

Aperio Algorithm Framework (AAF) Programmer's Reference



©Copyright 2007 Aperio Technologies, Inc.
 Part Number/Revision: MAN-0067, Revision B
 Date: December 3, 2007

This document applies to software versions Release 8 and later.

All rights reserved. This document may not be copied in whole or in part or reproduced in any other media without the express written permission of Aperio Technologies, Inc. Please note that under copyright law, copying includes translation into another language.

User Resources

For the latest information on Aperio Technologies products and services, please visit the Aperio Technologies website at: <http://www.aperio.com>.

Disclaimers

This manual is not a substitute for the detailed operator training provided by Aperio Technologies, Inc., or for other advanced instruction. Aperio Technologies Field Representatives should be contacted immediately for assistance in the event of any instrument malfunction. Installation of hardware should only be performed by a certified Aperio Technologies Service Engineer.

ImageServer is intended for use with the svf file format (the native format for digital slides created by scanning glass slides with the ScanScope scanner). Educators will use Aperio software to view and modify digital slides in Composite WebSlide (CWS) format.

Aperio products are FDA cleared for specific clinical applications, and are intended for research use for other applications.

Trademarks and Patents

ScanScope is a registered trademark and ImageServer, TMA Lab, ImageScope, and Spectrum are trademarks of Aperio Technologies, Inc. All other trade names and trademarks are the property of their respective holders.

Aperio products are protected by U.S. Patents: 6,711,283; 6,917,696; 7,035,478; and 7,116,440; and licensed under one or more of the following U.S. Patents: 6,101,265; 6,272,235; 6,522,774; 6,775,402; 6,396,941; 6,674,881; 6,226,392; 6,404,906; 6,674,884; and 6,466,690.

Contact Information

Headquarters: Aperio Technologies, Inc.
 1430 Vantage Court
 Vista, CA 92081
 United States

European Office: Aperio
 3 The Sanctuary
 Eden Office Park
 Ham Green
 Bristol BS20 0DD, UK

United States of America

Tel: 866-478-4111 (toll free)

Fax: 760-539-1116

Customer Service Tel: 866-478-4111 (toll free)

Technical Support Tel: 866-478-4111 (toll free)

Email: support@aperio.com

Europe

Tel: +44 (0) 1275 375123

Fax: +44(0) 1275 373501

Customer Service Tel: +44 (0) 1275 375123

Technical Support Tel: +44 (0) 1275 375123

Email: europesupport@aperio.com

Introduction

This document is intended to provide the level of detail necessary for an algorithm developer to understand and develop image processing algorithms using the Aperio Algorithm Framework (AAF) Software Development Kit. Part 1 of this document contains information on the AAF architecture and the general procedures for working with the AAF in developing an algorithm. Part 2 contains a practical, detailed set of instructions for starting a new Visual Studio project. A sample algorithm project is also included with the SDK, which may be built, registered, and run using the Aperio ImageScope application.

Part 1: The Aperio Algorithm Framework

The Aperio Algorithm Framework (AAF) facilitates running algorithms on scanned images. It acts as a single-point contact to all users/applications that need to run algorithms and for all algorithms to return their output.

The advantages of using the AAF include:

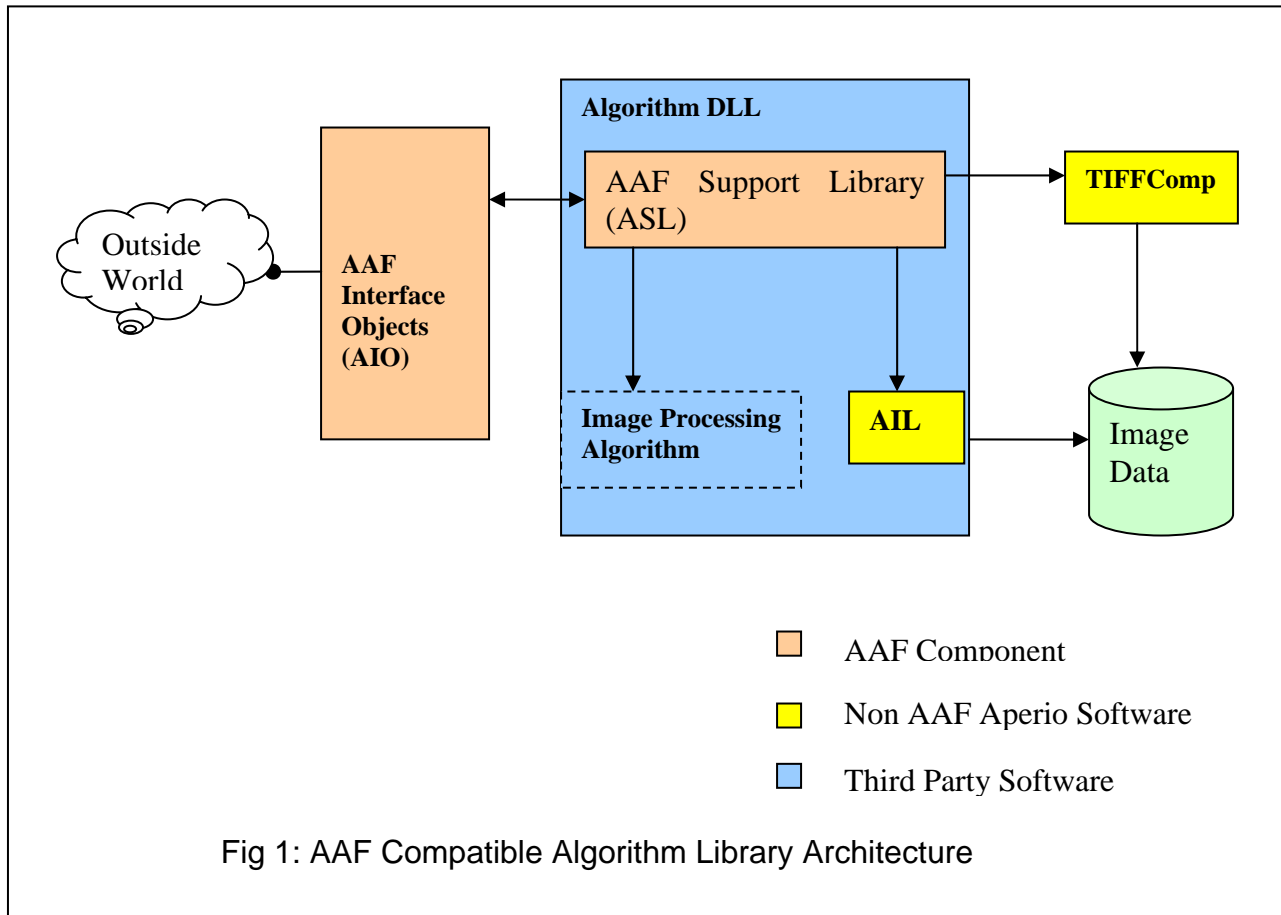
- Abstraction from file access logic, independent of file format and location.
- Abstraction from view and overlap logic to handle large size images.
- Enable storing of results in a standard format to be viewed with Aperio ImageScope.
- Enable processing of images at any specified zoom level.
- Enable “Region Of Analysis” processing.

