Adding Intelligence to Media

PARTNER'S GUIDE TO XMP FOR DYNAMIC MEDIA

Adobe

Partner's Guide to XMP for Dynamic Media: An introduction to models for dynamic-media metadata

Contents

Preface

This document discusses additions to the Extensible Metadata Platform (XMP), that provide a model for dealing with digital dynamic media, such as movies that contain many audio and video elements. The document model for dynamic media requires *asset management*, to allow tracking the history and composition of complex resources. Further, dynamic media objects contain time-related elements, and the metadata for these must be able to model temporal values such as start-time and duration.

This document provides guidance to developers writing applications that read, write, and modify dynamic media documents, so that those applications can maintain the integrity of the composition and editing history and temporal metadata in composed documents, and assign document identifiers correctly and unambiguously.

How this document is organized

This document has the following sections:

- Chapter 1, "Asset Management for Composed Documents," explains how XMP schema properties and types are used in dynamic-media documents to track the composition and history of a composed document and retain the metadata of the components.
- Chapter 2, "Temporal Information in Dynamic Media," explains how XMP schema properties and types are used to represent timing information in dynamic-media documents and their components.

Additional documentation

For complete details of the XMP model, schemas, and file format, see the *XMP Specification*. The specification has three parts:

- *Part 1, Data and Serialization Model*, covers the basic metadata representation model that is the foundation of the XMP standard format. The Data Model prescribes how XMP metadata can be organized; it is independent of file format or specific usage. The Serialization Model prescribes how the Data Model is represented in XML, specifically RDF.

  This document also provides details needed to implement a metadata manipulation system such as the XMP Toolkit (which is available from Adobe).

- *Part 2, Standard Schemas*, provides detailed property lists and descriptions for standard XMP metadata schemas; these include general-purpose schemas such as Dublin Core, and special-purpose schemas for Adobe applications such as Photoshop. It also provides information on extending existing schemas and creating new schemas.

- *Part 3, Storage in Files*, provides information about how serialized XMP metadata is packaged into XMP Packets and embedded in different file formats. It includes information about how XMP relates to and incorporates other metadata formats, and how to reconcile values that are represented in multiple metadata formats.

The metadata workflow

With metadata as part of your workflow, flowing through all stages of production, you can speed up the entire production process and open up new opportunities in marketing and search when delivering your final product.

A simplified workflow of metadata from acquisition to output in Production Premium CS4 is as follows:

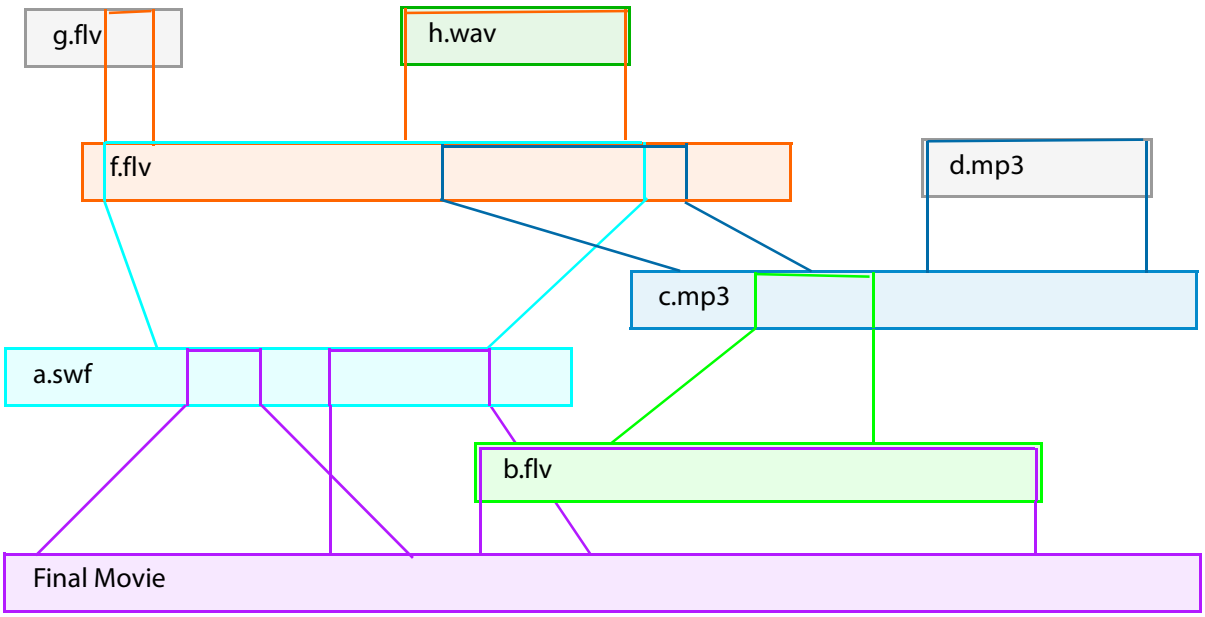
1. Metadata originates in either an Adobe OnLocation™ shot list or in an imported file-based format. You can also manually enter metadata during the capture process in Premiere.
   - ➢ When you capture from tape, you also generate some critical metadata (time, length, and so on).
   - ➢ Other file formats, such as Adobe Photoshop® and Adobe Illustrator® files, can also contain metadata.
2. Metadata can be modified or added to using the Metadata panels in any application. Metadata is embedded directly into dynamic media files whenever possible. (When the file format or usage model does not allow embedding, the metadata can be in a sidecar file.)
3. Metadata is used by various applications in their own workflows. For example:
   - ➢ Premiere Pro displays metadata and uses it for sorting and finding clips (quicksearch).
   - ➢ Premiere Pro, Soundbooth, and After Effects use speech data to find specific parts of a clip.
4. Metadata can be accessed and used directly by scripters and plug-in developers.
   - ➢ After Effects provides scripting support for metadata, so that users or plug-in developers can use it to extend the application's built-in functionality.
   - ➢ Metadata can also be accessed and used in Flash® and Flex™ applications.
5. Metadata flows out through (and is compiled together) when rendering using Adobe Media Encoder.

Production Premium CS4 is focused on capturing metadata during the creation of media. This method is more favorable than trying to retrieve metadata from a finished product. Extracting metadata from finished media is expensive, time-consuming, and prone to error due to heavy reliance on user input. In contrast, capturing metadata during creation is automatic, fast, and more accurate.

1 Asset Management for Composed Documents

Dynamic-media objects such as movies are typically composites. A composed document is created by building up a collection of individual parts, or by importing content from one document into another. Even a static document can contain an imported image, for example, whose metadata should be available through the new parent document. A dynamic document, such as a video, can include parts of other resources, which can range from, for example, a single frame to an extended clip from another file.

Each included part, called a *resource* or *asset*, can have its own XMP metadata. When an asset is added to or imported into a document using an application such as Adobe® Premiere® Pro, its metadata is also included in a hierarchy that reflects its position and history. The asset inclusion hierarchy allows you to reference and access metadata for both still and moving images that are included in the parent document.



When compositing occurs, it is important that metadata from the included assets is not simply flattened into the top-level document metadata, but retains its original association with the data it describes. The XMP metadata inclusion model retains the hierarchical structure of the assets. You can traverse the full structure of the original information, even after the rendering process has flattened the data itself.

For video and audio inclusions, the parent document's metadata also provides access to the timed values in its components, and the mapping between the timeline of the source and that of the destination. See Chapter 2, "Temporal Information in Dynamic Media".

Composed-document metadata

The XMP model for composed documents uses the metaphor of a composed dish that has *ingredients*. The ingredients are the assets used in the composition. Continuing the cooking metaphor, the composed document contains a *pantry*, where you can find the metadata for ingredients. This inclusion model can be used for print documents and workflows, as well as for dynamic media.

The properties that are used for this model are defined in the XMP Media Management namespace, `http://ns.adobe.com/xap/1.0/mm/`, with the preferred schema namespace prefix of `xmpMM`.

- The metadata for the top-level object (the movie, for example) contains a property `xmpMM:Ingredients`, which references all of the assets used in the composition (such as audio tracks or video clips contained in a movie). The referenced assets might in turn be composed, and contain their own ingredients lists.
- The metadata for the top-level object contains a property `xmpMM:Pantry`, whose value is an array that contains a copy of the XMP metadata for all of the ingredients, enumerated recursively with duplicate entries excluded. See "Retaining metadata from ingredients" on page 9.

Resource references

Each asset in the ingredients list is represented by *resource reference* (the **ResourceRef** datatype). The properties used for this type are defined in the namespace, `http://ns.adobe.com/xap/1.0/sType/ResourceRef#`, with the preferred schema namespace prefix of `stRef`.

The resource reference for an asset contains the management information from the original document's media-management metadata; the `xmpMM:manager` becomes the `stRef:manager`, for example.

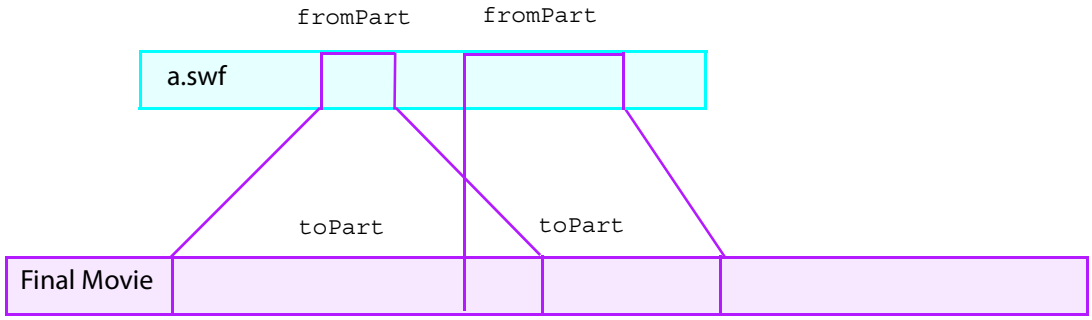
This information includes the unique identifiers for the asset (document, instance, and version ID). Each asset used in the construction of a document has, at the time of inclusion, one and only one *instance ID*, which uniquely identifies it. The `instanceID` value can be used as a unique key to find the appropriate pantry entry for any ingredient. See "Identifying documents and parts uniquely" on page 16.

Document parts

When creating a composed dynamic-media document, an author typically includes only a part of another document; a clip or even a single frame of a video, for instance; or an excerpt of a speech or song. When this occurs, it is important to identify the part used, and, if possible, trim the included metadata to that pertaining to the part used. Otherwise (particularly if the included asset comes from another composed document) the metadata recursion can quickly get out of hand.

