Evolution Plugin Development Manual

Novell, Inc.
Michael Zucchi

Evolution Plugin Development Manual

by Novell, Inc. and Michael Zucchi

Table of Contents

List of Figures

Preface

This document is work-in-progress. Its structure and design is still as fluid as the underlying strucutre and design of some parts of EPlugin. There's no guarantee it will be updated at regular intervals, particularly this version.

The API documentation is currently generated using the Linux kernel-doc script. The stylesheets used to generate the HTML you're seeing seems to have bugs which duplicates some sections. It is also ugly and difficult to navigate.

1. Conventions

The following conventions are used in the manual ... (insert details here).

1.1. XML Annotation

XML definitions are annotated with BNF-style markers to indicate alternative (|), multiples (* or +), and optional (?) items. If no annotation is present then the item must be present once.

|	Indicates an alternative option. Only one of the items separated by | is to be chosen.
*	Following an item, * indicates the item may occur any number of times, including no times (0 or more multiple).
+	Following an item, + indicates the item must occur at least once, but may occur more than ones (1 or more multiple).
?	Following an item, ? indicates the item may occur at most once, if present (0 or 1 times).

Part I. EPlugin

Table of Contents

Chapter 1. Introduction

This book aims to be a comprehensive technical manual for the development of plugins for Evolution, a personal information manager for GNOME.

Up-to, and including, Evolution version 2.0, Evolution contained limited extensibility interfaces. There were only two ways to extend Evolution; by implementing a new top-level component, or by implementing a Camel provider. When implementing a top-level component, there was still little integration, and in effect it was merely a more complex way of writing a separate GNOME application. Camel providers were only designed to be e-mail storage backends, so were of limited use for general extensibility. Despite this, both mechanisms were used for example for the Exchange Connector, although the system made the integration clumsy and difficult.

This lack of extensibility has severaly stifled external developer contributions by forcing any extensions to be considered as core features. Evolution being a commercial product, it has tight usability and quality requirements that limits the ability to experiment with the core feature set in this way. As a result, very few lines of code or new features have been implemented by external contributors.

One of the major goals for the 2.2 release was to implement an extensibility system, given the working name of EPlugin, which must provide a frame-work for both providing extensibility hooks, and for extending the functionality of Evolution.

1. Plugin System

Any plugin system will generally have a number of goals:

- Provide a language independent invocation mechanism
- Allow extension of parts of the user interface and processing elements
- Require minimal extra or foreign code to implement in the core application
- Require minimal interface code to implement the extensions
- Not to impact performance or increase resource usage unduly
- Versioning
- Be able to be extended itself fairly easily.

EPlugin manages to fulfill these goals in most cases. EPlugin isn't a single object or interface in itself, although there is an object titled EPlugin, it is a synergistic [1] collection of integrated and continually evolving objects which work together to achieve these goals (and that will be definitly be the end of the MarketSpeak). It consists of a loader to invoke extension callbacks, hooks to resolve these callbacks, targets to identify context, and managers which are used by the core code to provide functionality and merging points for the extensions.

EPlugin's design was inspired and influenced by the Eclipse project. It aims at a lower target however, so it was able more easily implemented in a practical time-frame.

EPlugin was chosen as an approach to the problem of adding scriptability to Evolution. Instead of just linking to Perl, or Python, or even Mono by itself an approach was taken which focuses on the application end of the system. So instead of making every part of the application export its functionality and have to deal with whatever script engine is present, EPlugin addresses the hooking part of the equation

[1] I've always wanted to use *synergistic* in a sentence since I read it on the back of the Commodore 64 Users Guide.

in a language-independent manner. It also attemps to do it in a way which doesn't impact on the application development either.

The EPlugin world is awash with its own language. The next few sections will introduce the basic plugin nomenclature and high-level view of this world.

2. Loaders

The core of EPlugin is a light-weight object loader and callback invocation system. Because of the varied calling conventions of different languages, and to reduce the overhead of the plugin system itself, all callbacks only receive and return a single argument. By using structures to pass complex arguments, native C plugins require no extra overhead, and marshalling details are moved into the plugin implementation itself where required. It also simplifies memory management issues significantly. For example, the C plugin handler merely loads a shared library using GModule, and resolves a symbol by name; and is so all of 50 lines of code, total. The loaders are the only modules which need to interace with non-native code or conventions.

The other task of the plugin core is to load XML definitions of the plugins. Extension hooks are registered with the plugin core before the plugins are scanned, and are automatically instantiated to load each definition appropriately as they are encountered.