For the Algorithm Developer to use the above features, the algorithm logic has to be integrated with the AAF. This document describes the steps required to achieve this integration.

Contents

Introduction.....	1
Part 1: The Aperio Algorithm Framework.....	3
Contents	3
Algorithm DLL Architecture	4
Description of the cAAFAAlgorithm Methods, Properties and Data Structures	5
Data Structures.....	5
Methods and Properties.....	6
Steps Required for Integrating Algorithm Logic into the AAF.....	11
Running Your Algorithm Using ImageScope.....	14
Debugging Your Algorithm DLL	14
Steps to Debug Your Algorithm Code.....	15
Part 2: Instructions for Starting a New Project.....	17
Step 1: Create New Project Folder.....	18
Step 2: Define DLL Name and Version Info	18
Step 3: Define Algorithm Information.....	19
Step 4: Define Algorithm Input Parameters and Properties.....	19
Step 5: Define Algorithm Output Parameters	22
Step 6: Image Processing Logic.....	22
Step 7: Image Zoom Correction.....	23
Step 8: Define Summary Results	24
Step 9: Report Summary Results	24
Step 10. Reporting and Working with Image Objects	26

Algorithm DLL Architecture



The Algorithm DLL makes use of the AAF Support Library (asl.lib) to interact with AIO. ASL uses AIL for reading images and TIFFComp for writing out images. It also calls the image processing logic. The `cAAFAlgorithm` class in asl.lib provides the logic to interact with various components. Other classes like `cAAFResult` and `cAAFParameter` provide the data-structure support needed. An algorithm developer can make use of all the logic provided by the `cAAFAlgorithm` class and customize the input and output to meet the needs of the algorithm. `cAAFAlgorithm` allows any part of the logic to be completely over-ridden or partially customized by using C++ virtual functions. Algorithm developers would normally derive a class from `cAAFAlgorithm` and customize any part they want to suit their needs.

Description of the cAAFAlgorithm Methods, Properties and Data Structures

Data Structures

```
typedef struct tagView {
    UCHAR *pucView;
    float fZoom;
    CPoint    cpP1;
    CPoint    cpP2;
    int       nChan;
} VIEW;
```

This structure contains the current view details and a pointer to the view data. This is passed to processViewport function with the relevant view. No release of memory is required.

```
typedef struct tagViewPos {
    int    nRoaCount;
    int    nCurRoa;
    int    nCurRoaViewCount;
    int    nCurView;
} VIEWPOS;
```

This structure contains the position of the current view in the area that needs to be processed. It is passed to processViewport. This structure must be used by the algorithm developer to determine whether a result should be reported.

```
typedef struct tagBox {
    int    xmin, xmax, ymin, ymax;
} BOX;
```

Structure used to define a rectangular area. See description of InCurView function for the use of this structure.

Methods and Properties

Method/Property	Description
DWORD registerMe(void)	Used to register the Algorithm DLL. Call this in DllRegisterServer() to perform registration.
DWORD unregisterMe(void)	Used to un-register the Algorithm DLL. Call this in DllUnRegisterServer().
LPCAAFInfo getTitleInfo()	Used by AIO to obtain information about the DLL. cAAFAAlgorithm implementation calls describeAlgorithm() and returns the member pointer to cAAFInfo class. Over-ride to change behavior. If using the default implementation, the derived class must update the variables that hold algorithm information in the constructor of derived class.
int describeAlgorithm()	cAAFAAlgorithm implementation initializes internal cAAFInfo structure and the parameters by calling defParameter().
defParameter(int nIndex)	Called by describeAlgorithm to initialize the parameters to their default values. cAAFAAlgorithm implementation handles only the AAF parameter. Over-ride this function to handle all parameters that the algorithm needs and any of the AAF parameters that you need to modify. Call cAAFAAlgorithm implementation for any AAF parameters.
int defParameterCount()	Used to obtain the parameter count. cAAFAAlgorithm implementation returns m_nParams.
LPCAAFInfo defTitle()	Called by describeAlgorithm to initialize cAAFInfo. cAAFAAlgorithm implementation calls necessary functions to initialize a cAAFInfo object.
LPCTSTR getDllPath()	Used by registration function. cAAFAAlgorithm implementation returns the current DLL path.
LPCTSTR getOnlyTitle()	cAAFAAlgorithm implementation returns the title of the algorithm stored in m_csTitle.

