

MetaIO

Medical Image I/O Made Simple

Abstract

Our goal was to create a simple, flexible, cross-platform image file format that supported medical image application development.

This meant that the image file format and its associated image I/O library had to support:

- 1) Image acquisition information essential to the correct processing and alignment of medical images (e.g., voxel/element size and spacing; and image position and orientation)
- 2) MSB (Most-significant-bit = Mac/Sun) and LSB (Least-significant-bit = PC) byte ordering
- 3) Arbitrary atomic pixel types (char, unsigned char, short, float, etc.)
- 4) N dimensional data
- 5) Image data stored in one file or in one file per sub-dimensional (e.g., N-1) slice
- 6) Text-based headers that are easily read and edited
- 7) Optionally storing data in a file(s) separate from the header to simplify conversion to/from other formats

MetaImage is the text-based tagged file format for medical images that resulted. We have now extended that file format to support a variety of objects that occur in medicine such as tubes (for vessels, needles, etc.), blobs (for arbitrary shaped objects), cubes, spheres, etc. The complete library is known as **MetaIO**.

The central code of MetaImage/MetaIO is quite stable. MetaImage has been in use for several years by a wide range of research at UNC, Chapel Hill. New features are occasionally added, but backward compatibility will always be maintained.

Current info on MetaObjects is available from

<http://caddlab.rad.unc.edu/technologies#metaObjects>

Table of Contents

Abstract	1
Table of Contents	2
Introduction and Installation.....	3
Obtaining MetaIO.....	3
Installing The MetaIO Package	3
MetaImage as a stand-alone package	3
MetaIO with the NLM's Insight toolkit (ITK)	4
Quick Start: Data conversion via MetaHeaders.....	4
Reading a Brick-of-Bytes	5
Reading DICOM and Other One-Slice-Per-File Data Formats	6
MetaIO Library.....	7
Reference: Tags of MetaImage.....	8
MetaObject Tags	8
Tags Added by MetaImage.....	8

Introduction and Installation

Obtaining MetaIO

MetaIO is being distributed over the WWW by its developers in the CADDLab at UNC:

<http://caddlab.rad.unc.edu/technologies#metaObjects>

MetaIO is also being distributed with the National Library of Medicine's Insight Toolkit (ITK) for medical image segmentation and registration:

<http://www.itk.org>

Installing The MetaIO Package

MetaIO is a hierarchy of C++ classes and functions. We have yet to find a modern C++ compiler that does not compile MetaIO. Known compatible compilers include G++ v2.95 and beyond (and probably previous), Microsoft Visual C++ 6.0, Sun's CC on Solaris 2.6 and beyond, and compilers on other workstations including HPs, SGIs, and Alpha systems. Please contact us (Stephen R. Aylward, aylward@unc.edu or via <http://caddlab.rad.unc.edu>) if you encounter any incompatibilities between our code and your compiler.

The hierarchy of the software in the stand-alone MetaIO package is as follows:

```
MetaIO/  
    docs/  
    tests/
```

The top level contains the source files, the header files, and the CMakeLists.txt file that is used by the CMake program to compile MetaIO. This document and the MetaObjects www pages are in the docs directory. A sequence of simple tests is available in the tests directory.

The hierarchy of the software in the ITK distribution is as follows:

```
Insight/Code/Utilities/MetaIO/  
    docs/  
    tests/  
  
Insight/Code/IO/  
Insight/Examples/  
    MetaImageReadWrite/  
    MetaImageViewer/  
    MetaImageColorViewer/  
    MetaImageImporter/
```

The files in Insight/Code/Utilities exactly match those in the stand-alone MetaIO package. Routines to wrap metaImage so that it can be accessed via the itkImageIO Object Factory are in /Insight/Code/IO. The details of the Examples are provided later in this documents.

MetaImage as a stand-alone package

If you downloaded MetaIO separate from Insight, you must also download and install CMake from www.CMake.org. That www site provide details regarding installing CMake and using it to compile another program. Follow those directions to create the MetaIO library and tests.

A minimal description of the process is presented next.

