Plug-in Developer's Tool Kit

v 1.1

Electric Image, Inc.

Index

General Introduction

Overview of the Plug-in Tool Kit

In version 2.0 of the ElectricImage™ Animation System (EI), two classes of plug-ins were implemented for internal and 3rd party developers. The purpose of plug-ins is to allow for expansion of EI's feature set between major updates of the software. Since a plug-in may be developed by 3rd party companies and individuals, features which are needed for specific purposes can be added whenever they are needed. The two classes are model generator and light flare effects. The following plug-ins are shipped with EI 2.1.2:

StandardShapes	This model plug-in creates planes, cubes, cylinders, cones and spheres.
Mesh	This model plug-in creates a flat mesh and animated sine wave, breathing cylinder, screw thread and sphere wrap meshes.
Particle	This model plug-in generates an animated particle system with motion-blurred points. This plug-in can create a variety of effects including a fountain, water stream, bomb burst and a shower of sparks.
Mr. Nitro™	This model plug-in explodes the groups which are linked to it in ElectricImage. It can generate one or two pass explosions which rip apart the groups into fragments. The fragments' motion can be affected by gravity, air resistance and turbulence. Each fragment can be motion blurred.
Mesher	This model plug-in converts all polygons in attached groups into evenly sized meshes. This allows any model to be animated with ElectricImage's deformation tools.
Mr. Blobby™	This model plug-in uses an implicit surface algorithm to generate spherical blobs from attached groups, facets or vertices. It demonstrates how to allocate memory and pre-generate data during the setup plug-in command.
LensFlare	This light flare plug-in generates standard lens flare effects. There are two lens flare effects built into the plug-in. The plug-in uses data tables in its resource fork to store the ring and spike elements of each lens flare. The plug-in itself can be expanded in the future to provide additional lens-flare types. This plug-in creates animation channels in ElectricImage for each of its ring elements.

The Mesh, Particle and Mr. Blobby™ plug-ins are provided in the developer kit as source code examples. In addition, two more sample plug-ins are provided as source code:

Morph	This model plug-in uses two or more groups which are linked to it in ElectricImage and generates an in-between morphed group. It has the limitation that it only works with groups which have the same number of vertices.
Glow	This light flare plug-in generates a simple circular shaped flare at the location of the light source. It uses the light source's color and size to control the appearance of the flare.

Software Licensing

The sample source code plug-ins may be used and modified by plug-in developers. A developer may release modified versions of these plug-ins in the Electric Image forum on America Online as freeware but must obtain a license from Electric Image, Inc. for any code they wish to use from the body of these plug-ins in their own commercial products. The plug-in libraries and headers may be used without a license. The plug-in developer kit will have two folders for source code to clearly separate the unlicensed plug-in libraries from the licensed plug-in sample code.

Plug-in Hosts

The ElectricImage™ Animation System version 2.0 and 2.1 consists of three main applications:

ElectricImage	This is the main user interface application. ElectricImage allows users to design animations by adding objects to a project list and animating a wide range of values for each object.
Camera	This application renders control files it receives from ElectricImage. It has almost no user interface of its own. It creates animations in EI's own multi-frame image file format.
Projector	This application provides several image and animation utilities. It can display images and animations. It can convert animations into FastLoad files which can be played back in real time. It can convert other file formats to and from EI's own image file format.

EI plug-ins can be "hosted" by both ElectricImage and Camera. ElectricImage adds plug-ins to its project list and allows users to interact with the plug-in's user interface and to preview the results before rendering. ElectricImage also keeps track of any animation channels which are created by the plug-ins. Camera calls each plug-in at every rendered frame in the animation in order to generate model data or light flare effects.

The plug-in host must perform two functions. It must send commands to the plug-in and it must provide services by receiving commands the plug-in sends to it. In addition to ElectricImage and Camera, there are test applications provided with the EI plug-in developers kit which will act as hosts to EI plug-ins. It is possible that other companies or individuals may write their own host applications which could accept EI plug-ins in the future.

In addition, each of the sample code plug-ins comes with a simple test shell which calls the plug-in directly as a function. The test shell acts as a very simple host application and can be used by developers to debug their plug-ins. The utility applications ModelHost and FlareHost are provided in the EI Plug-in Developers Kit. These applications are used to both test the plug-ins in a host application environment, to setup default plug-in values and to copy a plug-in's PowerPC shared library code into the plug-in's resource fork.

Plug-in Operations

Model plug-ins can create vertex and facet geometry either algorithmically or by referencing groups which have been linked to them in ElectricImage. Particle is an example of a model plug-in which generates geometry algorithmically. It creates a series of particles. For each particle it computes its position at the current frame by taking into consideration the time the particle was created, the particle's initial direction and velocity, the force of gravity and the life span of the particle. It then generates a vertex and a facet for the particle and sends them to the host where they are displayed or rendered.

Mr. Nitro is an example of a plug-in which generates geometry by referencing the groups which were linked to it in ElectricImage. A linked group is referred to as a child group. Mr. Nitro computes the radius of a simulated explosion at the current frame in the animation. It then determines which facets of its child groups are inside the radius. Facets which are outside the blast radius are simply copied and sent back to the host without modification. Facets which are partially or completely inside the blast radius are broken up into smaller fragments. Each fragment's position and rotation is calculated based upon the force and speed of the shock wave, rotation speed, gravity, air resistance and turbulence. Each fragment is then sent to the host be displayed or rendered. It is possible that hundreds of fragments or more may be generated by a single facet in a child group. If instead of rendering the fragments, the host wrote each fragment to a data file, even a short animation could generate hundreds of millions of fragment facets which would occupy many gigabytes of disk storage. This demonstrates how model plug-ins can save time and disk space by generating geometry as it is being displayed or rendered.

The light flare plug-in allows special visual effects to be added to an image after it has been rendered. The LensFlare plug-in contains a database of ring and spike elements. Each ring element has a record which describes its size, location and an array of color values. LensFlare uses the ring element record to generate a circular image of the ring which is then added to the flare plug-in's frame buffer. By creating a series of these rings in different colors along the diagonal line between a light source and the center of the image, a lens flare effect is created. The spike elements are just thin lines drawn radially from the light source's position. Each spike element also has a record in LensFlare's resource file which contains the spikes length, direction, position and color table. In addition to sending the LensFlare plug-in the information it needs to draw the effect, Camera can also determine the visibility of the flare's light source and will turn off the flare plug-in if it is obscured by another object in the frame.

Interfacing with ElectricImage

EI plug-ins are compiled as code resources. The plug-ins contain separate code resources for 68K and PowerPC code. The 68K version of EI can only communicate with the plug-in's 68K code and the PowerPC version of EI can only communicate with the plug-in's PowerPC code. The plug-in's PowerPC code resource is compiled as a shared library and is then copied to a resource by the ModelHost or FlareHost test applications. The test applications also display the plug-in's user interface dialog box and save the plug-in's default control parameters in the plug-in's resource file. ModelHost creates a sample model for the plug-in which can be used to examine the geometry the plug-in is creating by opening it in ElectricImage as a model file instead of adding it as a plug-in.

The finished plug-in is placed into the "EI Sockets" folder which is in the same folder as ElectricImage. ElectricImage will allow the user to add model and flare plug-ins located in the "EI Sockets" folder to their animation project. The user can create several copies of a plug-in and then use the ModelHost and FlareHost applications to set different default control values for each copy. Every model plug-in in the "EI Sockets" folder will appear as a pop-up menu item in the add model dialog and as an icon in the model palette. Flare plug-ins appear in a pop-up menu in the light information dialog box.

Each plug-in has a single `main` routine which the host calls to send it commands. The plug-in may send commands back to the host by calling a function which is provided as part of the plug-in's command record.

Plug-ins may access Macintosh toolbox functions but this should be done as little as possible to allow the plug-in to be easily ported to the SGI and other computer platforms. Toolbox routines are commonly used when the plug-in is displaying its user interface or about box. Since the interface and about box commands are never sent to the plug-in by Camera, a portable version of the plug-in can be compiled for use with the SGI Slave Camera which does not use any of the Macintosh toolbox routines.

Terminology

The text and source code will use Pascal types. This is done to provide consistency between the original C and Pascal headers. These types are as follows:

Integer	2 byte short integer
UInteger	2 byte short unsigned integer
LongInt	4 byte long integer
ULongInt	4 byte unsigned long integer
Real	4 byte single precision float
Double	8 byte double precision float
Extended	8 byte float on PowerPC and SGI; 12 byte float on 68K Macintosh

All string types conform to Pascal string conventions with a length byte followed by characters.

The term 'record' is used to refer to a 'struct' in C.

The term 'Host' will refer to the application which is hosting the plug-in. The host applications are ElectricImage, Camera, TestModel and TestFlare. It is possible that other applications may also host ElectricImage plug-ins in the future.

What's New

Version 1.1 adds new capabilities to the animation channel control. Now, all plug-in types can create and maintain their own animation channel hierarchy. A previous limitation on the number of hierarchy levels have been lifted with the release of Electric Image version 2.7.5. Two new animation channel data types have been added: Unsigned floats and 3D coordinates. Also, an additional data type has been added to allow modification of existing values using string type operators (see `hostSetChannelKey` command). There are new commands added to allow the plug-in greater control on key frame creation and maintenance.

In addition to the `hostAddChannel` command there is a new `hostInsertChannel` command which allows the developer to insert a new animation channel without rebuilding the animation channel structure.

Animation channel can be queried using the `hostGetChannelInfo`. This command provides valuable information about the type and state of an animation channel.

The system's animation track can be queried using the `hostGetTimingInfo` command. This allows the plug-in a "peek" at the current length and number of frames of the project's animation.

Greater key frame control is allowed using `hostAddChannelKey`, `hostDeleteChannelKey`, `hostGetChannelKey`, and `hostSetChannelKey`.

Abstract

The Communication Process

An ElectricImage plug-in can be thought of as a simple program. The host application can be thought of as a mini operating system for the plug-ins. The plug-in is "launched" every time the host sends it a command. The plug-in code continues to execute until the plug-in is finished processing the command and exits to the host application. While the plug-in is executing a command, it may send its own commands to the host application to request various services. The host application provides a function pointer to its plug-in service routine as part of the plug-ins command record. The host will never send another command to the plug-in while it is processing a plug-in service command.

The structure of a plug-in command or a host service command is identical. Each command consists of a four character command identifier, a command record size and a command record pointer. Each command identifier is defined as a unique constant in the file EIPlugin.h. The command record size is used by the plug-in and host to verify the contents of the command record.

The first command sent to a plug-in, for example, is called `pluginInitialize`. Its record is defined in EIPlugin.h as `PluginInitializeRec`. The command record size for this command would be `sizeof(PluginInitializeRec)`. When the plug-in is called by the host, it will examine the plug-in command and then call the appropriate function to process it. In this case, it would call a routine to process the `pluginInitialize` command.

While it is processing the command, the plug-in may need to request services from the host. If the plug-in needed to allocate memory, for example, it would send it a `hostAllocate` command. The record for this command is defined as `HostAllocateRec` in EIPlugin.h. The command record size for this command would be `sizeof(HostAllocateRec)`. The host's plug-in service routine would then call the appropriate function to process the memory allocation request. After processing the plug-in service command, the host would return to the plug-in which would then complete its initialization and return to the host.

Both the plug-in and host return result codes. The result codes are defined in EIPlugin.h. If the plug-in detects an error in a command record or cannot process a command for some reason, it should return an appropriate result code to the host. The host's plug-in service function may also return an error result code when it cannot process a plug-in request. When the plug-in receives an error result code from the host, it should immediately stop processing the current command and return the same result code back to the host. The host will act upon the returned result code. It may display the result code in an error message to the user. If the result code is not defined in EIPlugin.h, it will send a `pluginDescribe` to the plug-in. The plug-in will then return a description string to the host if it recognizes one of its own custom result codes.

Memory Allocation and Initialization

Memory

Since a plug-in acts as a mini program which is launched every time the host sends it a command, the plug-in needs a way to keep track of its information between commands. Two data records have been created to solve this problem. The data records are maintained by the host application. This allows the host to load and unload plug-ins from memory, and to send commands to a plug-in at any time. The plug-in may use the data records to keep track of the information the host sent to it during previous commands and to maintain its own variables and user editable parameters.

Even though the host is responsible for maintaining the data records, the plug-in defines their contents. The host application cannot directly modify the contents of the data records. The plug-in is responsible for allocating memory for the data records by sending `hostAllocate` service commands to the host. The host will respond by allocating the requested memory size and returning new pointers to the data records to the plug-in. In some cases, a plug-in may not know how much memory it will need before it begins to process data. A plug-in can allocate all available memory by sending a `hostFree` command to the host. It can then process its information and return the unused memory to the host by sending it another `hostAllocate` command.

Due to the requirements of Camera, the `hostAllocate` command may only be sent during certain plug-in commands. If the `hostAllocate` command is sent during other plug-in commands, the host will return an error result code to the plug-in.

The two data records are described as follows:

The ControlData Record This record is used to store information which must be saved with the project file for use the next time it is opened. It is also used to pass plug-in information from one host to another. Most commonly, the information stored in the control data record was entered by the user in the plug-in's interface dialog box. In the case of the Mesh plug-in, for example, the mesh type and density is entered by the user in the plug-in's interface dialog and then stored in the control data record. This control data record is saved with the project to which the plug-in has been added. When Camera is launched to render an image or animation, the control data record is written to Camera's control file. The same control record will be read by Camera and passed to the plug-in during the rendering process so that the correct mesh is generated at each frame of the animation. To keep the size of the project and Camera control files to a minimum, the control data record should be made as small as possible.

The PluginData Record This record is used to store information which the plug-in only needs during a series of commands within a single host application. Most commonly, the information stored in the plug-in data record is pre-computed values and information which the plug-in received in earlier commands for use in later commands. In the case of the Mesh plug-in, for example, the current frame index, and time is received in the `pluginModelSetup` command and stored in the plug-in data record for later use during the `pluginModelGenerate` command.

Initialization

Every plug-in command contains a plug-in information record. The host uses this record to send the size and pointer for the ControlData and PluginData records to the plug-in.

```
typedef struct {
  Ptr theControlData;          /* Pointer to the plug-in's ControlData record */
  LongInt theControlDataSize;    /* Size of the plug-in's ControlData record */
  Ptr thePluginData;             /* Pointer to the plug-in's PluginData record */
  LongInt thePluginDataSize;     /* Size of the plug-in's PluginData record */
  EIPluginRoutine theHostProc; /* The host application's command routine */
  Ptr theHostReference;        /* The host application's reference */
} PluginInfoRec, *PluginInfoPtr;
```

When a plug-in receives a command which will require it to access one or both of its data records, it should check to make sure that the data records have been correctly allocated and initialized. The first command a plug-in will receive from a host is the Initialize command. At this time, the PluginData record will not yet have been allocated so its size will be zero and its pointer will be NULL. If the plug-in has just been added to a new project file in ElectricImage, the ControlData record will also have a zero size and a NULL pointer. The plug-in should determine the correct size for the PluginData and/or the ControlData records and then request that the host allocate the records by sending a `hostAllocate` command. After the records are allocated, the plug-in should initialize their contents. When the next command is sent to the plug-in, the data records will have the same size and contents as they did when the plug-in completed the previous command. The pointers to the records may be different if the host has moved them in memory since the last plug-in command. The last command the plug-in will receive from the host is the Finish command. After the plug-in returns from this command, ElectricImage will save the contents of the ControlData record in the project file. It will have the same size and contents the next time the project file is opened and the Initialize command is sent to the plug-in.