At each layer, a level of indirection is used so that new loaders and new hooks can be added transparently, and extend the plugin definition freely with any information they require.

3. Hooks

The hooks [2] which are registered with the loader provide meta-data for the management implementation layer for extending it at run-time. Their primary functions are to load the detail of the XML plugin definition, map it to the implementation, and marshal the implementation callbacks to the common plugin interface. How they do this depends on the implementation itself, and ranges from registering factory methods to simply adding the items directly.

In most cases the physical object need not be loaded until the callback is invoked, since the plugin definitions provide enough contextual information to build the interface or determine when they need to be invoked.

4. Managers

Managers [3] provide tools for the core code to extend itself at specific points, and in many cases are the objects used directly in the code to implement core features. In other cases they simply provide the hooks with an entry point into Evolution. For example, for the main menu hook, the manager is a thin layer to BonoboUI. On the other hand, EPopup is a complete implementation of a popup menu management system which was already used in Evolution 2.0. Some managers are one-off objects used as constructors for other objects, others are view-dependent, and some are static objects, such as the Event routers.

5. Items

Each manager uses a number of items to describe the object they control or create. The items are added to each manager instance from the plugins or from core code. The items from all of these sources are then merged together when required and processed accordingly. For example, menu items are merged

[2] A hook is something you can hang your stuff on.
[3] Unlike real managers, these are the ones that do the heavy lifting.

into a tree of GtkMenus. Events on the other hand are simply ordered and then invoked in the order of their priority. Items are part of the manager implementation, and in EPlugin they are all extensible objects too, which the hooks use to perform mapping to the plugin. Items may be extended by code hooking into the implementation, either the plugin hooks, or the core code.

6. Targets

Targets [4] are view or component specific context objects. They contain enough information to be used as stand-alone contexts to implement callbacks for both core functions and plugin hooks. For example for the mail view, a select target contains a folder and a list of selected messages. An attachment (part) target contains the Camel representation of the part and the mime-type for that part. Targets are part of the manager implementation and are extended by subclassing the manager.

[4] Think of a target as the target of interest.

Chapter 2. Plugin Loaders

Plugin loaders implement a hool to a new language, or loading system in the plugin system. The actual binding of new languages to the plugin system or other parst of Evolutions api's are beyond the scope of this document, some languages make this easier than others.

1. Base Plugin

The EPlugin base class is an abstract class which provides the basic services for plugin implementations. The main services are:

- Resolve plugin type and instantiate an EPlugin object to represent and manage it.
- Load the base structure of the XML plugin definition files.
- Resolve plugin hook types and instantiate a EPluginHook to represent and manage it.
- Provide a simple, language-independent api for invoking plugin callbacks
- Provide I18N context for plugins.
- Some simple static helper methods to simplify each implementing class.

See the Chapter 8, *EPlugin* for these details.

1.1. Definition of a Plugin

The base plugin XML definition. Subclasses of EPlugin extend this basic structure with additional parameters or elements as they require.

Note that there may be any number of `e-plugin` elements in a given plugin file, this may be used to simplify distribution of plugin packages.

```
<?xml version="1.0">
<e-plugin-list>
  <e-plugin
    id="unique id"
    type="loader type"
    domain="translation domain" ?
    name="plugin name"
    ...>
    <description>long description</description> ?
    <hook
      class="hook class"
      ...>
    ...
    </hook> +
  </e-plugin> +
</e-plugin-list>
```

`id`	A unique string identifying this plugin. By convention this will follow the java-like class namespace system. e.g. `com.ximian.evolution.test-plugin`

type	The type name of the plugin loader. Currently shlib and mono are the only supported values. If no known handler is registered for this type, the plugin definition is silently ignored.
domain	The translation domain for this plugin, as passed to the `dcgettext` call of the gettext package. If not supplied then the default application domain is used (i.e. "evolution"). This is used to translate translatable strings for display.
name	A short name for the plugin. "Bob's Wonder Extender" might be suitable. This value will be translated.
description	A longer description of the plugin's purpose. This value will be translated.
hook	This is a list of all of the hooks that this plugin wishes to hook into. See the Plugin Hooks section for the details of the basic hook types defined. The hook *class* is resolved using the registered hook types, and if none can be found, or a version mismatch occurs, then the hook is silently ignored.

2. Shared Library Loader

The shared library loader EPluginLib implements a concrete EPlugin type which loads GNU shared libraries via the GModule api. It simply resolves symbols directly from the loaded shared object and invokes them expecting a function signature of EPluginLibFunc.