- 1) Install CMake
- 2) On Windows:
 - a. Run CMake.exe
 - b. In the source directory field, browse to the top-level of the MetaIO directory
 - c. In the binary directory field, enter the path and name of the source directory, but add “-VC++” to the directory name (this assumes that you are compiling using VC++, otherwise add an equally descriptive name for the directory to store the binaries in such as “MetaIO-bcc” for Borland).
 - d. Press the “configure” button. It will ask if you want to create the binary directory, press “yes.”
 - e. Specify your compiler in the pull-down menu field.
 - f. Press the “configure” button a second time, and then press “okay” to generate the Makefiles for your compiler (e.g., MetaIO.dsw and related files for VC++) in the binary directory.
- 3) On *nix/Cygwin machines:
 - a. At an appropriate spot, create a directory “MetaIO-g++” or give it an equally descriptive name based on your compiler
 - b. “cd MetaIO-g++”
 - c. Run “cmake <path to MetaIO source directory>”
 - d. Press “g” to generate the makefiles for your compiler in the binary directory.

To use MetaIO in your stand-alone programs:

- 1) add the MetaIO directory to your include paths,
- 2) add “#include <metaImage.h>” to the top of the your file that performs your image IO,
- 3) add the MetaIO binary directory to your link paths, and
- 4) link with the MetaIO library.

See the files in MetaIO/tests for examples of how to read/write MetaImages and other MetaObjects.

MetaIO with the NLM’s Insight toolkit (ITK)

If you have downloaded and are installing MetaIO as part of the Insight toolkit, follow the standard installation procedure of Insight. This will create the MetaIO library and the ITK specific wrapping, examples and tests. Certain examples, such as the MetaImageViewer, also require FLTK (a cross-platform user interface library available from <http://fltk.org>). Install FLTK and then ITK to have every MetaIO example built. Numerous other ITK examples also rely on FLTK.

See the file /Insight/Examples/MetaImageReadWrite for a working example on how to develop a program using MetaImage for IO. Work is underway to add access to other MetaObjects (e.g., tubes, spheres, etc.) via ITK’s SpatialObjects in /Insight/Code/SpatialObject.

Quick Start: Data conversion via MetaHeaders

This section assumes that you have some data that you wish to process using an application that reads MetaImages. This section gives examples on how “convert” your data to the MetaImage format.

For uncompressed data (i.e., data stored in a raw format as is often done for DICOM, BMP, and PNG formats), “conversion” to MetaImage is actually just a matter of specifying a MetaImage Headerfile (a “MetaHeader”) that describes and points to the file(s) containing your data.

For compressed data (i.e., data stored in JPEG or GIF formats), you must first convert your data to a non-compressed format. Currently, no data compression methods are supported by MetaImage, but adding support for .gz, RLE, and other compression formats is high on our To-Do list. One of the most robust

image conversion software packages is ImageMagick (<http://www.imagemagick.org/>; Unix and PC versions available). It has an application called “convert” that handles most of the popular 2D image formats.

IF YOU HAVE INSTALLED METAIO AS PART OF ITK, USE THE PROGRAM Insight/Examples/MetaImageImporter. IT WILL ASK A SERIES OF QUESTIONS AND PRODUCE A METAIMAGE HEADER FOR YOU.

Reading a Brick-of-Bytes

A “brick of bytes” is a volume of image data stored in a single file possibly with preceding and trailing non-image data.

To correctly load these images, the minimal information that you need to know is:

- 1) Number of dimensions
- 2) Size of each dimension
- 3) Data type
- 4) Name of the data file

For example, let’s say the data was 3 dimensional, had 256 x 256 x 64 voxels, used an unsigned short to represent the value at each voxel, and was stored in the file “image.raw”. The resulting MetaHeader (our naming convention would call this file “image.mhd”) file would read

```
NDims = 3
DimSize = 256 256 64
ElementType = MET_USHORT
ElementDataFile = image.raw    (this tag must be last in a MetaImageHeader)
```

That’s it, but this assumes quite a bit about the image data. Specifically, it assumes

- 1) There are not any non-image data bytes (header data) at the beginning of the image data file “image.raw”.
- 2) The voxels are cubes – the distance spanned by and between a voxel in each coordinate direction is 1 “unit”, e.g., 1x1x1mm voxel size and voxel spacing
- 3) The byte-order of the data in image.raw matches the byte ordering native to the machine the application is running on (e.g., PC’s use LSB ordering and Suns/Macs use MSB ordering).