Dynamic Memory Allocation

A plug-in should only attempt to allocate or change the size of its ControlData and PluginData records during the Initialize, Interface and Setup commands. If it sends a `hostAllocate` command to the host during any other commands, it should expect the command to fail with an error code. Some plug-ins may not know how much memory will be required to process data during the Generate command until it has received a Setup command. Most plug-ins should allocate temporary memory storage in their PluginData record. During the Initialize command, they will allocate a minimum sized PluginData record and initialize its contents. When the Setup command is received, the plug-in will determine the correct size for the PluginData record and send a `hostAllocate` command to the host.

Some plug-ins may not know what their memory requirements will be ahead of time. In this case, during the Setup command a plug-in should determine how much free memory their is by sending a `hostFree` command to the host. The plug-in can then send a `hostAllocate` command to request all free memory from the host, process the data and then send another `hostAllocate` command to return the unused memory. During the Generate command, the plug-in will use the contents of its PluginData record to generate its model or effect.

Frame to Frame Data Storage

Plug-ins which create complex simulations may need to keep state information from past frames in order to generate data during later frames. An example of this is a particle simulator plug-in which can leave trails of particles behind itself as its location is animated. There are no commands which allow a plug-in to determine its position at an earlier point in time. Even if such commands were provided, a plug-in may need to base its operation upon the position of its child groups which could have been created by other plug-ins. If the particle generator plug-in used the positions and motion of facets in its child groups to create new particles, it would be possible to link one copy of the plug-in to itself. This would allow the particles to emit new particles as they moved. There would be no way for the plug-in to know where the particles in its child group were located at an earlier frame without keeping some form of data storage from frame to frame. In ElectricImage, this can be done by simply keeping the state data in the PluginData record since it is kept for each plug-in while the project file is open.

Camera opens and closes each plug-in at each frame so the contents of the PluginData record are lost after every frame is rendered. In order for the plug-in to keep state data from frame to frame, it must create a temporary data file. The plug-in should create this file if it does not already exist. After each Setup command, the plug-in should write the contents of the PluginData record to the temporary file. During the next Setup command, the plug-in should open the temporary file and read the PluginData record back into memory. Camera will delete the temporary file automatically before it quits if the file's creator is "EIAR", its type is "EIDT", the file's name ends in ".temp" and it is located in the Temporary Items folder on the system disk.

The Model Plug-in Interface

Data Structure for Model Objects

The purpose of a model plug-in is to create new model data and/or to modify model data in its child groups. The model data is comprised of one or more groups. Each group contains a list of vertices, a list of facets and several information, shading, texture mapping and reflection mapping records. A facet is a structure containing between one and four vertices, and a color. A facet may represent a point, line, triangle or convex quadrangle. The facet vertices index the group's vertex list. All vertex indices are one based.

```
typedef struct {
  ARGB theColor;
  Integer theNumVertices;
  LongInt theVertices[4];
} Facet;
```

A vertex always contains a three dimensional position and may also contain a color, surface normal, texture position and motion blur position. Each group has a flag for each optional value to determine if it is to be computed and stored at each vertex.

```
typedef struct {
  ARGB theColor;
  DCoordinate theNormal;
  DCoordinate thePosition;
  DCoordinate theBlurPosition;
  DCoordinate theTexturePosition;
} Vertex;
```

Colors are represented by a union containing a 32 bit integer and four byte channel values for alpha, red, green and blue.

```
typedef union {
  ULongInt colorValue;
  struct {
    Byte a, r, g, b;      /* alpha, red, green, and blue */
  } argb;
} ARGB;
```

Positions and normals are represented as 64 bit double precision X, Y and Z coordinates.

```
typedef struct {
  Double x, y, z;          /* x, y and z coordinates */
} DCoordinate;
```

During the Generate command, a simple plug-in, such as Particle, can create model data by sending a series of `hostModelAddVertex` commands and then sending a series of `hostModelAddFacet` commands. A more complex plug-in, such as Mr. Nitro™, will loop for each of its child groups creating a new group with the `hostModelAddGroup` command, copying and modifying the vertex and facet data with the `hostModelGetVertex` and `hostModelGetFacet` commands, and then deleting the original child group with the `hostModelDeleteGroup` command. A plug-in can find out how many vertices and facets a child group contains by sending a `hostModelGetGroup` command. This command also gets the group's reference pointer, vertex flags, transformation matrix, rotation matrix and motion blur matrix.

Morph is an example of a model plug-in which generates new model data by referencing the model data in its child groups. Morph blends the vertex values of two child groups together by performing the following algorithm:

```
send hostModelGetGroup for group 1
send hostModelGetGroup for group 2

for i = 1 to (number of vertices) {
  send hostModelGetVertex for vertex i in group 1
  send hostModelGetVertex for vertex i in group 2
  calculate a new vertex by interpolating the group 1 and group 2 vertex values
  Send hostModelAddVertex to add the new vertex to the plug-in group
}
for i = 1 to (number of facets) {
  send hostModelGetFacet for facet i in group 1
  send hostModelAddFacet to copy the facet to the plug-in group
}
send hostModelDeleteGroup for group 1
send hostModelDeleteGroup for group 2
```

This algorithm works by first calculating and copying the vertex list and then copying the facet list.

The following algorithm would allow for a differing number of vertices in each group as long as the facet list was the same:

```
send hostModelGetGroup for group 1
send hostModelGetGroup for group 2

for i = 1 to (number of facets) {
  send hostModelGetFacet for facet i in group 1
  send hostModelGetFacet for facet i in group 2
  create a new facet record based upon the group 1 facet
  for j = 1 to (number of vertices of facet i) {
    get vertex j of facet i in group 1
    get vertex j of facet i in group 2
    calculate a new vertex by interpolating the group 1 and group 2 vertex values
    Send hostModelAddVertex to add the new vertex to the plug-in group
    Store the new vertex index in the new facet record
  }
  send hostModelAddFacet to add the new facet record to the plug-in group
}
send hostModelDeleteGroup for group 1
send hostModelDeleteGroup for group 2
```

This algorithm creates a larger number of vertices in the final model but is more robust when applied to models imported into ElectricImage.

Static and Dynamic Model Types

A model plug-in may generate either static or dynamic models. A static model is one which never changes. The Standard Shapes plug-in, for example, creates several types of basic three dimensional primitives whose shape never changes unless the user picks a new shape from the plug-in's user interface. A dynamic model may change over time or may change in relation to changes in its own plug-in group or one of its child groups. The Particle plug-in, for example, creates a stream of particles which changes at every frame.

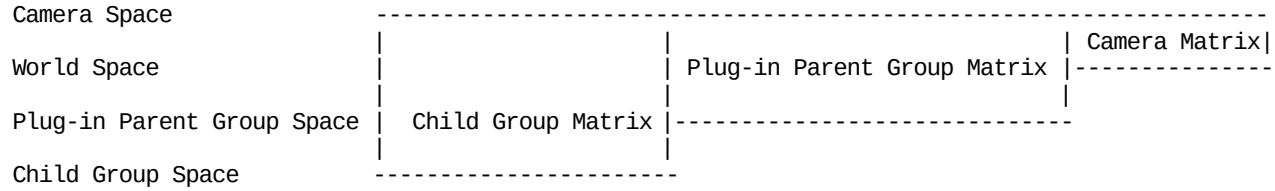
When a model plug-in receives a `pluginInterface` command and determines that it must re-generate its model data due to changes the user has made in its user interface dialog, it should send a `hostModelReset` command to the host. When the host receives this command, it will send a `pluginModelSetup` command and a `pluginModelGenerate` command to the plug-in.

A model plug-in should set the appropriate sensitivity flags in the `thePluginPermission` field of the `pluginInformation` command record. The sensitivity flags cause the host to send a `pluginModelSetup` command to the plug-in whenever the time, plug-in parent group or a plug-in child group changes. When a dynamic model plug-in receives a `pluginModelSetup` command, it will either send a `hostModelReset` command every time or only when it detects a change which will force it to re-generate its model data. A static model plug-in should disable its sensitivity flags and should not send a `hostModelReset` when it receives `pluginModelSetup` command.

Both static and dynamic plug-ins should always re-generate their model data when they receive a `pluginModelGenerate` command.

Matrix Operations

Model data is created and stored in camera space coordinates. Depending upon the needs of a plug-in, these coordinates may need to be converted into child group, plug-in parent group or world space coordinates. This can be done by using the matrices which are provided in the pluginModelSetup, and hostModelGetGroup commands. The pluginModelSetup commands provides the plug-in parent group and camera matrices. The hostModelGetGroup command supplies the child group matrices. The following chart shows the different spaces and how the different matrices are used to get from one to another:



Since all vertex positions are in camera space coordinates, a plug-in must perform matrix operations if it wishes to compute vertex positions in world, plug-in parent group or child spaces. If a plug-in needs to work in child group space, for example, it must first get a coordinate in camera space coordinates using hostModelGetVertex. It should then apply the inverse of the child group matrix to the vertex position. A new vertex position is computed and then the child group matrix is applied to it to bring it back into camera space coordinates. The vertex can then be added to the new group using the hostModelAddVertex command. Since the inverse of the child group matrix is not supplied as part of the hostModelGetGroup command, the plug-in is responsible for inverting the child group matrix. A matrix inversion routine is provided in PluginMatrix.c which is included with the EI Plug-in Developer Kit.

There is a set of three matrices provided with the camera, plug-in parent group and child group matrices. These are the position, rotation and motion blur matrices. The position matrices work as described above. The rotation matrices contain only the rotation components of the position matrices and are used to compute vertex normals. The motion blur matrices contain the position matrices from the previous frame and are used to compute vertex motion blur positions.

The Lens Flare Plug-in Interface

The Lens Flare Plug-in Interface was designed to allow a developer to generate and superimpose 2D graphical images over an image as it is being rendered. This allows the simulation of the interaction of light sources with the camera's lens elements. Lens flare plug-ins also have access to the host channel commands in ElectricImage. The plug-in can create animation channels for any of its user interface values. This gives the user the ability to apply ElectricImage's advanced animation features to the plug-in's editable values.

The Setup Phase

The pluginFlareSetup command contains the information a lens flare plug-in needs to draw its lens flare effect during the pluginFlareGenerate command. This information includes the image buffer resolution, light source location in space and in screen coordinates, and the current time and frame number. The plug-in should store this information in its PluginData record for use during the pluginFlareGenerate command.

The Generate Phase

The pluginFlareGenerate command contains information about the buffer into which the flare plug-in is to draw its effect. The host may choose to send the plug-in a pluginFlareGenerate command for each scanline in the buffer or it may send the plug-in a single pluginFlareGenerate command to generate the entire buffer. A flare plug-in should be developed to allow it to efficiently generate its effect on a scanline by scanline basis. This will enable the plug-in to work well regardless of how the host chooses to generate the flare buffer. Camera currently generates the lens flares on a scanline basis. ElectricImage currently generates the entire buffer in a single pass. The following structures are used to define the flare buffer:

```
typedef struct {
  LongInt top, left, bottom, right;
  LongInt width, height;
} L_Rect;

typedef struct {         /* The flare buffer contains pixels of type FlareColor (0..16384) */
  Ptr bufferBase;          /* The buffer's base address */
  LongInt bufferRowBytes;/* The number of bytes in a single scanline of the buffer */
  L_Rect bufferRect;       /* The enclosing rectangle of the buffer */
  Integer bufferDepth;     /* The bits per pixel of the flare buffer (8 or 48)  */
  LongInt bufferIndex;     /* The index color of flare rings for 8 bit wire frame buffers  */
} FlareBuffer;
```

The FlareBuffer's bufferDepth value can be either 8 or 48 bits. Other bit depths may be added in the future but for now these are the only ones a flare plug-in needs to be able to draw into. When the buffer is 48 bits deep, each pixel contains 16 bit red, green and blue channels. The following record defines the structure of a 48 bit flare buffer pixel:

```
typedef struct {
  UInteger red, green, blue;   /* 0..16384 */
} FlareColor, *FlareColorPtr;
```

The host program begins by initializing the buffer to zero (black) and then sends a `pluginFlareGenerate` command to each of its flare plug-ins. Each flare plug-in should calculate a color value and sum it with the contents of the flare buffer at every pixel. The values in the 16 bit channels range from 0 (black) to 16384 (white). The plug-in should limit each channel's value to 16384 to prevent overflow. After calling each flare plug-in, the host will sum the buffer's contents with the rendered image.

When the buffer is 8 bits per pixel, the flare plug-in should draw an outline of its effect into the buffer. The plug-in should draw the outline with the `bufferIndex` specified in the `FlareBuffer` record. In this mode, the host simply initializes the flare buffer to zero and the flare plug-in replaces the pixel values in the buffer with the `bufferIndex` value as it draws its outline. 8 bit flare buffers are only used in ElectricImage to provide a quick preview for the size and position of the flare effect. The LensFlare plug-in generates a circle with the radius and location of each of its flare elements. The Glow plug-in generates a square with the size and location of its glow circle.

The Project Channel Interface

Plug-ins may have their own animation channels.

Plug-in channels are displayed in ElectricImage's project window following the parent light source's channels. A plug-in may create channels to animate the following data types:

```
#define channelLabel          'LABL'   /* A container label (contains other channels) */
#define channelEnd            'LEND'   /* The end of a container label */
#define channelReal          'REAL'   /* Real (floating point) data (8 byte Double) */
#define channelCRGB          'CRGB'   /* RGB 16 bit color data (6 byte RGBColor) */
#define channelARGB          'ARGB'   /* ARGB 16 bit color data (8 byte ARGBColor) */
#define channelCHSV          'CHSV'   /* HSV 16 bit color data (6 byte HSVColor) */
#define channelAHSV          'AHSV'   /* AHSV 16 bit color data (8 byte AHSVColor) */
#define channelInteger        'SINT'   /* Signed integer count data (4 byte LongInt) */
#define channelUnsigned      'UINT'   /* Unsigned integer count data (4 byte ULongInt) */
#define channelBoolean        'BOOL'   /* Boolean flag data (2 byte Boolean) */
#define channelUnsignedReal  'UREL'   /* Unsigned Real (floating point) data (8 byte Double) */
#define channelCoordinate    'CORD'   /* 3 Axis Coordinate data (3 * 8 byte Double)  */
```

The channelLabel and channelEnd types are used to define container channels. The Plug-in creates a channel container for each of its sub elements. This helps to visually organize the channel data when a large number of animation channels is required. All of the animation channels created between the channelLabel and channelEnd types are nested inside a hierarchical block. The user can open this block to display the sub-channels by clicking on the channelLabel name in the project window.

The plug-in creates its channels when the host sends it a pluginInitializeChannel command. The plug-in sends a hostAddChannel for each of its animation channels. The plug-in sends the host the name, type and default value of the animation channel. The host returns a channel reference to the plug-in. The plug-in saves this reference number in its PluginData record and uses it to later retrieve data values from the animation channel. The plug-in can delete some or all of its animation channels with the hostDeleteChannel and add new ones if the user changes parameters during the pluginInterface command. If a plug-in determines that its animation channels have become invalid for some reason, it can send a hostResetChannel command to cause the host to delete all of the plug-in's animation channels and send the plug-in another pluginInitializeChannel command.