Method/Property	Description
LPCTSTR getDllName()	cAAFAAlgorithm implementation returns the DLL name stored in m_csDllName.
int getParameterCount()	Used by AIO to obtain Parameter count.
LPCAAFPParameter getParameter(int nIndex)	cAAFAAlgorithm implementation returns pointer to the specified cAAFPParameter object. Index is zero based. Used by AIO to get parameters.
setParameter(LPCAAFPParameter p, int nIndex)	cAAFAAlgorithm implementation sets the parameter specified by Index to the value specified in p. Index must be zero based. Used by AIO to set parameters to user specified values.
LPCAAFRoa getRoa()	cAAFAAlgorithm implementation returns a pointer to the ROA object. If an ROA has not been specified, the object returned has no vertices. Used by AIO.
DWORD setRoa(LPCAAFRoa pNewRoa)	cAAFAAlgorithm implementation sets the local copy of the ROA to the one passed. Used by AIO.
int setImage(LPCTSTR szImageName)	Used by AIO to set the path to the image to be processed.
int logging(BOOL enable, LPCTSTR szPath="")	Not used currently.
int setSaveOutput(BOOL bSOutput)	Used to specify if a result image must be saved.
int getSaveOutput(BOOL *pbSOutput)	cAAFAAlgorithm implementation returns the result image state.
int setUseThread(BOOL bUThread)	Used to specify if a separate thread must be started to process the image. Used by AIO.
int getUseThread(BOOL *pbUThread)	States whether algorithm will be processed in a separate thread. Used by AIO.
DWORD start()	cAAFAAlgorithm implementation starts algorithm processing. Behavior depends on whether thread must be used.
DWORD pause()	Not Implemented.

Method/Property	Description
DWORD abort()	cAAFAAlgorithm implementation sets up the thread to stop processing after the current view is processed. Used only if a separate thread is used for processing.
DWORD resume()	Not Implemented.
int getState()	cAAFAAlgorithm implementation returns the state the algorithm DLL is in. Possible states are enumerated in AAF_STATE.
int initProcessThread()	Called by start() if processing needs to be done in a separate thread. cAAFAAlgorithm implementation sets up OLE initialization and starts the thread.
DWORD processThread()	Called by either in the main thread by start or in a worker thread by a static function doProcessThread(). This function initializes the state and calls processImage().
DWORD initImageTraversal(LPAIC pImage)	This function is called to setup motion of image traversal. cAAFAAlgorithm implementation defines how frames within the image are requested from AIL
DWORD traverseImage(LPAIC pImage)	traverseImage is main loop used to process full image broken into views which are returned by AIL cAAFAAlgorithm implementation moves through the image sequentially in row first order.
DWORD processViewport(VIEW* psvCur, VIEWPOS svp)	processViewport is called by traverseImage to do the actual processing of one view within the image. Two structures are passed to this function. The VIEW structure contains details about the current view and the VIEWPOS structure contains details of the position of the current view in the entire processing region.. If an ROA is specified, the required mask will have been applied. Currently the mask is white. Later this will be customizable. Call your own implementation of the algorithm here and depending upon the needs of your algorithm, use the VIEWPOS structure to determine when to return a result.

Method/Property	Description
DWORD setProgressCallback(AAFProgressCallback p, void * pObj)	Used by AIO to register its progress callback routine.
DWORD setResultCallback(AAFResultCallback p, void * pObj)	Used by AIO to register its result callback routine.
LPAIC getImage()	cAAFAAlgorithm implementation returns pointer to CAIC (represents AIL) object.
BOOL isAbortRequested()	cAAFAAlgorithm implementation returns abort request state. Call this function to verify if the algorithm must abort. cAAFAAlgorithm implementation of traverseImage always checks after every view if algorithm must abort.
BOOL InitStdParams()	Called to transfer AAF parameter values from cAAFParameter objects to the actual member variables.
BOOL InitAlgoParams()	Called to transfer algorithm values from cAAFParameter to actual member variables. Developers are expected to over-ride this function to do their own parameter transfer. This is guaranteed to be called before the first view processing begins.
BOOL InitResTitles()	Called to fill algorithm info structure with the summary result titles that the algorithm reports. Developers must over-ride this function to support running algorithms in TMA Lab.
void sendProgressEvent(int status, int perCent)	Call this to report progress in algorithm processing. cAAFAAlgorithm implementation of traverseImage calls this after every view. Passes the AAFProgress object to AIO. Variable status is currently not used.

Method/Property	Description
void sendResultEvent(long ImgId, int score, int status, void *metric, eMetricType mtType)	Call this to report results. Depending on MetricType, cAAFAAlgorithm implementation of this function calls either setRegionMetric or setSummaryMetric with the void pointer passed to it. Call this function with the proper MetricType setting to send results back to the client program. Variables; score, status and ImgId are not used currently.
BOOL setRegionMetric(LPCAAFAResult pResult, void *metric)	Called by sendResultEvent when the MetricType is eREGION. For every result metric you need to report, call addmetric() on the cAAFAResult object pointer passed, with strings indicating metric name and value.. Developers should implement this function. Results reported in this function appear in the “Regions” section of the annotations window of Imagescope.
BOOL setSummaryMetric(LPCAAFAResult pResult, void *metric)	Called by sendResultEvent when the MetricType is eSUMMARY. For every result metric you need to report, call addmetric() on the cAAFAResult object pointer passed with strings indicating metric name and value. cAAFAAlgorithm implementation of this function just returns. Over-ride to report summary results for the algorithm run. Results reported in this function appear in the “Annotation Attributes” section of the annotations window of Imagescope.
BOOL GetBoundingBox(VIEW *sBox)	Used to determine the bounding box if ROA is specified. If specified, it fills in the VIEW structure passed. Used by cAAFAAlgorithm::traverseImage()
BOOL GetMask(unsigned char *pucMask, unsigned char *pucState, int nW, int nH, int nXOff, int nYOff, int nOverlap)	Returns the mask to be applied to the current view. Memory must be allocated for the mask before the function is called.
BOOL ApplyMask(unsigned char *pucMask, unsigned char *pucImg, int nW, int nH);	cAAFAAlgorithm implementation applies the mask to the view specified.