In the example show above, for instance, the part of `c.mp3` that is used in Final Movie does not overlap with the part that uses `d.mp3`. This means you do not need to retain the metadata of `d.mp3` in the pantry of Final Movie.

The **Part** datatype identifies a portion of a document. The resource reference identifies both the part of the asset used in the composed document (`stRef:fromPart`), and the part of the composed document that incorporates the asset (`stRef:toPart`).



For dynamic media, the part specifications must include temporal information, identifying the exact stretch that is included, and the destination in the timeline of the parent. A resource reference can also specify a mapping function (`stRef:partMapping`) that is used to transform the time sequence of the `fromPart` into a new time sequence in the `toPart`. See "Timed ingredients" on page 21.

A **Part** value is also used in the event history to identify a position at which the document has been changed; see "Tracking document history" on page 14.

Part specifications

Part names are a hierarchy of arbitrary depth, specified using path syntax where levels in the hierarchy are indicated by the slash "/" character. The slash may not be used for any other purpose in these strings.

Paths (including partial paths) must always start from "/" (meaning *all* or *root*). A partial path is assumed to encompass all further descendents of the last level specified. For example, `/metadata` includes all descendants of metadata, whereas `/metadata/crs` includes all camera raw settings, but excludes metadata that is not descendent from camera raw settings. Additional levels of sub-parts or alternatives for existing levels may be defined; for example `/content/audio/channels/left` or `/content/audio/FFTaudio/high`.

NOTE: This syntax description is not normative; for definitive syntax, see the *XMP Specification Part 2, Standard Schemas*.

The following part specifications are explicitly defined:

Part specification	Part that changed or is referenced
/	Any (specific part unknown) or all (all parts of the content and metadata).
`/metadata`	Portions of the metadata.
`/content`	Any or all of the content (non-metadata).
`/content/audio`	Any or all sound.
`/content/visual`	Some image data (video or still).
`/content/visual/video`	Video or animation.
`/content/visual/raster`	Static raster image.
`/content/visual/vector`	Static vector image.

Part specification	Part that changed or is referenced
/content/visual/form/data	Form field data.
/content/visual/form/template	Form template.
/content/visual/annots	Applied annotations (comments).

A part specification for a dynamic document can include a time-range specification appended to the path, or can consist entirely of a time-range specification. If the time range is specified instead of a path, the asset includes all parts starting at the time or within the range specified. If it is added to the path, only those parts of the specified data that fall within the range are part of the included asset.

See "Timed ingredients" on page 21 for details of how to specify a time range in a part.

Retaining metadata from ingredients

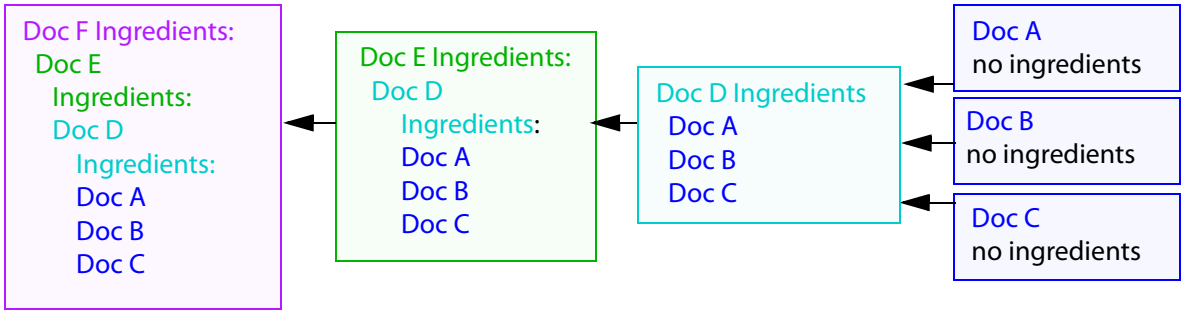
The ingredients of a composite document form a hierarchy, since each ingredient can itself be a composed document that contains ingredients. The ingredients tree is formed indirectly, however; both the ingredients and pantry metadata properties contain simple lists, with no nesting. The ingredients list of a top-level document contains only those ingredients that have been directly included in that document. The hierarchy is formed through pointers from each ingredient to its pantry entry, which contains the ingredients list for that ingredient.

The pantry is a simple array of entries. Each pantry entry corresponds to exactly one instance ID. Instance IDs are found in the ingredients lists. Each pantry entry encapsulates the metadata for the corresponding ingredient, including the `xmpMM:Ingredients` property that lists the documents used in its composition. It also contains its own `xmpMM:Pantry` property, but to avoid duplication and redundancy as much as possible, the ingredient metadata from the contained pantry items has been moved out to the top-level document.

When you add an ingredient to a composite document, all of the metadata for that ingredient is added to the pantry of the new container. If the component was itself a composed document, its pantry contains a set of entries for the metadata of each ingredient. When an Adobe application imports a document into a new container, the pantry entries for ingredients are all moved to the top level of the new container's pantry. When you perform such import operations, you should use the same technique to maintain the integrity of the metadata.

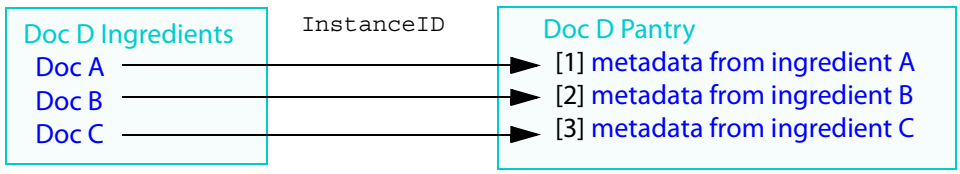
Except for moving pantry entries into the metadata of the top-level document on import or inclusion, an application should consider the pantry entries of ingredients to be invariant (read-only). If you made any changes to the metadata for an included document, it would no longer match the unique instance identified by the ID.

The following example shows the process of promoting pantry entries, using this inclusion hierarchy which results from a series of import operations:



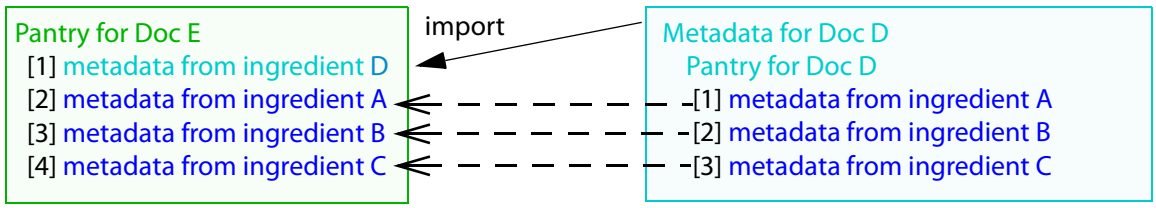
- The identifying information for a newly imported document becomes an entry in the new container's `xmpMM:Ingredients`. The metadata for that ingredient is imported into the new container's metadata by adding an entry to `xmpMM:Pantry`.
- The ingredients of a newly imported document are included in its own metadata, in the `xmpMM:Ingredients` property contained in its pantry entry. If that metadata includes any pantry entries, they are moved into new entries in the container's pantry, and removed from the pantry entry for the ingredient.

Stepping through this process at each stage of this series of import operations, the documents with no ingredients contain no pantry entries. When each component was imported into Doc D, an entry was added to the pantry of Doc D, containing the instance ID and metadata of that ingredient. Each pantry entry is associated with exactly one instance ID in the ingredients list of Doc D:



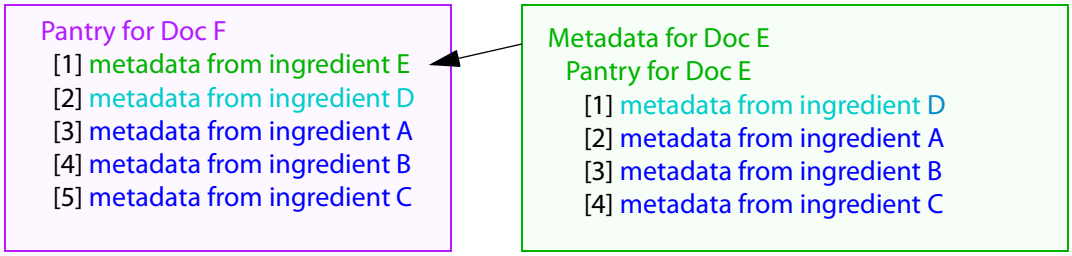
The pantry entry contains the serialized XMP Packet from the ingredient, with the `x:xmpmeta` and `rdf:RDF` wrappers removed.

When Doc D is imported into Doc E, an entry for D is added to the pantry for E, containing Doc D's metadata. The metadata for Doc D includes that document's `xmpMM:Pantry` property, which in this case does contain entries. These pantry entries from Doc D's metadata (for A, B, and C) must be moved to the pantry of the new container, becoming individual entries in Doc E:



To avoid duplication, the entire `xmpMM:Pantry` property must be removed from the pantry entry for Doc D. The entry for Doc D should contain only the metadata for Doc D, not for any of its ingredients.

The process continues up the chain of inclusions:



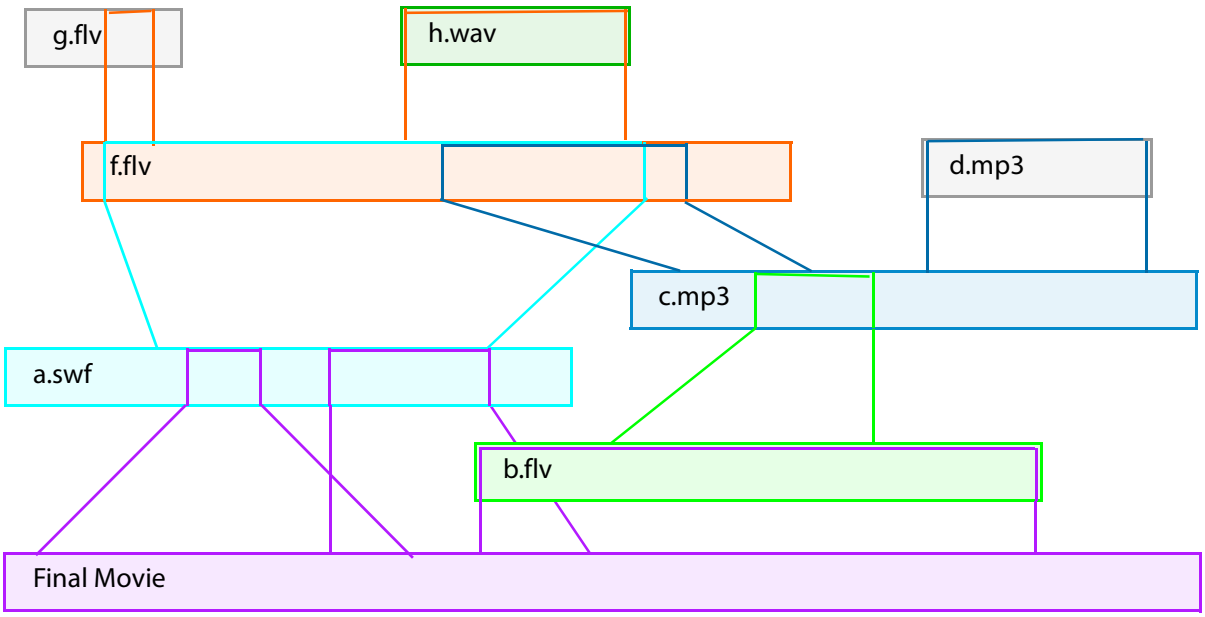
For dynamic-media documents, the metadata contains temporal information associating it with a particular part of the resource it describes. This adds a layer of complexity to the process of moving the metadata into a new container. For details of this process, see "Retaining markers from nested ingredients" on page 28.

