties, including the sync relationship of the new Chunk to the already-running Chunks, relative level, fade-out or continuation of the old Chunk(s), and fade-in (if any) of the new Chunk.

## 3. Scripting Language

This section describes the semantics of the scripting language, the built-in statements (opcodeSpace 0x00), and the optional debugging statements (opcodeSpace 0x01). The binary representation of Scripts is described in the chapter 6, Data Structures.

### 3.1 Overview

Any Cue can include any number of numbered, and optionally named, Scripts.

#### Examples:

Typical ScriptIDs and names might look like the following:

```
8: CueIxmFUnload  
666: MusicHasEnded
```

A Script contains a sequence of any number of script statements in binary form, which in an iXMF authoring tool may be presented to the content author as a simple programming language with a simple expression syntax. iXMF Editors may optionally be designed to hide the details of the programming language from the content creator in various ways, however all Scripts are saved using the same binary representation. Simple control flow statements (conditional branch and labels) are available for looping and conditional execution of statements.

#### 3.1.1 Extensibility Mechanism

The binary representation of the scripting language can be extended with new statements. When extension statements are installed, they can be freely intermixed with the built-in statements. Extensions can include both future extensions of the interoperable standard and proprietary/non-interoperable extensions. See chapter 6, Data Structures, for details.

### 3.2 ScriptIDs and Names

Every Script has a numeric ScriptID, and may also be given a name. Certain ScriptIDs and names are predefined by the iXMF standard and will be called by the IXE whenever particular conditions occur in a Cue. Examples are 1: CueStart and 8: CueIxmFUnload.

All other ScriptIDs and names are free for the content creator to define, and will be called whenever:

- Another Script uses the `CallScript()` statement to call that ScriptID as a subroutine;
- Another Script uses the `LaunchScript()` statement to run that ScriptID as a parallel process; or
- A marker callback referencing that Script name is executed in the Cue.

The only case in which a Script must have a name is when it is intended to be called via marker callback.

Examples of ScriptIDs and names created by a content creator include 34: `ThisIsMyScriptName` and 666: `MusicHasEnded`.

#### 3.2.1 Predefined ScriptIDs and Names

When used in a Cue, the following ScriptIDs and names are predefined and will be called by the IXE whenever the indicated condition occurs in that Cue:



Script ID	Script Name	IXE Calls Script When	Required / Optional
0	CueIxmLoad	The iXMF File containing the Cue is first loaded	Optional
1	CueStart	The Cue is first started by the IXE	Required
2	ChangeOver	Any ChangeOver point for any Transition in this Cue is reached	Optional
3	ChunkEnd	Any Native Media Player in this Cue encounters the end of any playable Chunk	Optional
4	CueCancel	The Cue is canceled by the IXE	Required
5	CueEnd	The Cue naturally ends (by playing through to the end of the last playable Chunk)	Optional
6	CueStop	The Cue is stopped by the IXE	Optional
7	CueLastInstanceTeardown	The last running instance of the Cue has finished. No instance of the Cue is running any more,	
8	CueIxmUnload	The iXMF File containing the Cue is about to be unloaded by the IXE	Optional
9	NoSuchScript	A desired ScriptID or name was not found within the Cue and not found found within the SharedScripts folder.	Optional
10-19	(Reserved for future definition by IASIG/MMA)		
20 - up	(any other name is available for user Scripts)	<ul style="list-style-type: none"> <li>• Another Script uses the <code>CallScript()</code> statement to call that ScriptID as a subroutine;</li> <li>• Another Script uses the <code>LaunchScript()</code> statement to run that ScriptID as a parallel process; or</li> <li>• A marker callback referencing that Script name is executed in the Cue</li> </ul>	Optional

### 3.2.2 Special Treatment of CueStart Script

A Cue's `CueStart` Script is always called when the IXE starts an instance of the Cue. When execution of the `CueStart` Script completes, the IXE examines the value of the `ChunkID` held in `NextChunk`, and (so long as `NextChunk` does not contain zero) loads a Player with the indicated Chunk and starts it playing. (If another Cue in the same `SyncGroup` is already running, the start may be delayed until the next synch point is encountered.) Therefore, a Cue will not play any media unless the audio artist includes a `SetNextChunk( ChunkID )` Script statement, with the desired `ChunkID`, in that Cue's `CueStart` Script. A Cue that is meant to perform other non-playback operations should not use the `SetNextChunk()` statement in its `CueStart` Script.



## 3.3 iXMF Variables

iXMF provides variables, and Script statements for accessing them, and expressions for manipulating them, for several reasons:

- To support programming within a single Script
- To support programming within a single running instance of a Cue
- To support communication between multiple running instances of a single Cue
- To support communication between multiple simultaneously loaded Cues and/or their running instances
- To support communication between active Cues and/or instances, and the host game or other application controlling the IXE.

All iXMF variables may be set or accessed only by running instances of Cues (by means of Script Statements in their Scripts); and by the host game or other client application controlling the IXE.

### 3.3.1 Variables Data Format

All iXMF variables are stored internally as signed 32-bit integers.

When an iXMF variable is interpreted as a Boolean, logical FALSE is represented by the integer value 0 (0x00000000) and any other value represents logical TRUE. To set any iXMF variable to Boolean value of logical TRUE, it is recommended to use the integer value 1 (0x00000001).

### 3.3.2 Number, Kinds, and Scope of Variables

There are three kinds of iXMF variables: Global variables, Cue variables, and Local variables.

There are always exactly 32 Global variables. There are 32 Cue variables for every currently loaded Cue. There are 32 Local variables for every running Cue instance. In the runtime binary form of Script statements, variables are selected not by name but by a 2-part identifier: Type (Global, Cue, or Local) and Index.

Each running instance of a Cue (and all of that Cue's Scripts) may set or get all 32 Global variables, only the 32 Cue variables that belong to that Cue, and only the 32 Local variables that belong to that particular instance of that Cue.

The host game or other client application controlling the IXE has access to all iXMF variables at all times.

#### 3.3.2.1 Global Variables

The 32 Global variables may be set or get by any running instance of any Cue (and all their Scripts); and by the host game or other client application controlling the IXE.

Global variables come into existence at the time the IXE is initialized, with a value of 0. To initialize Global variables to any other value, include an `IxmLoad` Script for at least one of the Cues in at least one of your iXMF files. Once a Global variable is set to a given value, it retains that value until replaced by a different value or the IXE is dismantled. All Global variables cease to exist at the time the IXE is dismantled.

#### 3.3.2.2 Cue Variables

The 32 Cue variables belonging to each loaded Cue may only be set or get from any running instance of that same Cue (and all their Scripts); and by the host game or other client application controlling the IXE. Running instances of one Cue may not access the Cue variables of a different Cue.

Cue variables come into existence at the time the IXE loads the Cue to which they belong from an iXMF file, with a value of 0. To initialize a particular Cue's Cue variables to any other values, include an `IxmLoad` Script for that Cue. Once a Cue variable is set to a given value, it retains that value until replaced by a different value, or until the IXE unloads that Cue. All Cue variables for a given Cue cease to exist at the time the IXE unloads that Cue.

### **3.3.2.3 Local Variables**

The 32 Local variables belonging to each running instance of a given Cue may only be set or get from that same running instances of that same Cue (and all its Scripts); and by the host game or other client application controlling the IXE. Other running instances of the same Cue may not access a different instance's Local variables. Running instances of one Cue may not access the Local variables of any instance of a different Cue.

Local variables come into existence at the time the IXE creates the running Cue instance to which they belong, with a value of 0. To initialize a particular Cue instance's Local variables to any other values, include a `CueStart` Script for that Cue. Once a Local variable is set to a given value, it retains that value until replaced by a different value, or until the IXE dismantles that running Cue instance.

## **3.4 Expressions in Script Statements**

In the iXMF Engine, expressions are evaluated by a very simple stack-based expression engine, driven by a small set of expression opcodes. Each expression opcode is 1 byte long. These expression opcodes are wholly separate from the scripting language opcodes.

### **3.4.1 The Expression Stack and Expression Engine**

Expressions are used in the script statements `ConditionalBranch()` and `EvaluateExpression()`. It's up to the authoring tool designer to decide how (if at all) to display expressions to the audio artist, and how to edit them, including how to display any grouping of operations according to parentheses and/or operator precedence rules. However, all iXMF authoring tools emit the same kind of data (tiny programs) to drive the expression engine at runtime, and all iXMF implementations will produce the same results.

When an expression is used in a `ConditionalBranch( expression, label )` script statement, the branch is taken if the result left on the top of the stack is logical `TRUE`, or not taken if logical `FALSE`. When an expression is used in a `var = EvaluateExpression( expression )` script statement, the result is left on the top of the stack, where it can be used for any purpose by the next script statement, for example it can be assigned to any iXMF variable or Cue property.

Expression opcodes are defined for the just basic set of C language operators:

- **2-operand ('binary') operators:**

Arithmetic: +, -, \*, /, %

Relational: >, >=, <, <=, ==, !=

Logical: &&, ||

Bitwise Logical: &, |, ^, <<, >>

- **1-operand ('unary') operators:**

Logical: !

Increment & Decrement: ++, --, ++n, --n

Bitwise Logical: ~

All operands on the stack are 32-bit integers, the same size and format as all the global, Cue, and local variables. Except for `operatorPushLiteral`, all operators that take operands pop their operand(s) from the top one or 2 positions of the stack, and produce a result, which is pushed back onto the top of the stack. For example, the `+` operator takes 2 operands by popping the top 2 stack positions, adds them, and puts the sum back on the stack, leaving the stack one item shorter than before.

The expression opcode `operatorPushLiteral` enables the use of constants in expressions. This is the only opcode with an operand that is stored directly in the opcode stream, rather than on the stack. The operand is a 32-bit value like any iXMF variable.

#### Example 1:

An iXMF authoring tool might display a given expression using constants as:

( 7 \* 8 )

and that expression would be encoded as a 3-operation expression program:

```
operatorPushLiteral 7      // stack is now { 7 }
operatorPushLiteral 8      // stack is now { 8, 7 }
operatorArithmeticMultiply // stack is now { 56 } which is the final result of the
expression.
```

The three 'pushVariable' opcodes allow the current value of any of the iXMF variables (Local, Cue, or Global) to be used in expressions. These opcodes pull the value on the top of the stack and use it as index for the desired kind of variable, then push the variable's value onto the stack.