Method/Property	Description
BOOL InCurView(BOX *psBox)	Applies overlap logic to determine if an object specified by the BOX structure must be counted in the current view. Algorithm developers can call this function for every valid object to determine if the object must be counted in the current view. Use this function if using overlap logic to avoid over counting. The BOX structure passed must be a tight fitting box that completely contains the object in question.

Steps Required for Integrating Algorithm Logic into the AAF

The following steps are required for integrating your algorithm logic into the AAF. Part 2 of this document further explains how to modify the included sample project to suite your particular requirements.

1. Create a compatible project.

Algorithm developers can use the Microsoft Visual Studio App-wizard to create a project compatible with AAF. Developers need to choose a standard MFC AppWizard (DLL). After choosing the MFC AppWizard DLL, in the next screen of the wizard choose Automation. At this stage you can hit finish.

2. Add self-registering functions.

There are two functions needed to make the Algorithm DLL a self-registering DLL: DllRegisterServer() and DllUnRegisterServer(). These functions are called by registering applications (like regsvr32.exe) for registering the DLL. This function can either do its own registering or add a call to registerMe() method of cAAFAAlgorithm class to delegate the registration process. For un-registering, call unregisterMe() in cAAFAAlgorithm class.

3. Provide a class-factory.

AIO uses the class factory function to create an instance of the algorithm class. AIO calls a function called CreateAAFInstance() to obtain the instance of Algorithm class. This must be a global exported function from the DLL that accepts no parameters and returns a pointer to your own algorithm class derived from cAAFAAlgorithm cast to a void pointer. The function declaration is:

```
extern "C"
{
    void __declspec(dllexport) *CreateAAFInstance();
}
```

and the definition is

```
void *CreateAAFInstance()
{
    return static_cast< void* > (new CYourAlgorithm);
}
```

Where CYourAlgorithm is the class derived from cAAFAAlgorithm.

4. Override methods and define structures specific to the algorithm. Typically, algorithm developers will have to override the following functions:

CYourAlgorithm constructor():

In the derived class constructor in addition to any algorithm initialization, the class needs to initialize some AAF specific parent class member variables that provide info about the algorithm. Algorithm class must also specify the number of algorithm specific parameters. This can be done by adding to m_nParams, the number of parameters the algorithm expects. The variables that must be initialized are:

```
m_csDllName = "Dc_Sample.dll";
m_csTitle = "Dc Sample Algorithm";
m_csDescription = "Algorithm Description goes here";
m_csVersion = "1.0";
m_nParams += nAlgoParameters;
```

defParameter(int nWhichOne)

This function is called with a zero based index of the parameter for which the description is required. There are AAF related parameters and Algorithm related parameters. AAF currently uses 4 parameters, view width, view height, zoom and overlap. Though these are AAF specific parameters, developers might want to have control over what these values must be. For instance not all algorithms can be run at 0.5 zoom. Developers can handle these parameters in addition to algorithm parameters in this function. For each parameter, developers must allocate a new cAAFPParameter object, fill the parameter info and return a pointer to this parameter object. The cAAFPParameter object allows specification of parameter type, default value, minimum, maximum and the step size. It is not logical to set all these for all parameter types. For e.g., list type supports a list of strings but the value itself is the index of the string in the list on the other hand integer type has a default, min, max and step size. Note unbounded parameters are also supported. If you have an unbounded type, just do not set the upper or lower limit.

processViewport(VIEW* psvView, VIEWPOS svp)

processViewport is called by traverseImage to do the actual processing of one view within the image. Two structures are passed to this function. The VIEW structure contains details about the current view and the VIEWPOS structure contains details of the position of the current view in the entire processing region.. If an ROA is

specified, the required mask will have been applied by the parent implementation. Currently the mask is white. Later this will be customizable. Call your own implementation of the algorithm here and depending upon the needs of your algorithm, use the VIEWPOS structure to determine when to return a result.. Call `sendResultEvent` to report results. Depending on the metric type, either `setRegionMetric` or `setSummaryMetric` is called with the same void pointer that you pass to `send result event`. Also `cAAFAAlgorithm` provides a function that can be used to implement overlap logic that avoids over-counting of objects found. This is implemented in the function `InCurView()`. This accepts a bounding box for each object and returns a Boolean that specifies if the object must be counted in the current view or not.

```
setRegionMetric(LPCAAFResult pResult, void *metric)
```

This function is called by `sendResultEvent` when the `MetricType` is `eREGION`. For every result metric you need to report, call `addmetric()` on the `cAAFResult` object pointer passed, with strings indicating metric name and value. Developers should implement this function. Results reported in this function appear in the “Regions” section of the annotations window of Imagescope.

```
setSummaryMetric(LPCAAFResult pResult, void *metric)
```

This function is called by `sendResultEvent` when the `MetricType` is `eSUMMARY`. For every result metric you need to report, call `addmetric()` on the `cAAFResult` object pointer passed with strings indicating metric name and value. `cAAFAAlgorithm` implementation of this function does not report any. Developers must Over-ride to report summary results for the algorithm run. Results reported in this function appear in the “Annotation Attributes” section of the annotations window of Imagescope.

```
InitAlgoParams()
```

This function called to transfer algorithm values from `cAAFParameter` to actual member variables. Developers are expected to over-ride this function to do their own parameter transfer. This is called by `cAAFAAlgorithm` implementation of `traverseImage` before the first view processing begins.

```
InitResTitles()
```

Called to fill algorithm info structure with the summary result titles that the algorithm reports. Developers must over-ride this function to support running algorithms in TMA Lab.