Excluding metadata from unused parts

The pantry allows an application to encapsulate all of the metadata for an asset and carry it into the composed document. To reduce overhead, however, you can include only the metadata for *visible* components. Visible components are those that can be seen (or heard) by the user; they appear in the timeline for the document. Most components have short portions before the actual visible portion that are included in the clip to allow for transitions or minor adjustments in the editing; if these are not present in the end result (that is, not used in a transition) they are typically not visible.

In the previous example, if Doc F contains only a fragment of the timespan of Doc D that does not include any of the data imported from Docs A and C, then the pantry for Doc F can exclude the entries for Docs A and C altogether. The ingredients list must still contain entries for Docs A and C, which are part of the composition history for Doc D.

To give another example, here is the inclusion diagram for Final Movie again.



- The ingredients list for Final Movie includes the file `a.swf` twice, referencing the different parts that are used in the movie (see "Document parts" on page 7), and the file `b.flv`.
- The pantry list contains metadata from the used parts of those files, and also retains metadata from parts of files used in their composition (`c.mp3`, `f.flv`, and `h.wav`),and in turn from the files used in the composition of the components (`d.mp3`, `f.mp3`). The pantry would have these entries:

Pantry entry	contains pantry entries from ingredients
`[1] a.swf`	`f.flv`
`[2] b.flv`	`c.mp3`
`[3] c.mp3`	`d.mp3, f.mp3`
`[4] d.mp3`	none
`[5] f.flv`	`g.flv, h.wav`
`[6] g.flv`	none
`[7] h.wav`	none

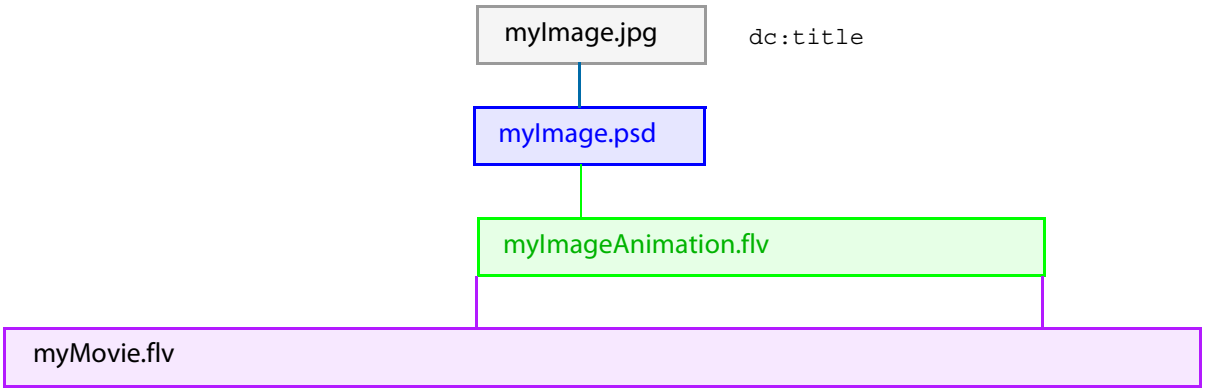
In this case, you can remove the pantry entry for the file `d.mp3`, because the part of `c.mp3` derived from that file is not used in the final composition.

For a further discussion of metadata for timed ingredients, see "Retaining markers from nested ingredients" on page 28.

Finding metadata for an ingredient

Because of the way pantry entries are promoted when one document is imported into another, the pantry entries for all of the ingredients of a document are in the top-level metadata property, `xmpMM:Pantry`. However, the ingredients list in the top-level metadata (`xmpMM:Ingredients`), contains only the documents that have been directly imported into the outermost document. To find an ingredient at a deeper level of nesting, you must find the parent document, locate its pantry entry in the outermost document's metadata, and retrieve the ingredients list stored in the pantry entry. If you have a tree of nested documents, you must perform this operation recursively to get to the metadata of the document you want.

Suppose, for example, you have an Flash file, `myMovie.flv`, that contains a sequence created from an image (`myImage.jpg`) that was touched up in Photoshop (`myImage.psd`), animated in After Effects, and then exported as Flash (`myImageAnimation.flv`).

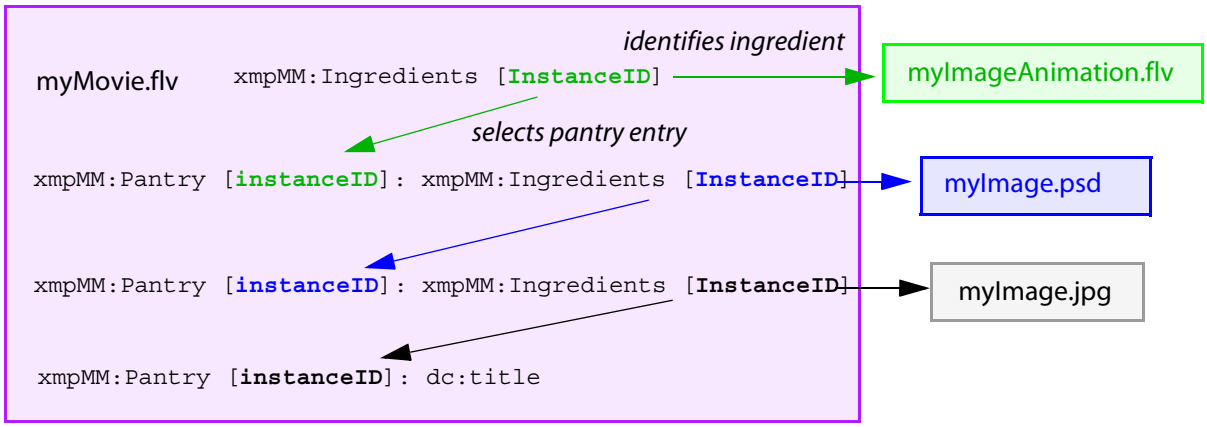


When you created `myImageAnimation.flv`, its ingredients included `myImage.psd`, whose ingredients included `myImage.jpg`. When you imported `myImageAnimation.flv` into `myMovie.flv`, the animation file instance became an ingredient of `myMovie`, and each entry in its pantry was moved to the `myMovie` pantry list.

The metadata is retained through each step of this process, so the metadata for each component has a pantry entry in the metadata for `myMovie.flv`. To find the value of an XMP property from `myImage.jpg`, you must work through the ingredients lists in each parent to find the instance ID for `myImage.jpg`, and match that ID to a pantry entry in `myMovie.flv`.

Chapter 3, "Code Example: Walking the Ingredient/Pantry Tree provides a complete code example (using the XMP Toolkit API) of how to recurse through the ingredients within the pantry entries to find the instance ID for a particular component, and then iterate through the pantry list to find a metadata value for that component. This is the basic procedure, as it applies to the example document:

- The metadata for the outermost document, `myMovie.flv`, is accessible directly through an accessor such as `SXMPMeta.GetProperty()`. Use such an accessor to retrieve the *ingredients list* from the top-level property `xmpMM:Ingredients`, and the *pantry list* from the top-level property `xmpMM:Pantry`.
- Find the `instanceID` (note lowercase 'i') for the component that is a direct ingredient (`myImageAnimation.flv`).
- Find the pantry entry with an `InstanceID` (note uppercase 'I') that matches the ingredient's `instanceID`.
- In that entry (which contains the metadata for `myImageAnimation.flv`), find the `xmpMM:Ingredients` property, and get the `instanceID` for the ingredient at the next level of nesting (`myImage.psd`).
- Find the pantry entry with a matching `InstanceID`.
- In that entry (which contains the metadata for `myImage.psd`), find the `xmpMM:Ingredients` property, and get the `instanceID` for the ingredient at the next level of nesting (`myImage.jpg`).
- Find the pantry entry with a matching `InstanceID`. Use an XMP accessor to retrieve the desired property value (`dc:title`) from that pantry entry.



You can use the same technique to find metadata for any ingredient at any step of the composition. You can find, for instance, the file type of a component file, the path where it was last stored, and the history of changes to that file; see "Tracking document history" on page 14.

Tracking document history

In addition to the ingredients list of assets, the metadata for a composed document tracks the *history* of how the document was created. You can track both the change history (modifying operations performed on the document), and the derivation history (other documents from which the document was derived through branching).

- The `xmpMM:History` property contains an ordered array of high-level user actions that resulted in this instance of the document. These actions are represented by the `ResourceEvent` data type; see "Recording change events" on page 15. This history is intended to give human readers a description of the steps taken to make the changes from the previous version to this one. It is an abstract list, not an exhaustive keystroke or other detailed history. The description is intended to be sufficient for metadata management and workflow enhancement.
- The `xmpMM:DerivedFrom` property references the source document from which this one is derived, typically through a Save As operation that changes the file name or format. It is a minimal reference; missing components can be assumed to be unchanged. For example, a new version might only need to specify the instance ID and version number of the previous version, or a rendition might only need to specify the instance ID and rendition class of the original.

Tracking document history correctly requires meticulous use of correct document identifications; see "Identifying documents and parts uniquely" on page 16.

Using document history

"Finding metadata for an ingredient" on page 12 gave the example of a movie that contains an animated sequence derived from a still image through several steps. Suppose that, after some time and work on the client side, you wanted to determine if a later version of the animation exists, requiring an update of the movie.