To manage plugin lifecycle, the function `e_plugin_lib_enable` will be invoked which allows the plugin to initialise itself. Its signature should match EPluginLibEnableFunc, and it will be called with *enable=1*. If the enable function returns non-zero it is assumed to have failed intialisation and will not be invoked further.

2.1. Definition

The shared library loader only requires one extra parameter in the base plugin definition.

```
<e-plugin
  ...
  type="shlib"
  location="/full/path/name.so"
  ...
  <hook class="...">
   ...
</e-plugin>
```

type	The type name of the shared library plugin is `shlib`.
location	The location parameter contains the full path-name of a shared object to load.

2.2. Invocation

Function specification

Where a function spec is required in a plugin hook definition, it should simply be the full name of an exported symbol in the shared object.

Callback signature

```
void * function(ep, data);
EPlugin * ep;
void * data;
```

`function`	The callback function.
`ep`	The container EPlugin representing this plugin.
`data`	Hook context data. It is part of the hook's api to specify the type of this pointer.
return value	Return data. It is part of the hook's api to specify the type of this pointer.

3. Mono Assembly Loader

The mono assembly loader EPluginMono implements a concrete EPlugin type which loads C# assemblies using Mono. Apart from loading the assembly, it can optionally instantiate a class to implement the callback or invoke static methods directly.

3.1. Definition

The mono assembly loader needs the name of the assembly and optionally the name of the class for handling the callbacks.

```
<e-plugin
  ...
  type="mono"
  location="/full/path/name.dll"
  handler="PluginClass" ?
  ...
  <hook class="...">
   ...
</e-plugin>
```

`type` The type name of a mono assembly plugin is `mono`.

`location` The location parameter contains the full path-name of an assembly to load.

`handler` If supplied, the handler contains the fully qualified name of the class which handles all callbacks for this plugin. If a handling class is used, then the function specifications become relative to this class.

This class will be instantiated once upon the first callback invocation, and remain active for the life of the plugin (or application).

3.2. Invocation

Function specification

If no *`handler`* class is specified, then the function specification must match a static method in the assembly. This is passed to `mono_method_desc_new` and `mono_method_desc_search_in_image`, typically `FunctionName(intptr)`.

If the handler is specified, then the function specification is relative to the handler class. This is passed to `mono_method_desc_new` and `mono_method_desc_search_in_class`, typically `:MethodName(intptr)`.

Callback signature

```
IntPtr function(data);
IntPtr data;
```

`function`	The callback method.
`data`	The hook context data. This is a pointer to unmanaged data, and it is up-to the plugin to interpret this data right now, although some helper binding classes are planned. FIXME: hook-up when they and doco are done.
return value	The callback return data. It is up to the hook's api to define the type of this pointer. It may be a simple boxed value type, or a memory pointer allocated in unmanaged memory (e.g. a GObject handle or a CamelObject cobject value).

Chapter 3. Plugin Hooks

This chapter will introduce the available plugin hook types. A given plugin can hook into any of these hooks any number of times. Some refer to specific instances of objects and others are implicitly defined.

By design, there is considerable similarity and orthogonality amongst all of the various hook types and management objects.

1. Popup Menus

The popup menu hook lets you hook into any of the context menus in Evolution, by name and context. Complex, dynamic, and multi-level menus are created on the fly by merging the items for a given menu as it is being shown. Each component provides its own context targets to self-describe the situation under which the menu is invoked. Plugins and core code alike are then invoked at the user's direction. The popup manager and all context data lives as long as the menu and until a choice is made, simplifying memory management.

The menu is merged from multiple plugins and core application code by using a simple lexiographical sort of an absolute path to the menu item. This merged list is then scanned and expanded into a tree of menus. Individual items can be hidden or inactive based on the target and a simple mask which is defined by the component itself. A rich collection of menu item types are possible, from simple, to checkboxes or images. The popup code is simple, and easy to use, and simplifies the use of popup menu's in the core application anyway, that they are pluggable is a free-bonus.

1.1. Defining a popup hook

Not sure if this fits here as such. Probably temporary placeholder.

```
<hook class="com.ximian.evolution.mail.popup:1.0">
 <menu id="menuid" target="targettype">
  <item
   type="item | toggle | radio | image | submenu | bar"
   active ?
   path="foo/bar"
   label="menu text"
   icon="icon name" ?
   visible="target mask" ?
   enable="target mask" ?
   activate="function spec"/> *
 </menu> *
</hook>
```

Need to define menu tag

type	The menu item type. The type maps directly to the corresponding EPopupItem types.
active	If present, then radio or toggle menu items are active when first shown. After the first instantiation, they will remember their active state.
path	A '/' separated path used to position the item within menu and in the right submenu. Each menu and plugin should define how its menu's are layed out so other plugins can determine what value to use here.