5. Add Image Processing logic.

For each view, the function `processViewport` is called by `traverseImage` with the view data, some related information about the view dimension and the position of the view in the entire set of regions to be processed. Developers can add all image-processing logic in `processViewport` or by calling other functions. After processing the image, developers can call `sendResultEvent` function to report result when required.

Running Your Algorithm Using ImageScope

For Imagescope to recognize and load your algorithm DLL, it needs to be registered. To do this:

1. Open command window:
 - a. Click Start->Run
 - b. Type “command” and click OK
2. Navigate to the directory you have your algorithm DLL. For e.g., if you just built the release version of sample algorithm, you must type:

```
cd <path_to_AAF_SDK>\Algorithms\Color Filter Algorithm\Release
```

and press the Return key.

3. Type “regsvr32 youralgorithmdll.dll.” For example, if you just built the release version of the sample algorithm, you would type

```
regsvr32 ColorFilter.dll
```

and press the return key.

4. If there are any spaces in your <path to AAF_SDK>, the command processor may not allow you to navigate to that folder. In this case, create a batch file in the folder containing your algorithm DLL (e.g., RegisterDll.bat) with the following line (you need to include the quotes):

```
"C:\WINNT\system32\REGSVR32.EXE ColorFilter.dll"
```

If you are using Windows XP, the command would be:

```
"C:\Windows\system32\REGSVR32.EXE ColorFilter.dll"
```

5. Double-click on this batch file to run it.
6. If you are familiar with regedit (registry editor), you will find your algorithm registered under the key name:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Scanscope\AAF\Algorithms
```

Debugging Your Algorithm DLL

Since a DLL is not a self-loading binary file, you need another executable to load the DLL and call it. The executable to be used when debugging the algorithm DLL can be specified in the

debug options of Microsoft Visual Studio, project settings dialog. If you point this to Aperio ImageScope, then Visual studio brings up ImageScopewhen you start debugging. You can then put appropriate breakpoints and start processing an image. When the logic hits your breakpoint, Microsoft Visual Studio breaks into your code.

An important point to remember while using the SDK is that the AAF and Algorithm must be of the same type i.e., Debug/Release

Steps to Debug Your Algorithm Code

1. Open your algorithm project in Microsoft Visual Studio. Change the “Active Configuration” in Microsoft Visual Studio to Debug and build the project to get the debug version of your algorithm DLL.
2. Set ImageScope as the executable for debug session. To do this:
 - a) Go to the project settings dialog in Microsoft Visual Studio. Click the settings option in the Project Menu.
 - b) In the tab control, go to the Debug tab.
 - c) Click the **Browse** button next to the **Executable for debug session** edit box.
 - d) Navigate to the folder you installed ImageScope in and select **Imagescope.exe**.
 - e) Click **OK**.
3. Register the debug version of the algorithm DLL. See “Running Your Algorithm Using ImageScope” above on page 14 for steps to do this.
4. Register the debug version of AAF DLL. When you install the SDK, two versions of the AAF DLLs are copied to the AAFObjects sub_directory. This DLL enables your algorithm to interface with Imagescope and other applications. The type (Debug/Release) of this DLL has to coincide with that of the algorithm DLL. To register the debug version (in the “Debug” sub directory of the AAFObjects directory) of the AAF DLL, follow the steps listed in the “Running Your Algorithm Using ImageScope” above on page 14, except type **regsvr AAFObjects.dll** in the last step.
5. In the code of your algorithm DLL, place your breakpoints.
6. Choose the Build->Start Debug->Go menu option in Microsoft Visual Studio. ImageScope should now start running.
7. Open an image in ImageScope and load your algorithm. Set the parameters and start running the algorithm. Code execution must stop where you placed the breakpoint.

Part 2: Instructions for Starting a New Project

The SDK contains a sample Visual Studio project in a folder named “Color Filter Algorithm.” Before beginning a new project of your own, you are advised to build the Release version of ColorFilter.dll, register it (see Part 1), and use the ImageScope application to run the algorithm on the provided sample image. ImageScope is the primary application for running an algorithm—it allows the user to open an image for processing, select an algorithm, adjust the input parameters, select ROAs, run the algorithm, and view results. Once you are familiar with how you interact with the algorithm’s input and output, you will be in a much better position to understand how this functionality is implemented in the Visual Studio project.

To begin a new project, you will copy the sample project folder and give it a new name. This new folder should reside at the same level in your file system as the sample project folder, since the build settings include references to libraries located there. None of the filenames in the project need to be changed—their names reflect their functionality in a generic way. For example, all of the image processing logic is implemented in functions contained in the files Image_Processing.cpp and Image_Processing.h. Logic associated with initializing input and saving output is contained in the file CAlgorithm.cpp. Other than modifying the project settings to reflect a new DLL name and updating the version information, you will not need to modify any other files.

The separation of input/output logic from image-processing logic enables you to easily build other applications that implement your algorithm. For example, you could build a simple console application which initializes the algorithm input parameters and prints the results to a text file. This application could use the same Image_Processing.cpp and Image_Processing.h files, without any changes to their source code. These two files are totally defined by the algorithm developer and do not depend upon any SDK-supplied libraries or code files.

The SDK uses the Aperio Annotation format for recording algorithm input/output in an xml-file. Algorithm results are reported at two levels: Region-level reporting for each ROA and Annotation-level (Summary) reporting for the entire algorithm run (all ROAs). Results at each level consist of name-value pairs. An example of a name is “Number of Nuclei” and its value might be “998.” For morphological image processing, where individual objects are detected and quantified, each object can be reported as a new region at the Region level. However, if the number of objects becomes quite large, the Annotation format is not an efficient format to use for reported individual objects. In this case, the developer is advised to implement a new reporting mechanism, to augment the Annotation results. The new reporting format would not be supported by other AAF-enabled applications, so the developer would need additional tools to access this information.

The following outline for starting a new project is divided into 10 steps. The number of each step represents the order in which the steps would normally be done. However, you will likely revisit some of the steps as you decide that additional input or output parameters are needed, as well as when modifications of the image processing logic become necessary. Changes you must make to the file CAlgorithm.cpp are enclosed by the two character strings *UserDefined and *End

UserDefined. A search on these text strings will help you verify that all code sections have been modified appropriately.

Step 1: Create New Project Folder

- **Create New Folder:** Open Windows Explorer and navigate to the folder that contains the SDK sample project folder **Color Filter Algorithm**. Copy-Paste the folder **Color Filter Algorithm** to a new folder and rename it to the name you have chosen for the new algorithm; for example **YourAlgorithmName**. You can rename the folder at any time. It is a good idea to make the folder name correspond to the name used in the version info and DLL name (see Step 2).
- **Open the Project:** Navigate into the newly created folder and double-click on the file **AlgorithmDLL.dsp**. This should start Visual Studio so that you will be ready for the next step.
- **Important:** Don't rename any files in the project folder, including the dsp and dsw files. This is not necessary.

Step 2: Define DLL Name and Version Info

You should already have the project AlgorithmDLL.dsp open in Visual Studio from Step 1. If you cannot see the workspace files, Click **View/Workspace**.

To define the DLL filename:

- Click Project/Settings/Link/General and change the "Output file name" value to the name you have chosen for your algorithm. You must do this for both the Release and Debug Builds: for example, Release/YourAlgorithmName.dll and Debug/YourAlgorithmName.dll.
- Double-click on the file AlgorithmDLL.def and on the line with the tag "LIBRARY," replace ColorFilter with YourAlgorithmName.

To set the version info:

- Double-click on AlgorithmDLL.rc to open the resource editor. In the Version folder, double-click on **VS_VERSION_INFO** and edit the version labels. In particular, the label "OriginalFilename" should be set to YourAlgorithmName.dll.
- Other labels can be set as desired. These are visible as a properties of the DLL in Windows Explorer, by Right-click/Properties/Version on the DLL-filename.

- Edit defineAlgoParameter():** Double-click on CAlgorithm.cpp to open this file for editing. Scroll down until you are positioned to edit the function defineAlgoParameter(). This function is called by AAF-enabled applications to define the algorithm input parameter properties. The switch statement shown in the sample program has four cases, corresponding to the four input parameters. The case index matches the order in which the parameters are declared in the ALGORITHM_INPUT structure above. The three allowable AAF-parameter types are Integer, Double, and List. A portion of this source code is shown here:

```

                                                                    /*UserDefined
case 0:
    pParam->setName("Color Filter");
    pParam->setDescription("Select the Color of Pixels");
    pParam->setRequired(m_cbParamReq);
    pParam->setType(cAAFPParameter::eList);
    pParam->addString("Red");
    pParam->addString("Green");
    pParam->addString("Blue");
    pParam->setValue((int)0);
    break;

case 3:
    pParam->setName("Color Saturation Threshold");
    pParam->setDescription("Pixels with ...");
    pParam->setRequired(m_cbParamReq);
    pParam->setType(cAAFPParameter::eDouble);
    pParam->setValue((double)0.0);
    pParam->setMin((double)0.0);
    pParam->setMax((double)1.0);
    pParam->setStep((double)0.01);
    break;
                                                                    /*End UserDefined

```

Case 0 shows the properties of the input parameter associated with the variable `icolor`. The permissible values for `icolor` are 0, 1, or 2, corresponding to the strings “Red,” “Green,” or “Blue.” The default value will be initialized to 0 (“Red”) in this example. This type of property control is implemented as a List-Box in AAF-enabled programs.

Case 3 shows the properties of the input parameter associated with the variable `dsaturation_threshold`. The permissible values for `dsaturation_threshold` may range from 0.0 to 1.0. This type of property control will be implemented as a Slider, with step increments of 0.01. The default value for this parameter will be initialized to 0.0. Either or both of the calls to `setMin/setMax` may be omitted—in which case, the property control will be implemented as an Up/Down (Spin) control, instead of a Slider.

- **Edit defineAAFPParameter():** Continue to scroll down in the file CAlgorithm.cpp until you are positioned to edit the function defineAAFPParameter(). These are the input parameters that control how AAF does the block processing (views) of the image. For image processing algorithms that find objects, you will need to set the default value for the overlap size to be as big as the diameter of the largest object you expect to find. In the sample application, the default value is zero (setValue((int)0)).

```

case 2:
    pParam->setName("Overlap Size");
    pParam->setDescription("Size of the overlap region for each view");
    pParam->setRequired(m_cbParamReq);
    pParam->setType(cAAFPParameter::eInteger);

                                                                    /*UserDefined

    pParam->setValue((int)0);
    pParam->setMin((int)0);
    pParam->setMax((int)300);
    pParam->setStep((int)10);

                                                                    /*End UserDefined

```

- **Edit InitAlgoParams():** Continue to scroll down in the file CAlgorithm.cpp until you are positioned to edit the function InitAlgoParams(). This function is called by AAF-enabled applications to initialize the input parameters when running the algorithm.

```

                                                                    /*UserDefined

    pParam = getAlgoParameter(0);
    pParam->getValue(AlgInput.icolor);

    pParam = getAlgoParameter(1);
    pParam->getValue(AlgInput.intensity_max);

    pParam = getAlgoParameter(2);
    pParam->getValue(AlgInput.intensity_min);

    pParam = getAlgoParameter(3);
    pParam->getValue(AlgInput.dsaturation_threshold);

                                                                    /*End UserDefined

```

Each input parameter defined in the ALGORITHM_INPUT data structure, AlgInput, is initialized in order. This order is also the AAF-property order as defined above in function defineAlgoParameter().

Step 5: Define Algorithm Output Parameters

This step defines the output parameters for the image processing portion of the algorithm. In Step 8, these parameters are further summarized for result reporting.

- **Edit ALGORITHM_OUTPUT:** Double-click on Image_Processing.h to open this file for editing. The data structure ALGORITHM_OUTPUT contains the output parameters from the image processing portion of the algorithm. The following code segment should be changed to reflect your algorithm output parameters.