The plug-in can send either a hostAddChannel or a hostInsertChannel command to add animation channels. The command hostAddChannel is used to during a complete rebuild of the animation channel hierarchy after the host calls pluginInitializeChannel or the plug-in calls hostResetChannel.
The hostInsertChannel command can be used to insert a new animation channel into an already exisiting channel hierarchy without rebuilding a completely new list. The main addition to the hostInsertChannel command is the parent reference which must be know in order to add a child to the list.

When the plug-in receives a pluginLoadChannel command, it should load its ControlData values at the specified frame number from each of its animation channels with the hostGetChannel command. ElectricImage sends a pluginLoadChannel command for every frame to be rendered and then writes the plug-in's ControlData record to Camera's control file. This makes the information contained in the plug-in's animation channels available to the plug-in when it is called from Camera.

After the host sends a `pluginInterface` command, it will send a `pluginUpdateChannel` command to the plug-in. The plug-in should respond by sending a `hostSetChannel` command for each of its animation channels (other than the `channelLabel` and `channelEnd` channels). The host will create a new key frame if the project window is set to time or key frame mode and the new value does not match the value in the animation channel at the current frame. In frame mode, ElectricImage will create a custom frame when a difference is detected in the channel value.

The host will send a `pluginFinishChannel` command to the plug-in if its animation channels have been removed. The plug-in should reset its channel reference numbers to indicate that the channels are no longer valid.

User Interface

The host will send a `pluginInterface` command to the plug-in when the user wishes to edit the plug-in's parameters. This command is sent by ElectricImage but not by Camera. ElectricImage sends this command when the plug-in is first added to the project or when the user clicks the Options button in the plug-in group or light source information window. The plug-in should respond to this command by displaying a standard Macintosh dialog box and accepting user input. If the plug-in needs to access its own resources, it should send a `hostOpenResource` command to the host. After this command is sent, it can access its dialog box template by calling the system's GetNewDialog command. The plug-in should only send a `hostCloseResource` command if it needs to open another file's resource fork.

The host may send a `pluginAbout` command to the plug-in in response to a request by the user for information about a plug-in. This command is not currently used by either ElectricImage or Camera. ElectricImage currently sends a `pluginInformation` command to get information strings from the plug-in when the user requests information.

During time consuming processing, a plug-in should send `hostStatus` commands at least four times a second. This command allows the host to display the plug-in's current task and progress. It also allows the user to abort the plug-in at any time.

Exception Handling

Each plug-in and host command returns an error result code. Several result codes are defined in EIPlugin.h as follows:

```
#define pluginNoErr         noErr/* Zero result means no error */
#define pluginMemoryErr     1    /* Cannot allocate enough memory */
#define pluginCommandErr      2    /* Unknown command received */
#define pluginParameterErr   3    /* Error discovered in passed parameter record */
#define pluginCancelErr     4    /* The user canceled or aborted the operation */
#define pluginSequenceErr   5    /* The command was received in the wrong sequence */
#define pluginResourceErr   6    /* A required resource could not be found */
#define pluginProtectionErr  7    /* The copy protection could not be verified*/
#define pluginFeatureErr     8    /* A requested feature or mode is not supported */
```

If a host command returns a non-zero error code to the plug-in, the plug-in should return the same error code back to the host. The host should report any errors to the user. The host will call the function `DescribePluginResult` to convert a result code into a description string. A plug-in can have its own custom error codes. If the code is not recognized by `DescribePluginResult` , it will be sent back to the plug-in in a `pluginDescribe` command. The plug-in should call the function `DescribePluginError` to convert the pre-defined result codes into a description string. The functions `DescribePluginResult` and `DescribePluginError` are defined in EIPlugin.c.

Software Protection

A plug-in can get the host application's serial number string by sending a `hostChallenge` command. The serial number is stored as a seven character Pascal string and begins with two ASCII characters followed by 5 numeric digits. It is up to the plug-in author to develop their own copy protection scheme based upon the host serial number. One method is to create an authorization string which is coded with the contents of the serial number string. In order to activate the plug-in the user must enter this string in a modal dialog box. The plug-in then stores the authorization string in its resource file and validates it during every generate command.

Reference

The Communication Process

Calls from the Host to the Plug-in

All calls made by the host to a plug-in are done through a single function of the plug-in. The function's declaration is as follows:

```
pascal LongInt Plug-in (OSType theCommand, LongInt theCommandRecSize, void* theCommandRec);
```

theCommand

The first parameter, theCommand, allows the plug-in to determine which operation the host is expecting it to perform. This parameter is a four character token. There is a unique token for each command the plug-in may accept from the host. Upon entry to the plug-in's main routine, the command token is examined and a sub function is then called based upon its value. The following sample code shows how a simple plug-in would handle an incoming command token:

```
pascal LongInt Plug-in (OSType theCommand, LongInt theCommandRecSize, void* theCommandRec)
{
  LongInt theError = noErr;

  if (theCommand == pluginInitialize)
    theError = Initialize(theCommandRecSize, theCommandRec)
  else if (theCommand == pluginInterface)
    theError = theInterface(theCommandRecSize, theCommandRec)
  else if (theCommand == pluginModelSetup)
    theError = ModelSetup(theCommandRecSize, theCommandRec)
  else if (theCommand == pluginModelGenerate)
    theError = ModelGenerate(theCommandRecSize, theCommandRec);
  else
    theError = pluginCommandErr;
  return theError;
} /* Main */
```

The plug-in command constants such as pluginInitialize are defined in the file EIPlugin.h.

When a command is passed to the plug-in which the plug-in does not support, the plug-in should return the result code pluginCommandErr to tell the host that the command could not be accepted by the plug-in. The host software should not report an error to the user in this case unless the requested command was essential to the operation of the host. The required command set may change in the future but for now only the Initialize and Generate commands are considered essential for correct plug-in operation. A plug-in author should try to support as many commands as possible for future compatibility and smooth plug-in operation.

theCommandRec

The theCommandRec is a generic pointer to a plug-in command record. Each command has its own record. A plug-in must examine the command token before it is able to examine the contents of the command's record. For example, here are two typical plug-in commands that the host will make to a plug-in along with their respective records:

(An entire listing of all the command and record types available, including these, can be found in 'EIPlugin.h')

```
#define pluginCheck 'CHEK'

typedef struct {
  PluginInfoRec theInfo;
  OSType theCommand;
} PluginCheckRec, *PluginCheckPtr;

#define pluginModelSetup 'SETM'

typedef struct {
  PluginInfoRec theInfo;
  LongInt theFrameIndex;
  Double theFrameTime;
  Double theFrameTimeDelta;
  DMatrix4 theCameraMatrix;
  DMatrix4 theCameraRotMatrix;
  DMatrix4 theCameraBlurMatrix;
  DMatrix4 theGroupMatrix;
  DMatrix4 theGroupRotMatrix;
  DMatrix4 theGroupBlurMatrix;
} PluginModelSetupRec, *PluginModelSetupPtr;
```

theCommandRecSize

This parameter is a long integer which contains the size of the record that theCommandRec points to. It is used to help validate the command's parameter record. The following sample shows how a plug-in would validate the parameters for a model setup command:

```
static LongInt DoModelSetup (LongInt theCommandRecSize, void* theCommandRec)
{
  LongInt theError = noErr;
  PluginModelSetupPtr theParameters = (PluginModelSetupPtr)theCommandRec;

  if (theCommandRecSize< sizeof(PluginModelSetupRec)) {
    theError = pluginParameterErr;
  else {
    /* Setup the model plug-in */
  }
  return theError;
}  /* DoModelSetup */
```

By comparing the parameterSize against the size of PluginModelSetupRec, the plug-in can determine if enough data was sent for the pluginModelSetup command.

The incoming record size is only checked to see if it is smaller than the size of the expected type. This means the Plug-in would allow records of a larger size than expected. In future versions of the host software, record definitions may be expanded to include other parameters and information. Since all parameters which are expected to return information are initialized to their defaults by the host, earlier version plug-ins should remain compatible with later versions of the host software.

Calls from the Plug-in to the Host

When the plug-in processes a command, it may need to request services from the host. These requests consist of commands to the host, just as the host sends commands to the Plug-in. These services cover a multitude of activities which include allocating memory, creating or deleting groups, adding vertex or facet data, and others. Most services typically require three steps to successfully complete a call to the host application.

(1) Initialize the command record

(2) Use `CallPluginRoutine` to send the command to the Host

(3) Extract the data which the Host has returned in the record, if any

Initialize the command record

The first thing a Plug-in must do before sending a command to the host is to initialize the command's data record. For example, say a Plug-in would like to have a block of memory for storing data that survives in between calls from the host. This would require sending a hostAllocate command to the host, since only the host application can allocate memory. A record of the type HostAllocateRec must be passed along with this command. The record's fields must be initialized before sending the command to the host. Take a look at the record definition below:

```
#define hostAllocate 'ALOC'

typedef struct {
  Ptr theReference;            /* Sends:  the private data used by host callbacks */
  Ptr theControlData;         /* Returns:  the new control parameter data location */
  LongInt theControlDataSize;  /* Sends:  the requested new control parameter data size */
  Ptr thePluginData;           /* Returns:the new plug-in private data location */
  LongInt thePluginDataSize;   /* Sends:    the requested new plug-in private data size */
} HostAllocateRec, *HostAllocatePtr;
```

The comments contained in the type definitions (found in 'EIPlugin.h' and the Reference section of this manual), describe the purpose of each field in the record.

The first field in this structure is theReference. This field is found in every host command record. It is a pointer defined by the host and passed to the plug-in in the plug-in command's theInfo record.

The fields theControlData and thePluginData will return information to the plug-in. They should be initialized to their current values from the plug-in command's theInfo record. This way, should the host fail and return an error code, these fields will still contain valid information.

The fields theControlDataSize and thePluginDataSize send information to the host. They should be initialized to the desired memory sizes of the control and plug-in data records respectively. The host will use this information to allocate, increase or decrease the size of the plug-in's permanent and temporary data records.

Use CallPluginRoutine to send the command to the Host

All host command calls are of the form:

```
pascal LongInt Host (OSType theCommand, LongInt theCommandRecSize, void* theCommandRec);
```

In this example, the call would look like this:

```
theError = theInfo.theHostProc(hostAllocate, sizeof(HostAllocateRec), &theHostAllocate);
```

The first parameter is the host command token. The second is the size of the host command record. The third is the address of the host command record.

The host's command function pointer is passed to the plug-in in the plug-in's theInfo record.

If the host returns an error code to the plug-in, the plug-in should abort its current command and return the error code back to the host. If pluginNoErr is returned (0), the plug-in can use the records referenced by theControlData and thePluginData. Error codes are defined in 'EIPlugin.h'. See section 'Exception Handling' for more information.

Extract the data which the Host has returned in the record, if any

If the call to the host returns pluginNoErr, the plug-in can use the fields in the command record which return information. In the case of the memory allocation example, the values in the theControlData and thePluginData fields would be stored in theInfo record and used by the plug-in during the current and later commands from the host. See section 'Memory Allocation and Initialization' for more information.

Summary

What follows is a procedure called ModelGenerate which summarizes and demonstrates the three steps described in this section. This procedure really does nothing useful since the pointer to the new group isn't used for anything, but it does illustrate the process. It would be called by the Plug-in's main routine whenever theCommand token it received was pluginModelGenerate. In the example below, the record of type HostModelAddGroupRec is initialized, the hostModelAddGroup command is sent to the host, and the pointer to this new group is retrieved from the record if pluginNoErr is returned.

```
static LongInt ModelGenerate (LongInt theCommandRecSize, void* theCommandRec)
{
  Ptr theNewGroup;
  PluginModelGeneratePtr thePluginModelGeneratePtr = (HostModelGeneratePtr)theCommandRec;
  HostModelAddGroupRec theHostModelAddGroup;
  LongInt theError = pluginNoErr;     /* Initialize theError variable */

  /* Check to see that plug-in command record is of the expected size */

  if (theCommandRecSize < sizeof(PluginModelGenerateRec))
    theError = pluginParameterErr;  /* If it's not the right size, return an error */
  else {

    /* Initialize the host command record's fields */

    theHostModelAddGroup.theReference =
      thePluginModelGeneratePtr->theInfo.theHostReference;
    theHostModelAddGroup.theGroupTemplate = nil;
    theHostModelAddGroup.theGroupVertexFlags = 0;
    theHostModelAddGroup.theGroup = nil;

    /* Make the call to the host */

    theError = thePluginModelGeneratePtr->theInfo.theHostProc(hostModelAddGroup,
    sizeof(HostModelAddGroupRec), &theHostModelAddGroup);

    /* If no errors occurred, extract the information that the Host 'returns' */

    if (theError == pluginNoErr )
      theNewGroup = theHostModelAddGroup.theGroup;
  }
  return theError;
}   /* AddGroup */
```

Memory Allocation and Initialization

The Plug-in Info Record

This record is included in every plug-in command record as `theInfo`. The host uses this record to pass the temporary plug-in data and permanent control data records to the plug-in. This record also includes the host's command function pointer and plug-in reference pointer. These values are used by the plug-in to send commands back to the host.

```
typedef struct {
  Ptr theControlData;
  LongInt theControlDataSize;
  Ptr thePluginData;
  LongInt thePluginDataSize;
  EIPluginRoutine theHostProc;
  Ptr theHostReference;
} PluginInfoRec, *PluginInfoPtr;
```

`theControlData`	Sends a pointer to the control data record, or nil if it has not yet been allocated.
`theControlDataSize`	Sends the size of the control data record if it has already been allocated. It sends zero if it has not yet been allocated.
`thePluginData`	Sends a pointer to the plug-in data record, or nil if it has not yet been allocated.
`thePluginDataSize`	Sends the size of the plug-in data record if it has already been allocated. It sends zero if it has not yet been allocated.
`theHostProc`	Sends a pointer to the host's command function. This function is called by the plug-in to send commands to the host.
`theHostReference`	Sends a reference value which is used by the host to identify the plug-in. This reference is passed back to the host when the plug-in sends it commands. The plug-in's reference is a private value defined by the host.

Calls from the Host to the Plug-in

PluginInitialize

This is the first command in the plug-in command sequence. A plug-in's activation is bounded by `pluginInitialize` and `pluginFinish` commands. While a plug-in is active, the host will maintain its control data and plug-in-data records.

The ElectricImage will also maintain the plug-in's control data record between activations. ElectricImage keeps each instance of a plug-in active while the project it has been added to is open.

Camera will activate each instance of a plug-in once for every frame (or sub-frame) it renders. If a Particle plug-in is rendered in an animation with shadows for example, it will be activated twice per frame: once for the shadow and once for the final shaded frame. Camera does not currently save the contents of a plug-in's control data record between activations. If a plug-in needs to be able to "remember" what occurred on an earlier frame, it must create its own temporary storage file on the computer's hard disk and access that file at every frame.

```
#define pluginInitialize 'INIT'

typedef struct {
  PluginInfoRec theInfo;
} PluginInitializeRec, *PluginInitializePtr;
```

`theInfo` Sends the PluginInfoRec. See ‘PluginInfoRec’ in ‘Memory Allocation and Initialization.’

Typical Activities

Memory Allocation

The control data and plug-in data records should be allocated during this call. The sizes of these records may need to be adjusted in later plug-commands but they should be at least allocated to size zero during the `pluginInitialize` command. The control data record will usually already be allocated with its contents from the plug-in's last activation or from the plug-in's default values. The plug-in must be able to handle the case when this record is empty (size zero). The plug-in should also have a way of examining the contents of the control data record to determine if it contains the correct information and belongs to the correct version of the plug-in.