label	The text to be displayed on the menu item. This will be translated based on the plugin translation domain.
icon	The name of a gnome-icon-theme standard icon, or the full path-name of an icon image to use as menu item icon. This will be blank if not supplied.
visible	A comma separated list of mask enumeration values used to define when this item is shown. What values are valid depend on the menu hook class of the menu being hooked onto.
enable	A comma separated list of mask enumeration values used to define when this item is enabled. What values are valid depend on the menu hook class of the menu being hooked onto. This is currently unimplemented.
activate	A plugin-type specific function specification. This function will be resolved and called when the menu item is activated.

1.2. Merging Plugin Items

A very simple algorithm is used to form the menu by merging the plugin's menu items with the system menu items for a given menu. What follows is a simple example of how this is done. It will be demonstrated using a simplified menu from the message-list, as used in the Evolution Mail component, and a simple plugin which adds a single menu item and menu separator into the middle of the menu, when appropriate.

When the application wishes to show a specific popup menu, it creates a new EPopup object with a unique menu id to manage it. It adds all of the items it wishes to add to the menu (see "Builtin Items" in the following diagrams). The application then asks for the menu to be created. The menu building process adds all of the menu items from all plugins that target this specific menu into a flat list, discarding those which don't match the current Target qualifications. The result is then sorted using a simple ASCII sort, and then a menu built from the remaining items. This is probably best described by some diagrams.

The following two diagrams show how a popup menu is automatically customised depending on the context. On the left of each diagram are all of the menu items which apply to the example menu. The menu label, with the qualifiers listed underneath, with the menu item path along-side. On the right-hand side of each diagram is the result of:

- Selecting items based on the target qualifiers
- Sorting the remaining items based on their path.
- Building this sorted list into a menu.

The actual list of target qualifiers are defined by the application itself. Generally a specific menu will have only one possible target, and a list of matching target qualifiers. The example shows how a plugin can insert a menu item anwhere it wishes in the menu system. Submenus are also supported, and they work in exactly the same manner, with / characters used to separate submenu paths. A submenu must sort into the position immediately before the definition of its items.

Figure 3.1. Merging a menu with many items selected.

The first diagram shows when the target qualifiers are *many*, and *mark_unread*. The menu items which operate on only one selected message are not shown. Similarly for those able to be marked as unread (i.e. they are currently read).

Figure 3.2. Merging a menu with one item selected.

This diagram shows when the target qualifiers are `one`, and `mark_read`. The menu items which operate on only many selected messages are not shown. Similarly for those able to be marked as read.

2. Main menus

The main menu hook lets you hook into various main menus in Evolution, based on the current active view (component). The system works by piggy-backing on the existing use of the BonoboUI menu system used by all of the Evolution components. Bonobo handles the menu merging and user input, and the hook resolves the verb being invoked and redirects it to the plugin. Each view defines a single target which describes the appropriate context. For the Mail view, this is the current folder and currently selected message(s).

Each view keeps track of its own manager object. When it is (de)activated, it also (de)activates the management object which dynamically adds and removes the menu items from the BonoboUIContainer via a supplied BonoboUI XML definition file <perhaps it should embed the bonobouixml>. If the target changes, the view lets the manager know, and it updates the visibility and sensitivity of objects appropriately, allowing reasonably dynamic user-interfaces to be managed automatically. The plugin itself isn't loaded until the menu item in question is invoked

Simple menu items and toggle menu items are supported currently. Also, because actual menu display is driven by BonoboUI, then toolbar items can also be added using this mechanism.

2.1. Defining a menu hook

Not sure if this fits here as such. Probably temporary placeholder.

```

<hook class="com.ximian.evolution.mail.bonoboMenu:1.0">
 <menu id="menuid" target="targettype"
  <ui file="/path/to/bonobo-ui-menu-definition.xml"> +
  <item
   type="item | toggle | radio"
   active ?
   path="/commands/FooBar"
   verb="FooBar"
   visible="target mask" ?
   enable="target mask" ?
   activate="function spec"/> *
 </menu> *
</hook>
```

Need to define menu tag

`ui`	The `ui` element contains a filename of the BonoboUI XML menu definition to load when the view is activated. Any number of `ui` elements may be defined, and they are all loaded.
`type`	The menu item type. The type maps directly to the corresponding EMenuItem types. `ra-`

dio is currently not implemented.

active	If present, then radio or toggle menu items are active when first shown. After the first instantiation, they will remember their active state.
path	The BonoboUI element path corresponding to this menu item.
verb	The BonoboUI verb corresponding to the item to be listened to.
visible	A comma separated list of mask enumeration values used to define when this item is shown. What values are valid depend on the menu hook class of the menu being hooked onto.
enable	A comma separated list of mask enumeration values used to define when this item is sensitive. What values are valid depend on the menu hook class of the menu being hooked onto.
activate	A plugin-type specific function specification. This function will be resolved and called when the menu item is activated. The funciton's parameters will depend on the type of menu item being invoked.

