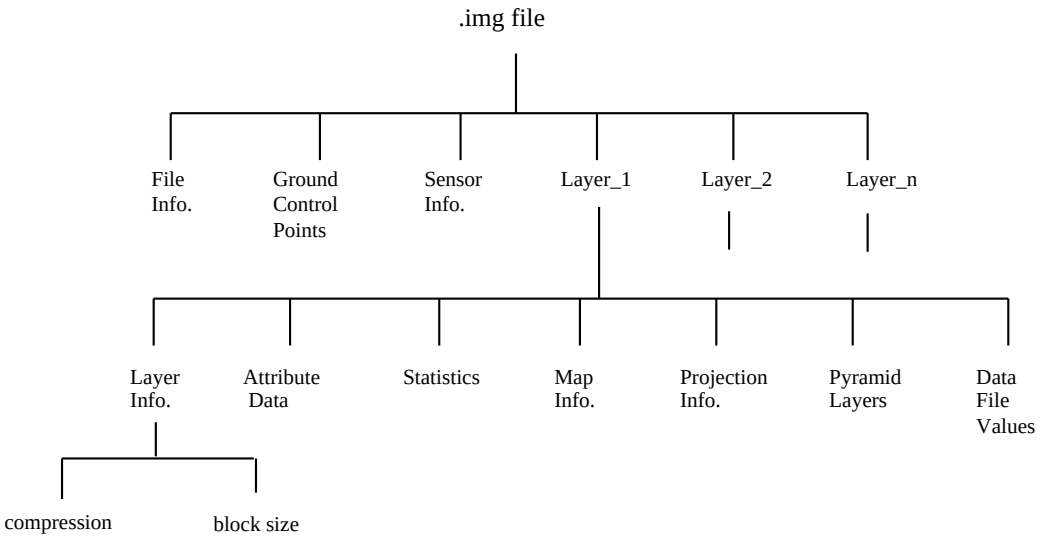
ERDAS IMAGINE .img Files

ERDAS IMAGINE uses .img files to store raster data. These files use the ERDAS IMAGINE Hierarchal File Format (HFA) structure. Figure 1, below, shows the different objects of data stored in a .img file. The contents of the .img file is not fixed. Many of the items shown below are optional. In addition, because of the open nature of the file format, other developers may create and add new types of items to the file.

The C Programmers' Toolkit provides programmers with the functions needed to read and write ERDAS IMAGINE .img files. Even though the file format is documented in the **HFA Object Directory** chapter, it is recommended that you read this section because it will greatly reduce development time. In addition, it will guarantee compatibility with future versions of the .img file format.



- file name
- layer name
- number of layers

- date the file was last modified

This information applies to all of the layers.

Sensor Information

When you import satellite imagery from a tape or CD–ROM, there is usually a header file preceding the image data on the input medium which is extracted by the import program and stored as an object in the .img file. This object contains ephemeris information about the sensor, such as:

- date and time scene was scanned
- calibration information of the sensor
- orientation of the sensor
- original dimensions for data
- data storage format
- number of bands

The data presented are dependent upon the sensor. Each sensor provides different types of information. The sensor object is named:

<format type>_Header

Sensor	Sensor Object
ADRG	ADRG_Header
ADRI	ADRI_Header
DEM	DEM_Header
DTED	DTED_Header
Landsat TM	TM_Header
Landsat MSS	MSS_Header
NOAA AVHRR	AVHRR_Header
SPOT	SPOT_Header

Raster Layer Information

Each raster layer within a .img file has its own ancillary data, including the following parameters:

- height and width (rows and columns)
- layer type (continuous or thematic)
- data type (signed 8–bit, floating point, etc.)
- compression (see below)
- block size (see below)

This information is usually the same for each layer.