#### Example 2:

An iXMF authoring tool might display a different expression as:

( localVariable[ 2 ] + 3 )

and that can be encoded as a 4-operation expression program:

```
operatorPushLiteral 2      // stack is now { 2 }
operatorPushLocalVariable // assuming localVariable[ 2 ] holds 7, stack is now { 7 }
operatorPushLiteral 3      // stack is now { 3, 7 }
operatorArithmeticAdd      // stack is now { 10 } which is the final result of the
expression.
```

#### Example 3:

A more complex expression that an iXMF authoring tool might display thus:

( ( localVariable[ 2 ] + 3 ) \* ( globalVariable[ 17 ] / 5 ) ) % 23

can be encoded as an 11-operation program:

```
// ( globalVariable[ 17 ] / 5 )
operatorPushLiteral 5      // stack is now { 5 }
operatorPushLiteral 17     // stack is now { 17, 5 }
operatorGetGlobalVariable // assuming globalVariable[ 17 ] holds 20,
                           // stack is now { 20, 5 }
operatorArithmeticDivide   // stack is now { 4 }

// ( localVariable[ 2 ] + 3 )
operatorPushLiteral 3      // stack is now { 3, 4 }
operatorPushLiteral 2      // stack is now { 2, 3, 4 }
operatorGetLocalVariable   // if localVariable[ 2 ] holds 5, stack is now { 5, 3, 4 }
operatorArithmeticAdd      // stack is now { 8, 4 }

// ( ( localVariable[ 2 ] + 3 ) * ( globalVariable[ 17 ] / 5 )
operatorArithmeticMultiply // stack is now { 32 }

// % 23
operatorPushLiteral 47     // stack is now { 47, 32 }
operatorArithmeticModulo   // stack is now { 15 } which is the final result
                           // of the expression.
```

### 3.4.2 ExpressionOpcodeIDs

ExpressionOpcodeIDs are 1 byte long. The following expression opcodes are defined:

Val	Name	Suggested Symbol	Operation Performed
<b>Data Source Operators:</b>			
1	operatorPushLiteral	a	push( literal ) <b>Note:</b> no stack operands, but a 32-bit operand immediately follows the opcode
2	operatorPushLocalVariable	localVariable[ a ]	push( localVariable[ pop() ] )
3	operatorPushCueVariable	cueVariable[ a ]	push( cueVariable[ pop() ] )
4	operatorPushGlobalVariable	globalVariable[ a ]	push( globalVariable[ pop() ] )
<b>Two-Operand ('Binary') Operators</b>			
5	operatorArithmeticAdd	a + b	push( pop() + pop() )
6	operatorArithmeticSubtract	a - b	push( pop() - pop() )
7	operatorArithmeticMultiply	a * b	push( pop() * pop() )
8	operatorArithmeticDivide	a / b	push( pop() / pop() )
9	operatorArithmeticModulo	a % b	push( pop() % pop() )
10	operatorRelationalGreater	a > b	if( pop() - pop() ) > 0 push( TRUE ) else push( FALSE )
11	operatorRelationalGreaterOrEqual	a >= b	if( pop() - pop() ) >= 0 push( TRUE ) else push( FALSE )
12	operatorRelationalLess	a < b	if( pop() - pop() ) < 0 push( TRUE ) else push( FALSE )
13	operatorRelationalLessOrEqual	a <= b	if( pop() - pop() ) <= 0 push( TRUE ) else push( FALSE )
14	operatorRelationalEqual	a == b	if( pop() - pop() ) == 0 push( TRUE ) else push( FALSE )
15	operatorRelationalNotEqual	a != b	if( pop() - pop() ) != 0 push( TRUE ) else push( FALSE )
16	operatorLogicalAnd	a && b	push( logical( pop() ) && logical( pop() ) )
17	operatorLogicalOr	a    b	push( logical( pop() )    logical( pop() ) )
18	operatorBitwiseLogicalAnd	a & b	push( pop() & pop() )
19	operatorBitwiseLogicalOr	a   b	push( pop()   pop() )
20	operatorBitwiseLogicalXor	a ^ b	push( pop() ^ pop() )

21	operatorBitwiseLogicalLeftShift	a << b	push( pop() << pop() )
22	operatorBitwiseLogicalRightShift	a >> b	push( pop() >> pop() )
<b>One-Operand ('Unary') Operators:</b>			
23	operatorLogicalNot	!a	push( !logical( pop() ) )
24	operatorIncrement	++a	push( pop() + 1 )
25	operatorDecrement	--a	push( pop() - 1 )
26	operatorBitwiseLogicalInvert	~a	push( ~pop() )

## 3.5 Built-In Statements: OpcodeSpace 0x00

The following statements must be built into all IXE implementations, using OpcodeSpace 0x00. Statements that return a value, as indicated below by '**var =**', may write the return value to any one Global, Cue, or Local variable. This destination is indicated by a 1-byte **var** parameter, as described in section 3.5.4 Format of "var" Parameters.

### 3.5.1 Statements

This section documents all opcodes in OpcodeSpace 0x00. The statements are grouped by function.

#### 3.5.1.1 Global

**void SetGlobalDefaultFade( FadeDirectionID, FadeCurveShapeID, Duration )**

Sets the global default fade-in or fade-out settings for all Cues.

**void SetMasterDspParameter( DspParameterID, Value )**

Sets the indicated master DSP parameter. See DSP ParameterIDs below. Applies to master audio mix. The format of the value parameter depends on the target DspParamID.

#### 3.5.1.2 Cue

**void CallCue( CueID )**

Causes the IXE to run a new instance of the indicated Cue. This Statement allows one Cue to invoke other Cues. The new Cue will run in parallel to this Cue.

**var = GetCueID( void )**

Returns (to any Local, Cue, or Global variable) the CueID of this Cue. This functionality is most useful in shared scripts, where it is not possible to know the CueID at authoring time.

**var = GetCueParameter( void )**

Returns (to any Local, Cue, or Global variable) the 32-bit parameter passed to this Cue instance in the IXE /Host API CallCue() call.

**void SetCueMaxInstanceCount( CueID, CountMax )**

Sets the maximum number of instances of the indicated Cue that the IXE will allow to exist at the same time.

**var = GetCueMaxInstanceCount( CueID )**

Returns (to any Local, Cue, or Global variable) the maximum number of instances of the indicated Cue that the IXE will allow to exist at the same time.

**var = GetCueInstanceCount( CueID )**

Returns (to any Local, Cue, or Global variable) the current number of instances of the indicated Cue.

**void SetPreBufferDuration( DurationInSeconds )**

For all Chunks for this Cue that are used in streamed Native Media Players (see media handling type), sets the player's pre-buffer size to hold enough data to play at natural speed for the indicated amount of time.

**void SetCueCancelScript( ScriptID )**

Causes the indicated Script to be called when the Host calls `CancelCue()` (rather than always using the Cue's `DefaultCancelScript`).

**void ReleaseCue( void )**

Frees all resources currently in use by this Cue. If any Native Media Players are not already stopped, this stops them first.

**void Prebuffer( void )**

For Chunks used in streaming Native Media Players, causes the streaming buffer to be preloaded, using the current duration: either the inherited setting, or as previously set via `SetPreBufferDuration( t )` statement.

**void InvokeCallbackFunction( CallbackID )**

Invokes the indicated API-level callback function. To function correctly, the game or other host application must have previously used the IXE API to register a callback handler function. The callback handler function will generally receive a pointer or handle to this Cue instance, and pointers or handles to all Native Media Players used by this Cue instance.

**void SetCueInstancePriority( Priority )**

Sets the priority of this instance of the Cue.

**var = GetCueInstancePriority( void )**

Returns (to any Local, Cue, or Global variable) the current priority of this instance of the Cue.

**void SetCuePriority( CueID, Priority )**

Sets the priority of the indicated Cue. All subsequently started instances of the Cue will have this priority unless an instance of that Cue sets a different priority with `SetCueInstancePriority()`.

**var = GetCuePriority( CueID )**

Returns (to any Local, Cue, or Global variable) the current priority of the indicated Cue.

**void SetCueDefaultFade( CueID, FadeDirectionID, FadeCurveShapeID, Duration )**

Sets the default fade-in or fade-out settings for the indicated Cue.

**void SetCueDspParameter( CueID, DspParamID, Value )**

Sets the indicated DSP parameter for the indicated Cue. Applies to all Chunks and Native Media Players used for the indicated Cue. The format of the value parameter depends on the target `DspParamID`.

**void SetCueSyncGroup( CueID, SyncGroupID )**

Associates the indicated Cue with the indicated SyncGroup. A SyncGroupID of 0 means no SyncGroup.

**void SetCueMixGroup( CueID, MixGroupID )**

Associates the indicated Cue with the indicated MixGroup. A MixGroupID of 0 means no MixGroup.

**void SetCueMediaHandling( CueID, MediaHandlingTypeID )**

Sets the media handling method for the indicated Cue.

**void PreloadCueMedia( CueID )**

Preloads all media files and/or file images (or relevant section thereof) for all Chunks used by this Cue. For all-in-memory Chunks, this means wholly loading into memory; for streamed Chunks, this means prebuffering.

**void UnloadCueMedia( CueID )**

Unloads all media files and/or file images used for this Cue. For Chunks, the semantic is not Unload but 'Release' – since all Chunks using a playable media file or file image must be released before the data can be unloaded.

**3.5.1.3 Variables and Functions**

**void SetVariable( VariableTypeID, VariableID, Value )**

Sets the indicated Local, Cue, or Global variable to the indicated value. The destination variable is determined by the combination of `variableTypeID` (Local, Cue, or Global) and `variableID` (0-31 index within the indicated `variableType`).

**var = GetVariable( VariableTypeID, VariableID )**

Returns (to any Local, Cue, or Global variable) the current value of the indicated iXMF Global, Cue, or Local variable.

**var = EvaluateExpression( Expression )**

Evaluates the indicated expression and returns (to any Local, Cue, or Global variable) the result. See section 3.4 Expressions in Script Statements for expression syntax and binary representation.

**var = GetTime( void )**

Returns (to any Local, Cue, or Global variable) the current master time of the IXE.

**var = GetRandom( MinValue, MaxValue )**

Returns (to any Local, Cue, or Global variable) a random value between `minValue` and `maxValue`.

**3.5.1.4 Script Control Flow**

**void Label( LabelID )**

Target for `ConditionalBranch()` statements with matching values of `labelID`.

**void ConditionalBranch( Expression, LabelID )**

If the indicated expression evaluates to logical `TRUE`, execute a branch to the indicated label. See section 3.4 Expressions in Script Statements for expression syntax and binary representation.

**void TimeDelay( Duration )**

Wait for the indicated amount of time before proceeding to the next Script statement. Duration is in terms of IXE master time units.



**void CallScript( ScriptID )**

Call the indicated Script as a subroutine. The calling Script will not proceed to its next Script statement until the called Script completes.

**void LaunchScript( ScriptID )**

Start the indicated Script running as a parallel process. Proceed immediately to the next Script statement without waiting for the launched Script to complete.

**3.5.1.5 Chunk Pool and Chunk Sequencing**

**var = GetTotalPoolSize( void )**

Returns (to any Local, Cue, or Global variable) the total (unnarrowed) number of Chunks in this Cue instance's pool.

**void IncludeInNarrowing( ChunkID, NarrowingInclusionRuleID )**

Overrides all narrowing criteria including 'Select All'.

**void NarrowPool( NarrowingRuleID, NarrowingParameter )**

Create a subset of this Cue instance's ChunkPool. Meaning of narrowingParameter depends on the narrowingRuleID being used.

**var = GetNarrowedPoolSize( void )**

Returns (to any Local, Cue, or Global variable) the number of Chunks in this Cue instance's narrowed pool.