Default Control Data Values and Resources

After allocating the control data and plug-in data records, the plug-in should initialize their contents with default values. During this initialization, the plug-in may need to access its own resources. To do this, the plug-in should send a `hostOpenResource` command to the host.

The plug-in may not know how much memory it needs to allocate during the `pluginInitialize` call. In this case, a plug-in should just validate and/or initialize the contents of the control data record and set the size of the plug-in data record to zero.

PluginFinish

This is the last command in the plug-in command sequence. It tells the plug-in that it is about to become deactivated.

```
#define pluginFinish 'FINI'

typedef struct {
  PluginInfoRec theInfo;
} PluginFinishRec, *PluginFinishPtr;
```

theInfo — Sends the PluginInfoRec. See 'PluginInfoRec' in 'Memory Allocation and Initialization.'

Typical Activities

The plug-in should release any resources and close any files it has opened during the earlier commands. It is not necessary for the plug-in to dispose of its control data and plug-in data records since they will be taken care of by the host. ElectricImage will save the contents of the control data record and dispose of the plug-in data record. Camera disposes of both records. Most plug-ins will not need to do anything during the pluginFinish command.

Calls from the Plug-in to the Host

HostAllocate

This command is used by the plug-in to increase or decrease the size of the control data and plug-in data records.

```
#define hostAllocate 'ALOC'

typedef struct {
  Ptr theReference;
  Ptr theControlData;
  LongInt theControlDataSize;
  Ptr thePluginData;
  LongInt thePluginDataSize;
} HostAllocateRec, *HostAllocatePtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plug-in command's theInfo record.
theControlData	Sends and returns pointers to memory allocated for the ControlData record. This field should always be initialized with its corresponding field in theInfo record (found in whatever record was passed to the plug-in during the current call from the host).
theControlDataSize	Sends the amount of memory needed for the ControlData record. This value is typically sizeof(ControlDataRec).
thePluginData	Sends and returns pointers to memory allocated for the PluginData record. This field should always be initialized with its corresponding field in theInfo record (found in whatever record was passed to the plug-in during the current call from the host).
thePluginDataSize	Sends the amount of memory needed for the PluginData record. This value is typically sizeof(PluginDataRec), but may vary if the plug-in's memory needs need to fluctuate.

Usage

A plug-in may allocate memory during the initialize, setup or interface commands. A plug-in may not allocate memory during the generate and other commands. A hostAllocate command will return an error code if called at the wrong time in the plug-in command sequence.

A plug-in may decide to send more than one hostAllocate command to the host during a single plug-in command. If the plug-in is unsure how much memory it will need to process a resource or data file, it may decide to allocate all of available memory (see the hostFree command) for the plug-in data record. It can then process the data and call hostAllocate again when it is done to set the plug-in data record to the size which was actually required. A plug-in should not allocate all available memory and then return to the host without giving some of it back. If it does, the host may have to quit if it runs out of memory.

HostFree

This command will return the largest block of free memory available to the plug-in. The plug-in can use this information when allocating the control data and plug-in data records. If the plug-in attempts to allocate more than the free memory, the hostAllocate command may fail and return an error code to the plug-in.

```
#define hostFree 'FREE'

typedef struct {
  Ptr theReference;
  LongInt theFreeSize;
 } HostFreeRec, *HostFreePtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plug-in command's theInfo record.
theFreeSize	Returns the largest block of free memory available to the plug-in.

Usage

This call should be used when the plug-in needs to allocate all or most of free memory. This should only be done if the plug-in is about to read in a data file or setup some data whose size is unknown ahead of time. The plug-in must give back the unused memory to the host before returning from the current plug-in command. If this is not done, the host may run out of memory and quit.

The free plug-in memory should not be confused with free application heap memory. In Camera, there may be many megabytes of free plug-in memory but only a few hundred kilobytes of heap memory. This is why allocation of memory in the host's application heap by the plug-in is strongly discouraged.

The Model Plug-in Interface

Calls from the Host to the Plug-in

PluginModelSetup

This command is sent to the model plug-in to allow it to prepare for a pluginModelGenerate command.

```
#define pluginModelSetup 'SETM'

typedef struct {
  PluginInfoRec theInfo;
  LongInt theFrameIndex;
  Double theFrameTime;
  Double theFrameTimeDelta;
  DMatrix4 theCameraMatrix;
  DMatrix4 theCameraRotMatrix;
  DMatrix4 theCameraBlurMatrix;
  DMatrix4 theGroupMatrix;
  DMatrix4 theGroupRotMatrix;
  DMatrix4 theGroupBlurMatrix;
} PluginModelSetupRec, *PluginModelSetupPtr;
```

theInfo	Sends the PluginInfoRec. See 'PluginInfoRec' in 'Memory Allocation and Initialization.'
theFrameIndex	Sends the frame number of the current frame in the animation. This number is usually in the range of frames in the animation. Most animations start at frame index zero and count upwards by one.
theFrameTime	Sends the time in seconds of the current frame in the animation. Most animations start at time zero and count upwards in 1/30 of a second increments.
theFrameTimeDelta	Sends the duration in seconds of each frame of the animation. A 30 frames per second animation would have a frame duration of 1/30 of a second or 0.033333.
theCameraMatrix	Sends the camera's transformation matrix.
theCameraRotMatrix	Sends the camera's rotation only matrix.
theCameraBlurMatrix	Sends the camera's motion blur transformation matrix (at the time of the previous frame).
theGroupMatrix	Sends the group's transformation matrix.
theGroupRotMatrix	Sends the group's rotation only matrix.
theGroupBlurMatrix	Sends the group's motion blur transformation matrix (at the time of the previous frame).

Typical Activities

If a plug-in requires any information from the pluginModelSetup command record during the following pluginModelGenerate call, it must save it in its plug-in data record.

Simple dynamic model plug-ins should send a hostModelReset command when they receive a pluginModelSetup command. More advanced dynamic model plug-ins may attempt to determine if a change has been made which will force them to re-generate their data before sending a hostModelReset command. Static model plug-ins should not send a hostModelReset command.

Some model plug-ins do not know how much memory they will need before generating their model data. These plug-ins should pre-generate their data during the pluginModelSetup command in order to avoid allocating memory during the pluginModelGenerate command. To do this, a plug-in can allocate all of available memory for its PluginData record, generate data based upon time or its child groups and then re-allocate the its PluginData record to the amount it actually used. During the pluginModelGenerate command it can create model data based upon the contents of its PluginData record.

Reference

PluginModelGINF

PluginModelGATR

PluginModelGBLR

PluginModelTMAP

PluginModelTATR

PluginModelBMAP

PluginModelBATR

PluginModelRMAP

These commands are sent by Camera to the model-plug-in before the pluginModelGenerate command. These commands give model plug-ins the opportunity to access and modify the contents of various attribute records for the plug-in group before generation of any model data. These commands have become obsolete because the plug-in can now access these records by sending commands to the host during the pluginModelGenerate command. They are described here because they are sent to the plug-in in the current version of Camera. Future versions of Camera will not send these commands so it is recommended that a plug-in simply return with a noErr error code when it receives one of them and not to rely upon receiving them for future compatibility.

```
#define pluginModelGINF'GINF' /* Set the values of the group's GINF record */
#define pluginModelGATR'GATR' /* Set the values of the group's GATR record */
#define pluginModelGBLR'GBLR' /* Set the values of the group's GBLR record */
#define pluginModelTMAP'TMAP' /* Set the values of the group's TMAP record */
#define pluginModelTATR'TATR' /* Set the values of the group's TATR record */
#define pluginModelBMAP'BMAP' /* Set the values of the group's BMAP record (second TMAP) */
#define pluginModelBATR'BATR' /* Set the values of the group's BATR record (second TATR) */
#define pluginModelRMAP'RMAP' /* Set the values of the group's RMAP record */
```

All of these commands have the same record structure. The record for pluginModelGINF is as follows:

```
typedef struct {
  PluginInfoRec theInfo;
  GINFPtr theGINF;
  Boolean theChangeFlag;
} PluginModelGINFRec, *PluginModelGINFPtr;
```

theInfo	Sends a PluginInfoRec. See ‘PluginInfoRec’ in ‘Memory Allocation and Initialization.’
theGINF	Sends a pointer to the appropriate group attribute record.
theChangeFlag	Returns a Boolean flag. The plug-in should set this flag to True if it has changed any values in the group attribute record.

Typical Activities

A plug-in should ignore these commands and return noErr to the host without accessing their contents. A plug-in should not rely upon receiving these commands because they will be removed from future versions of the ElectricImage™ Animation System.

PluginModelGenerate

This command is sent to the model plug-in following a pluginModelSetup command. It tells the plug-in to begin generating a new model using the settings in the model setup record.

```
#define pluginModelGenerate 'GENM'

typedef struct {
  PluginInfoRec theInfo;
  LongInt theChildGroups;
  ULongInt theVertexFlags;
} PluginModelGenerateRec, *PluginModelGeneratePtr;
```

theInfo	Sends a PluginInfoRec. See 'PluginInfoRec' in 'Memory Allocation and Initialization.'
theChildGroups	Sends the number of child groups linked to the plug-in's group. The plug-in may access its child groups to generate new model data.
theVertexFlags	Sends the vertex interpolation flags for the plug-in's group. These flags tell the plug-in that it should generate color, texture and/or normal vertex information. For more information of mapping flags, see Appendix B.

Typical Activities

When a model plug-in receives a generate command, it should generate model data based upon the settings in the model setup record. The plug-in can add groups, vertices and facets to the host to accomplish this. It can either produce the data algorithmically, or it may extract the information from its child groups if any were linked to it.

Model data generated by the plug-in during the previous generate command is automatically deleted by the host before it sends the new generate command to the plug-in. The plug-in simply generates new model data from scratch every time it receives a generate command.

Calls from the Plug-in to the Host

HostModelReset

This command tells the host to send a pluginModelGenerate command to the plug-in so that a new model is generated for the current frame. It is sent by the plug-in in response to a pluginModelSetup command or a change in the plug-in's user interface during a pluginInterface command.

```
#define hostModelReset 'RSET'

typedef struct {
  Ptr theReference;
} HostModelResetRec, *HostModelResetPtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plug-in command's theInfo record.

Usage

ElectricImage will respond to a pluginModelReset command by sending a pluginModelGenerate command to the plug-in. See the section 'Static and Dynamic Model Types' for details about when a model plug-in should send a pluginModelReset command.

HostModelGetGroup

The get group command is used to get information about a child group of a model plug-in. This command may only be sent to the host during a model generate command.

```
#define hostModelGetGroup 'GETG'

typedef struct {
  Ptr theReference;
  LongInt theGroupIndex;
  Ptr theGroup;
  ULongInt theGroupVertexFlags;
  LongInt theGroupVertices;
  LongInt theGroupFacets;
  DMatrix4 theGroupMatrix;
  DMatrix4 theGroupRotMatrix;
  DMatrix4 theGroupBlurMatrix;
} HostModelGetGroupRec, *HostModelGetGroupPtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plug-in command's theInfo record.
theGroupIndex	Sends the index value of the desired child group. This value is in the range from 1 to the number of child groups linked to the model plug-in. The number of child groups is found in the theChildGroups field of the PluginModelGenerateRec.
theGroup	Returns a reference for the requested child group. Once this group reference is retrieved, it can be used in other commands such as hostModelGetVertex and hostModelGetFacet to get more information about the child group. The group's reference is a private value defined by the host.
theGroupVertexFlags	Returns the vertex mapping flags for the requested child. These flags tell the plug-in if the child group contains color, texture and/or normal vertex information. For more information of mapping flags, see Appendix B.
theGroupVertices	Returns the number of vertices of the child group.
theGroupFacets	Returns the number of facets of the child group.
theGroupMatrix	Returns the child group's transformation matrix.
theGroupRotMatrix	Returns the child group's rotation only matrix.
theGroupBlurMatrix	Returns the child group's motion blur transformation matrix (at the time of the previous frame).

Usage

To begin retrieving data from a child group, a group pointer must be retrieved using this command. The children are indexed from 1 to the number of child groups linked to the plug-in. The number of children is found in the theChildGroups field of the pluginModelGenerateRec. A model plug-in such as Mr. Nitro will loop for each child group which is attached to it. The pointer to each child is retrieved with the hostModelGetGroup command and then each vertex and facet is accessed with hostModelGetVertex and hostModelGetFacet respectively. Mr. Nitro uses the vertex and facet information to create explosion fragments which are sent back to the host with hostModelAddVertex and hostModelAddFacet commands.

HostModelSetGroup

This command is used to copy a group's position, shading and texture attribute records into the current group. This command may only be sent to the host during a model generate command.

```
#define hostModelSetGroup 'CPYG'

typedef struct {
  Ptr theReference;
  Ptr theGroupTemplate;
  LongInt theGroupVertexFlags;
} HostModelSetGroupRec, *HostModelSetGroupPtr;
```

`theReference`	Sends the plug-in's reference. This reference can be found in `theHostReference` of the plug-in command's `theInfo` record.
`theGroupTemplate`	Sends a reference to the group which will be used as a template. All of the texture and shading information will be copied from the template group to the current group.
`theGroupVertexFlags`	Sends the mapping flags for the current group. For more information of mapping flags, see Appendix B.

Usage

When a model generate command is sent to a plug-in, the host has already created a work group for the plug-in to add vertex and facet information to. The position, shading and texture attributes of the default work group are copied from the plug-in's group which the user setup in ElectricImage. If the plug-in wishes instead to assign attributes from one of the plug-in's child groups, it may send a `hostSetGroup` command to the host passing the pointer to the child group in `theGroupTemplate`. The plug-in will usually pass the value it received from the `pluginModelGenerate` command record in `theGroupVertexFlags`. If the plug-in cannot process one or more of the vertex options, it may choose to disable their flags in this field.

HostModelAddGroup

This command is used to add a new work group to the model plug-in. This command may only be sent to the host during a model generate command.

```
#define hostModelAddGroup 'ADDG'

typedef struct {
  Ptr theReference;
  Ptr theGroupTemplate;
  LongInt theGroupVertexFlags;
  Ptr theGroup;
} HostModelAddGroupRec, *HostModelAddGroupPtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plug-in command's theInfo record.
theGroupTemplate	Sends a pointer to a group which will be used as the template. All of the texture and shading information will be copied from this group to the new group. A pointer value of null will cause the plug-in's own attributes to be copied to the new group.
theGroupVertexFlags	Sends the mapping flags for the new group. For more information of mapping flags, see Appendix B.
theGroup	Returns a reference to the new group. The group's reference is a private value defined by the host.

Usage

All data added to the host by the model plug-in with the hostModelAddVertex and hostModelAddFacet commands will be placed into the current work group. Most plug-ins which generate their own model geometry and are not dependent upon the geometry in their child groups will not need to use this command. They can use the default work group which was created by the host before the PluginModelGenerate command was sent. Plug-ins which do depend upon the geometry in their child groups may wish to generate one new work group for each of their child groups. This will allow the shading and texture information in the child groups to be preserved in the new work groups. Mr. Nitro, for example, uses the hostModelAddGroup command before fragmenting each of its child groups. The fragments keep the texture and shading information which was applied to the child groups.

Reference

HostModelDeleteGroup

This command is used to tell the host to hide a child group linked to the plug-in. This command may only be sent to the host during a model generate command.