2.2. Merging Plugin Items

Merging is performed by BonoboUI, and the source of the menu data is defined by the *ui* file.

3. Configuration Pages and Wizards

Configuration pages are somewhat more complex than any of the other types of hookable object. This is reflected in the complexity of the items and callbacks involved.

Essentially, the EConfig object is used in combination to both instrument existing windows and building new content. Each configuration window comprises of several basic elements with some minor variations allowed. It consists of a number of pages in a specific order, each containing a number of titled sections in a specific order, each containing a number of items. The variations are that the top-level widget may be a GtkNotebook or a GnomeDruid; and each section may instrument a GtkBox, or a GtkTable. The definition of the available hooks will define what form they take.

The EConfig manager uses the description of all the items supplied to it to build the complete window. It can also drive various aspects of the UI, such as navigating through a druid or handling instant-apply vs. modify-and-save dialogues.

Figure 3.3. Event and Data Flow in EMConfig

3.1. Defining a configuration page hook

Not sure if this fits here as such. Probably temporary placeholder.

```
<hook class="com.ximian.evolution.mail.config:1.0">
 <group
  id="window id"
  target="targettype"
  check="function spec"?
  commit="function spec"?
  abort="function spec"?>
  <item
   type="book | druid | page | page_start | page_finish | section | section_table
   path="/absolute/path"
   label="name" | factory="function  spec"
  /> *
 </menu> *
</hook>
```

3.1.1. Group Element Properties

id The name of the configuration window to which this hook applies.

target The type of target this configuration window applies too. This will normally be tied directly to the specific configuration window itself.

check A callback which will be invoked to validate the configuration or a specific page of the configuration. It will be invoked with a EConfigHookPageCheckData structure, and is expected to return a non-NULL value if the page validates.

The callback will be expected to handle all *pageid*'s present in the configuration window, and should return `TRUE` for pages it does not recognise. If *pageid=""* (an empty string), then the *check* function should validate all settings. See also ???.

commit A callback which will be invoked to commit the configuration data, if the configuration page isn't an instant-apply one. This callback can write any configuration changes to permanent storage. It is not used for instant-apply windows.

abort A callback which will be invoked to abort the configuration process. This callback is called when the Cancel button is pressed on stateful configuration windows.

3.1.2. Item Element Properties

type The menu item type. The type maps directly to the corresponding EConfigItem types. Only one of *book* and *druid* may be supplied for the entire configuration page, and this will usually already be defined by the application.

path The path to the configuration item in question. This is a simple string that when sorted using an ASCII sort will place the items in the right order. That is, sections before items before pages before the root object.

label The textual label of this item. This may only be supplied for the section and page types. For sections it will be the section frame text. For pages this will be the druid page title or the notebook tab text. If a *factory*is supplied then this value is not used. This will be translated based on the plugin translation domain.

factory If supplied, the factory method used to create the GtkWidget elements for this configuration item. Factories may be supplied for any of the item types. If no *label* is set then the *factory* must be set.

3.2. Generating Configuration Pages

Configuration items essentially spam 3 dimensions, but are merged in a similar fashion to the way Popup items are merged. The main difference is that there are no target qualifiers used to select which items are shown, it is up to the item factory to either create or not create the item as it sees fit. The EConfig manager takes care of the rest, including removing un-used sections or pages.

All items for a given configuration screen are converted into a list and sorted based on the `path`. The configuration builder then goes through each item, creating container widgets or calling factories as required. If a given page or section is empty, then it is removed automatically. This process isn't only a one-off process. For certain complex configuration screens, items or even pages and sections need to be dynamic based on a previous setting. EConfig supports this mode of operation too, in which case it rebuilds the configuration screen the same way, and automatically destroys the old widgets [5] and even re-orders pages and sections where appropriate to make the user-interface consistent.

The following few examples some of the flexibility of the EConfig system.

Figure 3.4. The application defined, unaltered configuration page.