**void AddChunkToNarrowedPool( ChunkID )**

Adds the indicated Chunk to the already narrowed pool, independent of the ordinary narrowing mechanism. The position of the newly added Chunk within the narrowed pool is not defined until a Sort statement is executed.

**void SetChunkOrderTag( ChunkID, OrderTag )**

OrderTag is a user field in the Chunk structure, used for controlling sort and pick operations. This opcode allows the DefaultOrderTag value to be overridden when desired.

**var = GetChunkOrderTag( ChunkID )**

Returns (to any Local, Cue, or Global variable) the ChunkOrderTag for the indicated Chunk.

**void SetChunkLRUOrder( ChunkID, LruOrder )**

Each Chunk's LRUOrder property is ordinarily set dynamically at runtime by the IXE based on order in which Chunks are played. This statement allows the audio artist to alter the LRU number, which will perhaps most likely be used to move Chunks up in probability when desired.

**var = GetChunkLRUOrder( ChunkID )**

Returns the LRUOrder for the indicated Chunk.

**void Sort ( SortTypeID )**

Sort the narrowed subset of this Cue instance's ChunkPool according to the indicated type of sort. Any number of sort operations can be applied at any stage.



**void SetChunkPickWeight( ChunkID, PickWeight )**

Pick weights may optionally be used in the `Pick()` step when determining what `NextChunk` to play.

**var = GetChunkPickWeight( ChunkID )**

Returns the `ChunkPickWeight` for the indicated `Chunk`.

**void SetChunkPickWeights( MaxPickWeight, MinPickWeight, WeightingCurveID )**

Applies a weighting curve to this Cue instance's current narrowed/ordered pool. Can be applied at any time before or after ordering.

**var = Pick( PickTypeID, PickParameter )**

Pick one `Chunk` from the narrowed and sorted subset of this Cue instance's `ChunkPool`, and returns (to any Local, Cue, or Global variable) its `ChunkID`. The meaning of the `pickParameter` depends on the `pickTypeID`.

**void SetNextChunk( ChunkID )**

Designate one `Chunk` (from this Cue) as the `NextChunk`. Important: This overrides any previously set `NextChunk`.

**void SetNextTransition( TransitionID )**

Sets the transition to be used at this Cue's next changeover.

**3.5.1.6 Playback Control**

**void PlayChunk( ChunkID )**

Immediately allocates a Native Media Player, loads it with the indicated `Chunk`, and starts playback. The new player is associated with this Cue.

**void SetPlayerTransportState( TransportStateID )**

Puts this Cue's Native Media Players into play, pause, stop, or wait-for-sync state.

**var = GetPlayerTransportState( void )**

Returns (to any Local, Cue, or Global variable) the transport state of this Cue's Native Media Players.

**void Fade( void )**

Start fading all Native Media Players currently in use by this Cue out to zero over the duration, and using the curve, set by `SetChunkFade( FadeOut, ... )`.

**void ReleasePlayer( void )**

Frees all Native Media Players currently in use by this Cue for reallocation. If not already stopped, this stops them.

**void FadeAndReleasePlayer( void )**

Start fading all Native Media Players currently in use by this Cue out to zero over the duration, and using the curve, set by `SetChunkFade( FadeOut, ... )`. When each Native Media Player reaches zero volume, the Player is freed for reallocation.

### 3.5.1.7 Chunk

#### **void SetChunkFade( FadeDirectionID, FadeCurveShapeID, Duration )**

Sets the duration and curve of the fade-in or fade-out that `FadePlayer()` or `FadeAndReleasePlayer()` will start.

#### **void SetChunkPriority( ChunkID, Priority )**

Sets the priority of the indicated Chunk.

#### **var = GetChunkPriority( ChunkID )**

Returns (to any Local, Cue, or Global variable) the priority setting of the indicated Chunk.

#### **void SetChunkDspParameter( ChunkID, DspParamID, Value )**

Applies only to the indicated Chunk and its Native Media Player. A `chunkID` of 0 applies the DSP parameter setting to the ChunkID of the Chunk from which the Script was invoked. The format of the value parameter depends on the target `DspParamID`.

#### **void SetChunkSyncGroup( ChunkID, SyncGroupID )**

Associates the indicated Chunk with the indicated SyncGroup. A `SyncGroupID` of 0 means no SyncGroup.

#### **void SetChunkMixGroup( ChunkID, MixGroupID )**

Associates the indicated Chunk with the indicated MixGroup. A `MixGroupID` of 0 means no MixGroup.

#### **void SetTrackMute( ChunkID, TrackID, MuteOrUnmuteID )**

Mute (silence) or un-mute the indicated Track of the indicated Chunk.

#### **void SetChunkChunkGroup( ChunkID, ChunkGroupID )**

Associates the indicated Chunk with the indicated ChunkGroup.

#### **var = GetChunkChunkGroup( ChunkID )**

Returns (to any Local, Cue, or Global variable) the `ChunkGroupID` of the ChunkGroup with which the indicated Chunk is currently associated.

#### **void SetChunkMediaHandling( ChunkID, MediaHandlingTypeID )**

Sets the media handling method for the indicated Chunk.

#### **void SetChunkCancelScript( ScriptID )**

Causes the indicated Script to be called when the Host calls `CancelCue()` (rather than always using the Chunk's `DefaultCancelScript`).

#### **void SetChunkTempoMap( ChunkID, SmfChunkID )**

Associates a tempo map (which must be an SMF with conductor Track) with any audio or MIDI Chunk.

#### **void SetChunkEntryPointsRule( ChunkID, PositionRuleID )**

Sets the entry points rule to be used at this Cue's next changeover.

**void SetChunkExitPointsRule( ChunkID, PositionRuleID )**

Sets the exit points rule to be used at this Cue's next changeover.

**void PreloadChunkMedia( ChunkID )**

Preloads the media file or file image (or relevant section thereof) for the indicated Chunk. For all-in-memory Chunks, this means wholly loading into memory; for streamed Chunks, this means prebuffering.

**void ReleaseChunkMedia( ChunkID )**

Releases one reference to the indicated Chunk. When the media file (or file image) that the indicated Chunk uses is no longer referenced by any unreleased Chunks, the media file will be unloaded.

#### **3.5.1.8 ChunkGroup**

**void LoadMediaForChunkGroup( ChunkGroupID )**

Causes all media files for the indicated ChunkGroup to be loaded into memory in preparation for playback.

**void UnloadMediaForChunkGroup( ChunkGroupID )**

Causes all media files for the indicated ChunkGroup to be released, permitting subsequent automatic unloading from memory. For Chunks, the semantic is not Unload but 'Release' – since all Chunks using a playable media file or file image must be released before the data can be unloaded.

#### **3.5.1.9 MixGroup**

**void SetMixGroupDspParameter( MixGroupID, DspParameterID, Value )**

Sets the indicated DSP parameter for the indicated MixGroup. Applies to all Chunks and Native Media Players used for the indicated MixGroup. The format of the value parameter depends on the target DspParamID.

#### **3.5.1.10 SyncGroup**

**void SetSyncGroupDspParameter( SyncGroupID, DspParameterID, Value )**

Sets the indicated DSP parameter for the indicated SyncGroup. Applies to the sync group's master time clock. The format of the value parameter depends on the target DspParamID.

**var = GetSyncGroupBars( SyncGroupID )**

Returns (to any Local, Cue, or Global variable) the current musical position in Bars of the indicated SyncGroup.

**var = GetSyncGroupBeats( SyncGroupID )**

Returns (to any Local, Cue, or Global variable) the current musical position in Beats of the indicated SyncGroup.

**var = GetSyncGroupTicks( SyncGroupID )**

Returns (to any Local, Cue, or Global variable) the current musical position in Ticks of the indicated SyncGroup.

### 3.5.2 Parameter Formats

This section defines the data formats and value lists used as scripting statement parameters.

#### 3.5.2.1 CallbackID

16-bit unsigned integer. CallbackID of a CallbackDefinition resource in this Cue's namespace.

#### 3.5.2.2 ChunkGroupID

16-bit unsigned integer. ChunkGroupID of a ChunkGroup in this Cue's namespace. ChunkGroups are defined by the audio artist and are represented only by ChunkGroupIDs, not by any data structures in the iXMF file.

#### 3.5.2.3 ChunkID

16-bit unsigned integer. ChunkID of a ChunkDefinition resource in this Cue's namespace.

#### 3.5.2.4 CountMax

16-bit unsigned integer. Maximum number of instances allowed for the indicated Cue.

#### 3.5.2.5 CueID

16-bit unsigned integer. CueID of a Cue Definition resource in this Cue's namespace.

#### 3.5.2.6 DspParameterID

4-byte, 4-character code. See section 4.13 below DspParameterID Type Format and Value Definitions.

#### 3.5.2.7 Duration

32-bit unsigned integer, in IXE master clock ticks.

#### 3.5.2.8 DurationInSeconds

32-bit unsigned integer. Duration in whole seconds.

#### 3.5.2.9 Expression

Variable length. Expressions are created by audio artists and/or their Script development tools. See section 3.4 Expressions in Script Statements for format.

#### 3.5.2.10 FadeCurveShapeID

4 bytes, 4-character code. See section 4.12 FadeCurveShapeID Type Format and Value Definitions.

#### 3.5.2.11 FadeDirectionID

1 byte unsigned integer.

<b>fadeDirectionID Value</b>	<b>Fade Direction</b>
0	Fade In
1	Fade Out
2-127	Reserved for future definition by IASIG/MMA
128-255	Extension Area

#### 3.5.2.12 LabelID

16-bit unsigned integer. An ID of a position within the current Script that is intended to be the target of a ConditionalBranch() Script statement in the same Script. All LabelIDs within a Script must be mutually unique.

### 3.5.2.13 LruOrder

16-bit unsigned integer. A replacement for a Chunk's current ranking in the IXE's list of Least Recently Used Chunks.

### 3.5.2.14 MediaHandlingTypeID

1 byte unsigned integer.

mediaHandlingTypeID D Value	Media Handling Behavior	Notes
0	In-Memory, loaded from local file system	Whole media file is loaded into memory before playback starts
1	In-Memory, loaded from network	
2	Streamed, from local file system	Streaming via a small in-memory buffer
3	Streamed, from network	
4-127	Reserved for future definition by IASIG/MMA	
128-255	Extension Area	

### 3.5.2.15 MinValue, MaxValue

16-bit unsigned integer. Lower and upper bounds, respectively, on the value to be returned by the random number generator.

### 3.5.2.16 MixGroupID

16-bit unsigned integer. MixGroupID of a MixGroup in this Cue's namespace. MixGroups are defined by the audio artist and are represented only by MixGroupIDs, not by any data structures in the iXMF file.

### 3.5.2.17 MuteOrUnmuteID

1 byte unsigned integer.

<b>muteOrUnmuteID Value</b>	<b>Operation</b>
0	Unmute
1	Mute
2-127	Reserved for future definition by IASIG/MMA
128-255	Extension Area

### 3.5.2.18 NarrowingInclusionRuleID

1 byte unsigned integer.