You could look in your asset management system for later versions of the animation (that is, the one with the same document ID as the movie ingredient). You could then walk through the change history of the animation, back to the instance that is currently included (as identified by the instance ID), in order to determine whether the changes warrant an update. You may need to do more detailed comparisons to

determine whether a specific change warrants an update. The history records only the general nature of a change from one instance to another, and the order of instances.

Alternatively, you could look in your asset management system for an earlier form of the document, who's instance ID corresponds to one stored in the derivation history. You could then find the `OriginalDocumentID` and `DocumentID` for that source document, and use them to determine if later versions exist in the source thread, in order to choose whether the document you currently have should be reconverted from one of those later versions.

Recording change events

Modifying actions in the `xmpMM:History` property are represented by the **ResourceEvent** data type, whose fields are defined in the namespace `http://ns.adobe.com/xap/1.0/sType/ResourceEvent#`, with the preferred field namespace prefix of `stEvt`.

Event information includes the instance ID of the modified resource, a timestamp, and the *agent* (application) that performed the action.

The modifying action itself (`stEvt:action`) is described by a verb in the past tense. Applications can define their own actions; these action values are predefined in the schema:

```
converted
copied
created
cropped
edited
filtered
formatted
version_updated
printed
published
managed
produced
resized
saved
```

Each history entry can also contain an optional list of specific parts of the document that were changed since the previous event (`stEvt:changed`). This is a semicolon-separated list of **Part** objects, each of which can indicate a hierarchy of any depth (see "Part specifications" on page 8).

If no changed-parts list is present, it can mean that the scope of changed components is unknown, or that every part of the document has changed. To be safe when tracking changes, you must assume that if no changed-parts list is present, anything in the document might have changed.

An application can use the parts-changed list together with the timestamp (`stEvt:when`) to avoid performing costly operations unnecessarily. For example, a video editor can use this information to avoid re-rendering the data from a clip if the metadata was changed, but the content was not. In this case, the file time stamp will have changed, but the metadata event record shows that it was a metadata-only change, so the full re-rendering of the video/audio data is not required.

Maintaining the history list

It is easy for nested document editing histories to grow to an unmanageable size. You should trim the list whenever possible by removing intermediate entries that are referenced by some specific operation. You can further condense the list by accumulating the changed parts from all merged entries.

For example, a sequence such as `"new/created, saved, edited, saved, edited, saved, edited, saved, edited, saved, edited, saved, branch/converted, saved"` could be collapsed by removing most of the redundant `"edit, save"` pairs, resulting in `"new/created, edited, saved, branch/converted, saved"` or even `"new/created, branch/converted, saved"`. This is especially true if the save operations were auto-saves.

Identifying documents and parts uniquely

Globally unique identifier (GUIDs) are generally used to identify objects unambiguously. The contents of the identified file, however, can change over time. Depending on the situation, it might be desirable to refer to either:

- a *specific* state of the file as it exists at a point in time, or
- the file in general, as a *persistent* container whose content can change.

Some characteristics of a file (such as the application that created it) would normally be expected to be persistent over its life. Other characteristics (such as word count) would be expected to change as the content of the file changes. Some characteristics (such as copyright information or authors' names) would not normally be expected to change, but might change due to the addition of new copyrighted material, or some external event such as the copyright holder being acquired by another company.

For composed dynamic-media projects, the question of what constitutes a uniquely identifiable document becomes even more complicated. You must be able to track assets in various workflows. You might need to decide, for instance, how a document was derived, what other documents were used in its composition, or whether it was used in the composition of some other document; or you might want to track how many times you have used a particular asset in various projects throughout a suite of applications.

When files and parts of files can be included in a composite, you must identify each instance of an object, which can easily branch from another instance that originated from the same file. If a portion of document A is included in document B, then document A is edited, the portion that was included has diverged from the original. Whether the two versions are considered to be the same document depends on the context of the task at hand. It is very important that, when an application creates or edits any dynamic-media document, you create or maintain the various identifiers in strict accordance with the requirements described here.

> **TERMINOLOGY NOTE:** The XMP media-management schema uses the term resource to refer to the "thing the metadata is about" in a general sense. Depending on the context, properties may refer to either specific or persistent aspects of the described file. In order to refer unambiguously to a specific state of the file, we use the term instance.
>
> This terminology should be distinguished from HTTP terminology, where resource is most often used in the sense of "container", while entity or entity-part is always used to mean "the current content of all or part of a resource at some point in time."

Assigning document identifiers

The XMP Media Management schema defines these different identifiers:

- `xmpMM:OriginalDocumentID`: As media is imported and projects is started, an *original-document ID* must be created to identify a new document. This identifies a document as a conceptual entity.

- `xmpMM:DocumentID`: When a document is copied to a new file path or converted to a new format with Save As, another new *document ID* should usually be assigned. This identifies a general version or branch of a document. You can use it to track different versions or extracted portions of a document with the same original-document ID.
- `xmpMM:InstanceID`: To track a document's editing history, you must assign a new *instance ID* whenever a document is saved after any changes. This uniquely identifies an exact version of a document. It is used in resource references (to identify both the document or part itself and the referenced or referencing documents), and in document-history resource events (to identify the document instance that resulted from the change).

NOTE: The Media Management schema defines the `VersionID` and `Version` properties, as well as the specific document identifiers. These are not GUIDs for the document. They are meant to associate the document with a product version that is part of a release process. They can be useful in tracking the document history, but should not be used to identify a document uniquely in any context.

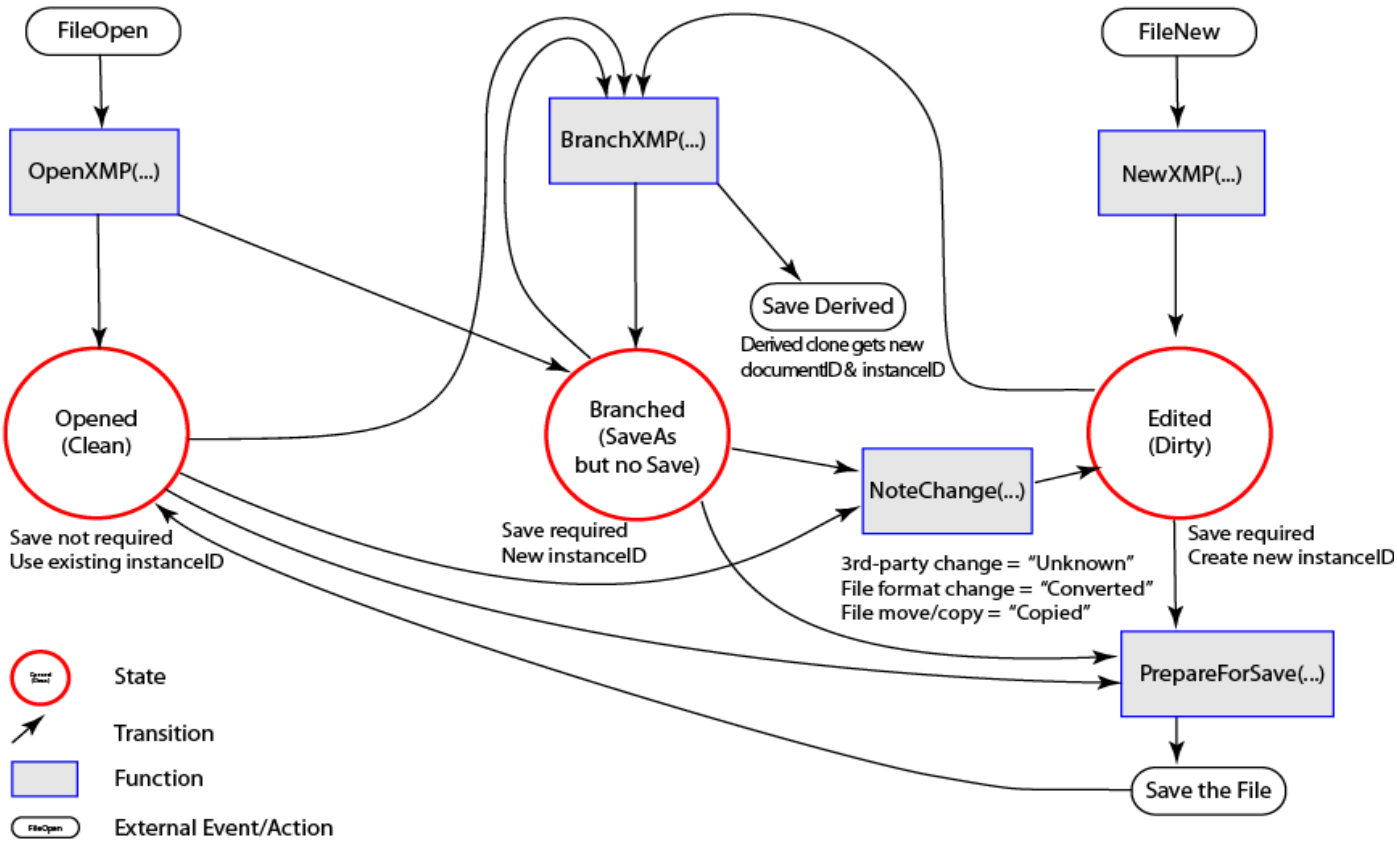
There are three change states that determine when these IDs should be assigned as the document is opened, edited, and saved. Follow these guidelines in assigning identifiers

- When a new document is created (including when it is created from a template), create a new GUID as the value of both `documentID` and `originalDocumentID`. This indicates that this is the first document in the history/derivation chain. This document should also receive a new `instanceID` (which might or might not match the `documentID`).
- As the document is modified or worked on, create a new `instanceID` every time the file is saved or checkpointed.
- If the document is modified sufficiently that it is no longer similar to the original, create a new `documentID`, and insert a `DerivedFrom` record into the document.
- If the document is saved to a new name, but then stripped sufficiently that it no longer would be considered a derivation of the original document, create a new `originalDocumentID` and copy it to `documentID`.

State		ID status
`Opened`	Clean, no changes since the file was opened and its metadata retrieved.	Save is not required, the object maintains the existing `DocumentID` and `InstanceID`.
`Edited`	Dirty, there are unsaved changes in the file, or the file has been newly created but not yet saved.	Save is required. Upon save the object gets a new `InstanceID`. ➤ If it was loaded and edited, this new ID distinguishes it from the version of the file that was loaded. ➤ If it was created, this is the initial `InstanceID`.