```
typedef struct algorithm_output
{
    // *UserDefined
    long nColorPixels;
    long nImgPixels;
    long iIntensity_Sum;
    long iSaturation_Sum;
    // *End UserDefined
}ALGORITHM_OUTPUT;
```

Step 6: Image Processing Logic

The image processing logic is implemented in a single function process_one_view(). A declaration of this function is given in the header file Image_Processing.h:

```
void process_one_view(unsigned char* rgb_img,
                    long w,
                    long h,
                    ALGORITHM_INPUT* pAlgInput,
                    ALGORITHM_OUTPUT* pAlgOutput
                    );
```

This function prototype should not be changed. The purpose of this function is to process a rectangular block (view) of image data (width w, height h), with input/output as defined above. The AAF handles caching of the entire image into blocks (views) and making repeated calls to this function in order to process very large images.

- **Edit process_one_view():** Double-click on Image_Processing.cpp to open the file for editing. The entire contents of the function process_one_view() are replaced by the developer. You may add other functions to this file, or make calls to function libraries that you include in the project settings.

The sample function illustrates how you may call a library routine TiffOut() to output debugging images to disk. This is provided as an additional development tool. The input

image data pointed to by `*rgb_img` is also modified to reflect the behavior of the algorithm. AAF-enabled applications, such as ImageScope, will display this “mark-up” image when running the algorithm in "Current Screen" mode. This is useful for obtaining visual feedback as input parameter values are changed.

Step 7: Image Zoom Correction

Image zoom is the resolution for running the algorithm. For example, `zoom=1.0` will process each native image pixel. `Zoom=0.5` will process a down-sampled image, with half the number of pixels in each direction. Running an algorithm at reduced zoom will take less time, since there are fewer pixels to process. However, you will need to confirm that your algorithm produces substantially the same results when run at reduced zoom levels. The default value for zoom is 1.0, and is set in function `defineAAFParameter()`. The actual value can be changed at run-time using an AAF-enabled application, such as ImageScope.

Since `process_one_view()` only knows the size of an object in pixels, you will need to correct for the zoom factor before reporting summary results. This will result in the reported numbers corresponding to native pixel units, no matter what zoom level the algorithm is run at. For example, when reporting number of pixels at `zoom=0.5`, you would divide the `process_one_view()` result by 0.25 (i.e., $zoom^2$)—this is because there are four native pixels for each processed pixel. For image coordinates, you would divide by 0.5, since linear dimensions have two native pixels for each processed pixel.

- **Edit `processViewport()`:** Double-click on `CAAlgorithm.cpp` to open this file for editing. Scroll down until you are positioned to edit the function `processViewport()`. The following code segment makes the zoom adjustment for the sample algorithm parameters:

```
                                                                    /*UserDefined  
  
AlgOutput.nColorPixels    = (int)((AlgOutput.nColorPixels)  /(zoom*zoom));  
AlgOutput.nImgPixels      = (int)((AlgOutput.nImgPixels)    /(zoom*zoom));  
AlgOutput.iIntensity_Sum  = (int)((AlgOutput.iIntensity_Sum)/(zoom*zoom));  
AlgOutput.iSaturation_Sum = (int)((AlgOutput.iSaturation_Sum)/(zoom*zoom));  
  
                                                                    /*End UserDefined
```

Since each algorithm output parameter in the sample program is proportional to the number of pixels, the proper correction is to divide by $(zoom*zoom)$.

Step 8: Define Summary Results

Image processing results must be accumulated as each block (view) of image data is processed. In many cases, the accumulation of results will also involve summarization, in order to reduce the amount of output to be reported.

- **Edit SUMMARY_OUTPUT:** Double-click on Image_Processing.h to open this file for editing. This is where you define the structure SUMMARY_OUTPUT. Change the variable names accordingly. There are no limitations on the type of data this structure may contain.

```
typedef struct summary_output
{
    Int    nColorPixels;
    int    nImgPixels;
    double dPercentColor;
    double dIntensity_Avg;
    double dSaturation_Avg;
}SUMMARY_OUTPUT;
/*UserDefined
/*End UserDefined
```

Note that the SUMMARY_OUTPUT structure is similar to, but not identical to, the ALGORITHM_OUTPUT structure.

Step 9: Report Summary Results

When running an algorithm in an AAF-enabled application, the user selects a number of ROAs (Regions of Analysis). Summary results are reported for each ROA (Region level) and for all ROAs collectively (Summary/Annotation level). As each view in each ROA is processed, the results must be summarized and accumulated. The developer must supply the programming logic for converting the image processing results into summary results, as well as defining the format for reporting the results to the AAF-enabled application.

- **Edit summarize_output():** Double-click on Image_Processing.cpp to open the file for editing. Scroll down until you are positioned to edit the function summarize_output(). The entire contents of this function must be replaced by the developer. A declaration of this function is shown here:

```
void summarize_output(ALGORITHM_OUTPUT* pAlgorithm_Output,
                     SUMMARY_OUTPUT** ppSummary_Output
                     );
```

This function prototype should not be changed. The purpose of this function is to accumulate the image processing results (*pAlgorithm_Output) into the summary results

(*ppSummary_Output). The developer **must** add the relevant code section to allocate memory to the SUMMARY_OUTPUT structure, as well as initialization of its contents, as in this sample code. De-allocation of memory is handled elsewhere automatically.