<b>narrowingInclusionRuleID Value</b>	<b>Narrowing Inclusion Rule</b>	<b>Notes</b>
0	Always	Always include the Chunk in the Narrowed pool, even if it doesn't satisfy the chosen Narrowing Rule
1	Follow Narrowing Rule	Only include the Chunk in the Narrowed pool if it satisfies the chosen Narrowing Rule
2	Never	Never include the Chunk in the Narrowed pool, even if it satisfies the chosen Narrowing Rule
3-127	Reserved for future definition by IASIG/MMA	
128-255	Extension Area	

### 3.5.2.19 NarrowingParameter

16-bit unsigned integer. Generic numeric input to the narrowing operation. Interpretation depends on the NarrowingRuleID parameter.

### 3.5.2.20 NarrowingRuleID

1 byte unsigned integer.

<b>narrowingRuleID</b> value	<b>Narrowing Rule</b>	<b>Use of narrowingParameter</b>
0	Select All	(Not used – all Chunks in Cue are selected)
1	N Least Recently Used	N
2	Has Metadata Tag	MetadataTagID
3	Doesn't Have Metadata Tag	MetadataTagID
4	OrderTag > N	N
5	OrderTag < N	N
6	Select Chunk Group	ChunkGroupID
7	Exclude N Most Recently Used	N
8-127	Reserved for future definition by IASIG/MMA	
128-255	Extension Area	

### 3.5.2.21 OrderTag

16-bit unsigned integer. A custom order ranking to be assigned to a Chunk for later use in chunk pool narrowing operations.

### 3.5.2.22 PickParameter

16-bit unsigned integer. Generic numeric input to the pick operation. Interpretation depends on the PickTypeID parameter.

### 3.5.2.23 PickTypeID

1 byte unsigned integer.

pickTypeID value	Pick Rule	Use of pickParameter
0	Random	weightingCurveID (See Note below)
1	Index	Literal index into narrowed pool
2	Index From Local Variable	variableID for Local Variable
3	Index From Local Variable with Auto Decrement	
4	Index From Local Variable with Auto Increment	
5	Index From Cue Variable	variableID for Cue Variable
6	Index From Cue Variable with Auto Decrement	
7	Index From Cue Variable with Auto Increment	
8	Index From Global Variable	variableID for Global Variable
9	Index From Global Variable with Auto Decrement	
10	Index From Global Variable with Auto Increment	
11-127	Reserved for future definition by IASIG/MMA	
128-255	Extension Area	

**Note:** For pickTypeID value 0, the Random number function is automatically scaled to the size of the narrowed pool, and the pickParameter is interpreted as a weightingCurveID as described in section 3.5.2.37 WeightingCurveID.

#### **3.5.2.24 PickWeight, MaxPickWeight, MinPickWeight**

16-bit integer. A relative probability of being selected in a Pick operation, to be assigned to a Chunk. A weight of 1000 is considered ordinary. Larger weights increase the probability that the Chunk will be selected, and smaller weights decrease the probability.

#### **3.5.2.25 PositionRuleID**

16-bit unsigned integer. PositionRuleID of a PositionRuleDefinition resource in this Cue's namespace.

#### **3.5.2.26 Priority**

16-bit unsigned integer. A relative priority for use in internal IXE resource allocation decisions, to be assigned to a Chunk or Cue. A priority of 1000 is considered ordinary. Larger priorities increase the probability that the Chunk or Cue will receive contested resources and run, and smaller priorities decrease the probability. A priority of 0 means "don't care."

#### **3.5.2.27 ScriptID**

16-bit unsigned integer. ScriptID of a ScriptDefinition resource in this Cue's namespace. If no matching ScriptID is found, this Cue's ScriptID 9 (No Such Script) will be called instead, if present. If this Cue does not include such a Script, then the IXE will search for one in this iXMF file's SharedScripts folder. If not found there, the IXE will search for it in the SharedScripts folders of any and all other currently loaded iXMF files in this Cue's namespace, in an undefined order.

#### **3.5.2.28 SmfChunkID**

16-bit unsigned integer. A ChunkID that must refer to a Chunk Definition resource in this Cue's namespace which uses Standard MIDI File format, for use as a TempoMap.

#### **3.5.2.29 SortTypeID**

1 byte unsigned integer.

<b>sortTypeID value</b>	<b>Sort Rule</b>
0	As Is (no operation, do not sort)
1	By Least-Recently-Used (LRU) Order
2	By Order Tag
3	By ChunkID
4	By Name
5	Reverse List
6	Randomize
7	By First Use
8	By Duration
9	By Media Size
10	By Chunk Group ID
11-127	Reserved for future definition by IASIG/MMA
128-255	Extension Area

#### **3.5.2.30 SyncGroupID**

16-bit unsigned integer. SyncGroupID of a SyncGroup in this Cue's namespace. SyncGroups are defined by the audio artist and are represented only by SyncGroupIDs, not by any data structures in the iXMF file.

#### **3.5.2.31 TrackID**

8-bit unsigned integer, values 0-99. Index of a Track within the indicated Chunk.

#### **3.5.2.32 TransitionID**

16-bit unsigned integer. TransitionID of a TransitionDefinition resource in this Cue's namespace.

### 3.5.2.33 TransportStateID

1 byte unsigned integer.

<b>transportStateID</b> Value	<b>Transport State</b>	<b>Notes</b>
0	Play	
1	Pause	
2	Stop	
3	Wait For Sync	
4-127	Reserved for future definition by IASIG/MMA	
128-255	Extension Area	

### 3.5.2.34 Value

4 bytes. Interpretation depends on context: 32-bit signed integer, 32-bit unsigned integer, or 32-bit IEEE floating point number.

### 3.5.2.35 VariableID

8-bit unsigned integer, values 0-31. Index of a Global, Cue, or Local variable, as determined by the VariableTypeID parameter.

### 3.5.2.36 VariableTypeID

1 byte unsigned integer.

<b>variableTypeID</b> Value	<b>Variable Type</b>
0	Local Variable
1	Cue Variable
2	Global Variable
3-255	Reserved for future definition by IASIG/MMA

### 3.5.2.37 WeightingCurveID

1 byte unsigned integer. When the `Pick()` opcode uses `pickTypeID` value 0 (Random) to select a probability curve, the `pickParameter` is interpreted as a `weightingCurveID`. Different weighting curves makes different ranges of random values more likely to occur than others.

<b>weightingCurveID</b> value	<b>Weighting Curve Rule</b>	<b>Notes</b>
0	Equal Probability	
1	Decreasing	Lower values are more likely to occur than higher values
2	Increasing	Higher values are more likely to occur than lower values
3	Gaussian	Mid-range values are more likely to occur than higher or lower values
4	Either End	Higher and Lower values are more likely to occur than mid-range values
5-127	Reserved for future definition by IASIG/MMA	
128-255	Extension Area	

## 3.5.3 OpcodeID and OperandsLen Values

As described in the Script Resource Format section of section 4 Data Structures, the built-in scripting language described in this section occupies opcode space 0x00, and every scripting language statement has a unique binary representation consisting of an opcode and an operands field with a length in bytes and zero or more parameters. Those details are specified in the following table.



Interactive XMF: File Format Specification  
**PUBLIC REVIEW DRAFT – NOT FINAL – NOT FOR IMPLEMENTATION**

OpcodeID	Returns	Statement	OperandsLen	Operands
0	void	SetGlobalDefaultFade	9	1: FadeDirectionID [1 byte] 2: FadeCurveShapeID [4 bytes] 3: Duration [4 bytes]
1	void	SetMasterDspParameter	8	1: DspParamID [4 bytes] 2: Value [4 bytes]
2	void	CallCue	2	2: CueID [2 bytes]
3	var	GetCueID	1	1: var [1 bytes]
4	var	GetCueParameter	1	1: var [1 byte]
5	void	SetCueMaxInstanceCount	4	1: CueID [2 bytes] 2: CountMax [2 bytes]
6	var	GetCueMaxInstanceCount	3	1: var [1 byte] 2: CueID [2 bytes]
7	var	GetCueInstanceCount	3	1: var [1 byte] 2: CueID [2 bytes]
8	void	SetPreBufferDuration	4	1: DurationInSeconds [4 bytes]
9	void	SetCueCancelScript	2	1: ScriptID [2 bytes]
10	void	ReleaseCue	0	Void
11	void	Prebuffer	0	Void
12	void	InvokeCallbackFunction	2	1: CallbackID [2 bytes]
13	void	SetCueInstancePriority	2	1: Priority [2 bytes]
14	var	GetCueInstancePriority	1	1: var [1 byte]
15	void	SetCuePriority	4	1: CueID [2 bytes] 2: Priority [2 bytes]
16	var	GetCuePriority	4	1: var [2 bytes] 2: CueID [2 bytes]
17	void	SetCueDefaultFade	11	1: CueID [2 bytes] 2: FadeDirectionID [1 byte] 3: FadeCurveShapeID [4 bytes] 4: Duration [4 bytes]
18	void	SetCueDspParameter	10	1: cueID [2 bytes] 2: DspParameterID [4 bytes] 3: Value [4 bytes]
19	void	SetCueSyncGroup	4	1: CueID 2: SyncGroupID [2 bytes]
20	void	SetCueMixGroup	4	1: CueID [2 bytes] 2: MixGroupID [2 bytes]
21	void	SetCueMediaHandling	3	1: CueID [2 bytes] 2: MediaHandlingTypeID [1 byte]
22	void	PreloadCueMedia	3	1: CueID [2 bytes]
23	void	UnloadCueMedia	2	1: CueID [2 bytes]
24	void	SetVariable	6	1: VariableTypeID [1 byte] 2: VariableID [1 byte] 3: Value [4 bytes]

Interactive XMF: File Format Specification  
**PUBLIC REVIEW DRAFT – NOT FINAL – NOT FOR IMPLEMENTATION**