State		ID status
`Branched`	This state results when: ➤ A clean or edited file was copied (SaveAs), but not saved. ➤ Upon opening the file, the XMP parser found external changes in the metadata. External changes can be: ➢ Third-party (`Unknown`) ➢ File format (`Converted`) ➢ File move or copy (`Copied`)	Save is required. Upon SaveAs, the clone always gets a new `DocumentID` and `InstanceID`. ➤ If the original document was branched while clean, maintains its `instanceID`. ➤ If the original document was changed before branching, changes are still flagged in the original file, and it becomes Edited again, and gets a new `instanceID` when saved. ➤ If external changes are found upon load, the document gets a new `InstanceID` immediately, and retains it until edited. If it is branched before editing, the original can still be saved as a unique instance.

The following state-transition diagram illustrates implementation considerations for responses to particular events, and the resulting states.



Generating identifiers

An ID should be guaranteed to be globally unique (in practical terms, this means that the probability of a collision is so remote as to be effectively impossible). Typically 128- or 144-bit numbers are used, encoded as hexadecimal strings.

XMP does not require any specific scheme for generating the unique number. There are various common schemes available for that purpose, such as:

- Using physical information such as a local Ethernet address and a high resolution clock. When creating a unique ID, applications must consider trade-offs between privacy and the desire to create an audit trail. Adobe applications favor privacy and do not include Ethernet addresses.
- Using a variety of locally unique and random data, then computing an MD5 hash value. This avoids privacy concerns about the use of Ethernet addresses. It also allows for regeneration of the ID in some cases; for example if the MD5 hash is computed using the image contents for a resource that is a digital photograph.

Identifier URI schemes

The following URI schemes are used in some versions of XMP software to ensure unambiguous ID references:

- `xmp.iid`: Used for `InstanceID`
- `xmp.did`: Used for `DocumentID`

  These have the same syntax as `InstanceID` and `DocumentID`; globally unique 128-bit numbers encoded in hexadecimal. However, each is used only in the correct context. For example, the following unambiguously refers to "a file that has this ID as its `InstanceID` in its XMP Packet":

  ```
  xmpMM:InstanceId="xmp.iid:b9db20421f30bb3fe10e5f90"
  ```

  Similarly:

  ```
  xmpMM:DocumentId="xmp.did:b9db20421f30bb3fe10e5f90"
  xmpMM:OriginalDocumentId="xmp.did:b9db20421f30bb3fe10e5f90"
  ```

  The `xmp.iid` and `xmp.did` URIs are used only for files that have XMP actually stored in the file, not in sidecar files. Files that do not have embedded XMP still have `InstanceID` and `DocumentID`, but not in this form. Some documents that do have embedded XMP might still have a `DocumentID` that does not use `xmp.did`, because the document started out as a file without XMP but with some other kind of unique document identifier.

  These are URIs (that is, they uniformly identify resources), but they are only useful as URLs (Uniform Resource Locators) if there is an index of files with XMP indexed by the IDs found within them. In the case of `xmp.did`:, it might be necessary to examine modification dates and history to determine which is the latest of multiple instances with the same document ID.

In addition, the following URI scheme may be found within XMP applications:

- `hashuri`: Of the form `hashuri:xxxxxxxxxxxxxxxxxxxxxxxx`.

  Generated by taking a URI (either absolute or relative), hashing it (using MD5), and representing the result as a hexadecimal string. Identifies the same resource that was originally identified by the URI that was hashed.

  This might be used anywhere you would otherwise use a URI but need to obscure the actual URI string for privacy reasons. A URI is only useful as a URL if you have an index that caches the URI and the associated files. However a `hashuri` can be compared to another (calculated with the same hash scheme) to determine if you have the same file.

2 Temporal Information in Dynamic Media

Temporal information in metadata is used to track two different kinds of temporal relationships:

- There are time relationships between a parent document and its ingredients. For instance, the included part of an ingredient video clip might be a stretch from the 15th to the 30th frame, and this might be copied into the 110th to 115th frame of the containing movie. Temporal metadata gives you the ability to track the placement sequence of assets into the parent object

  The ways in which XMP specifies these relationship are discussed in "Timed ingredients" on page 21.

- Within any dynamic-media document or component, you need to mark specific points or spans of time within the timeline of that document. XMP accomplishes this with time *markers*, which can contain descriptions of events, indicate go-to points in the document, record synchronized speech or lyrics, and so on. Markers can be grouped into named *tracks* that hold markers of a specific type or from a specific source or stage of production.

  The syntax and usage of markers and tracks are discussed in "Representing temporal markers in metadata" on page 23.

- These usages interact when you need to retain timed values (markers) from ingredients in the metadata of composite documents. In order to enhance performance, or to retain editability in parts that would otherwise be invariant, you may need to flatten the marker hierarchy when copying ingredient metadata into the pantry of a new container.

  These techniques are discussed in "Retaining markers from nested ingredients" on page 28.

Timed ingredients

All or part of one dynamic document can be included in another, as in the example given in "Finding metadata for an ingredient" on page 12, where a movie contains an animation derived from a still image. In this case, the ingredient list for parent document (the movie) contains a **ResourceReference** entry for the included component (the animation). The entry must specify the time stretch that is included (in the `stRef:fromPart`), while the `stRef:toPart` must specify the time location where the part now resides in the parent document (the movie).

Specifying time values for parts

A time-range specification can be the entire **Part** value, or it can be appended to the specification. For example, a part can be specified as `"/content/time:0d15"`, or simply as `"time:0d15"`. If there is no time modifier, the default time range is `"all"`.

The time-range value takes one of these forms:

`[/]time:##d##`	A duration, specified by a start time, and a duration time (a frame-count value).

`[/]time:##r##`	A range, specified by a start time and an end time, both expressed as a number of frames relative to the beginning of the part.
`[/]time:##`	A frame count, as for a time value in a marker. This specifies a start time, with the default end time of `"maximum"`.

- In these syntax statements, the `##` notation can be replaced by any form of a frame-count specifier, as described in "Specifying time values for markers" on page 27.
- In a `fromPart` or `toPart` value, the leading `/` is optional. For an `stEvt:changed` part descriptor in a history record, the leading `/` is required.
- For a `fromPart` value, the start time is an offset from the start of the current ingredient's file. For a `toPart` value, the start time is measured from the start of the destination file.

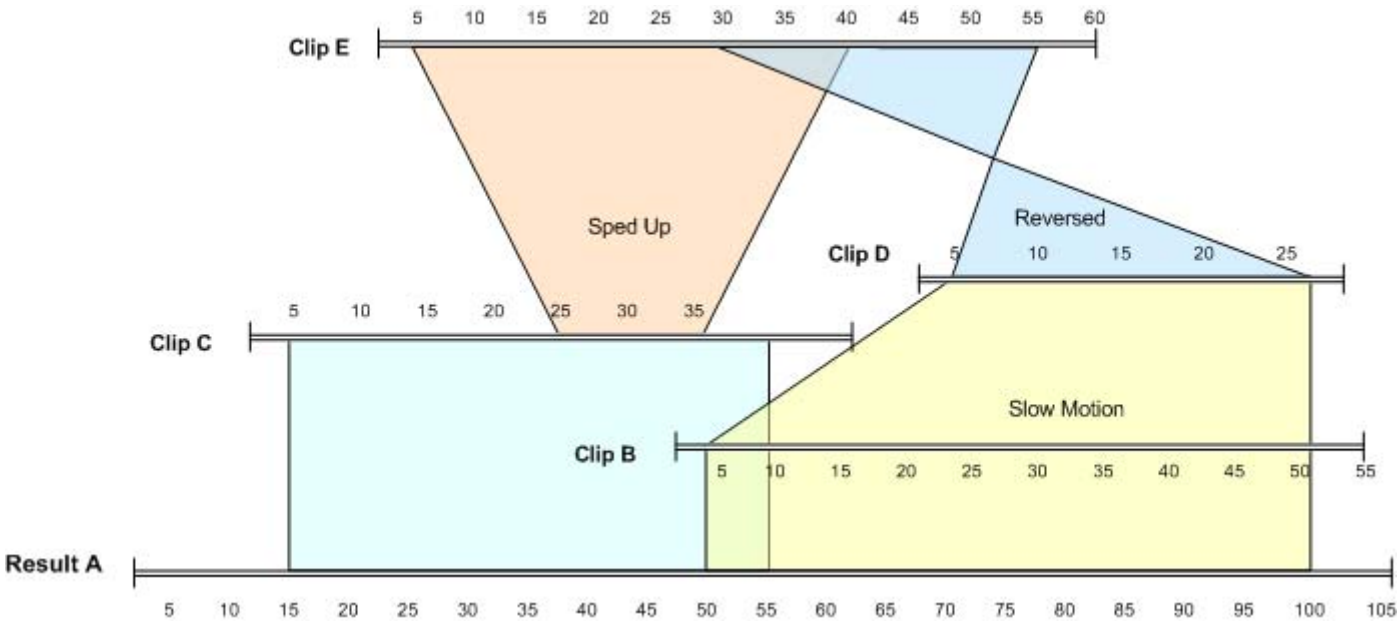
If time values are not specifically given, the default start time is 0, meaning the beginning of the relevant file. The default duration for a member of the ingredients list is "maximum", meaning the full length of the part. (This differs from the default duration for a marker, which is 0.)

Mapping times from source to target parts

A resource reference can specify a mapping function (`stRef:partMapping`) that is used to transform the time sequence of the `fromPart` into a new time sequence in the `toPart`.

The default for time mappings is `"linear"`, meaning that the `fromPart` time sequence is mapped into a specified portion in the timeline of the file in which it now resides. Depending on frame-rate and duration, the result may be a frame-by-frame mapping, or a rough-motion/slow-motion playback. The linear mapping scales the part uniformly to fill the new duration. A mapping of `"linear-reversed"` results in backward motion, which is also uniformly scaled in time if necessary.

The following figure shows some ways in which parts can be mapped into the timeline of a composed dynamic-media document.



