Appscio(tm) Media Processing Framework (MPF) Development Guide

Table of Contents

1. Document Information

This section contains information about this Appscio document.

1.1 Document History

Version	Date	Author	Changes
1.0	Feb 15, 2009	Appscio, Inc.	Add MPF documentation
1.1	May 9, 2009	Appscio, Inc.	Minor modifications to multiple sections
1.2	Jun 3, 2009	Appscio, Inc.	Code modifications in section 6
1.3	Jul 8, 2009	Appscio, Inc.	Updates to section 3 through 6
1.4	Jul 13, 2009	Appscio, Inc.	Miscellaneous minor modifications

1.2 Document Conventions

The Appscio MPF Development Guide will observe the following conventions:

- Informational text will be provided using Arial, 10 point
- Instructions for any user input, including commands will be given in ***bold, italicized*** form
- Output from any actions will be shown using `Courier,10 point`
- References to components and modules will be made using `Courier-New, 10 point`

2. Introducing the Appscio Media Processing Framework (MPF)

Appscio is developing a new class of software to address the issues involved in capturing, integrating, synchronizing, evaluating and consuming semantic data generated by media algorithms.

Appscio approached the challenges associated with audio and video data (AV data) with the following set of key objectives:

- Promote rapid, widespread adoption
- Leverage existing technologies
- Simplify the development and deployment of media algorithms
- Allow portability across the spectrum from mobile devices to server farms
- Maximize the scalability of solutions
- Encourage the emergence of standards

The Appscio Media Processing Framework (MPF) is an open software framework that is independent of media formats and data types. To promote adoption it is distributed as open source software. To leverage existing work it is implemented as an extension to one of the most mature and respected open source projects – gstreamer (http://www.gstreamer.net) – and it employs the W3C (www.w3.org) standard for expressing metadata – Resource Description Framework (RDF). More information on RDF can be found at: (www.w3.org/RDF).

2.1 Understanding Basic Appscio MPF Terminology

At its heart, the Appscio MPF is a tool for configuring and executing "pipelines" composed of semantic media algorithms. A **Pipeline** is a sequence of **Algorithms** (wrapped with Appscio MPF code) configured to perform an application-specific **Service**. The services implemented in Pipelines can be as simple as transcoding a media file from one format to another or as complex as generating a transcript from the audio track, or recognizing people in the video stream.

Algorithms perform a wide variety of elementary functions and fall into distinct categories. **CODECs** compress (encode) and decompress (decode) audio and video (AV) signals. **Filters** modify frames in AV signals. **Detectors** identify objects (e.g. words, faces, text) and behaviors (e.g. motion) in AV channels. **Classifiers** and **Recognizers** associate structured fragments of metadata with objects and behaviors. **Analytics** distill, improve, combine and/or summarize fragments of metadata.

Appscio MPF provides a mechanism for communicating semantic metadata between Algorithms. Metadata is "published" and consumed through **Pads** implemented in the Appscio MPF software **wrappers** around elementary media algorithms. Algorithms are assembled in Pipelines and metadata flows "downstream" through Pads forming a "metadata channel."

2.2 About Appscio MPF and gstreamer

Appscio MPF extends gstreamer – an open source media handling toolkit. Since its inception, the gstreamer community has created large libraries of elements – algorithms typically developed for media playback applications. Most of the elements in the gstreamer libraries are CODECs and Filters.

Appscio MPF wrappers are designed to hide the complexity of the gstreamer environment and to add support for transporting metadata. Semantic algorithms can be turned into gstreamer elements by wrapping them with Appscio MPF code. Pipelines are defined by specifying combinations of native gstreamer and wrapped Appscio MPF elements in a particular order. Which elements are chosen and the order in which they are executed depends on the application.

2.3 Appscio MPF Architecture Overview

The core of the Appscio MPF architecture is the Analytics Pipeline, which is illustrated in *Figure 1*. The Analytics Pipeline supports moving media streams (e.g. audio, video) through a network of elements and components.