Compression

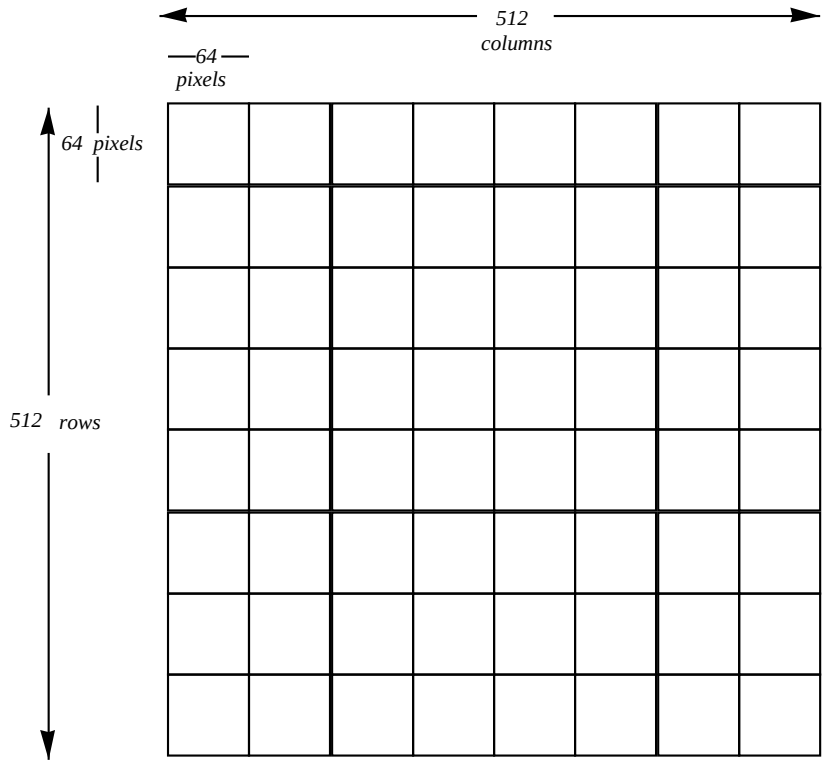
When you import a file into ERDAS IMAGINE you have the option to compress the data. Currently, ERDAS IMAGINE uses the run–length compression method. The amount that the data are compressed depends on data in the layer. For example, if the layer contains large homogenous areas (e.g., blocks of water) then compressing the layer would save you disk space. However, if the layer is very heterogenous, then run–length compression would not save you much disk space.

Data will be compressed only when it is stored. ERDAS IMAGINE automatically uncompresses data before the layer is run through a process. The time that it takes to uncompress the data is minimal.

Block Size

ERDAS IMAGINE software uses a ***tiled format*** to store raster layers. The tiled format allows raster layers to be displayed and resampled quickly. The raster layer is divided into tiles (i.e., blocks) when ERDAS IMAGINE creates or imports a .img file. You can define the size of these blocks when you either create the file or import it. The default block size is 64 pixels by 64 pixels.

i The default block size is acceptable for most applications and should not need to be changed.



- histogram
- contrast table

Thematic Raster Layer

For a thematic raster layer, the attribute table object, by default, includes the following information:

- histogram
- class names
- class values
- color table (red, green, and blue values).

Attribute data can also include additional information for thematic raster layers, such as the area, opacity, and attributes for each class.

Statistics

The following statistics are calculated for each raster layer:

- minimum and maximum data file values
- mean of the data file values
- median of the data file values
- mode of the data file values
- standard deviation of the data file values

You should create statistics for a layer if they do not exist. Certain Viewer functions (e.g., contrast tools) will not run without layer statistics. Rebuilding statistics for a raster layer may be necessary. For example, if you decide that you do not want to include zero file values in the statistics calculation (and they are currently included), you could rebuild the statistics without zero file values.

Map Information

Map information for a raster layer will be created only when the layer has been georeferenced. If the layer has been georeferenced, the following information will be stored in the raster layer:

- upper left X,Y coordinates
- pixel size
- map unit used for measurement (e.g., meters, inches, feet)

Map Projection Information

If the raster layer has been georeferenced, then the following projection information will be generated for the layer:

- map projection
- spheroid

- zone number

Pyramid Layers

ERDAS IMAGINE gives you the option to "pyramid" large raster layers for faster processing and display in the Viewer. When you generate pyramid layers, reduced, subsampled raster layers are created from the original raster layer. The number of pyramid layers that are created depends on the size of the raster layer and the block size.

For example, a raster layer which is 4k x 4k pixels could take a long time to display when using the Viewer **Fit To Window** option.Using the Compute Pyramid Layers option in ImageInfo will cause ERDAS IMAGINE to create additional raster layers successively reduced from 4k x 4k, to 2k x 2k, 1k x 1k, 512 x 512, 128 x 128, down to 64 x 64. Then ERDAS IMAGINE would select the pyramid layer size most appropriate for display in your Viewer window.

Reading and Writing .img Files

This section describes each of the C Toolkit packages that are needed to read and write ERDAS IMAGINE .img files. The detailed format of the .img file is documented in **Machine Independent Format** and **HFA Object Directory**.