```
#define hostModelDeleteGroup 'DELG'

typedef struct {
  Ptr theReference;
  Ptr theGroup;
} HostModelDeleteGroupRec, *HostModelDeleteGroupPtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plug-in command's theInfo record.
theGroup	Sends the reference of the plug-in's child group which should be hidden and not rendered in Camera or displayed in ElectricImage's drawing windows.

Usage

Child groups linked to a plug-in are usually rendered in Camera and displayed in the drawing windows of Electric Image. If a plug-in wishes to use the child groups to generate its own model data, it may wish to hide their original model data. In the case of Mr. Nitro, for example, the child groups are fragmented with a simulated explosion. If Mr. Nitro did not hide its child groups, both the original model data and the exploded fragments would be rendered and displayed. Instead, Mr. Nitro sends a hostModelDeleteGroup command for each child group so that they do not appear in the display or rendered animation. The child groups are not actually deleted. They will still be linked to the plug-in and will show up again during the next pluginModelGenerate command.

HostModelGetVertex

This command is used to get vertex information from a child group of the model plug-in. This command may only be sent to the host during a model generate command.

```
#define hostModelGetVertex 'GETV'

typedef struct {
  Ptr theReference;
  Ptr theGroup;
  LongInt theVertexIndex;
  ARGB theColor;
  DCoordinate theNormal;
  DCoordinate thePosition;
  DCoordinate theBlurPosition;
  DCoordinate theTexturePosition;
} HostModelGetVertexRec, *HostModelGetVertexPtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plug-in command's theInfo record.
theGroup	Sends the reference of the plug-in's child group from which the vertex is to be retrieved. The group reference is found by using the hostGetGroup command.
theVertexIndex	Sends the index number of the desired vertex. The vertices are ordered from 1 to the number of vertices. The number of vertices in a plug-in child group is found by using the hostGetGroup command.
theColor	Returns the ARGB vertex color. The color value is only valid if the reference group's color vertex flag is enabled. (For more on the relationship of the color attributes at the facet, vertex, or group level, see Appendix B.)
theNormal	Returns the 3D vertex normal in camera space. The normal value is only valid if the reference group's normal vertex flag is enabled.
thePosition	Returns the 3D vertex position in camera space. The position value is always valid.
theBlurPosition	Returns the 3D vertex motion blur position in camera space (at the time of the previous frame). The blur position value is only valid if the reference group's blur position vertex flag is enabled.
theTexturePosition	Returns the 3D vertex texture position. The texture position value is only valid if the reference group's texture vertex flag is enabled.

Usage

Model plug-ins which need to access the model data of their child groups can get information about a vertex by using the hostModelGetVertex command. The plug-in must put the index number of the desired vertex into theVertexIndex field of the HostModelGetVertexRec. The number of vertices in a child group is found in theGroupVertices field of the HostModelGetGroupRec. A plug-in can loop from 1 to theGroupVertices sending a hostModelGetVertex and a hostModelAddVertex command to copy all of the vertices from the child group to the current work group. In Mr. Nitro, the vertices are accessed one facet at a time by using the contents of theVertices in the HostGetFacetRec.

HostModelAddVertex

This command is used to add a new vertex to the plug-in's current work group. This command may only be sent to the host during a model generate command.

```
#define hostModelAddVertex 'ADDV'
typedef struct {
  Ptr theReference;
  DCoordinate thePosition;
  ARGB theColor;
  DCoordinate theNormal;
  DCoordinate theBlurPosition;
  DCoordinate theTexturePosition;
  LongInt theIndex;
} HostModelAddVertexRec, *HostModelAddVertexPtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plug-in command's theInfo record
thePosition	Sends the 3D vertex position in camera space. The position value is always used by the host.
theColor	Sends the ARGB vertex color. The color value is only used by the host if the current work group's color vertex flag is enabled. (For more on the relationship of the color attributes at the facet, vertex, or group level, see Appendix B.)
theNormal	Sends the 3D vertex normal in camera space. The normal value is only used by the host if the current work group's normal vertex flag is enabled.
theBlurPosition	Sends the 3D vertex motion blur position in camera space (at the time of the previous frame). The blur position value is only used by the host if the current work group's blur position vertex flag is enabled.
theTexturePosition	Sends the 3D vertex texture position. The texture position value is only used by the host if the current work group's texture vertex flag is enabled.
theIndex	Returns the index of the newly created vertex.

Usage

The model-plug-in is responsible for setting all of vertex information fields which are enabled in the current work group's vertex flags. The host uses the vertex information to create a new vertex and returns its index to the plug-in in theIndex. The new vertex index is passed back to the host in the hostAddFacet command. Because the vertex indexes used in the hostAddFacet command reference vertices which have already been created, the plug-in must add the vertices before it adds the facets which reference them.

Reference

HostModelGetFacet

This command is used to get information about a facet in a child group of a model plug-in. This command may only be sent to the host during a model generate command.

```
#define hostModelGetFacet 'GETF'

typedef struct {
  Ptr theReference;
  Ptr theGroup;
  LongInt theFacetIndex;
  ARGB theColor;
  Integer theNumVertices;
  Vertices theVertices;
} HostModelGetFacetRec, *HostModelGetFacetPtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plug-in command's theInfo record.
theGroup	Sends the reference of the plug-in's child group from which the facet is to be retrieved. The group reference is found by using the hostGetGroup command.
theFacetIndex	Sends the index number of the desired facet. The facets are ordered from 1 to the number of facets. The number of facets in a plug-in child group is found by using the hostGetGroup command.
theColor	Returns the color of the facet. (For more on the relationship of the color attributes at the facet, vertex, or group level, see Appendix B.)
theNumVertices	Returns the number of vertices contained in this facet. This number is in the range of 1 to 4 and represent a point, line, triangle or quadrangle respectively.
theVertices	Returns an array which contains the indexes to the vertices which define the facet. These indexes can be used to retrieve the vertices themselves. Only the array elements in the range of 0 to theNumVertices - 1 contain valid information.

Usage

Model plug-ins which need to access the model data of their child groups can get information about a facet by using the hostModelGetFacet command. The pug-in must put the index number of the desired vertex into theFacetIndex field of the HostModelGetFacetRec. The number of facets in a child group is found in theGroupFacets field of the HostModelGetGroupRec. A plug-in can loop from 1 to theGroupFacets sending a hostModelGetFacet and a hostModelAddFacet command to copy all of the facets from the child group to the current work group. To get information about the vertices of a facet, a plug-in send a hostModelGetVertex command for each vertex index in the facet's theVertices array. Mr. Nitro does this for each facet before breaking up a facet into smaller fragments.

HostModelAddFacet

This command is used to add a new facet to the current work group. This command may only be sent to the host during a model generate command.

```
#define hostModelAddFacet 'ADDF'

typedef struct {
  Ptr theReference;
  ARGB theColor;
  Integer theNumVertices;
  Vertices theVertices;
} HostModelAddFacetRec, *HostModelAddFacetPtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plug-in command's theInfo record.
theColor	Sends the color of the new facet. (For more on the relationship of the color attributes at the facet, vertex, or group level, see Appendix B.)
theNumVertices	Sends the number of vertices contained in this facet. This number is in the range of 1 to 4 and represent a point, line, triangle or quadrangle respectively.
theVertices	Sends an array which contains the indexes to the vertices which define the facet. These indexes were returned by the host in the hostModelAddVertex command when the vertices were added to the current work group. The host only sets the array elements in the range of 0 to theNumVertices - 1.

Usage

In order to add new model data to a plug-in's current work group, hostModelAddVertex and hostModelAddFacet commands must be sent to the host. The hostModelAddVertex adds a new vertex to the current work group and returns an index value which the plug-in should put into the facet's theVertices array. After each of the facet's vertices (1 for a point, 2 for a line, 3 for a triangle and 4 for a quadrangle) have been created, the facet itself can then be added to the current work group with the hostModelAddFacet command.

Reference

HostModelGINF

HostModelGATR

HostModelGBLR

HostModelTMAP

HostModelTATR

HostModelBMAP

HostModelBATR

HostModelRMAP

These commands allow a plug-in to access and modify the contents of various group attribute records.

```
#define hostModelGINF 'GINF' /* Set the values of the group's GINF record */
#define hostModelGATR 'GATR' /* Set the values of the group's GATR record */
#define hostModelGBLR 'GBLR' /* Set the values of the group's GBLR record */
#define hostModelTMAP 'TMAP' /* Set the values of the group's TMAP record */
#define hostModelTATR 'TATR' /* Set the values of the group's TATR record */
#define hostModelBMAP 'BMAP' /* Set the values of the group's BMAP record (second TMAP) */
#define hostModelBATR 'BATR' /* Set the values of the group's BATR record (second TATR) */
#define hostModelRMAP 'RMAP' /* Set the values of the group's RMAP record */
```

All of these commands have the same record structure. The record for `hostModelGINF` is as follows:

```
typedef struct {
       Ptr theReference;
       Ptr theGroup;
       GINFPtr theGINF;
} HostModelGINFRec, *HostModelGINFPtr;
```

`theReference`	Sends the plug-in's reference. This reference can be found in `theHostReference` of the plug-in command's `theInfo` record.
`theGroup`	Sends the reference of the plug-in's child group from which the requested attribute record is to be retrieved. The group reference is found by using the `hostGetGroup` command.
`GINFPtr`	Sends a pointer to the requested group attribute record.

Typical Activities

A model plug-in can send these commands to get attribute records from any of its child groups, its own plug-in group or any of the groups it has created. The plug-in may change the contents of its own groups or any of its created groups but should not attempt to change the contents of one of its child groups. This command should only be sent for the purpose of modifying the contents of plug-in group attribute records during the `pluginModelGenerate` command and only before the first vertex is added to requested group. The contents of the groups GINF, GATR, GBLR, TMAP, TATR, BMAP, BATR and RMAP records are described in FACTStuff.h.

It is rare that a model plug-in will need to call one of these commands since it is usually left up to the user to apply textures and shading attributes to a group in ElectricImage. Most model plug-ins allow the group attribute records to be set automatically when creating new groups by supplying a group template reference. With access to these commands, a model plug-in could modify the contents of these records over time. One example would be a plug-in which rotated the position of a reflection map over time.

The Lens Flare Plug-in Interface

Calls from the Host to the Plug-in

PluginFlareSetup

This call is sent to the flare plug-in to allow it to prepare for a flare generate command.

```
#define pluginFlareSetup 'SETF'

typedef struct {
  PluginInfoRec theInfo;
  LongInt theFrameIndex;
  Double theFrameTime;
  Double theFrameTimeDelta;
  L_Rect theFrameRect;
  Double theFrameXScale;
  Double theFrameYScale;
  Double theFrameXCenter;
  Double theFrameYCenter;
  Double theBlurShutter;
  DMatrix4 theCameraMatrix;
  DMatrix4 theCameraRotMatrix;
  DMatrix4 theCameraBlurMatrix;
  DCoordinate theLightWorldPosition;
  DCoordinate theLightBlurWorldPosition;
  DCoordinate theLightFramePosition;
  DCoordinate theLightBlurFramePosition;
  DCoordinate theLightDirection;
  DCoordinate theLightBlurDirection;
  ARGB theLightColor;
  Double theLightIntensity;
  Double theLightDropoff;
  Double theLightWorldRadius;
  Double theLightFrameRadius;
  Double theLightOuterAngle;
  Double theLightInnerAngle;
  Double theLightSoftExponent;
} PluginFlareSetupRec, *PluginFlareSetupPtr;
```

`theInfo`	Sends a PluginInfoRec. See ‘PluginInfoRec’ in ‘Memory Allocation and Initialization.’
`theFrameIndex`	Sends the frame number of the current frame in the animation. This number is usually in the range of frames in the animation. Most animations start at frame index zero and count upwards by one.
`theFrameTime`	Sends the time in seconds of the current frame in the animation. Most animations start at time zero and count upwards in 1/30 of a second increments.
`theFrameTimeDelta`	Sends the duration in seconds of each frame of the animation. A 30 frames per second animation would have a frame duration of 1/30 of a second or 0.033333.
`theFrameRect`	Sends the frame’s rectangle. The rectangle's units are in pixels. The lens flare will be expected to generate its effects into sub-rectangles of the frame's rectangle.
`theFrameXScale`	Sends the frame’s X scale value. This value is usually equal to the width of the frame's rectangle.

theFrameYScale	Sends the frame's Y scale value. This value is usually equal to the height of the frame's rectangle unless the frame was rendered with a non-square pixel aspect ratio.
theFrameXCenter	Sends the frame's X center location. This value is usually equal to the horizontal center of the frame's rectangle. If the flare plug-in generates a lens flare, the flare's elements would be calculated by the plug-in to pivot around the frame's center point.
theFrameYCenter	Sends the frame's Y center location. This value is usually equal to the vertical center of the frame's rectangle. If the flare plug-in generates a lens flare, the flare's elements would be calculated by the plug-in to pivot around the frame's center point.
theBlurShutter	Sends the motion blur shutter angle. This value is usually a number between 0 and 1. The default shutter angle is 0.5. The shutter angle is the percentage of time which the camera's shutter is open for each frame. A moving object would appear to travel half of its distance from the previous frame to the current frame in the rendered image with a 0.5 shutter angle.
theCameraMatrix	Sends the camera's transformation matrix.
theCameraRotMatrix	Sends the camera's rotation only matrix.
theCameraBlurMatrix	Sends the camera's motion blur transformation matrix (at the time of the previous frame).
theLightWorldPosition	Sends the light's 3D position in the world space coordinates. The camera's transformation matrix would have to be applied to this position to find the light's position in camera space coordinates.
theLightBlurWorldPosition	Sends the light's motion blur position in world space coordinates (at the time of the previous frame).
theLightFramePosition	Sends the light's position in frame coordinates. The host computes this point by applying perspective, scale and offsets to the light's position in camera space coordinates. The result is a 2D frame position. Only the X and Y coordinates are used by the flare plug-in when generating its effects into the frame buffer. The Z is computed by Camera as the distance from the camera to the light source in camera space coordinates.
theLightBlurFramePosition	Sends the light's motion blur position in frame coordinates (at the time of the previous frame).
theLightDirection	Sends the light's direction normal in camera space.
theLightBlurDirection	Sends the light's motion blur direction normal in camera space (at the time of the previous frame).
theLightColor	Sends the light's color. Only the R, G and B components should be used by the flare plug-in to generate its effects.
theLightIntensity	Sends the light's intensity. This value is 1.0 by default. The light intensity can be used with the light's drop-off distance to simulate more realistic lighting effects.

`theLightDropoff`	Sends the light's drop-off distance. This is the distance over which the light's should diminish towards black.
`theLightWorldRadius`	Sends the light's radius in world units.
`theLightFrameRadius`	Sends the light's radius in frame pixels.
`theLightOuterAngle`	Sends the spotlight outer cone angle. Both the inner and outer spotlight cone angles are 0.0 for non-spotlights.
`theLightInnerAngle`	Sends the spotlight inner cone angle. Both the inner and outer spotlight cone angles are 0.0 for non-spotlights.
`theLightSoftExponent`	Sends the spotlight transition exponent. This controls how fast the spotlight drops off between the inner and outer cones.

Typical Activities

When the flare plug-in receives a setup command, it should store any information it needs into its plug-in data record, allocate memory and setup its lookup tables. The LensFlare plug-in, for example, will read in its flare element resources and generate look-up tables based upon the frame X and Y scales. When it receives the flare generate command, it will use the lookup tables to rapidly calculate element colors. Flare plug-ins must be able to draw into any sub-rectangle of the frame's rectangle from individual scanlines up to the entire rectangle. Camera is currently designed to draw individual scanlines from the top of the frame (low Y value) to the bottom of the frame (high Y value).