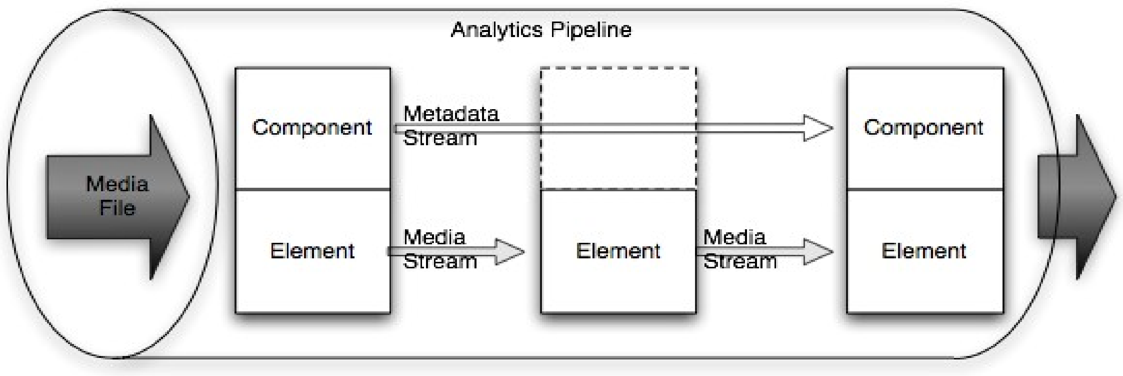

Figure 1 The Analytics Pipelines Uses Elements and Components

The Analytics Pipeline is built on top of gstreamer's pipeline and its associated packages of elements. The elements from gstreamer are as follows:

- **stream managers** (e.g. sources and sinks for HTTP or RTP, queues, multiplexers, or demultiplexers)
- **stream handlers** (e.g. codecs for getting streams in and out of various formats, or transformers to adjust color spaces, audio levels, video brightness, etc.)

Existing algorithms or media-processing software elements are wrapped using Appscio tools to become a component – a manageable, reusable code block with efficient connections into the metadata communication and management layer. The component wrapper layer depends on the software environment or on proprietary libraries to provide the transformative or analytic code.

Elements are low-level, while the components in the Analytics Pipeline recognize or react to higher-level objects (e.g. faces, actions, words). The components generate metadata and pass that metadata into the metadata stream while staying synchronized with the media stream. The metadata stream can either be exported to applications, or used by downstream components in recognizing yet-higher level events.

The low-level elements can be quickly intermixed with the components to meet specific data and format needs of the various components. The Pipeline manages the configuration negotiation to ensure each component gets input data of the type it specifies.

The primary architectural goals of Appscio MPF are:

1. pre-package repetitive code needed for creating media-centric applications
2. provide a pluggable framework that allows multiple vendor analytics modules to be joined together as application building blocks
3. gain cost effectiveness and security/quality improvements by adopting and participating in open source and open development practices
4. provide for future evolution in AV metadata availability and languages

Appscio MPF takes media input from files and produces both transformed media files and descriptive metadata.

3. Building Your First Component

In this section, let's explore the development of your first component.

3.1 Creating Your First Component

To create your first component, let's use the Terminal window and execute some commands to a) create a working directory to build components in and b) create your first component:

cd /home/<username>/

You should now be in your 'home' directory. Inside your home directory, let's create a working directory where we can build components:

mkdir mycomponents

Now, let's navigate to your component working directory:

cd mycomponents

pwd

The output from the pwd command should indicate that you are currently in your component working directory inside of your home directory (e.g. /home/<username>/mycomponents). Now, let's create your first component by executing the following in the Terminal window:

mpf-new-component.sh <your-component-name>

For example, you might create a component named *mycomponent1*, as follows:

```
./mpf-new-component.sh mycomponent1
```

3.2 Modifying Your First Component

Now that you have created your first component, let's modify it slightly and also note some of the characteristics of this new component. Using the Terminal window, execute the following commands:

cd /home/<username>

cd <component-working-directory>

cd <individual-component-directory>

So, for example, if you created the component working directory as *mycomponents* and your first component as *mycomponent1*, you could execute the commands, as follows:

```
cd /home/<username>
cd mycomponents
cd mycomponent1
```

In your first component directory (e.g. mycomponent1), edit the file ***<your-component-name>.c***, using your favorite text editor. For example, for a component named *mycomponent1*, you might do:

```
vi mycomponent1.c
```

Let's begin by modifying the name of the component author and its description using the editor. With the file open in the editor, locate these lines:

```
#define COMPONENT_DESC "Template component that passes data through unmodified"
#define COMPONENT_AUTH "Appscio, Inc. <info@appscio.org>"
```

Using the editor, change these two lines to read:

```
#define COMPONENT_DESC "This is my first component"
#define COMPONENT_AUTH "<your name><your e-mail address>"
```

While still in the editor, locate the following method called *component_class_init*:

```
component_class_init() {
 mpf_voidstar_add_input("input");
 mpf_voidstar_add_output("output");
```

In this method, the inputs and outputs are created.