These are the `fromPart`, `toPart`, and `partMapping` entries for the ingredients at the various levels of composition:

Document	Ingredients	Mapping

Result A	Clip C	`fromPart = /time:5d40'`
		`toPart = /time:15d40'`
		`partMapping = 'linear'`

	Clip B	`fromPart = '/time:5d45'`
		`toPart = '/time:50d45'`
		`partMapping = 'linear'`

Clip B	Clip D	`fromPart = '/time:5d20'`
		`toPart = '/time:5d45'`
		`partMapping = 'linear'`

Clip C	Clip E	`fromPart = '/time:5d35'`
		`toPart = '/time:25d10'`
		`partMapping = 'linear'`

Clip D	Clip E	`fromPart = '/time:30d25'`
		`toPart = '/time:5d20'`
		`partMapping = 'linear-reversed'`

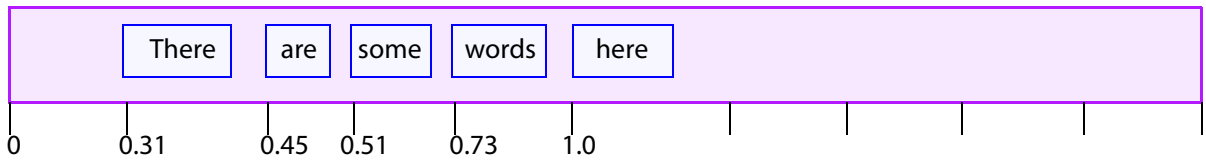
Clip E	none	

Representing temporal markers in metadata

An audio or moving image sequence is synchronized with other elements through the use of marked times or spans on a timeline. A marked time can be a simple point of reference that the editor or viewer can display or find. Timed sequences can be lyrics, captions, recognized speech, original scripts, and so on.

Temporal metadata provides a way for applications to represent the time-related data in its proper timed context. For example, assets that have audio tracks can have their audio transcribed to text, and the transcript written to the file's metadata. The Speech-to-Text feature of Adobe Premiere and Adobe Soundbooth™ allows these transcripts to be viewed in the Main Metadata panel. Similarly, Adobe After Effects® displays text-over-time using temporal markers which are derived from metadata and shown in the Timeline view.

Recognized speech, for example, is attached to the timeline of a dynamic-media document:



This particular speech fragment could be represented like this, using the format *word*`[`*start*`\`*duration*`]`:

```
There[0.31\0.14], are[0.45\0.6], some[0.51\0.22], words[0.73\0.37], here[1.1\0.4]
```

Tracks and markers

To maintain temporal marking information about dynamic media, such as audio and video, the XMP model uses the concepts of *tracks* and *markers*. The properties that are used for this model are defined in the XMP Dynamic Media namespace, `http://ns.adobe.com/xmp/1.0/DynamicMedia/`, with the preferred schema namespace prefix of `xmpDM`.

- A **Track** is a named set of markers, as for a clip in a video, or a song in an album. An unordered list of all tracks in a document is kept in `xmpDM:Tracks`. A track can specify a default frame rate and default marker type that apply to all markers in the set.
- A **Marker** is a datatype that contains information about a specific location in a timed sequence, with a start time, an optional duration, and other optional information, such as a name and type. Markers are generally contained in tracks. An ordered list of all markers in a track is kept in `xmpDM:markers`.

The example speech sequence, when recorded as a track in Premiere Pro, is described in the XMP metadata as follows. Each word is described by a value of the `xmpDM:markers` array. The times are given in milliseconds. The syntax and usage of the `xmpDM:Tracks` and `xmpDM:markers` properties are discussed in the following sections.

```
<xmpDM:Tracks>
    <rdf:Bag>
        <rdf:li>
            <rdf:Description
                xmpDM:trackName="Text transcription aggregated by Premiere Pro CS4"
                xmpDM:trackType="Speech"
                xmpDM:frameRate="f1000">
                <xmpDM:markers>
                    <rdf:Seq>
                        <rdf:li
                            xmpDM:startTime="310"
                            xmpDM:duration="140"
                            xmpDM:name="there"
                            xmpDM:speaker="Speaker0"
                            xmpDM:probability="30"/>
                        <rdf:li
                            xmpDM:startTime="450"
                            xmpDM:duration="60"
                            xmpDM:name="are"
                            xmpDM:speaker="Speaker0"
                            xmpDM:probability="80"/>
                        <rdf:li
                            xmpDM:startTime="510"
                            xmpDM:duration="220"
                            xmpDM:name="some"
                            xmpDM:speaker="Speaker0"
                            xmpDM:probability="41"/>
                        <rdf:li
                            xmpDM:startTime="730"
                            xmpDM:duration="370"
                            xmpDM:name="words"
                            xmpDM:speaker="Speaker0"
                            xmpDM:probability="61"/>
                        <rdf:li
                            xmpDM:startTime="1100"
                            xmpDM:duration="340"
                            xmpDM:name="here"
                            xmpDM:speaker="Speaker0"
```

```
                        xmpDM:probability="29"/>
                </rdf:Seq>
            </xmpDM:markers>
        </rdf:Description>
    </rdf:li>
  </rdf:Bag>
</xmpDM:Tracks>
```

This model allows you to search the metadata of, for example, song lyrics, without regard to the timing information, while still retaining the time-code values.

The model is extensible. The data representation can be extended to cover language alternatives (translated captions, for example). Attributes can be merged, combined (to make all the captions a single string, for instance), split (to break lyrics up into stanzas, for example) or otherwise manipulated. If there are multiple attributes for a given object that have time-code markers (for example, both recognized speech and lyrics), you could choose to interleave the values or to treat them as separate layers.)

Marker types

Markers are used in audio and video editing for many purposes. The simplest marker has neither a specific type nor a duration, and is used simply to mark a time (the `xmpDM:startTime` value). When editing, you can snap things, such as the current time indicator, to such a marker, or use it as a reminder or placeholder.

You can specify an optional *type* value to indicate how a marker or set of markers is intended to be used. The type tells you what other information is associated with it.

- An `xmpDM:trackType` value in a **Track** specifies a default type for all of the markers in the track. In the example, all of the markers in the track represent speech values, as specified by `xmpDM:trackType="Speech"`.
- You can also specify one or more types for an individual **Marker**, using the `xmpDM:type` value. This overrides the type given in the track, if any.

The type of a marker can control how an editor behaves when editing or exporting it. For example, `xmpDM:type="FLVCuePoint"` means that it is intended as a cue point when exported to Flash, and that it has cue-point information in it. Markers of particular types are typically displayed differently in an editor; in a particular color, or in a particular part of the UI, for instance.

A track or marker can have multiple types; for example, `xmpDM:type="Chapter,WebLink"` indicates a marker that is both a chapter marker and a link.

Predefined types for XMP markers and tracks include:

`AlertAudio...` (various types)	An audio marker with no duration, which associates a point in time with a condition, enabling the application to warn when that condition is encountered. For example, a clipping alert can warn when the top of the scale is reached. Audio alerts are used in OnLocation.
`AlertUserNote`	The marker contains a user-supplied description in the `xmpDM:comment` field.
`AlertVideoTooLow` `AlertVideoTooHigh`	A video marker with a duration, which allows an application to respond to the brightness level within the marked segment. This type of marker is displayed in OnLocation and After Effects.

`Beat`	An audio marker (cue type) that is set when Adobe Audition® automatically detects beats; the duration is always zero.
`Chapter`	Marks the start of a chapter, for example in a DVD. Corresponds to the "Encore Chapter Marker" option in Premiere Pro's Marker dialog.
`Cue`	A basic audio marker type, which can have a duration or not. This type maps directly to the RIFF standard `cue` chunk used for WAV and AVI formats.
`FLVCuePoint`	The marker will be converted to a Cue Point when exported to Flash (FLV), using the `xmpDM:cuePointType` and `xmpDM:cuePointParams` values in this marker. See following example. Corresponds to the "Flash Cue Point" option in Premiere Pro's Marker dialog.
`Index`	An audio marker (cue type) that identifies an index within a CD audio track, used by some high-end audio CD players. Adobe Audition writes these when burning CDs. The duration is always zero. This type of marker must be aligned on 75 fps boundaries.
`Speech`	The marker contains text created by speech-recognition technology, manually entered, or by other means. The marker's `xmpDM:name` contains the phrase, word, or syllable. The marker can also contain an `xmpDM:speaker` value, and, if created through speech recognition, an `xmpDM:probability` value.
`Track`	An audio marker (cue type) that identifies a CD audio track delimiter. The duration is that of the audio track. This type of marker must be aligned on 75 fps boundaries.
`WebLink`	The marker is a link to the URL specified in this marker's `xmpDM:location` value (such as `"http://www.mysite.com"`). The marker can also contain an `xmpDM:target` value with a specific frame that should be set to view the URL. See following example. Corresponds to the "Web Link" option in Premiere Pro's Marker dialog.

This example shows the XMP for a sequence that contains markers of various types:

```
<xmpDM:Tracks>
   <rdf:Bag>
      <rdf:l rdf:parseType="Resource">
         <xmpDM:trackName>layer_markers</xmpDM:trackName>
         <xmpDM:frameRate>f30</xmpDM:frameRate>
         <xmpDM:markers>
            <rdf:Seq>
               <rdf:li rdf:parseType="Resource">
                  <xmpDM:startTime>0</xmpDM:startTime>
               </rdf:li>
               <rdf:li rdf:parseType="Resource">
                  <xmpDM:startTime>10</xmpDM:startTime>
                  <xmpDM:duration>6</xmpDM:duration>
                  <xmpDM:comment>flyingthroughtheair</xmpDM:comment>
```

```
                    </rdf:li>
                    <rdf:li rdf:parseType="Resource">
                        <xmpDM:startTime>40</xmpDM:startTime>
                        <xmpDM:name>Chapter1</xmpDM:name>
                        <xmpDM:type>Chapter</xmpDM:type>
                    </rdf:li>
                    <rdf:li rdf:parseType="Resource">
                        <xmpDM:startTime>60</xmpDM:startTime>
                        <xmpDM:name>myfavoritecuepoint</xmpDM:name>
                        <xmpDM:type>FLVCuePoint</xmpDM:type>
                        <xmpDM:cuePointType>Event</xmpDM:cuePointType>
                        <xmpDM:cuePointParams>
                            <rdf:Seq>
                                <rdf:li rdf:parseType="Resource">
                                    <xmpDM:key>param1</xmpDM:key>
                                    <xmpDM:value>777</xmpDM:value>
                                </rdf:li>
                                <rdf:li rdf:parseType="Resource">
                                    <xmpDM:key>secondparam</xmpDM:key>
                                    <xmpDM:value>888</xmpDM:value>
                                </rdf:li>
                            </rdf:Seq>
                        </xmpDM:cuePointParams>
                    </rdf:li>
                    <rdf:li rdf:parseType="Resource">
                        <xmpDM:startTime>70</xmpDM:startTime>
                        <xmpDM:name>Chapter2</xmpDM:name>
                        <xmpDM:type>Chapter,WebLink</xmpDM:type>
                        <xmpDM:location>http://www.adobe.com</xmpDM:location>
                        <xmpDM:target>right_frame</xmpDM:target>
                    </rdf:li>
                </rdf:Seq>
            </xmpDM:markers>
        </rdf:li>
    </rdf:Bag>
</xmpDM:Tracks>
```

Specifying time values for markers

A time value in a marker (`xmpDM:startTime` or `xmpDM:duration`) is specified as a *frame count*, which must be associated with a *frame rate*. The default frame rate is 1 frame per second (fps).