PluginFlareGenerate

This command is sent to the flare plug-in following a pluginFlareSetup command. It tells the plug-in to begin generating its effect into a flare pixel buffer using the settings in the flare setup record.

```
#define pluginFlareGenerate 'GENF'

typedef struct {
  PluginInfoRec theInfo;
  FlareBuffer theFlareBuffer;
} PluginFlareGenerateRec, *PluginFlareGeneratePtr;
```

theInfo	Sends a PluginInfoRec. See 'PluginInfoRec' in 'Memory Allocation and Initialization.'
theFlareBuffer	Sends the flare pixel buffer. The flare plug-in will generate its effect into this buffer.

Typical Activities

The host will send one or more pluginFlareGenerate commands to the flare plug-in following a pluginFlareSetup command. Camera will send a generate command for each scanline in the frame rectangle. Flare plug-ins must be written to generate scanline effects efficiently. The use of look-up tables is a good way to do this. These tables can be pre computed from resolution independent data in the plug-ins resources during the flare setup command.

The FlareBuffer Record

```
typedef struct {
  Ptr bufferBase;
  LongInt bufferRowBytes;
  L_Rect bufferRect;
  Integer bufferDepth;
  LongInt bufferIndex;
} FlareBuffer, *FlareBufferPtr;
```

`bufferBase`	The flare pixel buffer's base address.
`bufferRowBytes`	The flare pixel buffer's scanline byte offset.
`bufferRect`	The flare pixel buffer's rectangle. This will always be a sub-rectangle of the frame's rectangle in the flare setup record.
`bufferDepth`	The flare pixel buffer's bit depth. Only 8 and 48 bit buffers are currently supported.
`bufferIndex`	The flare pixel buffer's drawing color index. This is only used to draw flare previews into 8 bit flare pixel buffers.

The `FlareBuffer` record's `bufferDepth` field can support any bit depth buffer, however only 8 and 48 bit buffers are currently implemented by ElectricImage and Camera.

When a flare plug-in is told to draw into an 8 bit buffer, it should only draw a preview sketch of its effect. In the case of LensFlare, the rings are drawn as circles with the requested `bufferIndex` color. Support of the 8 bit flare buffer is optional but recommend. The flare plug-in should try to draw some fast and simple representation of its effect such as its outline. ElectricImage will copy the 8 bit buffer into its camera window which will help users to place light sources to achieve their desired result.

Camera will only use 48 bit buffers. The buffer is composed of 16 bit red, green and blue component pixels. Camera begins by initializing the buffer to black (0, 0, 0) and then calls each flare plug-in for each scanline of the buffer. The flare plug-ins are responsible for generating pixel data for their effect and adding it into the accumulation buffer. The pixel values for white are (16383, 16383, 16383). The plug-in must clip the pixel components to 16383 to prevent overflow.

The Project Channel Interface

The project command interface provides a set of commands for the host and plug-in to allow plug-ins to add channels to ElectricImage's project window. These commands are not available in Camera.

Calls from the Host to the Plug-in

PluginInitializeChannel

The host sends this command to the plug-in to tell it to add animation channels to its object the project window.

```
#define pluginInitializeChannel 'ICHN'

typedef struct {
  PluginInfoRec theInfo;
  LongInt theStartFrame;
  LongInt theEndFrame;
  Double theStartTime;
  Double theFrameTime;
} PluginInitializeChannelRec, *PluginInitializeChannelPtr;
```

theInfo	Sends a PluginInfoRec. See 'PluginInfoRec' in 'Memory Allocation and Initialization.'
theStartFrame	Sends the animation's starting frame index.
theEndFrame	Sends the animation's ending frame index.
theStartTime	Sends the time in seconds of the animation's starting frame. Most animations start at time zero and count upwards in 1/30 of a second increments.
theFrameTime	Sends the duration in seconds of each frame of the animation. A 30 frames per second animation would have a frame duration of 1/30 of a second or 0.033333.

Typical Activities

ElectricImage will send this command to a plug-in after it has been added to a project or when the project is opened. When a plug-in receives this command, it should throw away any channel information it currently has and send a `hostAddChannel` command for each variable it wishes to make available for user animation. When a plug-in is first added to the project, ElectricImage will create new channels when it receives `hostAddChannel` commands. When a project is opened, it will use the `hostAddChannel` commands to validate the channels which were stored in the project file. This makes it possible to update a plug-in to a newer version without loosing animation information in existing project files. The LensFlare plug-in creates channels for each flare element. These channels control the flare element's visibility, size, color and intensity.

PluginFinishChannel

This command tells a plug-in that its channels are no longer valid and should no longer be accessed.

```
#define pluginFinishChannel 'FCHN'

typedef struct {
  PluginInfoRec theInfo;
} PluginFinishChannelRec, *PluginFinishChannelPtr;
```

theInfo — Sends a PluginInfoRec. See ‘PluginInfoRec’ in ‘Memory Allocation and Initialization.’

Typical Activities

Most plug-ins do not need to do anything when they receive this command. This command was provided for later expansion of the channel command set. ElectricImage currently initiates all channel activity so the plug-in should not be accessing its animation channels unless it is told to do so by the host.

PluginUpdateChannel

This command tells a plug-in to update the values in a range of frames of its animation channels.

```
#define pluginUpdateChannel 'UCHN'

typedef struct {
  PluginInfoRec theInfo;
  LongInt theStartFrame;
  LongInt theEndFrame;
  Double theStartTime;
  Double theFrameTime;
} PluginUpdateChannelRec, *PluginUpdateChannelPtr;
```

theInfo	Sends a PluginInfoRec. See 'PluginInfoRec' in 'Memory Allocation and Initialization.'
theStartFrame	Sends the animation's starting frame index.
theEndFrame	Sends the animation's ending frame index.
theStartTime	Sends the time in seconds of the animation's starting frame. Most animations start at time zero and count upwards in 1/30 of a second increments.
theFrameTime	Sends the duration in seconds of each frame of the animation. A 30 frames per second animation would have a frame duration of 1/30 of a second or 0.033333.

Typical Activities

This command is currently used by ElectricImage to update the contents of the plug-in's animation channels after the user has accessed the plug-in's interface dialog box. The plug-in should respond by sending a `hostSetChannel` command for each of its animation channels. The host compares each plug-in channel value to the one stored in the project at the current frame. If the values are different, either a key frame or custom frame is created at the current frame with the new value depending upon whether the project window is in key frame or frame mode.

When LensFlare receives this command it will update its channels with the variables stored in its control data record. The same control data record variables can be edited in the user interface dialog and loaded with the `pluginLoadChannel` command.

PluginLoadChannel

This command tells a plug-in to load the current frame's plug-in channel values into the plug-in's control data record.

```
#define pluginLoadChannel 'LCHN'

typedef struct {
  PluginInfoRec theInfo;
  LongInt theStartFrame;
  LongInt theEndFrame;
  Double theStartTime;
  Double theFrameTime;
  LongInt theFrame;
} PluginLoadChannelRec, *PluginLoadChannelPtr;
```

theInfo	Sends a PluginInfoRec. See 'PluginInfoRec' in 'Memory Allocation and Initialization.'
theStartFrame	Sends the animation's starting frame index.
theEndFrame	Sends the animation's ending frame index.
theStartTime	Sends the time in seconds of the animation's starting frame. Most animations start at time zero and count upwards in 1/30 of a second increments.
theFrameTime	Sends the duration in seconds of each frame of the animation. A 30 frames per second animation would have a frame duration of 1/30 of a second or 0.033333.
theFrame	Sends the current frame index.

Typical Activities

When the plug-in receives this command, it should send a `hostGetChannel` command to the host for each of its animation channels. This will update the values in the plug-in's control data record to match those in the animation channels at the current frame. ElectricImage sends this command to the plug-in before sending a `pluginInterface` command. It also sends this command to the plug-in as each frame is sent to Camera in the control file. Because the contents of the plug-in's control data record are written to the control file at each frame, the plug-in is able to have access to the animated values before rendering.

Calls from the Plug-in to the Host

HostResetChannel

This command tells the host to delete all of the plug-in's animation channels.

```
#define hostResetChannel 'RCHN'

typedef struct {
  Ptr theReference;
} HostResetChannelRec, *HostResetChannelPtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plug-in command's theInfo record.

Usage

If a plug-in determines that its channel data is no-longer valid for some reason (the control data record contained invalid data, for example) it should send a hostResetChannel command to the host. After the current plug-in command returns to the host, the host should send a pluginInitializeChannel to the plug-in so that new channels can be created. A plug-in should not attempt to access its animation channels after it has sent a hostResetChannel command.

HostAddChannel

This command tells the host to add a new plug-in animation channel.

```
#define hostAddChannel 'ACHN'

typedef struct {
  Ptr theReference;
  LongInt theChannelID;
  OSType theChannelType;
  Str255 theChannelName;
  Ptr theChannelDefault;
  Ptr theChannelMinimum;
  Ptr theChannelMaximum;
  Ptr theChannelBefore;
  Ptr theChannelReference;
} HostAddChannelRec, *HostAddChannelPtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plug-in command's theInfo record.
theChannelID	Sends the animation channel's unique ID. The ID is defined by the plug-in and will help to validate the channel in later commands.
theChannelType	Sends the animation channel's four character type. The channel types are defined in 'EIPlugin.h'. The channel type is used by the host to display and animate the channel values and also to validate the channel in later commands.
theChannelName	Sends the animation channel's name. The name is used by ElectricImage when displaying the channel in the project window.
theChannelDefault	Sends the pointer to the animation channel's default value. This value will be used by the host to initialize the contents of the animation channel. Initialize to 'nil' if the channel does not contain an animated value.
theChannelMinimum	Sends a pointer to the animation channel's minimum allowable value. The host should prevent the animated value from dropping below the minimum value. Initialize to 'nil' if the channel does not contain an animated value or if there is no minimum value for the channel (the host will automatically limit channel values to the largest possible ranges for their numeric type).
theChannelMaximum	Sends a pointer to the animation channel's maximum allowable value. The host should prevent the animated value from rising above the maximum value. Initialize to 'nil' if the channel does not contain an animated value or if there is no maximum value for the channel (the host will automatically limit channel values to the largest possible ranges for their numeric type).
theChannelBefore	Sends a reference to a previously added channel. The channel being added will appear immediately following the referenced channel. Sending null in this parameter will cause the new channel to be added to the end of the list.
theChannelReference	Returns a reference to the added channel. This reference should be stored for use in later commands.

Usage

This command is sent to the host to add each of the plug-in's animation channels when a plug-in receives a pluginInitializeChannel command. When ElectricImage adds a plug-in to the project, it will add the plug-in's animation channels following the plug-in's object in the project list. When a project is opened, ElectricImage will send the pluginInitializeChannel command to the plug-in and then validate the ID, and data type of each channel it receives against the channels which were stored in the project file. If the channels all match, it will keep the original channels. If the channels do not match, it will use the newly created channels and their default values. This will allow plug-in versions to be updated while still maintaining compatibility with older project files.

Animation channels may be added to either store data values or to form containers of other animation channels. This allows the plug-in to create a more structured display of its animation channels in ElectricImage's project window. Animation channels can contain a variety of data types:

```
/* Channel data types            Description                    Type     Size */

#define channelLabel         'LABL'   /* A channel container label    none       0 */
#define channelEnd           'LEND'   /* The end of a container label   none       0 */
#define channelReal         'REAL'   /* Real (floating point) data     Double     8 */
#define channelCRGB         'CRGB'   /* RGB 16 bit color data          RGBColor   6 */
#define channelARGB         'ARGB'   /* ARGB 16 bit color data       ARGBColor  8 */
#define channelCHSV         'CHSV'   /* HSV 16 bit color data          HSVColor   6 */
#define channelAHSV         'AHSV'   /* AHSV 16 bit color data       AHSVColor  8 */
#define channelInteger       'SINT'   /* Signed integer count data    LongInt  4 */
#define channelUnsigned     'UINT'   /* Unsigned integer count data  LongInt  4 */
#define channelBoolean       'BOOL'   /* Boolean flag data              Boolean  2 */
#define channelUnsignedReal  'UREL'   /* Unsigned Real (float) data     Double     8 */
#define channelCoordinate    'CORD'   /* Coordinate (float) data        3*Double  24 */
```

Each channel is added with a default value. The default is used to initialize the value of all of the frames in the channel when it is first added to the project. A channel may optionally have minimum and maximum value limits. The host will prevent the animated channel's values from exceeding these limits when they are provided by the plug-in.

Every channel should have a unique ID number. The plug-in is responsible for creating and maintaining the channel IDs. Both the plug-in and the host can use the channel ID to validate the contents of a particular animation channel.

HostInsertChannel

This command tells the host to insert a new plug-in animation channel into an already existing channel hierarchy.

```
#define hostInsertChannel          'PCHN'

typedef struct {
        Ptr theReference;
        LongInt theChannelID;
        OSType theChannelType;
        Str255 theChannelName;
        Ptr theChannelDefault;
        Ptr theChannelMinimum;
        Ptr theChannelMaximum;
        Ptr theChannelBefore;
        Ptr theChannelParent;
        Ptr theChannelReference;
} HostInsertChannelRec, *HostInsertChannelPtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plug-in command's theInfo record.
theChannelID	Sends the animation channel's unique ID. The ID is defined by the plug-in and will help to validate the channel in later commands.
theChannelType	Sends the animation channel's four character type. The channel types are defined in 'EIPlugin.h'. The channel type is used by the host to display and animate the channel values and also to validate the channel in later commands.
theChannelName	Sends the animation channel's name. The name is used by ElectricImage when displaying the channel in the project window.
theChannelDefault	Sends the pointer to the animation channel's default value. This value will be used by the host to initialize the contents of the animation channel. Initialize to 'nil' if the channel does not contain an animated value.
theChannelMinimum	Sends a pointer to the animation channel's minimum allowable value. The host should prevent the animated value from dropping below the minimum value. Initialize to 'nil' if the channel does not contain an animated value or if there is no minimum value for the channel (the host will automatically limit channel values to the largest possible ranges for their numeric type).

theChannelMaximum	Sends a pointer to the animation channel's maximum allowable value. The host should prevent the animated value from rising above the maximum value. Initialize to 'nil' if the channel does not contain an animated value or if there is no maximum value for the channel (the host will automatically limit channel values to the largest possible ranges for their numeric type).
theChannelBefore	Sends a reference to a existing sibling channel. The channel being added will appear immediately following the referenced channel. Sending null in this parameter will cause the new channel to be added to the end of the list of siblings.
theChannelParent	Sends the reference to an existing parent channel. Sending null in this parameter will cause the new channel to be added at the end of the complete hierarchy listing without any parents.

Usage

This command is sent to the host to insert a new animation channel into an already existing channel hierarchy. This command should be called during the pluginInterface command when the plug-in has determined that a new channel is needed. An existing animation channel hierarchy must exist. To build a new animation channel hierarchy, use hostAddChannel commands in response to a pluginInitialize channel.

See HostAddChannel command for additional information.

Reference

HostDeleteChannel

This command tells the host to delete one of the plug-in's animation channels.