ERDAS IMAGINE functions are grouped into packages named with four letter mnemonics that always begin with a lowercase e. Each package focuses on some specific area of functionality. The complete reference pages for each package are accessible from **C Programmers' Toolkit Packages**.

Each package has an associated header file whose name is constructed from the package name plus the .h extension. All of the data types and function prototypes for the package are defined in the header file. Most packages create one or more types of structures. There are functions provided in each package to free the specific data type. The name is either exxx_DataTypeFree or exxx_DataTypeDelete.

This section gives an overview of each package. Click on the package name to open the reference pages for that package.

eerr

The eerr package is the ERDAS IMAGINE error handling system. Most functions take a pointer to an Eerr_ErrorReport *. The Eerr_ErrorReport is a structure which contains an error code and an error message. If a NON–NULL value is returned for this argument then an error has occurred. The code can be checked and the message can printed. Whether or not the error report is printed, it is the caller's responsibility to free it.

emif

The lowest level of the I/O packages is the emif (ERDAS Machine Independent Format) package. Emif provides tools for creating definitions of C structures so that they can be written to a file on one machine and read back on another with a different architecture.

The emif package works with designs which reside in dictionaries. Each design is a description of a C structure. The design tells the emif package how to pack and unpack the data between the computers host memory and the storage medium. Each design is created and placed in a dictionary. The dictionary can be written out in a compact form.

The details of the emif disk file format and dictionary format are given in the Machine Independent Format chapter.

ehfa

The ehfa (ERDAS Hierarchal File Architecture) package is built on top of the emif package. It provides the tools to create a tree structured collection of objects in a single file which can be written and read as named objects.

Every node in an HFA file has a name. The full HFA node path name is formed by enumerating a colon separated list of all direct ancestors of that node starting from the root node. For example lanier.img(:Layer_1:Descriptor_Table:Histogram) is the name of the table which contains the histogram information for Layer_1 in the file lanier.img.

A directory of the objects stored in an IMAGINE .img file are given in the HFA Object Directory chapter.

edsc

The edsc package is used to create and maintain descriptor tables. A descriptor table is a node in the file which contains one or more columns of information. The main descriptor table in a .img file is used to store information which is specific to each pixel value, such as histograms or contrast tables. At other times, they are simply used to store a variable list of objects that have common attributes such as ground control points.

Each column in a descriptor table can be though of as an array of integers, floats, or strings. All columns in a given table must have the same number of elements.

eprj

The eprj package provides the functions used to create, manipulate, and maintain map and projection information. The map information describes the map origin and cell size for the pixels in an image which are used to convert from pixel row column to map x, y. The projection information is used to convert from map x,y to latitude, longitude.

The HFA Object Directory chapter contains a description of the information found in the projection record in the file.

esta

The esta package provides the functions used to create, manipulate, and maintain image statistics. These include single layer statistics such as mean, maximum, minimum, standard deviation, and multilayer statistics such as the covariance matrix.

eimg

The eimg package is the main interface to ERDAS IMAGINE .img files. It is built upon the previously described packages and, for the most common operations of reading and writing image data, it provides the complete interface to the ERDAS IMAGINE .img files.

estr

The estr (string) package provides a number of functions for manipulating strings of characters and arrays of strings of characters.