Now, while still in the editor, locate the following method named *component_process*:

```
 /* Push the results out to the next component in the pipeline. */
 mpf_voidstar_push("output", mpf_voidstar_pull("input"));
```

The *component_process* method is where the component processes inputs and passes on the output data.

Save the <your -component-name>.c file and exit the editor

3.3 Building Your First Component

To build your first component, execute the following commands in the Terminal window:

```
cd /home/<username/<component-working-directory>/<component-directory>
```

For example: *cd /home/<username>/mycomponents/mycomponent1.*

```
./autogen.sh
make
```

Note: The ./autogen.sh script builds all the autotools files in your package. Any time you add or remove 3rd party library references in your project, you will need to run ./autogen.sh. Once distributed, an end-user will instead run ./configure, as that is what ./autogen.sh creates.

3.4 Setting Up Your Testing Environment

Now that your component is built, you must tell the underlying gstreamer system where to find the resulting component. To do this, you must add the current location of the file to GST_PLUGIN_PATH:

```
export GST_PLUGIN_PATH=$GST_PLUGIN_PATH:<component-working-
directory>/<component- name>/.libs
```

For example:

```
export GST_PLUGIN_PATH=$GST_PLUGIN_PATH:/home/<username>/mycomponents/mycomponent1
```

Note: The .libs directory is where libtool (which manages the construction of most all shared libraries on Linux) puts the component during the build process. When installed in /usr, the GST_PLUGIN_PATH will not need modification as /usr/lib/gstreamer is always in the path.

3.5 Using Your New Component

The new component you have just created is now a gstreamer-compliant Plugin that is ready to use. To verify that your environment is set correctly, execute the following in the Terminal window:

```
gst-inspect |grep <your-component-name>
```

You should see:

```
<your-component-name>: <your-component-name>: This is my first component
```

Use the Terminal window to run a test to see that your component is working:

```
cd tests
./simpletest
```

You should see some output similar to this:

```
Setting pipeline to PAUSED ...
Pipeline is PREROLLING ...
mpf-newcomponent buffer 1
Pipeline is PREROLLED ...
Setting pipeline to PLAYING ...
New clock: GstSystemClock
Got EOS from element "pipeline0".
Execution ended after <number> ms.
Setting pipeline to PAUSED ...
Setting pipeline to READY ...
Setting pipeline to NULL ...
FREEING pipeline ...
```

Look at the *simpletest* script to see how the pipeline was created and run using *gst-launch*.

This concludes the creation of your first component. From here you'll want to modify your component further to have it do something useful.

3.6 Expanding the Capabilities of Your Component

There are a number of ways to expand the capabilities of your new component and we will discuss some of these here.

3.6.1 Adding Libraries to your Component

To add particular libraries that your component may require, you have to modify the *configure.ac* file. At this point the library must be supplied with a *pkg-config* (.pc) file in order to be used.

There's a section in the *configure.ac* file that brings in the mpf-core package:

```
PKG_CHECK_MODULES(MPF_CORE, mpf-core,
  ,  AC_MSG_ERROR(no MPF Core libraries found (mpf-core)))
dnl make _CFLAGS and _LIBS available
AC_SUBST(MPF_CORE_CFLAGS)
AC_SUBST(MPF_CORE_LIBS
```

Simply copy that entire section for each of your external dependencies, and change all instances of *'mpf-core'* to the name of your package.

Note: When replacing "mpf-core" you must maintain the capitalization and “-” or “_” that's found at each replacement site.

3.6.2 Adding Source to your Component

To add another source file to the project, simply put it in place and add it to the list of sources for the component in *Makefile.am*:

```
libmycomponent_la_SOURCES = mycomponent.c othersource.c
```

Note: After adding or changing libraries or source you must execute:

./autogen.sh

This will insure that your changes are picked up in configure.ac or Makefile.am.

Next, execute:

make

This will complete the build process.

4. Using Datatype Libraries With Your Component

This section assumes that you have created a new component (e.g. *mycomponent1*) by executing the following:

```
./mpf-new-component.sh <my-component-name>
```

For an example, we will use the mpf-iplimage data-type which is used by many of the Open Computer Vision Library (OpenCV) components.

4.1 Datatype Includes

Let's begin by editing the *<my-comp-name>.c* file using your favorite source code text editor.