```
#define hostDeleteChannel 'DCHN'

typedef struct {
  Ptr theReference;
  Ptr theChannelReference;
  LongInt  theChannelID;
} HostDeleteChannelRec, *HostDeleteChannelPtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plug-in command's theInfo record.
theChannelReference	Sends the animation channel's reference for validation This reference was returned to the plug-in in the hostAddChannel command.
theChannelID	Sends the animation channel's unique ID for validation.

Usage

If the plug-in needs to make changes in its animation channel list, it must first delete the unwanted channels. The plug-in does this by sending a hostDeleteChannel command for each animation channel to be deleted. The plug-in must pass the channel reference it received from the host in the HostAddChannelRec record. It must also send the channel's unique ID number for channel validation. When the hostDeleteChannel is called on an animation channel which is just a container (no animation values) then all of its offspring animation channels are deleted as well.

The LensFlare plug-in deletes all of its flare element channels when the user changes the lens flare type in the plug-in's user interface. It then adds the animation channels for the new flare type's elements to the project list. The LensFlare plug-in does not delete the animation channels which control the overall lens flare effects because they do not change by lens flare type.

HostGetChannel

This command tells the host to get the value for a plug-in animation channel at a particular frame number.

```
#define hostGetChannel 'GCHN'

typedef struct {
  Ptr theReference;
  Ptr theChannelReference;
  LongInt  theChannelID;
  OSType theChannelType;
  LongInt theChannelFrame;
  Ptr theChannelData;
} HostGetChannelRec, *HostGetChannelPtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plug-in command's theInfo record.
theChannelReference	Sends the animation channel's reference for validation. This reference was returned to the plug-in in the hostAddChannel command.
theChannelID	Returns the animation channel's unique ID for validation.
theChannelType	Returns the animation channel's data type for validation.
theChannelFrame	Sends the requested frame number. The host will return the animation channel's value at this frame.
theChannelData	Returns a pointer to the requested animation channel's value at the requested frame number.

Description

This message is passed to the host to extract the value of a particular frame in an animation channel. Although a pointer to this value is returned, it is not a pointer to the actual frame's value. It is a pointer to a copy of this value. The plug-in must use a hostSetChannel command to actually change the frame value's contents.

Usage

The plug-in should send a hostGetChannel command for each of its animation channels in response to a pluginLoadChannel command. The plug-in should validate the channel's ID and data type and then copy the channel data into its control data record. The plug-in must not attempt to change the contents of theChannelData. To change the value of an animation channel, the plug-in should use the hostSetChannel command.

Reference

HostSetChannel

This command tells the host to change the value of a plug-in animation channel at a particular frame.

```
#define hostSetChannel 'SCHN'

typedef struct {
  Ptr theReference;
  Ptr theChannelReference;
  LongInt  theChannelID;
  OSType theChannelType;
  LongInt theChannelFrame;
  Ptr theChannelData;
} HostSetChannelRec, *HostSetChannelPtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plug-in command's theInfo record.
theChannelReference	Sends the animation channel's reference for validation. This reference was returned to the plug-in in the hostAddChannel command.
theChannelID	Sends the animation channel's unique ID for validation.
theChannelType	Sends the animation channel's data type for validation.
theChannelFrame	Sends the requested frame number. The host will set the animation channel's value at this frame.
theChannelData	Sends a pointer to the new channel frame value for the requested frame number.

Usage

When the host sends a pluginUpdateChannel command to the plug-in, the plug-in should respond by sending a hostSetChannel command for each of its animation channels and for each frame in the specified range. ElectricImage validates the channel's ID and data type and then compares the new channel value to the current channel value. If the two values are different a new key frame is created if the project is set to time or key frame mode. A custom frame is created if the project is in frame mode. ElectricImage sends a pluginUpdateChannel for the current frame to the plug-in following a pluginInterface command.

HostGetTimingInfo

This command tells the host to return the most recent animatiom timing information.

```
#define hostGetTimingInfo          'TINF'

typedef struct {
        Ptr theReference;
        double theTimingStart;
        double TheTimingCurrent;
        double theTimingFPS;
        LongInt theTimingInterlace;
        LongInt theTimingTotal;
        LongInt theTimingSkip;
        LongInt theRenderFrameStart;
        LongInt theRenderFrameStop;
HostGetTimingInfoRec, *HostGetTimingInfoPtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plugin-in command's theInfo record.
theTimingStart	Send the system's start time of the animation. The default is 0.0. But the user may change it by dragging the start time icon in the time bar.
theTimingCurrent	Sends the system's current time.
theTimingFPS	Sends the system's frame-per-second factor. Video rates is usually 30 frames-per-second, or 60 fields-per-second (fields are treated as frames in EI). Film rates are 24 frames-per-second. The host may have a different FPS count, so the plug-in should call this command as frequently as possible.
theTimingInterlace	Sends the system's interlace option. 0 is no interlace, while 1 specifieds interlacing.
theTimingTotal	Sends the system's total number of frames. The stop time of an animation is computed as follows stopTime = theTimingStart + (theTimingTotal * theTimingFPS)
theTimingSkip	Sends the system's skip frames of an animation. The user may instruct the animation to skip frames during rendering. This is 0 if all frames are rendered, or positive if frames need to be skipped. For example, a value of 3 only renders every fourth frame.

theRenderFrameStart	Sends the system's frame start of a rendering. A value of 0 means that the animation starts at theTimingStart.
theRenderFrameStop	Sends the system's frame stop of a rendering. A value of theTimingTotal means that the animation stops at the end of the animation. NOTE: If theRenderFrameStart and theRenderFrameStop are the same value, look at theTimingCurrent for the current time to be rendered (single frame only).

Usage

This command is valuable to inform the plugin about the current state of the system's animation timing track. Usually, the plug-in might want to use this command before generating any model data to check how much needs to be done.

HostGetChannelInfo

This command tells the host to return information about a animation channel

```
#define hostGetChannelInfo          'GCHI'

typedef struct {
        Ptr theReference;
        Ptr theChannelOwner;
        Ptr theChannelReference;

        LongInt theChannelType;
        LongInt theChannelFilter;
        LongInt theChannelVelocity;
        LongInt theChannelKeyTotal;
        LongInt theChannelFlag;
        Ptr     theChannelMinimum;
        Ptr     theChannelMaximum;
} HostGetChannelInfoRec, *HostGetChannelInfoPtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plugin-in command's theInfo record.
theChannelOwner	Sends the pointer to the owner of the animation channel. A null represents the plug-in's reference. Reserved for future expansion.
theChannelReference	Sends a reference to specify the channel.
theChannelType	Returns the type of channel. See hostAddChannel and hostInsertChannel for more information about the types.
theChannelFilter	Returns the type of channel value filter. This will tell you about the range of the values this channel can handle.
theChannelVelocity	Returns 0 if no velocity computation are performed on this channel.
theChannelKeyTotal	Returns the number of key frames available in the channel.
theChannelFlag	Returns a flags field for the animation channel. Reserved for future expansion.
theChannelMinimum	Returns the a pointer to the animation channel's minimum allowable value. A null is returned if all possible values for this type (returned in theChannelType) are allowed.
theChannelMaximum	Returns the a pointer to the animation channel's maximum allowable value. A null is returned if all possible values for this type (returned in theChannelType) are allowed.

Usage

This command is send to the host to return important information about an animation channel. Remember that the host has the same privileges of modifing an animation channel than the plug-in.
The host as well as the plug-in can add, delete, or modify the channel's keys and state. Before the plug-in attemps to issue the more advanced channel commands, it should send this command to update its information about a key channel.

The filter type returns the following values:

```
#define PLUGIN_CHANNEL_FILTER_NONE              0
#define PLUGIN_CHANNEL_FILTER_BYTE_SIGNED       1
#define PLUGIN_CHANNEL_FILTER_BYTE_UNSIGNED2
#define PLUGIN_CHANNEL_FILTER_INTEGER_SIGNED    3
#define PLUGIN_CHANNEL_FILTER_INTEGER_UNSIGNED  4
#define PLUGIN_CHANNEL_FILTER_FLOAT_SIGNED      5
#define PLUGIN_CHANNEL_FILTER_FLOAT_UNSIGNED    6
#define PLUGIN_CHANNEL_FILTER_BOOLEAN           7
```

PLUGIN_CHANNEL_FILTER_NONE	allows all key values
PLUGIN_CHANNEL_FILTER_BYTE_SIGNED	allows key values between -127 and +128
PLUGIN_CHANNEL_FILTER_BYTE_UNSIGNED	allows key values between 0 and 255
PLUGIN_CHANNEL_FILTER_INTEGER_SIGNED	allows key values between -2,147,483,648 and +2,147,483,647
PLUGIN_CHANNEL_FILTER_INTEGER_UNSIGNED	allows key values between 0 and +2,147,483,647
PLUGIN_CHANNEL_FILTER_FLOAT_SIGNED	allows key values between -infinity and +infinity
PLUGIN_CHANNEL_FILTER_FLOAT_UNSIGNED	allows key values between 0 and +infinity
PLUGIN_CHANNEL_FILTER_BOOLEAN	allows key values of 0 (true) and 1 (false)

HostAddChannelKey

This command tells the host to add a key frame to an existing animation channel

```
#define hostAddChannelKey          'ACHK'

typedef struct {
        Ptr theReference;
        Ptr theChannelOwner;
        Ptr theChannelReference;
        LongInt theChannelID;
        OSType theDataType;
        Ptr theData;
        double theKeyTime;
} HostAddChannelKeyRec, *HostAddChannelKeyPtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plugin-in command's theInfo record.
theChannelOwner	Sends the pointer to the owner of the animation channel. A null represents the plug-in's reference. Reserved for future expansion.
theChannelReference	Sends a reference to specify the channel.
theChannelID	Sends a plug-in maintained ID for the channel.
theDataType	Sends the type of data. Along with theData, this represents the default value for that key.
theData	Sends the pointer to the default data.
theKeyTime	Sends the time of the key. When creating keys, care should be taken so that the plug-in does not create keys at the same time. The host will return an error if this happens.

Usage

This command allows the plug-in to maintain the key frames of an existing channel. The plug-in should send the same default data type as the channel can accept. Use the hostGetChannelInfo command to inquire about the channel's capabilities. If the plug-in tries to create a key frame with the same time as an existing one, the host will return an error.

HostDeleteChannelKey

This command tells the host to delete a key frame from an existing animation channel.

```
#define hostDeleteChannelKey 'DCHK'

typedef struct {
        Ptr theReference;
        Ptr theChannelOwner;
        Ptr theChannelReference;
        LongInt theChannelID;
        LongInt theKeyIndex;
} HostDelChannelKeyRec, *HostDelChannelKeyPtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plugin-in command's theInfo record.
theChannelOwner	Sends the pointer to the owner of the animation channel. A null represents the plug-in's reference. Reserved for future expansion.
theChannelReference	Sends a reference to specify the channel.
theChannelID	Sends a plug-in maintained ID for the channel.
theKeyIndex	Sends the index or types of keyframes to be deleted. A non-zero positive index indicates which key frame is to be deleted. The first key frame starts with 1 (one). A zero index indicates ALL key frames are to be deleted. A value of -1 for index indicates only SELECTED key frames are to be deleted.

Usage

This command allows the plug-in to maintain the key frames of an existing channel. When specifying a key frame you can use the following constants to delete more than one key frame at a time.

```
#define PLUGIN_CHANNEL_SPECIFY_ALL              0
#define PLUGIN_CHANNEL_SPECIFY_SELECTED         -1
```

PLUGIN_CHANNEL_SPECIFY_ALL	allows the plugin to delete all key frames in a channel.
PLUGIN_CHANNEL_SPECIFY_SELECTED	allows the plugin to delete only selected key frames in a channel.

Any positive number indicates the indexed key frame to be deleted.
The first key frame always starts with 1.
Use the hostGetChannelInfo to get the total number of key frames in an animation channel.

HostGetChannelKey

This command tells the host to get a value from the specified key frame(s) of an existing animation channel

```
#define hostGetChannelKey          'GCHK'

typedef struct {
        Ptr theReference;
        Ptr theChannelOwner;
        Ptr theChannelReference;
        LongInt theChannelID;
        OSType theDataType;
        Ptr theData;
        LongInt theKeyPart;
        LongInt theKeyIndex;
} HostGetChannelKeyRec, *HostGetChannelKeyPtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plugin-in command's theInfo record.
theChannelOwner	Sends the pointer to the owner of the animation channel. A null represents the plug-in's reference. Reserved for future expansion.
theChannelReference	Sends a reference to specify the channel.
theChannelID	Sends a plug-in maintained ID for the channel.
theDataType	Returns the type of data.
theData	Returns a pointer to the data. Make sure the plug-in checks the data type to extract the data with the correct structure.
theKeyPart	Sends the part of the key frame to be evaluated.
theKeyIndex	Sends the index of the key frame to be inquired. A non-zero positive index indicates the key frame. The first key frame starts with 1 (one).

Usage

This command is send to the host to get the value of the specified key frame of an existing animation channel. By using the following constants, the plug-in can extract different information:

```
#define PLUGIN_CHANNEL_MODE_PROCESS_VALUE       10
#define PLUGIN_CHANNEL_MODE_PROCESS_TIME        11
```

PLUGIN_CHANNEL_MODE_PROCESS_VALUE	extracts the value of the key frame. Check the data type returned for the correct type.
PLUGIN_CHANNEL_MODE_PROCESS_TIME	extracts the time of the key frame. The value is always a signed double (channelReal).

Before attempting to extract the key value, call the hostGetChannelInfo command to get the total number of key frames. If the plug-in attemps to extract information of a non-existing key frame, an error will be returned.

HostSetChannelKey

This command tells the host to modify a value of specified key frame(s) of an existing animation channel.

```
#define hostSetChannelKey          'SCHK'

typedef struct {
        Ptr theReference;
        Ptr theChannelOwner;
        Ptr theChannelReference;
        LongInt theChannelID;
        OSType theDataType;
        Ptr theData;
        LongInt theKeyPart;
        LongInt theKeySubPart;
        LongInt theKeyIndex;
} HostSetChannelKeyRec, *HostSetChannelKeyPtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plugin-in command's theInfo record.
theChannelOwner	Sends the pointer to the owner of the animation channel. A null represents the plug-in's reference.
	Reserved for future expansion.
theChannelReference	Sends a reference to specify the channel.
theChannelID	Sends a plug-in maintained ID for the channel.
theDataType	Returns the type of data.
theData	Returns a pointer to the data. Make sure the plug-in checks the data type to extract the data with the correct structure.
theKeyPart	Sends the part of the key frame to be evaluated.
theKeySubPart	Sends the component ID of the key frame value to be processed by an operator type.
theKeyIndex	Sends the index of the key frame to be modified. A non-zero positive index indicates the key frame. The first key frame starts with 1 (one).

Usage

This command allows the plug-in to modify the key frame values. There are several modes in which the plug-in can modify a value of a key.

```
#define PLUGIN_CHANNEL_MODE_ADD_SYSTEM          1
#define PLUGIN_CHANNEL_MODE_ADD_KEY             2
#define PLUGIN_CHANNEL_MODE_ADD_FRAME           3
```

PLUGIN_CHANNEL_MODE_ADD_SYSTEM	Adds a key frame value at the current time by using the system's mode. This means that key frames will either be modified or created, or that custom frames will either be modified or created, which mode the system is currently in. theKeyIndex is ignored in this case.
PLUGIN_CHANNEL_MODE_ADD_KEY	Adds a key frame value by either modifying an existing key frame at the current time, or creating a new key frame with this value at the current time. theKeyIndex is ignored in this case.
PLUGIN_CHANNEL_MODE_ADD_FRAME	Adds a key frame value by either modifying an existing custom frame at the current time, or creating a new custom frame with this value at the current time. theKeyIndex is ignored in this case.

```
#define PLUGIN_CHANNEL_MODE_PROCESS_VALUE       10
#define PLUGIN_CHANNEL_MODE_PROCESS_TIME        11
```

PLUGIN_CHANNEL_MODE_PROCESS_VALUE	modifies the value of the key frame. Check the data type returned for the correct type.
PLUGIN_CHANNEL_MODE_PROCESS_TIME	modifers the time of the key frame. The value specified is always a signed double (channelReal).