If these assumptions are false, the data will not be loaded correctly by the application. To fix these problems....

- 1) To skip the header bytes in the image data file, use
HeaderSize = X
where X is the number of bytes to skip at the beginning of the file before reading image data. If you know there are no trailing bytes (extra bytes at the end of the file) you can specify
HeaderSize = -1
and MetaImage will automatically calculate the number of extract bytes in the data file, assume they those bytes are at the head of the data file, and automatically skip them before beginning to read the image data.
- 2) To specify the spacing of the voxels, use
ElementSpacing = X Y Z
where X is the distance between of the centers of the voxels along the x-dimension, Y is the spacing in the y-dimension, and Z is the spacing in the z-dimension. Therefore, to specify a 1x1x3mm voxel spacing, use
ElementSpacing = 1 1 3

NOTE: If ElementSpacing is not specified, it is assumed to be equal to ElementSize. If neither is specified, both are assumed to be 1.

- 3) To specify a voxel size, use
ElementSize = X Y Z
where X Y Z represent the size in the x, y, and z-dimensions respectively.

NOTE: If ElementSize is not specified, it is assumed to be equal to ElementSpacing. If neither is specified, both are assumed to be 1.

- 4) To specify a particular byte ordering, use
ElementByteOrderMSB = True
or
ElementByteOrderMSB = False
MSB (aka big-endian) ordering is common to SPARC and Motorola processors (e.g., Macintoshes). LSB (aka little-endian) ordering is common to Intel processors (e.g., PC compatibles).

Putting it all together, to “convert” a file containing the image data in a continuous block at the end of the file, specify the header

```
NDims = 3
DimSize = 256 256 64
ElementType = MET_USHORT
HeaderSize = -1
ElementSize = 1 1 3
ElementSpacing = 1 1 1
ElementByteOrderMSB = False
ElementDataFile = image.raw
```

Reading DICOM and Other One-Slice-Per-File Data Formats

If the data is split to be one slice per file, as is done with DICOM data, only the ElementDataFile tag's option needs to change.

Since the MetaLibrary cannot automatically parse DICOM headers, those headers must be skipped and the user must specify the image dimensions and other essential image information. For DICOM files, the MetaLibrary must automatically calculate the header size of each file (luckily for almost every type of DICOM object in common use, the image data is stored at the end).

To specify which files comprise the volume, they can be specified as an ordered list in the MetaHeader using the ElementDataFile=LIST option. The filenames should be listed at the end of the MetaHeader, after the ElementDataFile option, and the filenames should be separated by whitespace:

```
NDims = 3
DimSize = 512 512 100
ElementType = MET_USHORT
HeaderSize = -1
ElementSize = 1 1 3
ElementSpacing = 1 1 1
ElementByteOrderMSB = False
ElementDataFile = LIST
filenameOfSlice1
filenameOfSlice2
filenameOfSlice3
filenameOfSlice4
.
```

. (one hundred filenames must be specified to specify the 100 slices in the volume)

The second way of specifying a series of files can be used if the filenames are numerically distinguished. That is, the files names should be able to be specified using a numeric substitution into a c-style printf-string, for a range of values. In pseudo-code:

```
for i=numBegin to numEnd step numStep
    sprintf(sliceName, "baseName.%03d", i);
end
```

The parameters of this system are numBegin, numEnd, numStep, and the c-style printf string (e.g., "baseName.%03d"). The begin, end, and step parameters appear in order after the c-style printf string:

```
NDims = 3
DimSize = 512 512 100
ElementType = MET_USHORT
HeaderSize = -1
ElementSize = 1 1 3
ElementSpacing = 1 1 1
ElementByteOrderMSB = False
ElementDataFile = baseName.%03d 1 100 1
```

This MetaImage will cause the files "baseName.001" to "baseName.100" to be read to create a 100-slice volume.

In this case, because of the overlap of the slices, it may be helpful to only consider every-other slice in the volume. Changing the slice spacing and the ElementDataFileNumStep enacts this...

```
NDims = 3
DimSize = 512 512 50
ElementType = MET_USHORT
HeaderSize = -1
ElementSize = 1 1 3
ElementSpacing = 1 1 2
ElementByteOrderMSB = False
ElementDataFile = baseName.%03d 1 100 2
```

The complete set of MetaImage Tags are given in the Reference section of this document. The next section discusses how to use the MetaImage Library for image reading and writing in your own programs.