25	var	GetVariable	3	1: var [1 byte] 2: VariableTypeID [1 byte] 3: VariableID [1 byte]
26	var	EvaluateExpression	variable 2-N	1: var [1 byte] 2: Expression [variable length]
27	var	GetTime	1	1: var [1 byte]
28	var	GetRandom	9	1: var [1 byte] 2: MinValue [4 bytes] 3: MaxValue [4 bytes]
29	void	Label	2	1: LabelID [2 byte2]
30	void	ConditionalBranch	variable 2-N	1: Expression [variable length] 2: LabelID [1 byte]
31	void	TimeDelay	4	1: Duration [4 bytes]
32	void	CallScript	2	1: ScriptID [2 bytes]
33	void	LaunchScript	2	1: ScriptID [2 bytes]
34	var	GetTotalPoolSize	1	1: var [ 1 byte]
35	void	IncludeInNarrowing	3	1: ChunkID [2 bytes] 2: NarrowingInclusionRuleID [1 byte]
36	void	NarrowPool	3	1: NarrowingRuleID [1 byte] 2: NarrowingParameter [2 bytes]
37	var	GetNarrowedPoolSize	1	1: var [1 byte]
38	void	AddChunkToNarrowedPool	2	1: ChunkID [2 bytes]
39	void	SetChunkOrderTag	4	1: ChunkID [2 bytes] 2: OrderTag [2 bytes]
40	var	GetChunkOrderTag	3	1: var [1 byte] 2: ChunkID [2 bytes]
41	void	SetChunkLRUOrder	4	1: ChunkID [2 bytes] 2: LruOrder [2 bytes]
42	var	GetChunkLRUOrder	3	1: var [ 1 byte] 2: ChunkID [2 bytes]
43	void	Sort	1	1: SortTypeID [1 byte]
44	void	SetChunkPickWeight	4	1: ChunkID [2 bytes] 2: PickWeight [2 bytes]
45	var	GetChunkPickWeight	3	1: var [1 byte] 2: ChunkID [2 bytes]
46	void	SetChunkPickWeights	5	1: MaxPickWeight [2 bytes] 2: MinPickWeight [2 bytes] 3: WeightingCurveID [1 byte]
47	var	Pick	4	1: var [1 byte] 2: PickTypeID [1 byte] 3: PickParameter [2 bytes]
48	void	SetNextChunk	2	1: ChunkID [2 bytes]
49	void	SetNextTransition	2	1: TransitionID [2 bytes]
50	void	PlayChunk	2	1: ChunkID [2 bytes]
51	void	SetPlayerTransportState	1	1: TransportStateID [1 byte]
52	var	GetPlayerTransportState	1	1: var [1 byte]

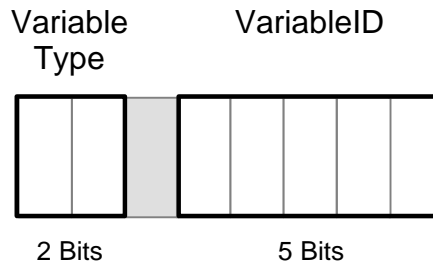
Interactive XMF: File Format Specification  
**PUBLIC REVIEW DRAFT – NOT FINAL – NOT FOR IMPLEMENTATION**

53	void	Fade	0	void
54	void	ReleasePlayer	0	void
55	void	FadeAndReleasePlayer	0	void
56	void	SetChunkFade	9	1: FadeDirectionID [1 byte] 2: FadeCurveShapeID [4 bytes] 3: Duration [4 bytes]
57	void	SetChunkPriority	4	1: ChunkID [2 bytes] 2: Priority [2 bytes]
58	var	GetChunkPriority	3	1: var [1 byte] 2: ChunkID [2 bytes]
59	void	SetChunkDspParameter	10	1: ChunkID [2 bytes] 2: DspParameterID [4 bytes] 3: Value [4 bytes]
60	void	SetChunkSyncGroup	4	1: ChunkID [2 bytes] 2: SyncGroupID [2 bytes]
61	void	SetChunkMixGroup	4	1: ChunkID [2 bytes] 2: MixGroupID [2 bytes]
62	void	SetTrackMute	4	1: ChunkID [2 bytes] 2: TrackID [1 byte] 3: MuteOrUnmuteID [1 byte]
63	void	SetChunkChunkGroup	4	1: ChunkID [2 bytes] 2: ChunkGroupID [2 bytes]
64	var	GetChunkChunkGroup	4	1: var [2 bytes] 2: ChunkID [2 bytes]
65	void	SetChunkMediaHandling	3	1: ChunkID [2 bytes] 2: MediaHandlingTypeID [1 byte]
66	void	SetChunkCancelScript	2	1: ScriptID [2 bytes]
67	void	SetChunkTempoMap	4	1: ChunkID [2 bytes] 2: SmfChunkID [2 bytes]
68	void	SetChunkEntryPointsRule	4	1: ChunkID [2 bytes] 2: PositionRuleID [2 bytes]
69	void	SetChunkExitPointsRule	4	1: ChunkID [2 bytes] 2: PositionRuleID [2 bytes]
70	void	PreloadChunkMedia	2	1: ChunkID [2 bytes]
71	void	ReleaseChunkMedia	2	1: ChunkID [2 bytes]
72	void	LoadMediaForChunkGroup	2	1: ChunkGroupID [2 bytes]
73	void	UnloadMediaForChunkGroup	2	1: ChunkGroupID [2 bytes]
74	void	SetMixGroupDspParameter	10	1: MixGroupID [2 bytes] 2: DspParamID [4 bytes] 3: Value [4 bytes]
75	void	SetSyncGroupDspParameter	10	1: SyncGroupID [2 bytes] 2: DspParamID [4 bytes] 3: Value [4 bytes]
76	var	GetSyncGroupBars	3	1: var [1 byte] 2: SyncGroupID [2 bytes]

77	var	GetSyncGroupBeats	3	1: var [1 byte] 2: SyncGroupID [2 bytes]
78	var	GetSyncGroupTicks	3	1: var [1 byte] 2: SyncGroupID [2 bytes]

### 3.5.4 Format of “var” Parameters

All `var` parameters are 1 byte long, and always indicate the destination to which the value returned by the script statement is to be written.



The upper two bits of a `var` parameter indicate what type of iXMF variable should receive the value returned by the script statement:

Upper 2 Bits of Var	Destination Variable Type
00	Local Variable
01	Cue Variable
10	Global Variable
11	(Do not use – Reserved for Future Definition by IASIG)

The lower five bits of a `var` parameter indicate the index to use within the indicated type of variable.

#### Examples:

Local Variable 1:            0b00 0 00001 or 0x01  
Cue Variable 1:             0b01 0 00001 or 0x41  
Global Variable 1:         0b10 0 00001 or 0x81  
Global Variable 31:        0b10 0 11111 or 0x9F

## 3.6 Debugging Statements: OpcodeSpace 0x01

OpcodeSpace 0x01 is reserved for debugging statements. Support for debugging statements is optional. It is expected that IXE implementations will include this OpcodeSpace only in debugging builds of the game or other host application, not in final release builds. For IXE implementations in audio artist tools, inclusion of the debugging statements OpcodeSpace is recommended.

The purpose of the statements in the Debugging OpcodeSpace is to allow the audio artist to inspect the internal operations of the Scripts he or she writes, similar to the way a C language programmer might use the `printf()` function.

#### Example:

To do the equivalent of `printf("At time of Footstep call, global var 3 = %d", globalVariable[ 3 ] )`, the audio artist could include the following Statements in his or her Script:

```
getGlobalVariable( 3 )           // Leaves val on stack

itoa()                           // Convert val to string on stack

pushStr( "At time of Footstep call, global var 3 = " )

                                // Pushes explanatory text on stack

strcat()                         // Makes a single string

debugPrint()                     // Emits (i.e. displays onscreen) the string
```

### 3.6.1 Statements

This section documents all opcodes in OpcodeSpace 0x01.

#### 3.6.1.1 void PushStr( stringLength, string )

Pushes a literal string onto the stack. The string is ASCII text up to 65536 bytes long, with no terminating null character. The string is delimited by a leading 16-bit length. It is recommended that iXMF authoring tools handle the length automatically to avoid user errors.

#### 3.6.1.2 void Itoa( void )

Converts the integer on top of the stack to a string, and then pushes the string onto the stack.

#### 3.6.1.3 void Strcat( void )

Pulls the top two strings off the stack, concatenates them, and then pushes the result onto the stack

#### 3.6.1.4 void DebugPrint()

Emits the string on top of the stack (and clears the stack). The display mechanism is implementation-specific.

### 3.6.2 OpcodeID and OperandsLen Values

As described in the Script Resource Format section of chapter 6, Data Structures, the optional debugging scripting language described in this section occupies opcode space 0x01, and every scripting language statement has a unique binary representation consisting of an opcode and an operands field with a length in bytes and zero or more parameters. Those details are specified in the following table.

OpcodeID	Returns	Statement	OperandsLen	Operands
0	void	PushStr	2-N	1: stringLength [2 bytes] 2: string [variable length]
1	void	Itoa	0	void
2	void	Strcat	0	void
3	void	DebugPrint	0	void

## 4. Data Structures

This section defines the format of each of the data resources contained in an iXMF File, except the playable media types such as audio and MIDI, in full detail. These items are contained using the XMF Meta File Format, and an iXMF File is a separate XMF File Type. iXMF players and iXMF Engine implementations must be able to parse the XMF Meta File Format, and must be able to parse all of the data resources defined in this section.

### 4.1 iXMF File Layout

The XMF Meta File Format allows nested hierarchical structures of ‘files’ held in ‘folders’, like a computer file system. This section uses slash-delimited path notation to describe the required folder tree structure inside an iXMF File. Data formats for all iXMF resource types shown here are detailed later in this chapter.

The XMF Meta File Format is specified in a separate MMA document [2]. It includes a metadata system that allows any number of metadata items to be attached to any file or folder in the XMF file. iXMF relies on this metadata system to uniquely identify each Cue, Script, Chunk, Transition, Callback, PositionRule, MediaType, and MediaFile resource within the iXMF file,

The filename extension and other external media type identifiers for XMF files, such as MIME Media Type and Mac OS File Type code, are specified at section 1.1.1 iXMF File Identification.

**Note:** The iXMF player profiles to which an iXMF file conforms may be attached to the Root folder using XMF metadata items with new Standard FieldID 16 iXMF Profile Conformance.

**Note:** If an XMF metadata item using new Standard FieldID 15 iXMF Namespace is attached to the RootNode, then the IXE will include all other loaded iXMF files using the same Namespace in all resource searches originating from Cues in this iXMF file. This allows for sharing of Cues, Scripts, Chunks, Transitions, Callbacks, PositionRules, MediaTypes, MetadataTags, MemorySpaces, and MediaFiles across multiple iXMF files.

#### 4.1.1 Folder Structure

The Root Node folder of an iXMF file contains a set of folders, one each for the following resource formats defined by this specification: Cues, SharedScripts, Chunks, Transitions, Callbacks, PositionRules, MemorySpaces, MediaTypes, MetadataTags, and MediaFiles. These folders are named using new XMF standard metadata FieldID 28, iXMF Resource Folder Name. Within any of these resource folders, the use of subfolders is permitted, for example to facilitate logical groupings of related material.

It is recommended but not required that these resource folders appear in the iXMF file in the same order as just listed. In any case the MediaFiles folder should appear last, because this folder is likely to be the largest in terms of bytes. If a given iXMF file has no need for any standard resources of a given type, then the folder for that resource format may be omitted from the iXMF file. For example, if there are no MemorySpaceDefinition resources, the MemorySpaces folder may be omitted.

An optional ExpansionArea folder containing non-standard data in any desired format may also appear in the Root folder; if present, it should appear just before the MediaFiles folder.

The Root folder of the iXMF file may also contain additional non-standard folders and/or files, so long as their names do not conflict with the pre-defined folder names shown here.