In addition, the plug-in call specifies which key frames are to be modified by using the following commands.

```
#define PLUGIN_CHANNEL_SPECIFY_ALL              0
#define PLUGIN_CHANNEL_SPECIFY_SELECTED         -1
```

PLUGIN_CHANNEL_SPECIFY_ALL	allows the plugin to modify all key frames in a channel.
PLUGIN_CHANNEL_SPECIFY_SELECTED	allows the plugin to modify only selected key frames in a channel.

Any positive number indicates the indexed key frame to be modified.
The first key frame always starts with 1.

In addition to the standard data types, the following type can be used to modify existing values:

```
#define channelPascalString  'PSTR'
```

This is a pascal string type with which the user can use relative operators on existing values. These are:

@		Current value.
@	<number>	Multiply the current value by a factor (implied).
@*	<number>	Multiply the current value by a factor.
@+	<number>	Add a number to the current value.
@-	<number>	Subtract a number from the current value.
@/	<number>	Divide the current value by a divisor.
@^	<number>	Apply a power to the current value.
		Adding a percent sign (%) after the number indicates a percentage of that number (<number> / 100.0).
@#		Compute the absolute of the current value ([value]).
@!		Negate the current value (-value).
@$		Reciprocal of the current value (1 / value).

All operators can be pipelined with the use of semi-colons (;). For example,

```
@*2;@+12;@!
```

will take the current value, multiply by 2, add 12 to the result, and, finally, negate the previous result.

Modifying the key times can lead two or more key frames to share the same time, which is impossible. You cannot exist in two places at once. Some of the modes allow more than one key frame to be modified. Care should be taken to make sure that this will not happen. In case it does happen, an error will be returned and the operation will not be performed. To extract all key frame times, use hostGetChannelKey and hostGetChannelInformation.

User Interface
Calls from the Host to the Plug-in

PluginInformation

This command is used by the host to get information about the plug-in.

```
#define pluginInformation 'INFO'

typedef struct {
  PluginInfoRec theInfo;
  OSType thePluginType;
  LongInt thePluginVersion;
  Str255 thePluginName;
  Str255 thePluginCopyright;
  Str255 thePluginAuthor;
  Str255 thePluginDescription;
  LongInt thePluginPermission;
} PluginInformationRec, *PluginInformationPtr;
```

theInfo	Sends a PluginInfoRec. See ‘PluginInfoRec’ in ‘Memory Allocation and Initialization.’
thePluginType	Returns the plug-in’s unique type.
thePluginVersion	Returns the plug-in’s version number.
thePluginName	Returns the plug-in’s name.
thePluginCopyright	Returns the plug-in’s copyright notice.
thePluginAuthor	Returns the plug-in’s author name, address, phone or any other desired data.
thePluginDecription	Returns a string which describes the plug-in’s capabilities.
thePluginPermission	Returns a flag which determines whether EI has permission to automatically copy this plug-in to slave machines for rendering.

Typical Activities

If the host needs to get information about a plug-in in response to request by the user or for other purposes, it should send a `pluginInformation` command to the plug-in. The plug-in should simply fill in all of the fields in the `PluginInformationRec` and return to the host. The "|" character is used to delineate a new line in the information strings. The host is responsible for displaying the information strings correctly to the user.

The thePluginPermission field is used to tell the host if it has permission to copy the plug-in to another machine for network rendering. This flag was provided to allow plug-in authors to decide if they wish to grant users a single or multi-machine license. This field also contains flags which indicate whether the plug-in is sensitive to changes in time, the plug-in's parent group or any of the plug-in's child groups. These flags are defined as follows:

```
#define pluginSensitivityFlag       0 /* The plug-in supports the sensitivity flags */
#define pluginTimeSensitivityFlag   1 /* Sensitive to changes in time */
#define pluginParentSensitivityFlag 2 /* Sensitive to changes to its own group */
#define pluginChildSensitivityFlag3 /* Sensitive to changes to its child groups */
#define pluginPermissionCopyFlag  31  /* Do not copy the plug-in to another system */
```

If the pluginSensitivityFlag for a model plug-in is set, the host will use the plug-in sensitivity flags to determine when to sent a pluginModelSetup command to the plug-in. If the pluginTimeSensitivityFlag is set, the plug-in will receive a pluginModelSetup command whenever the current time changes. If the pluginParentSensitivityFlag is set, the plug-in will receive a pluginModelSetup command when any change is made to the plug-in's own group. If the pluginChildSensitivityFlag is set, the plug-in will receive a pluginModelSetup command whenever any change is made to one of the plug-in's child groups. These changes can occur either by user interaction (such as dragging the time thumb or a group) or as part of an animated sequence (such as during a shaded preview).

When the plug-in receives a pluginModelSetup command, it must send the host a hostModelReset command in order to receive a pluginModelGenerate command. Most model plug-ins should send a hostModelReset command every time they receive a pluginModelSetup command. More advanced plugins could try to determine what has changed since the last pluginModelSetup command before deciding if it is necessary to re-generate their model data.

ElectricImage sends a pluginInformation command when the user requests information about a plug-in (by holding down the command key and clicking on the plug-in's icon in the object palette) and before rendering to a remote Camera.

PluginAbout

This command is sent to the plug-in to tell it to display its about box.

```
#define pluginAbout 'ABOU'

typedef struct {
  PluginInfoRec theInfo;
} PluginAboutRec, *PluginAboutPtr;
```

theInfo	Sends a PluginInfoRec. See ‘PluginInfoRec’ in ‘Memory Allocation and Initialization.’

Typical Activities

If the plug-in has an about box, it should display it in a modal dialog until the user clicks the OK button. This command is not currently supported by ElectricImage but is provided for future expansion.

PluginInterface

This command tells the plug-in to display its user interface dialog box and accept user input.

```
#define pluginInterface 'INTF'

typedef struct {
  PluginInfoRec theInfo;
} PluginInterfaceRec, *PluginInterfacePtr;
```

theInfo	Sends a PluginInfoRec. See 'PlugininfoRec' in 'Memory Allocation and Initialization.'

Typical Activities

Most plug-ins have user editable values. These values are presented to the user for editing with a user interface dialog box. When the plug-in receives a pluginInterface command, it should first send a hostOpenResource command to the host in order to gain access to its resource file. The plug-in should then display its user interface dialog box and allow the user to edit control values until the user hits the OK or Cancel button. If the user hit the OK button, the plug-in should store the edited control values into its control data record. It is not necessary to send a hostCloseResource command because the host does this automatically after calling the plug-in.

A model plug-in should send a hostModelReset command to the host if the new control values will cause the generated model data to change. The host should respond by sending the pluginModelSetup and pluginModelGenerate commands to the plug-in.

ElectricImage sends the pluginInterface command to the plug-in when it is first added to the project or when the user clicks on the Options button for a model plug-in's group or a flare plug-in's light source. ElectricImage will send a pluginUpdateChannel command to a flare plug-in following the pluginInterface command. This allows the plug-in to update its animation channel values at the current frame.

Calls from the Plug-in to the Host

HostStatus

This command tells the host to display or update the status box and to handle background events or allow the user to abort the current plug-in command.

```
#define hostStatus 'STAT'

typedef struct {
  Ptr theReference;
  Real theStatusPos; Real;
  Str255 theStatusUnit;
  Str255 theStatusInfo;
  LongInt theStatusMin;
  LongInt theStatusMax;
  LongInt theStatusValue;
  LongInt theStatusTick;
} HostStatusRec, *HostStatusPtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plug-in command's theInfo record.
theStatusPos	Sends the status box's completion percentage. This represents the percentage of the current plug-in operation which has been completed. The value can range between 0.0 and 1.0.
theStatusUnit	Sends the status box's unit name.
theStatusInfo	Sends the status box's information string. This will tell the user what operation the plug-in is currently performing. The plug-in may need to perform several operations in order to complete a single command.
theStatusMin	Sends the status box's minimum value. This value is displayed at the left side of the status bar.
theStatusMax	Sends the status box's maximum value. This value is displayed at the right side of the status bar.
theStatusValue	Sends the status box's current value. This value should be between the status box's minimum and maximum values. It is used as a display value only and to calculate the status bar's position.
theStatusTick	Sends zero to change the status box's contents. Returns the next status update tick. The plug-in should send another hostStatus command after TickCount() is greater than theStatusTick.

Usage

When a plug-in must process data for more than 1/3rd of a second, it should call the hostStatus command. This command tells the host to display a status dialog box to tell the user how much of the current operation is complete. It also allows background events to occur and gives the user a chance to abort the current plug-in command. When the plug-in first calls the hostStatus command or when it wishes to change the status strings, it should set theStatusTick to zero. The host returns the next status event tick in theStatusTick. The plug-in should send another hostStatus command after this tick. If the plug-in does not send any hostStatus commands and processes information for a long time, the user may believe that the plug-in has crashed.

HostOpenResource

This command tells the host to open the plug-ins resource fork so that the plug-in can access its own resources.

```
#define hostOpenResource 'ORES'

typedef struct {
  Ptr theReference;
 } HostOpenResourceRec, *HostOpenResourcePtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plug-in command's theInfo record.

Usage

If a plug-in needs to access one or more of its resources, it should send a hostOpenResource command to the host to tell it to open the plug-in's resource file. When a plug-in receives a pluginInterface command, for example, it will need to get its dialog box resources. The plug-in should respond by sending a hostOpenResource command to the host and then call the GetNewDialog function to open the dialog box. A plug-in should always assume that its resource fork is not open when it receives a command from the host.

HostCloseResource

This command tells the host to close the plug-in's resource file.

```
#define hostCloseResource 'CRES'

typedef struct {
  Ptr theReference;
} HostCloseResourceRec, *HostCloseResourcePtr;
```

`theReference` Sends the plug-in's reference. This reference can be found in `theHostReference` of the plug-in command's `theInfo` record.

Usage

Most plug-ins will not need to call the `hostCloseResource` command because the host will automatically close the plug-in's resource file after it has finished sending it commands. This command is provided to allow a plug-in to close its resource file in order to open another resource file. Currently none of our plug-ins need to do this but this command was provided for possible future plug-ins.

Exception Handling

Calls from the Host to the Plug-in

PluginDescribe

This command tells the plug-in to describe one of its error codes.

```
#define pluginDescribe 'DESC'

typedef struct {
  PluginInfoRec theInfo;
  LongInt theResult;
  Str255 theDescription;
} PluginDescribeRec, *PluginDescribePtr;
```

theInfo	Sends a PluginInfoRec. See ‘PluginInfoRec’ in ‘Memory Allocation and Initialization.’
theResult	Sends the plug-in error code which should be described.
theDescription	Returns the plug-in error code's description as a string.

Typical Activities

Most result codes returned by plug-ins are defined in the 'EIPlugin.h' file. If a plug-in returns its own custom result code, the host will send a pluginDescribe command to the plug-in. The plug-in should return a brief description of theResult in theDescription.

Error code descriptions have been designed to cascade from the host to the plug-in. If the host receives a non-zero result code from the plug-in, it should call the DescribePluginResult function which is provided in 'EIPlugin.c'. This function will return a description string for all of the plug-in result codes defined in 'EIPlugin.h'. If this function does not recognize the result code, it will send a pluginDescribe command to the plug-in.

Software Protection

Calls from the Plug-in to the Host

HostChallenge

This command tells the host to respond to a plug-in's copy protection challenge.

```
#define hostChallenge 'CHAL'

typedef struct {
  Ptr theReference;
  OSType theChallenge;
  ChallengeDataRec theChallengeData;
} HostChallengeRec, *HostChallengePtr;
```

theReference	Sends the plug-in's reference. This reference can be found in theHostReference of the plug-in command's theInfo record.
theChallenge	Sends the challenge type. Send 'SERL' in this field to access the rightmost 7 characters of the host's serial string.
theChallengeData	Sends the challenge data record and receives the challenge response.

The challenge data record contains an 8 byte challenge or response and is defined as follows:

```
typedef union {
  UChar b[8];
  struct {
    UChar b0, b1, b2, b3, b4, b5, b6, b7;
  } b0;
  struct {
    Integer i0, i1, i2, i3;
  } i;
  UChar s[8];  /* Pascal 7 character string (length + 7 characters) */
  struct {
    LongInt l0, l1;
  } l;
} ChallengeDataRec, *ChallengeDataPtr;
```

The plug-in's copy protection can be implemented by requesting the host's serial string. The serial string contains the rightmost seven characters of the host's serial number string (which is usually only seven characters in length). The plug-in can use this information to create an authorization code for the copy of the plug-in. ElectricImage and Camera have a unique serial number for each licensed copy of the animation system. Copies of Slave Camera also have their own unique serial numbers. The plug-in can request an authorization code from the user the first time it is run and then store the code into its own resource file. The next time the plug-in is opened, it can simply check to make sure that the serial string and authorization codes match.

The plug-in can set the thePluginPermission flag when it receives the pluginInformation command to prevent the host from copying it to remote systems running slave cameras. This will prevent the user from being forced to re-enter the authorization code every time a job is sent to the Slave Camera on the remote system. Alternatively, the plug-in could store the authorization code in its own preference file in the System:Preferences folder. This would allow the plug-in to be copied to remote systems or updated to a new version without overwriting the old authorization code.

Appendix

ElectricImage Calling Sequence

What follows is a list that contains events which trigger ElectricImage to call a plug-in, and the actual sequence of commands it sends to a plug-in in response to these events:

Note: A `pluginCheck` command is called by EI before each command shown below, but is not included here for the purpose of clarity.

A model plug-in is first added to a project.

(1) `pluginInitialize`

(2) `pluginInterface`

(3) `pluginModelSetup`

(4) `pluginModelGenerate`

A project containing a model plug-in is reopened.

(1) `pluginInitialize`

(2) `pluginModelSetup`

(3) `pluginModelGenerate`

The time thumb in the project window is dragged to a new frame for a model plug-in whose `pluginTimeSensitivityFlag` is set.

(1) `pluginModelSetup`

(2) `pluginModelGenerate` (if a `hostModelReset` command was received)

A preview is called upon for a model plug-in whose `pluginTimeSensitivityFlag` is set (equivalent to dragging the time thumb frame by frame from start to finish).

For each frame:

(1) `pluginModelSetup`

(2) `pluginModelGenerate` (if a `hostModelReset` command was received)

A change is made to the child group of a model plugin whose `pluginChildSensitivityFlag` is set.

(1) `pluginModelSetup`

(2) `pluginModelGenerate` (if a `hostModelReset` command was received)

The option box in the model plug-in's 'group info' window is clicked.

(1) `pluginInterface`

(2) `pluginModelSetup` (if a `hostModelReset` command was received from `pluginInterface`)

(3) `pluginModelGenerate` (if a `hostModelReset` command was received from `pluginInterface`)

The project is closed or the model plug-in is deleted from a project.

(1) `pluginFinish`