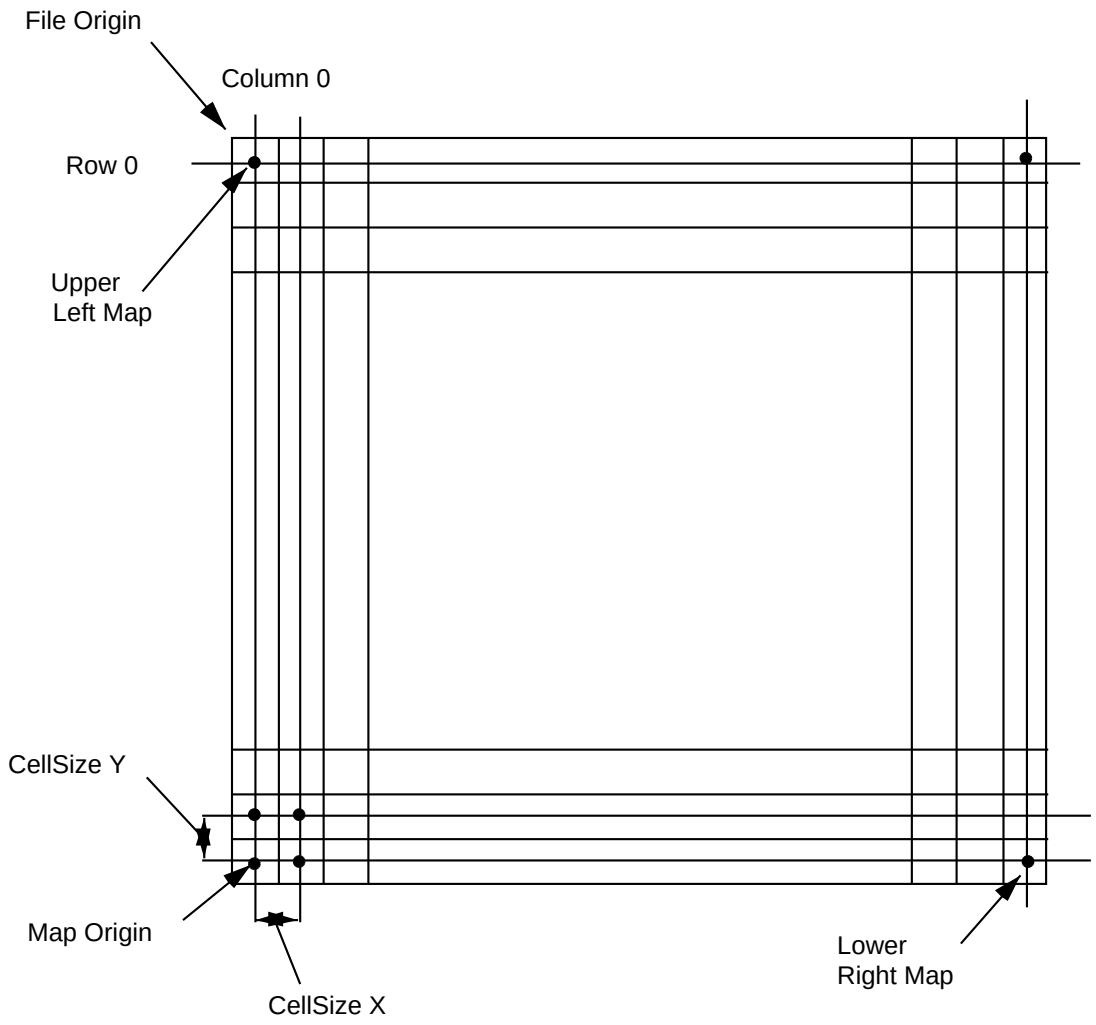
emap

The emap package contains functions for reading and writing map composition files.

Coordinate Systems

Raw Coordinate System

An IMAGINE image file is a rectangular array of pixels. The leftmost column of pixels is column 0 and the topmost row of pixels is row 0. The row(y) and column (x) numbers increase to the right and down. This is the coordinate system which is used with the eimg_LayerRead and eimg_LayerWrite functions.



origin at the lower left pixel in the image. The coordinate system is assigned by specifying the map coordinate which corresponds to the center of the upperleft pixel and the map coordinate which corresponds to the center of the lower right pixel. The pixel separation is also specified as the X Cellsize and the Y Cellsize. The specification of upperleft, lowerright, and cellsize is redundant. Only the upperleft and cellsize are actually used in practice.

The following formulas are used to convert between row/column and map x/y.

$$x = X_{ul} + c \times X_{size}$$

$$y = Y_{ul} - r \times Y_{size}$$

$$c = \frac{x - X_{ul}}{X_{size}}$$

$$r = \frac{y + Y_{ul}}{Y_{size}}$$

```
distance {
   meters 1.0 ;
   meter 1.0 ;
   feet 0.3048006099012192 ;
      :
}
```

This says that meters (or meter) is the standard unit of distance and that feet are converted to units by multiplying by 0.304800609901219. To add other units (surveyfeet for example) one would edit the units.dat file and add a new entry in the distance category which looks like "surveyfeet 0.314159....; " (whatever the proper conversion factor is). IMAGINE must be restarted for the new units to take effect. The new units will not show up in any dialogs automatically, but they will be understood by the system. The EML scripts must be edited to have the new unit appear to the user.