MetaIO Library

The base class of the MetaIO library is the MetaObject class. It defines a base set of tags that are common to all metaObjects such as MetaImages, MetaTubes, etc.

The tags are defined using the protected member functions SetupReadFields and SetupWriteFields. These functions create a list of MetaFieldRecords to define the name, type, interdependence, and necessity of each tag. Helper functions for defining the fields are in MetaUtils.cxx. The types are defined via enums in MetaTypes.h

The derived classes add tags to the list via their own SetupReadFields and SetupWriteFields member functions. The MetaImage subclass also re-implements the Read and Write methods since non tag data (i.e., the pixel values) must also be read. Compare the derived classes for MetaCube and MetaImage.

Reference: Tags of MetaImage

MetaObject Tags

The tags of MetaObject are:

Comment	– MET_STRING	– User defined - arbitrary
ObjectType	– MET_STRING	– Defined by derived objects – e.g., Tube, Image
ObjectSubType	– MET_STRING	– Defined by derived objects – currently not used
TransformType	– MET_STRING	– Defined by derived objects – e.g., Rigid
NDims	– MET_INT	– Defined at object instantiation
Name	– MET_STRING	– User defined
ID	– MET_INT	– User defined else -1
ParentID	– MET_INT	– User defined else -1
BinaryData	– MET_STRING	– Are the data associated with this object stored at Binary or ASCII – Defined by derived objects
ElementByteOrderMSB	– MET_STRING	
BinaryDataByteOrderMSB	– MET_STRING	
Color	– MET_FLOAT_ARRAY[4]	– R, G, B, alpha (opacity)
Position	– MET_FLOAT_ARRAY[NDims]	– X, Y, Z,... of real-world coordinate of 0,0,0 index of image)
Orientation	– MET_FLOAT_MATRIX[NDims][NDims]	– [0][0],[0][1],[0][2] specify X, Y, Z... direction in real-world of X-axis of image – [1][0],[1][1],[1][2] specify X, Y, Z... direction in real-world of Y-axis of image, etc.
AnatomicalOrientation	– MET_STRING	– Specify anatomic ordering of axis. For example, RAS mean x-axis is Right to Left, y-axis is Anterior to Posterior, and Z-axis is Superior to Inferior. Use only [R L] [A P] [S I] per axis.
ElementSpacing	– MET_FLOAT_ARRAY[NDims]	– The distance between voxel centers

Tags Added by MetaImage

In addition to the above tags, MetaImage provides the following tags:

DimSize	– MET_INT_ARRAY[NDims]	– Number of elements per axis in data
---------	------------------------	---------------------------------------

HeaderSize – MET_INT – Number of Bytes to skip at the head of each data file.
Specify –1 to have MetaImage calculate the header size based on the
assumption that the data occurs at the end of the file.

Modality – MET_STRING – One of enum type: MET_MOD_CT, MET_MOD_MR,
MET_MOD_US... See MetaImageTypes.h

SequenceID – MET_INT_ARRAY[4] – Four values comprising a DICOM sequence: Study, Series,
Image numbers

ElementMin – MET_FLOAT – Minimum value in the data

ElementMax – MET_FLOAT – Maximum value in the data

ElementNumberOfChannels – MET_INT – Number of values (of type ElementType) per voxel

ElementSize – MET_FLOAT_ARRAY[NDims] – Physical size of each voxel

ElementType – MET_STRING – One of enum type: MET_UCHAR, MET_CHAR... See
MetaImageTypes.h

ElementDataFile – MET_STRING – One of the following:

- Name of the file to be loaded
- A printf-style string followed by the min, max, and step values to be used to pass an argument to
the string to create list of file names to be loaded (must be (N-1)D blocks of data per file).
- LIST [X] – This specifies that starting on the next line is a list of files (one filename per line) in
which the data is stored. Each file (by default) contains an (N-1)D block of data. If a
second argument is given, its first character must be a number that specifies the
dimension of the data in each file. For example ElementDataFile = LIST 2D means that
there will be a 2D block of data per file.
- LOCAL – Indicates that the data begins at the beginning of the next line.