First we have the original configuration window. This is defined by the application, the application uses EConfig to build this window, and in the process EConfig instruments the sections that the application defines. This allows plugins to add new pages/sections/items anywhere on the page - to a granularity as defined by the application. For example the application may at minimum merely define the top-level notebook or druid object and a number of pages. When the pages are created the application could add as much content as it wants, which would still allow plugins to extend the user interface, but only by adding options to the end of each page. At the other end of the scale the application could enumerate every single item (i.e. row) in every section on every page, allowing plugins to put new items anywhere in the display.

Figure 3.5. A plugin adding a new section to an existing page.

In this case the plugin has merely added a new section on the bottom of the HTML Mail settings page. When the factory is called the plugin has a parent GtkTable (in this case, it could be a VBox) and borderless frame already defined, and it just has to instantiate its own control widgets, add them to the table, and return one of the widgets. The returned widget is used later if the window needs to be reconfigured, although this particular configuration page is static so it isn't needed.

Figure 3.6. A plugin inserting a new page for its settings.

And finally we have exactly the same plugin, which has exactly the same code. But a small change to the plugin definition allows the plugin to add an arbitrary new page (in an arbitrary position) into the whole window. If this was a druid, then new druid pages can also be inserted at arbitrary locations, and page navigation (in a strictly linear manner) is automatically controlled by EConfig as per Figure 3.3, “Event and Data Flow in EMConfig”.

In practice, EConfig provides more than it takes the application to use - generally little or no extra application code is required to use it. It also [6] enforces and simplifies HIG compliance. And as a side-benefit to the application it transparently provides extension hooks for external code to provide a seamlessly integrated user experience.

[5]In most cases, in some cases additional manual processing is required in the factory callback.
[6]Or it will - the code needs some tweaking.

4. Events

No extensibility framework would be complete without an event system. Events are used to reflect changes in internal state of the application, and track actions by the user. They can contain any information and additionally can be filtered based on the information itself. Special targets are used, as in the other plugin hooks, to hold this information.

Event managers are defined to contain the different event types that a given component can export. Only one event manager object is instantiated for each component, and each plugin listening to events from that component are registered on that event manager directly.

Events handlers have priorities, and can swallow events, allowing some level of complexity of event routing. This feature might not prove useful and may be removed in the future if it isn't.

4.1. Defining an event hook

Not sure if this fits here as such. Probably temporary placeholder.

```
<hook class="com.ximian.evolution.mail.events:1.0">
 <event
   target="target name"
   id="event name"
   type="pass | sink" ?
   priority="signed integer" ?
   enable="target mask" ?
   handle="function spec"/> *
</hook>
```

`target`	The target type of the event listener. This will normally match in a 1:1 relationship to the event *`id`* itself.
`id`	The name of the event to listen to. By convention the names will be of the form `target.event`. e.g. `folder.changed`, or `message.read`, etc. Although they are just simple case-sensitive strings.
`type`	The event listener type. The type maps directly to the corresponding corresponding EEventItem types.
`priority`	A signed integer specifying the priority of this event listener. 0 (zero) should be used normally, although positive and negative integers in the range -128 to 127 may aslo be used.
`enable`	A comma separated list of mask enumeration values used to qualify when this event listener is invoked. What values are valid depend on the event hook class.
`handle`	A plugin-type specific function specification. This function will be resolved and called when an event is routed to this listener.

5. Mail Formatter

The mail formatter plugin will invoke plugin code to format any part of an email based on mime-type. There are several formatters used internally by the mailer for different contexts, and each can be hooked into separately, providing extensible mail formatting for everything from the primary mail display, to printing, to reply quoting and more. If you are implementing a handler for a given mime-type, each formatter appropriate for the data-type should be hooked into, so that it displays properly in all contexts.

Since the management object in this case is the same formatting object as used by the core mail display engine, a plugin may override or reimplement complete new functionality seamlessly.

This plugin hook isn't strictly part of the core functionality as it is provided only by the mail component. It however demonstrates that the plugin system is extensible itself.

5.1. Defining a formatter hook

Not sure if this fits here as such. Probably temporary placeholder.

```
<hook class="com.novell.evolution.mail.format:1.0">
 <group id="formatter type">
  <item
    flags="handler flags"
    mime_type="major/minor"
    format="function spec"/> +
 </group> +
</hook>
```

id	The actual formatter this applies to. e.g. EMFormat for the base formatter class, EMFormatHTML for HTML output to a GtkHTML object, etc.
flags	Flags to define whether this is an attachment or inline content.
id	The name of the event to listen to.
mime_type	The type of object this handler formats.
format	A plugin-type specific function specification. This function will be invoked to format objects of the specified *mime_type*.