All instances of the Cues, SharedScripts, Chunks, Transitions, Callbacks, PositionRules, MemorySpaces, MediaTypes, MetadataTags, and MediaFiles resource types must appear within the corresponding resource folder, as XMF FileNodes. For example, all CueDefinition resources must appear inside the Cues folder. The IXE may not find standard resources located elsewhere in the iXMF file. In general, iXMF does not require these resource files to have names, however if names are desired, for example for

easier content authoring, the new XMF standard metadata FieldID 29, iXMF Resource Name may optionally be used.

Path within iXMF File	XMF Node Type	Description
<b>/</b>	folderNode	The RootNode of the iXMF File.
<b>/Cues</b>	folderNode	Folder containing all CueDefinition resources. Each resource is externally tagged with a CueID.  A few Cue names are reserved for handling built-in conditions, however providing CueDefinitions for them is optional.
<b>/FileIxmLoad</b> [optional] [CueID=0] [name optional]	fileNode	A CueDefinition resource.
<b>/FileIxmSetup</b> [optional] [CueID=1] [name optional]	fileNode	A CueDefinition resource.
<b>/FileIxmTeardown</b> [optional] [CueID=2] [name optional]	fileNode	A CueDefinition resource.
<b>/FileIxmUnload</b> [optional] [CueID=3] [name optional]	fileNode	A CueDefinition resource.
<b>/myCueDefinition</b> [CueID up to user] [name optional] [any number of these]	fileNode	User-defined CueDefinition resource.
<b>/SharedScripts</b>	folderNode	Folder containing all shared ScriptDefinition resources. Each resource is externally tagged with a ScriptID.
<b>/mySharedScriptDefinition</b> [ScriptID up to user] [name optional] [any number of these]	fileNode	A ScriptDefinition resource that can be accessed by any Cue. If a Script with the same number or name exists in a Cue, it will override the shared Script for that Cue.
<b>/Chunks</b>	folderNode	Folder containing all ChunkDefinition resources. Each resource is externally tagged with a ChunkID.
<b>/myChunkDefinition</b> [ChunkID up to user] [name optional] [any number of these]	fileNode	A ChunkDefinition resource.

<b>/Transitions</b>	folderNode	Folder containing all <code>TransitionDefinition</code> resources. Each resource is externally tagged with a <code>TransitionID</code> .
<b>/myTransitionDefinition</b> [TransitionID up to user] [name optional] [any number of these]	fileNode	A <code>TransitionDefinition</code> resource.
<b>/Callbacks</b>	folderNode	Folder containing all <code>CallbackDefinition</code> resources. Each resource is externally tagged with a <code>CallbackID</code> .
<b>/myCallbackDefinition</b> [CallbackID up to user] [name optional] [any number of these]	fileNode	A <code>CallbackDefinition</code> resource.
<b>/PositionRules</b>	folderNode	Folder containing all <code>PositionRuleDefinition</code> resources. Each resource is externally tagged with a <code>PositionRuleID</code> .
<b>/myPositionRuleDefinition</b> [PositionRuleID up to user] [name optional] [any number of these]	fileNode	A <code>PositionRuleDefinition</code> resource.
<b>/MemorySpaces</b>	folderNode	Folder containing all <code>MemorySpaceDefinition</code> resources. Each resource is externally tagged with a <code>MemorySpaceID</code> .
<b>/myMemorySpaceDefinition</b> [MemorySpaceID up to user] [name optional] [any number of these]	fileNode	A <code>MemorySpaceDefinition</code> resource.
<b>/MediaTypes</b>	folderNode	Folder containing all <code>MediaTypeDefinition</code> resources. Each resource is externally tagged with a <code>MediaTypeID</code> .
<b>/myMediaTypeDefinition</b> [MediaTypeID up to user] [name optional] [any number of these]	fileNode	A <code>MediaTypeDefinition</code> resource.
<b>/MetadataTags</b>	folderNode	Folder containing all <code>MetadataTagDefinition</code> resources. Each resource is externally tagged with a <code>MetadataTagID</code> .
<b>/myMetadataTagDefinition</b> [MetadataTagID up to user] [name optional] [any number of these]	fileNode	A <code>MetadataTagDefinition</code> resource.
<b>/ExtensionArea</b>	folderNode	Optional folder containing non-standard extension data.



<b>/MediaFiles</b>	folderNode	Folder containing all playable media file images. Each media file is externally tagged with a MediaFileID, using new XMF meta-data Standard FieldID 26, iXMF Media File ID. This MediaFileID value must match the MediaFileID field of the Chunk resource(s) that reference the media file.
<b>/myMediaFile</b> [MediaFileID up to user] [name optional] [any number of these]	fileNode	Image of a media file in a common playable format such as WAV, AIFF, MP3, or SMF.
(end of iXMF File)		

## 4.2 CueDefinition Resource Format

Externally tagged with a playable CueID (an integer), using new XMF meta-data Standard FieldID 17, iXMF Cue ID. May be optionally externally tagged with a Cue name, using new XMF meta-data Standard FieldID 27, iXMF Cue Name. Some Cue names and IDs are reserved for special functions:

CueID	Cue Name	IXE Calls Cue When	Required / Optional
0	FileIxmLoad	The iXMF File is first loaded by the IXE	Optional
1	FileIxmSetup	The Host calls SetUpIxmFile() for this iXMF File	Optional
2	FileIxmTearDown	The Host calls TearDownIxmFile() for this iXMF File	Optional
3	FileIxmUnload	The iXMF File is about to be unloaded by the IXE	Optional
4-10	(Reserved for future definition by IASIG/MMA)		
11 - up	(any other name)	client application or a running Cue instance asks the IXE to call a Cue of that name or CueID	Optional

Cues include several data fields, and a list of zero or more named Scripts. The format and list of reserved Script IDs and names, and the condition under which the iXMF Engine will call each reserved Script ID, appear at section 3.2.1 Predefined ScriptIDs and Names. In many cases, Script statements are provided to allow Scripts to override the settings in the data fields at runtime.

### 4.2.1 Fields

CueDefinition {

Field Name	Notes	Size in Bytes	Data Format
DefaultMaximumInstanceCount		1-3	VLQ
NeededPlayers {			
<length>		1-3	VLQ
{ MediaTypeID, NumOfPlayers }	[any number of these pairs]	2-4	VLQ, VLQ
}			
DefaultPriority	Integer. Priority of 0 means “Don’t care”, 1000 means ordinary priority, higher numbers mean higher priority and vice versa	1-2	VLQ
DefaultSyncGroup	SyncGroupID (or 0 for no Sync Group)	1-3	VLQ
DefaultTransition	TransitionID	1-3	VLQ
DefaultMixGroup	MixGroupID	1-3	VLQ
DefaultEntrySyncPoint	TimePositionRuleID	1-2	VLQ
DefaultEntrySyncRule	SyncTypeID (or token for ‘use Cue’s setting’)	1-2	VLQ

Interactive XMF: File Format Specification  
**PUBLIC REVIEW DRAFT – NOT FINAL – NOT FOR IMPLEMENTATION**

DefaultFadeIn	A FadeCurveShapeID (or token for 'use Cue's setting')	4	4-char code
DefaultExitSyncPoint	TimePositionRuleID	1-2	VLQ
DefaultExitSyncRule	SyncTypeID (or token for 'use Cue's setting')	1-2	VLQ
DefaultFadeOut	A FadeCurveShapeID (or token for 'use Cue's setting')	4	4-char code
GainTrimInDB	Signed integer representing relative gain in 1/100 dB	4	32 bit signed integer
DefaultDspParameters {			
<length>		1-3	VLQ
{ DspParamID, Value }	[any number of these pairs]		4-char, 4 byte
}			
MemoryUsage {			
<length>		1-2	VLQ
{ MemorySpaceID, RamInKB }	[any number of these pairs]	2-4	VLQ, VLQ
}			
BufferDurations {			
<length>		1-2	VLQ
{ MediaTypeID, BufDurInSeconds }	[any number of these pairs]	2-3	VLQ, VLQ
}			
Scripts {			
<length>		2-3	VLQ
ScriptID=0 (CueIxmLoad)	[required]	variable	VLQ, ScriptDefinition
ScriptID=1 (CueStart)	[required]	variable	VLQ, ScriptDefinition
ScriptID=2 (ChangeOver)	[optional]	variable	VLQ, ScriptDefinition
ScriptID=3 (ChunkEnd)	[optional]	variable	VLQ, ScriptDefinition
ScriptID=4 (CueCancel)	[optional]	variable	VLQ, ScriptDefinition
ScriptID=5 (CueEnd)	[optional]	variable	VLQ, ScriptDefinition
ScriptID=6 (CueStop)	[optional]	variable	VLQ, ScriptDefinition
ScriptID=7 (CueLastInstanceTeardown)	[optional]	variable	VLQ, ScriptDefinition

ScriptID=8 (CueIxmfmUnload)	[optional]	variable	VLQ, ScriptDefinition
ScriptID=9 (NoSuchScript)	[optional]	variable	VLQ, Script
<AnyOtherScriptID> (User-defined Script)	[optional] [any number of these]	variable	VLQ, ScriptDefinition
}			
ChunkPool {			
<length>		1-3	VLQ
{ ChunkIndex, ChunkID, dfltNxtChnk }	[any number of these triplets]	3-9	VLQ, VLQ, VLQ
}			
ExtensionArea {			
<length>		1-4	VLQ
ExtensionData		variable	Binary block
}			
}			

#### 4.2.2 MaximumInstanceCount Behavior

When trying to start a new instance of a Cue, if the current value of the Cue's MaximumInstanceCount would be exceeded, then the IXE will perform a priority test. If the new instance of the Cue has the same or higher priority as already running instances of the Cue, then the IXE will kill the oldest running instance of the Cue.

The default value of the MaximumInstanceCount is set at authoring time in the Cue resource, and can be changed at any time with the SetMaximumInstanceCount() Script statement.

For simple digital audio Cues, the MaximumInstanceCount caps the number of digital voices (polyphony) the Cue will consume.

## 4.3 ChunkDefinition Resource Format

Externally tagged with ChunkID, using new XMF meta-data Standard FieldID 19 iXMF Chunk ID.