Here is an example of reading the MapInfo header.

```
Eimg_MapInfo *mi;
double xul, yul, xsize, ysize;
double x, y;
double xm, ym;
int r, c;

mi = eimg_MapInfoRead(layer, NULL, &err);

if ( mi==NULL) /* No Map Information available */

xul = mi->upperLeftCenter->x;
yul = mi->upperLeftCenter->y;
xsize = mi->pixelSize->width;
ysize = mi->pixelSize->height;
/*
** Get the map coordiante of pixel 100, 100.
*/
r = 100;
c = 100;

x = xul + c*xsize;
y = yul - r*ysize;

/*
** x and y are in the units of the map header. To convert to meters,
```

```
** do the following:
*/
ecvt_UnitsConvertByName(&x, mi->units.data, &xm, "meters", 1, &err);
ecvt_UnitsConvertByName(&y, mi->units.data, &ym, "meters", 1, &err);

/*
** This is somewhat inefficient because it does the lookup of "meter" and
** the files units per number. Use ecvt_UnitSetByName to find the input
** and output units beforehand and then just use ecvt_UnitsConvert() to do
** the converstion. The x and y values can also be placed into arrays since
** the unit convert functions work on arrays.
*/
/*
** The following would be used to write a simple cartesian map system to
** file which is nrows by ncols.
*/
Eprj_MapInfo *mi;
double xul, yul, xlr, ylr, xsize, ysize;

/*
** Lets assume that the cellsize is 10 meters.
*/
xsize = 10.0;
ysize = 10.0;

xul = 1000000.0;
yul = 1000000.0;
xlr = xul + (ncols - 1) * xsize;
ylr = yul - (nrows - 1)*ysize;

mi = eprj_MapInfoCreate(&err);
eprj_MapInfoUpperLeftCenterSet(mi, xul, yul, &err);
eprj_MapInfoLowerRightCenterSet(mi, xlr, ylr, &err);
eprj_MapInfoPixelSizeSet(mi, xsize, ysize, &err);
eprj_MapInfoUnitsSet(mi, "meters", &err);
eprj_MapInfoProjectionNameSet(mi, "Unknown", &err);
/*
** Now write it to the layer.
*/
eimg_MapInfoWrite(layer, mi, &err0;
```

Radial Units

If the image coordinate system is radial, i.e., it is given as latitude longitude, then the default units are decimal degrees (given as "dd"). A file with a projection name of "Geographic" has radial units. The units package can be used to convert dd to radians just as it is used to convert feet to meters. However, there is nothing which forces the radial coordinate image file to be in decimal degrees. It could be in radians as well. The programmer must check the units.

Geographic Coordinates (Projections)

The MapInfo object also contains the name of the projection which is used to convert from the map system to geographic latitude / longitude coordinates. If the projection name is "Unknown" then the map system is a simple cartesian system which cannot be converted to lat/lon. However, if the name is other than "Unknown" then the map coordinates x and y can be converted to lat/lon using the projection package.

```
Eprj_ProParameters *pp;
Eprj_MapProjection *mapp, geop;
Eprj_Point pin, pout;


if (estr_Eq(mi->proName.data, "Unknown")... /* No Projection Info */

pp = eimg_ProParametersRead(layer, NULL, &err);

if (pp==NULL)... /* No projection info */

/*
** Convert the xm, ym to lat/lon (xm,ym must be in meters. The projection package
** expects all map coordinates in meters and all angle (lat/lon) in radians.
*/
geop = eprj_ProjectionInit(eint_GetInit(), eprj_ProParametersCreate(&err), &err);
mapp = eprj_ProjectionInit(eint_GetInit(), pp, &err);
pin.x = xm;
pin.y = ym;
eprj_Project(&pin, mapp, &pout, geop, 1, &err);

/*
** pout.x is now the longitude in radians.
** pout.y is now the latitude in radians.
**
** ecvt can be used as above to convert to degrees.
*/
```

Calibration

An IMAGINE image file can also have a Calibration Record. This calibration record can contain a polynomial which is used to convert from r,c to x,y and visa versa. This allows an arbitrary coordinate system to be established on the image. This transformation converts from row column to map x and map y. The calibration has a MapInfo object which is used to assign units to the coordinates and which is used to indicate whether there is an associated Map Projection. If there is a map projection it is used as described above.

This calibration node is accessed using the erec_CalibrationNode functions. erec_CalibrationNodeRead is used to read the calibration node. If the node is present then this functions returns the rc–>xy polynomial, the xy–>rc polynomial, the MapInfo and the ProParameters to be used with the calibration. The function efga_PolyTransCoords is used to apply the polynomial to the coordinate pairs.

NOTE: Calibrated files are discouraged at this point because the only application which deals effectively with them is the viewer. The calibration mechanism was created so that measurements could be taken directly from a file which had not been resampled. Using them effectively in applications such as Modeler or Mosaic requires on the fly Nth order polynomial resampling, which is not yet implemented throughout IMAGINE.