A different frame rate can be included in the time specification (a **FrameCount** value), or the frame-count value can be interpreted according to the frame rate of the containing track. Thus, a **FrameCount** specification can take any of these forms:

`"##"`	For a marker that is not in a track (or for a `time` value in a **Part**) a simple integer value is a number of frames, which is also a number of seconds, at the default frame rate of 1 fps. For a marker in a track, an integer value is interpreted as ticks/frames in the timescale specified by the track's `xmpDM:frameRate`. When the count is zero, no frame rate should be specified. The default start time and duration for a marker are both 0. The start time of a marker is measured from the start of the file that contains the track.

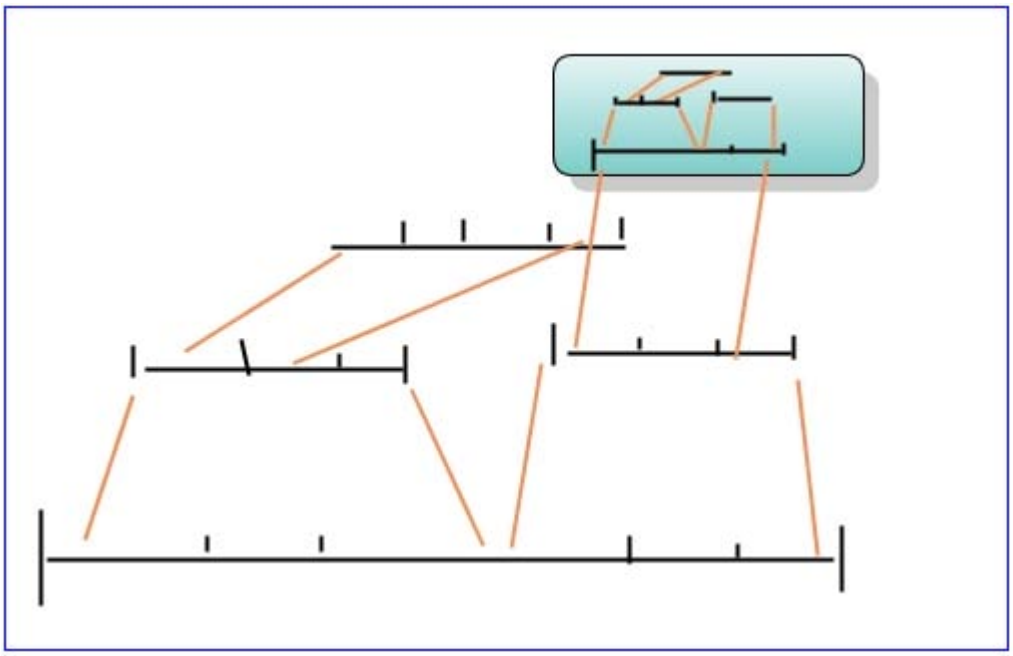
"##f###" "##f###s###"	A number of frames specified together with a frame rate, which can contain an optional rate basis. The rate basis defaults to 1.
"maximum"	Allowed for a duration value; indicates that the time span is unlimited, or is determined automatically up to the full duration of the source.

These examples show how a FrameCount value of 15 is expressed for common video and audio frame rates:

Film at 24 fps	"15f24"	0.625 seconds, frame rate = 24, rate basis = 1 Can be read as "15 24ths of a second or "15 frames at 24 fps"
Speech-to-text in milliseconds	"15f1000"	0.015 seconds, frame rate = 1000, rate basis = 1
NTSC at 29.97 fps	"15f30000s1001"	0.5005 seconds, frame rate = 30000, rate basis = 1001
DVATicks	"15f254016000000"	0.000000000059051398+ seconds, frame rate = 254016000000, rate basis = 1

Retaining markers from nested ingredients

The simple inclusion hierarchy for metadata can become complicated when an item that already has complex, hierarchical, temporal metadata is used in another composition.



The markers in the metadata for each ingredient contain time-code values that reference the timeline of that ingredient. When part of a dynamic document is imported, all of the markers must be mapped to the timeline of the outermost document, as described in "Mapping times from source to target parts" on page 22.

When you add an ingredient or part to the timeline of a document, you typically flatten the markers from the hierarchical ingredient structure of the part and its components into a simple set of markers, where each marker corresponds to some metadata start event. To flatten a set of markers:

- Walk the ingredient hierarchy and extract all markers from the pantry entries that pertain to the sequence used in the current container.
- Remap the times of these markers onto the time scale of the outermost document.

Normally, the resulting flattened view would be used for read-only data in an application; that is, a view that does not allow editing, and is not actually used during rendering. However, it is possible to *promote* the markers from an included hierarchy, so that the complex metadata tree from an original source is flattened into one or more sets of markers for the current container. To do this:

- Collect the extracted markers into a track.
- Insert the track into the pantry of the new container, in the pantry entry for the document being imported.
- Set the `stRef:maskMarkers` property in each ingredient in the outermost document to `"All"` (the default value is `"None"`), to prevent redundant processing of these same markers in those ingredients.

Promoting markers in this manner creates copies of them; these same markers are still to be found in the pantry entries for documents further down the ingredient hierarchy. If you do not mark the ingredients properly, the nested markers can be redundantly processed, next time this data is included somewhere.

This is how Adobe applications treat markers in metadata when parts are added to timelines, so if you mark included parts correctly, they will behave correctly when used with those applications. You should also check the mask-markers flag when walking the ingredient subtrees, and avoid regathering duplicated markers.