5.2. The formatting process

The formatting process is driven by the EMFormat object, although there are different subclasses of this object used for different purposes. These behave quite differently so each must be explained separately. There is the basic formatter type which converts a CamelMimeMessage into a stream of data, and there is a HTML formatter type which uses a GtkHTML object to parse the content and may request further information required to complete the formatting.

A basic formatter goes through the following steps:

1. Outputs pre-amble information. e.g. Flag-For-Followup status.
2. Invokes `format_message` to begin the message formatting. `format_message` displays the message header, then looks up the content object.
3. Using the mime-type of the content object (whether supplied or calculated), a handler is looked up from a per-class table to process the type.
4. If no handler exists, then the data is formatted as an attachment.

5. If a handler exists, then it is invoked to display that type. Depending on whether the data is to be displayed 'inline' or not, the data may also get an attachment expander and button.

6. The handler transforms the part's data, if need be, and writes the appropriate format output to a stream.

For conglomerate types, the formatting process is continued recursively, until all parts have been displayed, as appropriate.

A HTML formatter goes through the same basic steps, but has additional features and requirements. It uses multiple threads. At least one other thread is used for all of the Camel message content operations since some of them may block on remote I/O. This also simplifies cancellation processing. Also, because it has access to a full HTML rendering object, references to embedded content (images, buttons, etc.) are also processed.

Most format handlers don't need to know about all the fiddly details however. If they are just outputting HTML content with no out of band references, they work identical to the basic format handlers with the exception they cannot call any Gtk GUI code because of threading issues. This can still be done by using an IFRAME. If they want to embed an icon or other image, they simply need to insert the HTML IMG tag reference in their format handler, and setup a callback to handle it when GtkHTML requests it. EMFormat has some helper classes to make this only a few lines of code, including generation of the IMG SRC URL. IFRAMEs work identically to IMG tags, and similar process is involved with embedding custom widgets using the OBJECT tag. EMFormatHTML takes care of calling the right callbacks for the right embedded reference from the right thread.

Since format handlers are chained off a given type, then a plugin can also inherit formatting behaviour as well as override it. This gives much greater flexibility since the plugin need only implement its behaviour in specific situations. e.g. an OpenPGP message handler could fall-back to the normal text-formatter if it doesn't detect the ASCII armour in a text/plain part. Or another handler may disable itself based on configuration or state.

All format handlers for all types must also be fully re-entrant code (more or less write-once global and static variables) if they call any other formatting functions.

Part II. Evolution Hook Points.

This section enumerates all of the published hook points and target types available in each component in Evolution.

Table Format

Id	The hook point id.
Target	The target which this hook uses for its context data. Targets are described in a following section.
Items	If appropriate and defined, specifies identifying path names of items which make up the hook. e.g. popup menu items, and configuration pages. These item specifications allow the plugin writer to position their items appropriately.

Table of Contents

Chapter 4. Mail Hooks

Need to find out the right docbook to mark-up most of this text.

1. Popup menus

The mail popup menu class is `org.gnome.evolution.mail.popup:1.0`.

The plugin callback data will be the target matching the plugin menu itself, and the callback returns no value.

1.1. Folder Tree Context Menu

This is the context menu shown on the folder tree.

Id	org.gnome.evolution.mail.foldertree.popup
Class	org.gnome.evolution.mail.popup:1.0
Target	EMPopupTargetFolder
Defined	mail/em-folder-tree.c:2816

1.2. Message List Context Menu

This is the context menu shown on the message list or over a message.

Id	org.gnome.evolution.mail.folderview.popup.select
Type	EMPopup
Target	EMPopupTargetSelect
Defined	mail/em-folder-view.c:1022

1.3. Inline URI Context Menu

This is the context menu shown when clicking on inline URIs, including addresses or normal HTML links that are displayed inside the message view.

Id	org.gnome.evolution.mail.folderview.popup
Class	org.gnome.evolution.mail.popup:1.0
Target	EMPopupTargetURI
Defined	mail/em-folder-view.c:2226

1.4. Inline Object Context Menu

This is the context menu shown when clicking on inline content such as a picture.

Id	org.gnome.evolution.mail.folderview.popup
Class	org.gnome.evolution.mail.popup:1.0

Target	EMPopupTargetPart
Defined	mail/em-folder-view.c:2235

1.5. Attachment Button Context Menu

This is the drop-down menu shown when a user clicks on the down arrow of the attachment button in in-line mail content.

Id	org.gnome.evolution.mail.formathtmldisplay.popup
Class	org.gnome.evolution.mail.popup:1.0
Target	EMPopupTargetPart
Defined	mail/em-format-html-display.c:1099

1.6. Composer Attachment Bar Context Menu

This is the context menu on the composer attachment bar.