Add the include file for the data-type you want to use, for example:

```
/* Include the headers for any data-types needed. */
#include <mpf/mpf-voidstar.h>
#include <mpf/mpf-iplimage.h>
```

4.2 Adding Inputs

In the component_class_init() function, where input and output pads are created, add an IplImage input. You can limit your input by specifying particular IplImage formats, or as in this case, accept any format of IplImage.

```
component_class_init() {
mpf_voidstar_add_input("input");
mpf_voidstar_add_output("output");

/* Add an IplImage input */
mpf_iplimage_add_input("input_iplimage",
        MPF_IPLIMAGE_FORMAT_ANY);
```

You can also use mpf_iplimage_add_output("output_iplimage") to output IplImages to the next component in the pipeline.

4.3 Processing Streaming Data

In the component_process() function, where processing of streaming data occurs, pull in an IplImage from the input pad you created.

```
component_process() {
 /* Get the input */
 IplImage *src = mpf_iplimage_pull("input_iplimage");
```

Still in the component_process() function, do any processing of the IplImage, including making calls into the OpenCV library. Then, if you are not sending the IplImage out to the next component in the pipeline, you need to unreference it, as follows:

```
/* If the input image is not being sent down the pipeline then release it. */
mpf_iplimage_unref(src);
```

Otherwise, if you are sending it out to the next component in the pipeline, you would use the mpf_iplimage_push() function.

```
mpf_iplimage_push("output_iplimage", src);
```

4.4 Building The Component

To build your component using the mpf-iplimage data-type, you need to edit the configure.ac file, as follows:

Locate these lines:

```
dnl Uncomment these lines if you are using the IplImage data-type
dnl PKG_CHECK_MODULES(MPF_IPLIMAGE, mpf-iplimage,
dnl   ,  AC_MSG_ERROR(no MPF OpenCV IplImage libraries found (mpf-iplimage)))
dnl dnl make _CFLAGS and _LIBS available
dnl AC_SUBST(MPF_IPLIMAGE_CFLAGS)
dnl AC_SUBST(MPF_IPLIMAGE_LIBS)
```

Remove the "do not load" comment markers like this:

```
dnl Uncomment these lines if you are using the IplImage data-type
PKG_CHECK_MODULES(MPF_IPLIMAGE, mpf-iplimage,
, AC_MSG_ERROR(no MPF OpenCV IplImage libraries found (mpf-iplimage)))
dnl make _CFLAGS and _LIBS available
AC_SUBST(MPF_IPLIMAGE_CFLAGS)
AC_SUBST(MPF_IPLIMAGE_LIBS)
```

If you are using a different data-type library, you would add equivalent lines for that library. Then, you would rerun the autogen.sh script to recreate the build files.

```
./autogen.sh
```

Then, you would build your component, as follows:

```
make
```

Optionally, you can then install your component on the system, as follows:

```
sudo make install
```

You have now added the use of a data-type library to your component.

4.5 Creating Pipelines

To create pipelines using IplImage data, you can use the toiplimage and fromiplimage components, for example, as follows:

```
gst-launch filesrc location=/Media/GrandCanyonTrail.wmv ! \
decodebin ! ffmpegcolorspace ! toiplimage ! mycomp ! \
fromiplimage ! ffmpegcolorspace ! xvimagesink
```

5. Adding Parameters

In this section, we focus on adding parameters.

Begin by editing the *<my-component-name>.c* using your favorite source code text editor.

In the *component_class_init()* function, define and add the parameters for your component.

For example, to add an integer parameter called "object_count" that has a range of 0 to 32 and a default value of 3:

```
/* Add parameters (name, nickname, description, min, max, default) */
mpf_add_param_int("object_count", "ObjectCount",
"Number of objects", 0, 32, 3);
```

You can also use *mpf_add_param_float(name, nickname, description, min, max, default)* and *mpf_add_param_string(name, nickname, description, default)* to add float and string parameters.

In *component_init()* or *component_process()* you can get the value of the parameter.

```
int object_count = mpf_get_param_int("object_count");
```

You can also use *mpf_get_param_float("name")* and mpf_get_param_string("name") to get float and string parameters.

Specify parameters on the pipeline launch command line as name=value pairs.

For example:

```
gst-launch filesrc location=/Media/GrandCanyonTrail.wmv ! \
 decodebin ! ffmpegcolorspace ! toiplimage ! mycomp object_count=10 ! \
 fromiplimage ! ffmpegcolorspace ! xvimagesink
```

6. Adding Metadata I/O and Using Appscio Metadata Components

Note: Please read the RDF Primer (http://www.w3.org/TR/rdf-primer) to understand what Resource Description Framework (RDF) is and how it is used to represent metadata.

Edit *<my-component-name>.c* using your favorite source code text editor.