3 Code Example: Walking the Ingredient/Pantry Tree

This code example shows how to iterate through the inclusion hierarchy of nested ingredients of a composed document, and their associated pantry entries, to find a specific metadata value (`dc:title`) in each included document.

```
//-------------------------------------------------------------------------------------
// The Debug_PrintIngredientTree routine recursively walks the XMP object
// and prints a tree of all nested ingredients.
//-------------------------------------------------------------------------------------
// Call as follows:
//
//      Debug_PrintIngredientTree(
//            *var_XMPMetaPtr, // const SXMPMeta & in_BaseXMPMeta
//            NULL,           // XMP_StringPtr in_PathToSelectedPantryEntry,
//            0              // int in_NestLevel,
//      );
//-------------------------------------------------------------------------------------
// Output for the sample case of Docs A-F, illustrated in previous chapters:
//-------------------------------------------------------------------------------------
//  0: Base Document:
//      dc:title='DocF',
//      InstanceID='xmp.iid:961EBA74072068119457FF67DE63A97D'
//    Ingredient[1]: instanceID='xmp.iid:951EBA74072068119457FF67DE63A97D'
//    1: PantryEntry for ingredient was found:
//        dc:title='DocE',
//        InstanceID='xmp.iid:951EBA74072068119457FF67DE63A97D'
//       Ingredient[1]: instanceID='xmp.iid:941EBA74072068119457FF67DE63A97D'
//       2: PantryEntry for ingredient was found:
//           dc:title='DocD',
//           InstanceID='xmp.iid:941EBA74072068119457FF67DE63A97D'
//         Ingredient[1]: instanceID='xmp.iid:911EBA74072068119457FF67DE63A97D'
//         3: PantryEntry for ingredient was found:
//             dc:title='DocA',
//             InstanceID='xmp.iid:911EBA74072068119457FF67DE63A97D'
//           Has no ingredients
//         Ingredient[2]: instanceID='xmp.iid:921EBA74072068119457FF67DE63A97D'
//         3: PantryEntry for ingredient was found:
//             dc:title='DocB',
//             InstanceID='xmp.iid:921EBA74072068119457FF67DE63A97D'
//           Has no ingredients
//         Ingredient[3]: instanceID='xmp.iid:931EBA74072068119457FF67DE63A97D'
//         3: PantryEntry for ingredient was found:
//             dc:title='DocC',
//             InstanceID='xmp.iid:931EBA74072068119457FF67DE63A97D'
//           Has no ingredients
//-------------------------------------------------------------------------------------
// Function prototype
static void Debug_PrintIngredientTree(
    const SXMPMeta & in_BaseXMPMeta,             // The outermost doc's xmpMeta.
    XMP_StringPtr    in_PathToSelectedPantryEntry, // Path to selected Ingredient's
    //                                          pantryEntry. The path is
    //                                          initially NULL for outermost
    //                                          doc. It is filled in by this
    //                                          routine and passed to the
```

```
    //                                          recursive calls to access
    //                                          the pantry entries.
    int            in_NestLevel                // Used in formatting printf
    //                                          output, not needed for the
    //                                          actual traversal.
);

// The following additional XMP namespace identifiers are used.
#define kXMP_NS_IngredientsArray     kXMP_NS_XMP_MM
#define kXMP_NS_IngredientsEntry     kXMP_NS_XMP_ResourceRef
#define kXMP_NS_PantryArray          kXMP_NS_XMP_MM
#define kXMP_NS_PantryEntry          kXMP_NS_XMP_MM

// The following additional XMP property name identifiers are used.
#define kXMP_Property_IngredientsArray    "Ingredients"
#define kXMP_Property_PantryArray         "Pantry"

// Function implementation
void Debug_PrintIngredientTree(
    const SXMPMeta & in_BaseXMPMeta,             // The outermost doc's xmpMeta.
    XMP_StringPtr    in_PathToSelectedPantryEntry, // Path to selected Ingredient's
    //                                          pantryEntry. The path is
    //                                          initially NULL for outermost
    //                                          doc. It is filled in by this
    //                                          routine and passed to the
    //                                          recursive calls to access
    //                                          the pantry entries.
    int            in_NestLevel                // Used in formatting printf
    //                                          output, not needed for the
    //                                          actual traversal.
)
{
    // This code shows the differences in how to access data that is held directly in
    //  the base document's XMPMeta object (var_isTopLevel==true), as opposed to
    //  accessing the same data if it is held within a pantry entry
    //  (var_isTopLevel==false).

    bool var_isTopLevel = true;
    if(
        ( in_PathToSelectedPantryEntry == NULL )
        ||
        ( in_PathToSelectedPantryEntry == "" )
    )
    {
        // If the in_PathToSelectedPantryEntry entry is NOT supplied, it is
        //   accessing the baseDocument (sets var_isTopLevel=true).
        var_isTopLevel = true;
    }
    else
    {
        // If the in_PathToSelectedPantryEntry entry is supplied, this routine
        //  is accessing a pantryEntry (sets var_isTopLevel=false).
        var_isTopLevel = false;
    }


    // The var_ThisLevel_PrintIndent & var_NextLevel_PrintIndent variables
    //  are used in the formatting of the printf output,
    //  they are not needed for the actual traversal.
    TXMP_STRING_TYPE var_ThisLevel_PrintIndent = "";
```

```
for(
    int i = 0;
    i < in_NestLevel;
    ++i
)
{
    var_ThisLevel_PrintIndent += "   ";
}
TXMP_STRING_TYPE var_NextLevel_PrintIndent = var_ThisLevel_PrintIndent;
var_NextLevel_PrintIndent += "   ";

// This next block prints the InstanceID and dc:title of the baseDoc or
//  PantryEntry currently being processed.
if( var_isTopLevel )
{
    //-------------------------------------------------------------------------
    // This is how to access simple properties in the baseDoc (topLevel)
    //-------------------------------------------------------------------------

    // Find instanceID of base file
    TXMP_STRING_TYPE var_BaseDocInstanceID = "";
    in_BaseXMPMeta.GetProperty(
        kXMP_NS_XMP_MM, // Base NS differs between top-level and within-pantry.
        //                (The Base NS and the Field NS are the same for
        //               baseDoc access.)
        "InstanceID",
        &var_BaseDocInstanceID,
        NULL
    );

    // The title is not needed in the traversal, but is much more humanly
    //  readable than the InstanceID. It is used only in the printf output.
    TXMP_STRING_TYPE var_BaseDocTitle = "";
    in_BaseXMPMeta.GetProperty(
        kXMP_NS_DC,     // Base NS differs between top-level and within-pantry.
        "title",
        &var_BaseDocTitle,
        NULL
    );

    printf(
        "%s%d: Base Document: \n",
        var_ThisLevel_PrintIndent.c_str(),
        in_NestLevel
    );
    printf(
        "%s   dc:title='%s',\n",
        var_NextLevel_PrintIndent.c_str(),
        var_BaseDocTitle.c_str()
    );
    printf(
        "%s   InstanceID='%s'\n",
        var_NextLevel_PrintIndent.c_str(),
        var_BaseDocInstanceID.c_str()
    );
}
else
{
    //-------------------------------------------------------------------------
    // This is how to access simple properties in a pantryEntry.
```

```
    //---------------------------------------------------------------------

    // Find instanceID of requested pantry entry from current Ingredient entry
    TXMP_STRING_TYPE var_PantryEntryInstanceID = "";
    in_BaseXMPMeta.GetStructField(
        kXMP_NS_PantryArray,  //  Base NS differs between top-level and
        //                     within-pantry.
        in_PathToSelectedPantryEntry,
        kXMP_NS_XMP_MM,      // Field NS does not differ between top-level
        //                     and within-pantry.
        "InstanceID",
        &var_PantryEntryInstanceID,
        NULL
    );

    // The title is not needed in the traversal, but is much more humanly
    //  readable than the InstanceID. It is used only in the printf output.
    TXMP_STRING_TYPE var_PantryEntryTitle = "";
    in_BaseXMPMeta.GetStructField(
        kXMP_NS_PantryArray,  // Base NS differs between top-level and
        //                     within-pantry
        in_PathToSelectedPantryEntry,
        kXMP_NS_DC,          // Field NS does not differ between top-level
        //                     and within-pantry.
        "title",
        &var_PantryEntryTitle,
        NULL
    );

    printf(
        "%s%d: PantryEntry for ingredient was found:\n",
        var_ThisLevel_PrintIndent.c_str(),
        in_NestLevel
    );
    printf(
        "%s   dc:title='%s',\n",
        var_NextLevel_PrintIndent.c_str(),
        var_PantryEntryTitle.c_str()
    );
    printf(
        "%s   InstanceID='%s'\n",
        var_NextLevel_PrintIndent.c_str(),
        var_PantryEntryInstanceID.c_str()
    );
}


//-----------------------------------------------------------------------------
// To find all ingredients and nested ingredients in the current doc/pantryEntry,
//   find each ingredient's ingredietnEntry, then get its pantryEntry and
//   recursivly process the ingredients in the pantryEntry.
//-----------------------------------------------------------------------------
// 1.) Get the path to the Ingredients (this is different for top-level
//     and pantry access).
// 2.) Get the ingredient count
// 3.) For each entry in the IngredientList,
//     find the path to that IngredientEntry.
// 4.) Using that path, look up the ingredient's "instanceID"
//     (Note lowercase 'i')
// 5.) Now, given the path to the Pantry
```

```
//     (always in the top-level document),
//     find the count of the number of entries in the pantry.
// 6.) For each entry in the Pantry, find the path to that PantryEntry.
// 7.) Using that path, look up the PantryEntry's "InstanceID"
//     (Note uppercase 'I')
// 8.) If the ingredient's instanceID matches the
//     PantryEntry's InstanceID, you have the correct
//     PantryEntry to process further (recursively).
//-------------------------------------------------------------------------------
// These steps are implemented in the following code
//-------------------------------------------------------------------------------

// 1.) Get the path to the Ingredients (this is different for top-level
//     and pantry access).
TXMP_STRING_TYPE var_PathToIngredientsArray = "";
if( var_isTopLevel )
{
   // Set "IngredientArrayPath" within base file
   var_PathToIngredientsArray = kXMP_Property_IngredientsArray;
}
else
{
   // Set "IngredientArrayPath" within pantry entry
   SXMPUtils::ComposeStructFieldPath(
      kXMP_NS_PantryArray,
      in_PathToSelectedPantryEntry,
      kXMP_NS_IngredientsArray,
      kXMP_Property_IngredientsArray,
      &var_PathToIngredientsArray
   );
}

// 2.) Get the ingredient count
int var_IngredientCount = 0;
var_IngredientCount = in_BaseXMPMeta.CountArrayItems(
   kXMP_NS_IngredientsArray,
   var_PathToIngredientsArray.c_str()
);
if( var_IngredientCount == 0 )
{
   printf(
      "%sHas no ingredients\n",
      var_NextLevel_PrintIndent.c_str()
   );
}
for(
   int var_IngredientIndex = 1; // WARNING: This is a 1-based index
   var_IngredientIndex <= var_IngredientCount;
   ++var_IngredientIndex
)
{
   // 3.) For each entry in the IngredientList,
   //     find the path to that IngredientEntry.
   TXMP_STRING_TYPE var_PathToIngredientEntry = "";
   SXMPUtils::ComposeArrayItemPath(
      kXMP_NS_IngredientsArray,
      var_PathToIngredientsArray.c_str(),
      var_IngredientIndex,
      &var_PathToIngredientEntry
   );
```

```
// 4.) Using that path, look up the ingredient's "instanceID"
//     (Note lowercase 'i')
TXMP_STRING_TYPE var_IngredientInstanceID = "";
in_BaseXMPMeta.GetStructField(
   kXMP_NS_IngredientsArray,
   var_PathToIngredientEntry.c_str(),
   kXMP_NS_IngredientsEntry,
   "instanceID",
   &var_IngredientInstanceID,
   NULL
);
printf(
   "%sIngredient[%d]: instanceID='%s'\n",
   var_NextLevel_PrintIndent.c_str(),
   var_IngredientIndex,
   var_IngredientInstanceID.c_str()
);

//-----------------------------------------------------------------------------
// Find the pantryEntry that corresponds to that ingredient's instanceID.
//  (This code written for clarity, not for performance. May be slow for
//  files with large numbers of pantry entries.)
//-----------------------------------------------------------------------------

// 5.) Now, given the path to the Pantry
//     (always in the top-level document),
//     find the count of the number of entries in the pantry.
XMP_Index var_PantrySearchCount = in_BaseXMPMeta.CountArrayItems(
   kXMP_NS_PantryArray,
   kXMP_Property_PantryArray
);
bool var_MatchingPantryEntryWasFound = false;
for(
   XMP_Index var_PantrySearchIndex = 1; // WARNING: This is a 1-based index
   var_PantrySearchIndex <= var_PantrySearchCount;
   ++var_PantrySearchIndex
)
{
   // 6.) For each entry in the Pantry,
   //     find the path to that PantryEntry.
   TXMP_STRING_TYPE var_PathToPantrySearchEntry = "";
   SXMPUtils::ComposeArrayItemPath(
      kXMP_NS_PantryArray,
      kXMP_Property_PantryArray,
      var_PantrySearchIndex,
      &var_PathToPantrySearchEntry
   );

   // 7.) Using that path, look up the PantryEntry's "InstanceID"
   //     (Note uppercase 'I')
   TXMP_STRING_TYPE var_PantrySearchEntryInstanceID = "";
   in_BaseXMPMeta.GetStructField(
      kXMP_NS_PantryArray,
      var_PathToPantrySearchEntry.c_str(),
      kXMP_NS_XMP_MM,
      "InstanceID",
      &var_PantrySearchEntryInstanceID,
      NULL
   );
```

```
            if( var_PantrySearchEntryInstanceID == var_IngredientInstanceID )
            {
                // 8.) If the ingredient's instanceID matches the
                //     PantryEntry's InstanceID, you have the correct
                //     PantryEntry to process further (recursively).

                var_MatchingPantryEntryWasFound = true;


                // Recursively call for each matching/found pantry entry
                Debug_PrintIngredientTree(
                    in_BaseXMPMeta,        // outermost doc's xmpMeta
                    //                      const SXMPMeta & in_BaseXMPMeta,

                    var_PathToPantrySearchEntry.c_str(),  // path to pantryEntry
                    //                XMP_StringPtr in_PathToSelectedPantryEntry,
                    ( in_NestLevel + 1 )    // int in_NestLevel,
                );
            } // if( var_PantrySearchInstanceID == var_IngredientInstanceID )
        } // for( XMP_Index var_PantrySearchIndex = 1; ... )


        if( var_MatchingPantryEntryWasFound == false )
        {
            // This is not an error. The pantry can optionally be purged
            //  of entries where the timespan of the ingredient falls
            //  outside the visible timespan in of the result.
            printf(
                "%s%d: PantryEntry for Ingredient='%s' was not found.\n",
                var_NextLevel_PrintIndent.c_str(),
                ( in_NestLevel + 1 ),
                var_IngredientInstanceID.c_str()
            );
        }
    } // for( int var_IngredientIndex = 1; ... )
}
// End of sample code
```