### 4.3.1 Fields

Field Name	Notes	Size in Bytes	Data Format
MediaFileID	MediaFileID of the playable media file. The playable media file must be located in, or referenced from, the MediaFiles folder of the iXMF file.	1-3	VLQ
MetadataTags {			
<Length>		1-2	VLQ
MetadataTagID	[any number of these]	1-3	VLQ
}			
MediaType	[MediaTypeID of the Chunk's media format]	1-2	VLQ
DefaultMediaHandling	[a MediaHandlingTypeID]	1	VLQ
DefaultPriority	Integer. Priority of 0 means "Don't care", 1000 means ordinary priority, higher numbers mean higher priority and vice versa	1-2	VLQ
DefaultSyncGroup	[a SyncGroupID] (or 0 for no Sync Group)	1-3	VLQ
DefaultTempoMapID	[ChunkID of the SMF Chunk to use as TempoMap for this Chunk] (or 0 for no default tempo map)	1-3	VLQ
StartOffset	[Integer: Samples for audio, MIDI Ticks for SMF]	1-5	VLQ
EndOffset	[Integer: Samples for audio, MIDI Ticks for SMF]	1-5	VLQ
ExitPoints {			
<Length>		1-3	VLQ
{ TimePositionRuleID, SyncTypeID }	[any number of these pairs] The SyncType sets the condition that must be satisfied to select that TimePosition as the ExitPoint.	2-4	VLQ, VLQ
}			
EntryPoints {			
<Length>		1-3	VLQ

( TimePositionRuleID, SyncTypeID )	[any number of these pairs] The SyncType sets the condition that must be satisfied to select that TimePosition as the EntryPoint.	2-4	VLQ, VLQ
}			
GainTrim	[Signed integer representing relative gain in 1/100 dB]	4	32 bit signed integer
DefaultNextChunk	[a ChunkID]	1-3	VLQ
DefaultChunkGroup	[a ChunkGroupID]	1-3	VLQ
DefaultTransition*	[a TransitionID]	1-3	VLQ
DefaultIncludeInNarrowing	[a NarrowingInclusionRuleID ]	1	VLQ
DefaultOrderTag	[an integer]	1-3	VLQ
DefaultLRUOrder	[an integer]	1-3	VLQ
DefaultPickWeight	[an integer]	1-3	VLQ
DefaultCancelScript	[a ScriptID]	1-3	VLQ
DefaultDspParameters {			
<length>		1-3	VLQ
{ DspParamID, Value }	[any number of these pairs]		4-char, 4 byte
}			
ExtensionArea {			
<Length>		1-4	VLQ
ExtensionData		variable	Binary block
}			

**\*Note:** In the Chunk's DefaultTransition, the terms FadeIn and FadeOut are interpreted differently than in the other uses of the TransitionDefinition resource. Here, FadeIn means how this Chunk enters against another Chunk, and FadeOut means how this Chunk exits against another Chunk.

### 4.3.2 Chunk Synchronization

When preparing to synchronize a new Chunk to an already-running Chunk, the IXE shall determine the soonest possible (next occurring) match between the first Chunk's set of ExitPoints and the new Chunk's set of EntryPoints, and prepare the new Chunk to begin playback at the time necessary to synchronize the selected EntryPoint and ExitPoint. When calculating the necessary start time, the IXE must take into account the pre-lap duration of the selected Transition, as well as any media file loading latency. When the new Chunk's EntryPoints List is empty, the IXE shall treat the new Chunk's StartOffset as the EntryPoint, and shall use the current NextTransition (as set by the SetNextTransition script statement), or if none has been set, the Chunk's DefaultTransition.

Fade-in of the entering player is possible if the chosen EntryPoint is not at the start of its file. Actual playback starts with zero volume from the entry point minus the entry fade duration, fading up to full volume at the appropriate (per the sync & fade settings) time.

### **4.3.3 Sync Groups**

At the time the IXE starts playback of a given Chunk, if no other currently playing Chunks belong to the same SyncGroup, then the new Chunk begins playing immediately and becomes the master clock for that SyncGroup.

Alternatively, if the value of the new Chunk's SyncGroup is 0, then the new Chunk begins playing immediately and no SyncGroup master clock is established.

Alternatively, at the time the IXE starts playback of a given Chunk, if at least one other currently playing Chunk belongs to the same Sync Group, then the IXE will automatically adjust the start time of the new Chunk to achieve synchronization with that SyncGroup. Depending on the settings of the currently selected Transition, synchronization may require the new Chunk to be either delayed for some amount of time before starting, or else playback may have to start immediately but at some offset into the new Chunk.

## 4.4 ScriptDefinition Resource Format

When a ScriptDefinition resource appears in the SharedScripts folder, it is externally tagged with a ScriptID, using new XMF meta-data Standard FieldID 18 Script ID. When a ScriptDefinition resource appears within the Scripts area of a CueDefinition, it is externally tagged with a ScriptID in VLQ form.

See section 3.2 Predefined ScriptIDs and Names for predefined vs. freely usable ScriptIDs and names.

### 4.4.1 Fields

A ScriptDefinition resource is a list of ScriptInstruction structures, delimited by both an initial length (for easier file-level parsing) and a final EndOfScript token (to make it easy for the Script interpreter to identify the end of a running Script). A name is optional, though necessary for all Scripts intended to be called from marker callbacks. If the name is omitted, NameLengthInBytes will be zero.

```
Script {
  NameLengthInBytes      [1-3 byte VLQ]
  ScriptName             [variable length ASCII string, no terminating null]
  <Length>
  ScriptInstruction      [any number of these]
  EndOfScriptToken       [the byte 0xFF]
  ExtensionAreaLength    [integer]
  ExtensionArea          [variable length binary block]
}
```

### 4.4.2 ScriptInstruction Format

Every ScriptInstruction structure consists of one opcodeField structure and one operandsField structure. The operandsField may contain any number of operands, as detailed below.

```
ScriptInstruction {
  OpcodeField            [ 2 byte binary]
  OperandsField          [variable length binary block]
}
```

Expressed differently, a ScriptInstruction structure also looks like this:

opcodeSpace opcodeID operandsLen [op1 [op2 [op3 ... [ opN ] ... ] ] ]

#### 4.4.2.1 OpcodeField Format

The scripting opcode space is extensible for both future standard versions and proprietary extensions. This extensibility mechanism is facilitated by splitting the opcode field into two parts: an opcodeSpace identifying a set or suite of opcode definitions, and an opcodeID identifying a particular instruction within that opcodeSpace.

```
OpcodeField {
  OpcodeSpace            [1 byte unsigned integer]
  OpcodeID               [1 byte unsigned integer]
}
```

##### 4.4.2.1.1 OpcodeSpace Format and Assignments

The OpcodeSpace field is an 8-bit integer, leaving room for up to 256 simultaneous opcode spaces for a given iXMF Engine implementation.

The iXMF 1.0 scripting language defined in this document uses OpcodeSpace value 0x00.

The optional set of debugging statements defined in this document uses OpcodeSpace value 0x01.

Other opcode spaces should be used only as directed in the following table:



OpcodeSpace Value	Description
0x02–0x7F	<b>Do not use.</b> Reserved for future IA-SIG standardization.
0x80–0xCF	<b>Do not use.</b> Reserved for public extensions. IDs are to be defined only by IASIG. A registry will be maintained by the IASIG.
0xD0–0xFE	Available for proprietary extensions. A registry will be maintained by the IASIG.
0xFF	<b>Do not use.</b> 0xFF is used as the token for the end of the Script.

#### 4.4.2.1.2 OpcodeID

The value of the `OpcodeID` field indicates a particular operation within the given `opcodeSpace`. The `OpcodeID` field is an 8-bit integer, leaving room for up to 256 instructions per opcode space.

`OpcodeID` assignments for opcode space 0x00, the iXMF 1.0 scripting language, and opcode space 0x01, the optional debugging statements, are shown earlier in this chapter.

#### 4.4.2.2 OperandsField format

The `OperandsField` field is length-delimited to avoid placing unnecessary restrictions on the number, size, and format of operands supported by future and extension instructions.

```
OperandsField {  
    OperandsLen          [1 byte unsigned integer]  
    Operand              [variable length binary block]  
    ...[any number of these, in any format]...  
    Operand              [variable length binary block]  
}
```

##### 4.4.2.2.1 OperandsLen

The `OperandsLen` field is an 8-bit integer indicating the number of bytes of operand data immediately following, leaving room for up to 256 bytes of parameters per instruction. This allows Script parsers to skip over any unrecognized extension instructions, and continue executing recognized instructions. For an opcode with no operands, `OperandsLen` is zero.

##### 4.4.2.2.2 Operands

Format and size of each operand is left to the opcode's implementation code and the Script compiler for the relevant opcode space to sort out.

`OperandLen` values for all instructions in opcode space 0x00, the iXMF 1.0 scripting language, and opcode space 0x01, the optional debugging statements, are shown in section 3 Scripting Language.

## 4.5 PositionRuleDefinition Resource Format

Externally tagged with `PositionRuleID`, using new XMF meta-data Standard FieldID 22 iXMF Position Rule ID. Encodes an algorithm for determining available exit (and maybe entry) points for a Chunk. A `PositionRule` may contain any number of `PositionRuleItems`, all of which are simultaneously in effect. Any `PositionRuleDefinition` can be used by any number of Chunks by using the same `PositionRuleID`.

### 4.5.1 PositionRuleDefinition Fields

```
PositionRuleDefinition {  
    LengthInBytes          [1-3 bytes VLQ]  
    PositionRuleItem       [variable length]  
    ...[any number of these]...  
    positionRuleItem       [variable length]  
}
```

### 4.5.2 PositionRuleItem Fields

Encodes one item in a `PositionRule` by referring to either a Marker in the Chunk, or a regular time interval in the Chunk described by an initial offset in ticks from the start of the Chunk and the duration in ticks of the period; an `ItemType` field says which kind is being used. In C/C++ terms, this is a struct with a union. A `periodLengthInTicks` of 0 indicates a single, non-recurring time position at the `initialOffsetInTicks`.

`PositionRuleItem` resources are not externally tagged because they only appear within a `PositionRuleDefinition` resource.

```
PositionRuleItem {  
    LengthInBytes          [1-4 byte VLQ]  
    ItemType*              [1 byte VLQ enum: marker | periodicTicks | extension ]  
    If ItemType is 'marker': {  
        MarkerNameLengthInBytes [1-3 byte VLQ]  
        MarkerName              [variable length ASCII string, no null term.]  
    }  
    If ItemType is 'periodicTicks': {  
        initialOffsetInTicks    [1-5 byte VLQ]  
        periodLengthInTicks     [1-5 byte VLQ]  
    }  
    ExtensionAreaLength     [1-5 byte VLQ]  
    ExtensionArea           [variable length binary block]  
}
```

**\*Note:** For the `ItemType` enum, values 3-63 are reserved for future IASIG/MMA definition, and values 64-127 are the Extension Area.

## 4.6 TimePosition Resource Format

Encodes an exit or entry point for a Chunk by referring to either a Marker in the Chunk, a offset in ticks from the start of the Chunk, or the `PositionRuleID` of a `PositionRule` supplied in the iXMF file; a `PositionType` field says which kind is being used. In C/C++ terms, this is a struct with a union.

TimePosition resources are not externally tagged because they only appear within a `ChunkDefinition` resource.

### 4.6.1 Fields

```
TimePosition {
    LengthInBytes                [1-4 byte VLQ]
    PositionType*                [1 byte VLQ enum:
                                marker | tickOffset | positionRule | any ]
    If PositionType is 'marker': {
        MarkerLengthInBytes      [1-2 byte VLQ]
        MarkerName               [variable length ASCII string, no null termination]
    }
    If PositionType is 'tickOffset': {
        tickOffset               [1-5 byte VLQ]
    }
    If PositionType is 'rule': {
        PositionRule              [1-3 byte VLQ PositionRuleID]
    }
    If PositionType is 'any': {
        (No data)
    }
    ExtensionAreaLength          [1-4 byte VLQ]
    ExtensionArea                [variable length binary block]
}
```

**\*Note:** For the `PositionType` enum, values 3-63 are reserved for future IASIG/MMA definition, and values 64-127 are the Extension Area.