Add the include file for the mpf-rdf data-type.

```
/* Include the headers for any data-types needed. */
#include <mpf/mpf-rdf.h>
```

In the *component_class_init()* function, you can add an RDF output like this:

```
/* Add an RDF output */
mpf_rdf_add_output("output_rdf");
```

In the *component_process()* function, you can create an RDF graph similar to this:

```
GrdfGraph *graph;
GrdfNode *rdfface;
graph = mpf_rdf_new();
rdfface = grdf_node_anon_new(graph, "face");
grdf_stmt_new_nuu(graph, rdfface, "dcterms:type","urn:rdf:appscio.com/ver_1.0/roi");
grdf_stmt_new_nui(graph, rdfface, "urn:rdf:appscio.com/partners/pittpatt/ver_2.4/roi/x", x);
grdf_stmt_new_nui(graph, rdfface, "urn:rdf:appscio.com/partners/pittpatt/ver_2.4/roi/y", y);
grdf_stmt_new_nui(graph, rdfface, "urn:rdf:appscio.com/partners/pittpatt/ver_2.4/roi/width", w);
grdf_stmt_new_nui(graph, rdfface, "urn:rdf:appscio.com/partners/pittpatt/ver_2.4/roi/height", h);
```

Note: Please read the GRDF and mpf-rdf documentation for detailed instructions on creating GrdfGraphs.

Still in the *component_process()* function, you can push your RDF graph out to the next component in the pipeline, as follows:

```
mpf_rdf_push("output_rdf", graph);
```

You can use *mpf-rdf-dump* to see the RDF graph printed on the standard output, as follows:

```
gst-launch filesrc location=/Media/GrandCanyonTrail.wmv ! \
decodebin ! ffmpegcolorspace ! toiplimage ! \
mycomp object_count=10 ! mpf-rdf-dump
```

7. Building A New Datatype Library

A data-type library's purpose is to provide an API layer over the MPF core that focuses on a particular type of data, such as raw audio, or RDF. The complexity of such a library depends on how many variations there are to the data type, and whether the data is stored in a compound structure or a contiguous block of memory.

The functions provided to a component using the data-type library fall into the following categories:

- Creation of inputs and outputs
- Managing the subset of data formats allowed
- Pulling and pushing data
- Managing reference counts

Internally a variety of helper functions must be present in order to assist in the mapping between the data pointers and the GstBuffer structure that is actually passed along the pipeline.

Appendix A: Glossary

In this section we provide definitions for common terms you may uncover while working with Appscio MPF:

Term	Definition
Component	Appscio MPF basic unit of metadata-handling function (wraps Element)
Element	gstreamer's basic unit of media-handling function
IplImage	An OpenCV image definition
Metadata	Content description - or - data about data
MPF	Media Processing Framework
Ontology	A formal representation of a set of concepts within a domain and the relationships between those concepts. It is used to reason about the properties of that domain, and may be used to define the domain.
OpenCV	The Open Source Computer Vision Library
Pad	Interconnection points for passing data between components in a pipeline
Pipeline	A sequence of Algorithms (wrapped with Appscio $^{(tm)}$ MPF code)
ROI	Region Of Interest
RDF	Resource Description Framework
SDK	Software Development Kit

Appendix B: Recommended Background Material

The following background material is recommended to gain greater insight into how Appscio is approaching the management of AV data:

- gstreamer Application Development Manual
  http://gstreamer.freedesktop.org/data/doc/gstreamer/head/manual/html/index.html
- gstreamer Plugin Writer's Guide
  http://gstreamer.freedesktop.org/data/doc/gstreamer/head/pwg/html/index.html
- gstreamer Core Reference
  http://gstreamer.freedesktop.org/data/doc/gstreamer/head/gstreamer/html/
- gstreamer Libraries Reference
  http://gstreamer.freedesktop.org/data/doc/gstreamer/head/gstreamer-libs/html/
- gstreamer Core Design Documentation
  http://cgit.freedesktop.org/gstreamer/gstreamer/tree/docs/design/
- For useful information on debugging gstreamer:
  http://gstreamer.freedesktop.org/data/doc/gstreamer/head/gstreamer/html/gstreamer-GstInfo.html
- gstreamer includes a set of utility applications which are useful during development of components:
  http://gstreamer.freedesktop.org/data/doc/gstreamer/head/manual/html/section-checklist-applications.html
- RDF Primer
  http://www.w3.org/TR/rdf-primer

In addition to the above, visit the Appscio Community Web site (www.appscio.org) to learn more about Appscio MPF and to see what others in the Community are doing with Appscio MPF.

Alphabetical Index

# # # # # # # # # #