```

if (*ppSummary_Output == NULL)
{
    *ppSummary_Output = new SUMMARY_OUTPUT;

    (*ppSummary_Output)->nColorPixels      = 0;
    (*ppSummary_Output)->nImgPixels        = 0;
    (*ppSummary_Output)->dPercentColor     = 0.0;
    (*ppSummary_Output)->dSaturation_Avg   = 0.0;
    (*ppSummary_Output)->dIntensity_Avg    = 0.0;
}

```

- **Edit setRegionMetric():** Double-click on CAlgorithm.cpp to open this file for editing. Scroll down until you are positioned to edit the function setRegionMetric(). A call to addMetric must be made for each summary output parameter that you want to report. You don't need to report all parameters, or, you can combine several values into a single number for reporting. In each call, you provide a name and value in string format. The sample algorithm uses a switch statement, since the name of certain outputs was changed to reflect the type of color being filtered. These results will be reported after processing each ROA. A portion of the sample code is shown here:

```

switch(AlgInput.icolor)
{
    /*UserDefined
    case 0:
    pResult->addMetric("Red Pixels", LongStr( (long)prMet->nColorPixels));
    pResult->addMetric("Total Pixels",LongStr( (long)prMet->nImgPixels));
    pResult->addMetric("Pixel Ratio", FloatStr((double)prMet->dPercentColor));
    pResult->addMetric("Intensity",    FloatStr((double)prMet->dIntensity_Avg));
    pResult->addMetric("Saturation",   FloatStr((double)prMet->dSaturation_Avg));
    break;
    }
    /*End UserDefined

```

- **Edit setSummaryMetric():** Double-click on CAlgorithm.cpp to open this file for editing. Scroll down until you are positioned to edit the function setSummaryMetric(). The code in this function is analogous to the code in setRegionMetric(). These results will be reported after processing all ROAs. Notice that additional calls to addMetric() have been included here to report the input parameters. It is a good idea to record the input parameter values, along with the actual output results, to record the state of the algorithm run.

The reporting of results is handled automatically in function processViewport(). The developer does not need to be concerned with keeping track of views, when to begin accumulating results for a given ROA, or when an ROA is finished and ready for reporting.

Step 10. Reporting and Working with Image Objects

The sample ColorFilterAlgorithm reports each ROA as a Region-level result. There are times when it is desirable to report objects as regions, such as rare event detection. In this case, the following changes need to be made:

- **Edit processViewport():** Double-click on CAlgorithm.cpp to open this file for editing. Scroll down until you are positioned to edit the function processViewport(). You must edit two code segments here to accurately process image objects and report them as regions. The first code segment enables the AAF overlap logic to handle objects near view boundaries:

```

/*
                                                                    /*UserDefined
    for (int cnt=0; cnt<AlgOutput.NObjects; cnt++)
    {
        if (AlgOutput.Object[cnt].bValid)
        {
            sBBox.xmin = AlgOutput.Object[cnt].xmin;
            sBBox.ymin = AlgOutput.Object[cnt].ymin;
            sBBox.xmax = AlgOutput.Object[cnt].xmax;
            sBBox.ymax = AlgOutput.Object[cnt].ymax;
            if(InCurView(&sBBox) == FALSE)
            {
                AlgOutput.Object[cnt].bValid = FALSE;
            }
        }
    }

                                                                    /*End UserDefined
*/

```

To use this code segment, you must declare and save a bounding box for each object, as well as a boolean value (bValid) for validity testing. This information should be included in the ALGORITHM_OUTPUT structure. A call to function InCurView() determines whether the object contained in the bounding box should be counted in the current view or not. You must also make the appropriate changes to function summarize_output(), to accumulate the summary output for the valid objects.

The second code segment reports each object as a region.

```

/*
                                                                    /*UserDefined
for (int cnt=0; cnt<nObjects; cnt++)
{
    status = 1;
    score = 1;
    ImgId = 0;
    if (AlgOutput.Object[cnt].bValid)
    {
        AlgOutput.Object[cnt].xmin = AlgOutput.Object[cnt].xmin +
        image_xmin;
        AlgOutput.Object[cnt].ymin = AlgOutput.Object[cnt].ymin +
        image_ymin;
        AlgOutput.Object[cnt].xmax = AlgOutput.Object[cnt].xmax +
        image_xmax;
        AlgOutput.Object[cnt].ymax = AlgOutput.Object[cnt].ymax +
        image_ymax;

        sendResultEvent(ImgId, score, status, (void*)&AlgOutput.Object[cnt],
                        eREGION);
    }
}
                                                                    /*End UserDefined
*/

```

In this example, the bounding box information is used to enclose the object. To use this code segment, remove the nested if-statement that does ROA-region reporting and use this code segment instead. Note also that you will need to have zoom-corrected the object coordinates (xmin, xmax, ymin, ymax), as described above.

- **Edit setRegionMetric ():** Double-click on CAlgorithm.cpp to open this file for editing. Scroll down until you are positioned to edit the function setRegionMetric (). You must edit the following code segments here to report the boundary for each object. setRegionMetric is called indirectly when the call to sendResultEvent was made in the prior code segment with the argument eREGION.

```

                                                                    /*UserDefined
addRoatoResult(pResult);
/*
pResult->addPoint(prMet->xmin, prMet->ymin);
pResult->addPoint(prMet->xmin, prMet->ymax);
pResult->addPoint(prMet->xmax, prMet->ymax);
pResult->addPoint(prMet->xmax, prMet->ymin);
pResult->addPoint(prMet->xmin, prMet->ymin);
*/
                                                                    /*End UserDefined

```

The default action is to add the boundary for the region using a call to addRoatoResult(). To report objects as regions, remove this call and un-comment the code section shown.

The coordinates are ordered vertices describing the object boundary. The example shown is for a rectangular boundary.

Aperio Algorithm Framework (AAF) Programmer's Reference

MAN-0067, Revision B