## 4.7 TransitionDefinition Resource Format

Externally tagged with `TransitionDefinitionID`, using new XMF meta-data Standard FieldID 20 iXMF Transition ID

### 4.7.1 Fields

<code>TransitionDefinition {</code>	
<code>    SyncGroup</code>	[1-3 byte VLQ SyncGroupID, or 0 for 'none']
<code>    SyncType</code>	[1 byte VLQ enum SyncTypeID]
<code>    FadeInCurveShape</code>	[4 byte FadeCurveShapeID]
<code>    FadeInDuration</code>	[1-4 byte VLQ integer in 1/100 milliseconds]
<code>    FadeOutCurveShape</code>	[4 byte FadeCurveShapeID]
<code>    FadeOutDuration</code>	[1-4 byte VLQ integer in 1/100 milliseconds]
<code>    ExtensionAreaLength</code>	[1-5 byte VLQ]
<code>    ExtensionArea</code>	[variable length binary block]
<code>}</code>	

### 4.7.2 SyncTypeID enum

SyncTypeID Value	Sync Type	Notes
0	End	End the Cue when the current Chunk reaches its selected ExitPoint. Do not start any new Chunk.
1	No Sync	Start new Chunk playing from its start immediately, without waiting for the current Chunk to reach its selected ExitPoint.
2	Match Tick of Chunk	TimePosition must use tickOffset Both Chunks must either be in SMF format, or else have an associated SMF TempoMap Chunk.
3	Match Tick of Beat	
4	Match Tick of Bar	
5	Match Tick of Phrase	
6	Match Tick to Phrase End	
7	Match Any Tick	
8	Match Bar of Chunk	
9	Match Beat of Bar	
10	Match Beat of Phrase	
11	Match Beats to Phrase End	
12	Match Any Beat	
13	Match Bar Line of Chunk	
14	Match Bar Line of Phrase	
15	Match Bar Line to Phrase End	
16	Match Any Bar Line	
17	Match Time Offset within Chunk	TimePosition must use any
18	Match Time Offset to Chunk End	
19	Match Marker Name	TimePosition must use marker
20	Match Any Marker	Align any marker in the new Chunk to the current Chunk's selected ExitPoint.
21	Tail / Head Marker	Play the current Chunk through to or past its last marker, ignoring any other selected ExitPoint. Align the first marker in the new chunk to the last marker in the current chunk. Similar in concept to assembling model railroad track one piece at a time.

22	Exclude	Never use this <code>TimePosition</code> as an <code>EntryPoint</code> or <code>ExitPoint</code>
23-63	Reserved for future definition by IASIG/MMA	
64-127	Extension Area	

## 4.8 MediaTypeDefinition Resource Format

Externally tagged with `mediaTypeID`, using new XMF meta-data Standard FieldID 24 iXMF Media Type ID.

### 4.8.1 Fields

MediaTypeDefinition {	
bufferDuration	[1-5 byte VLQ integer: Amount of playing time to prebuffer in players; integer in milliSec]
MIMETypeLength	[1-3 byte VLQ integer]
MIMEType	[variable length ASCII string, no null termination: MIME Media Type of the content]
ExtensionAreaLength	[1-4 byte VLQ integer]
ExtensionArea	[variable length binary block]
}	

## 4.9 MetadataTagDefinition Resource Format

Externally tagged with `MetadataTagID`, using new XMF meta-data Standard FieldID 25 iXMF Metadata Tag ID.

### 4.9.1 Fields

MetadataTagDefinition {	
MetadataTypeLengthInBytes	[1-3 byte VLQ integer]
MetadataType	[variable length ASCII string, no null termination: Type of metadata (WAV, ID3, etc.) as ASCII string] <b>FORMAT TBD</b>
FieldIDLenInBytes	[1-3 byte VLQ integer]
FieldID	[variable length binary block: format depends on MetadataType]
ContentsLengthInBytes	[1-5 byte VLQ integer]
Contents	[variable length binary block: format depends on MetadataType and FieldID]
ExtensionAreaLength	[1-4 byte VLQ integer]
ExtensionArea	[variable length binary block]
}	

## 4.10 MemorySpaceDefinition Resource Format

Externally tagged with `MemorySpaceID`, using new XMF meta-data Standard FieldID 23 iXMF Memory Space ID. Examples of different memory spaces available within the same iXMF execution environment might include main memory, fast memory, media processor memory, DSP memory, slow memory, disk storage, network storage, etc.

### 4.10.1 Reserved MemorySpaceIDs

0	Use any available memory
1	Main memory
2	Media Processor memory
3	Disk memory
4-63	Reserved for future IASIG/MMA definition
64-127	Extension Area

### 4.10.2 Fields

```
MemorySpaceDefinition {  
    DescriptionLengthInBytes [1-3 byte VLQ integer]  
    Description               [variable length ASCII string, no null termination:  
                             Description of the memory space as ASCII string]  
    ExtensionAreaLength      [1-4 byte VLQ integer]  
    ExtensionAres            [variable length binary block]  
}
```

## 4.11 CallbackDefinition Resource Format

Externally tagged with `CallbackID`, using new XMF meta-data Standard FieldID 21 iXMF Callback ID. `CallbackDefinition` resources serve primarily to signal to the IXE what `CallbackIDs` it should expect to handle from `InvokeCallback()` Script statements, and to be registered by the game or other host application via the IXE API. It is optional to provide a name for the callback.

### 4.11.1 Fields

```
CallbackDefinition {  
    NameLengthInBytes [1-3 byte VLQ integer]  
    CallbackName       [variable length ASCII string, no null termination:  
                       Optional descriptive name, as ASCII string]  
    ExtensionAreaLength [1-4 byte VLQ integer]  
    ExtensionAres       [variable length binary block]  
}
```

## 4.12 FadeCurveShapeID Type Format and Value Definitions

Fade curve shapes are selected with a 4-character code. FadeCurveShapeIDs are not externally tagged because they only appear within TransitionDefinitions and Script statements.

FadeCurveShapeIDs use the following convention for prefixes (first character of the 4-character code):

I***	Fade-In
O***	Fade-Out

FadeCurveShapeIDs use the following conventions for suffixes (last 3 characters of the 4-character code):

*LdB	Linear ramp in dB
*LGn	Linear ramp in gain (multiplier)
*EqP	Equal Power
*NFd	No Fade (“butt splice”)
*X**	Extension Area (where ** varies)

### 4.12.1 Reserved FadeCurveShapeIDs

iLdB	Linear dB fade-in
oLdB	Linear dB fade-out
iLGn	Linear gain fade-in
oLGn	Linear gain fade-out
iEqP	Equal-Power fade-in
oEqP	Equal-Power fade-out

**Note:** More FadeCurveShapeIDs are expected to be defined and registered by IASIG/MMA in the future.



## 4.13 DspParameterID Type Format and Value Definitions

DSP parameters are selected with a 4-character code. DspParameterIDs are not externally tagged because they only appear within Script statements and in lists in CueDefinition and ChunkDefinition resources.

DspParameterIDs use the following convention for prefixes (first character of the 4-character code). The first character generally selects a target object type:

q***	Cue parameters
t***	Track parameters
m***	Master parameter
s***	SyncGroup parameters
x***	MixGroup parameters
g***	Global, i.e. all Cues in the same iXMF File
X***	Extension Area

DspParameterIDs use the following convention for suffixes (last 3 characters of the 4-character code). The last three characters generally select a particular function within the selected target object:

*Vol	Volume
*v00-*v99	Volume for sub-elements 0 through 99
*Pan	Stereo Pan
*p00-*p99	Stereo Pan for sub-elements 0-99
*Ptc	Pitch
*P00-*P99	Pitch for sub-elements 0-99
*GTr	Gain Trim
*G00-*G99	Gain Trim for sub-elements 0-99
*Bal	Stereo Balance
*B00-B99	Stereo Balance for sub-elements 0-99
*Mut	Mute
*M00-M99	99 Mute for sub-elements 0-99
*MPs	Musical Position (Bars, Beats, Ticks)
*MTS	Musical Time Signature (Numerator, Denominator)
*MTp	Musical Tempo (Beats per Minute)
*Trn	Transport State (Play, Pause, etc.)
*X**	Extension Area (where ** varies)

### 4.13.1 Reserved DspParameterIDs

qVol	Cue Volume
qPan	Cue Stereo Pan
qPtc	Cue Pitch
qGtr	Cue Gain Trim
qBal	Cue Stereo Balance
qMut	Cue Mute
qMPs	Cue Musical Position
qMTS	Cue Musical Time Signature
qMTp	Cue Musical Tempo
qTrn	Cue Transport State
tV00-tV99	Track 0 through 99 Volume
tP00-tP99	Track 0 through 99 Stereo Pan
tG00-tG99	Track 0 through 99 Gain Trim
tB00-tB99	Track 0 through 99 Stereo Balance
tM00-tM99	Track 0 through 99 Mute

mVol	Master Volume
mGTr	Master Gain Trim
mMut	Master Mute
sPtc	SyncGroup Pitch
sMPs	SyncGroup Musical Position
sMTS	SyncGroup Musical Time Signature
sVMTp	SyncGroup Musical Tempo
sTrn	SyncGroup Transport State
xVol	MixGroup Volume
xPan	MixGroup Stereo Pan
xGTr	MixGroup Gain Trim
xBal	MixGroup Stereo Balance
xMut	MixGroup Mute

**Note:** More DspParameterIDs are expected to be defined and registered by IASIG/MMA in the future.

## 4.14 VLQ Field Format

The acronym ‘VLQ’ stands for Variable Length Quantity, a number format first defined in the Standard MIDI File format, MMA Recommended Practice RP-001 (included in [1]). The XMF Meta File Format Specification [2] also includes a somewhat more flexible definition, and iXMF uses this XMF definition for VLQ. Developers should refer to [8] when implementing iXMF VLQs.

In general terms, a VLQ encodes an unsigned integer as a sequence of one or more bytes, where the high bit of each byte is used as a continuation flag and the low 7 bits of each byte are used for significant bits of the encoded integer. Thus a 1-byte VLQ can encode 7 bits for numbers 0-127; a 2-byte VLQ can encode 14 bits for numbers 0-16383; a 3-byte VLQ can encode 21 bits for numbers 0-2,097,151; a 4-byte VLQ can encode 28 bits for numbers 0-268,435,455; and so forth.

It is expected that every future iXMF profile definition document will specify both a maximum value and a maximum size in bytes for every VLQ format field in iXMF files that conform to that profile.

## 5. References

- [1] “*MIDI 1.0 Detailed Specification, Document Version 4.2.*” February 1996, In “The Complete MIDI 1.0 Detailed Specification, Document Version 96.1.” The MIDI Manufacturers Association Inc., Los Angeles, CA, USA.
- [2] “*Specification for XMF Meta File Format.*” November 2001, RP-030, The MIDI Manufacturers Association Inc., Los Angeles, CA, USA.

<End of Document>