Id	org.gnome.evolution.mail.composer.attachmentbar.popup
Class	org.gnome.evolution.mail.popup:1.0
Target	EMPopupTargetAttachments
Defined	mail/../composer/e-msg-composer-attachment-bar.c:517

1.7. Internal popup menus

The following popup menus are defined, but they are used with no target, and so provide no useful context if they were to be hooked onto.

`com.ximian.mail.messagelist.popup.drop` is used for the ASK drop type on the message list.

`com.ximian.mail.storageset.popup.drop` is used for the ASK drop type on the folder tree.

1.8. Mail Popup Targets

Not sure if this needs to explain the qualifier meanings, or leave it to the in-line comment stuff in the enumeration definition. Maybe it just needs a direct link to the enumeration.

1.8.1. Folder Target

This target is used to define actions on a folder context. Normally associated with the folder tree.

Name	`folder`
Structure	EMPopupTargetFolder

Qualifiers	`folder` = `EM_POPUP_FOLDER_FOLDER`
	`store` = `EM_POPUP_FOLDER_STORE`
	`inferiors` = `EM_POPUP_FOLDER_INFERIORS`
	`delete` = `EM_POPUP_FOLDER_DELETE`
	`select` = `EM_POPUP_FOLDER_SELECT`

1.8.2. Selection Target

This target is used to define context for actions associated with a selection of mail messages from a specific folder.

Name	`select`
Structure	EMPopupTargetSelect

Qualifiers	`one` = `EM_POPUP_SELECT_ONE`
	`many` = `EM_POPUP_SELECT_MANY`
	`mark_read` = `EM_POPUP_SELECT_MARK_READ`
	`mark_unread` = `EM_POPUP_SELECT_MARK_UNREAD`
	`delete` = `EM_POPUP_SELECT_DELETE`
	`undelete` = `EM_POPUP_SELECT_UNDELETE`
	`mailing_list` = `EM_POPUP_SELECT_MAILING_LIST`
	`resend` = `EM_POPUP_SELECT_EDIT`
	`mark_important` = `EM_POPUP_SELECT_MARK_IMPORTANT`
	`mark_unimportant` = `EM_POPUP_SELECT_MARK_UNIMPORTANT`
	`flag_followup` = `EM_POPUP_SELECT_FLAG_FOLLOWUP`
	`flag_completed` = `EM_POPUP_SELECT_FLAG_COMPLETED`
	`flag_clear` = `EM_POPUP_SELECT_FLAG_CLEAR`
	`add_sender` = `EM_POPUP_SELECT_ADD_SENDER`
	`mark_junk` = `EM_POPUP_SELECT_MARK_JUNK`
	`mark_nojunk` = `EM_POPUP_SELECT_MARK_NOJUNK`
	`folder` = `EM_POPUP_SELECT_FOLDER`

1.8.3. URI Target

This target defines context for operations on a URI, normally displayed inline somewhere in the message view.

Name	`uri`
Structure	EMPopupTargetURI

Qualifiers	`http` = `EM_POPUP_URI_HTTP`
	`mailto` = `EM_POPUP_URI_MAILTO`
	`notmailto` = `EM_POPUP_URI_NOT_MAILTO`

1.8.4. Message Part Target

This target defines context for operations on messages, or individual message parts. The same target is used for inline images or other content which can be encapsulated in a MIME part (i.e. anything).

Name	`part`
Structure	EMPopupTargetPart

Qualifiers	`message` = `EM_POPUP_PART_MESSAGE`
	`image` = `EM_POPUP_PART_IMAGE`

1.8.5. Attachments Target

This target is used to define context for operations on the mail composer attachment bar.

Name	`attachments`
Structure	EMPopupTargetAttachments

Qualifiers	`one` = `EM_POPUP_ATTACHMENTS_ONE`
	`many` = `EM_POPUP_ATTACHMENTS_MANY`

2. Main menus

The mail popup menu class is `org.gnome.evolution.mail.bonobomenu:1.0`.

The plugin callback data will be the target matching the plugin menu itself, and the callback returns no value.

2.1. Main Mail Menu

The main menu of mail view of the main application window.

Id	org.gnome.evolution.mail.browser
Class	org.gnome.evolution.mail.bonobomenu:1.0
Target	EMMenuTargetSelect
Defined	mail/em-folder-browser.c:295

2.2. Standalone Mssage View Menu

The main menu of standalone message viewer.

Id	org.gnome.evolution.mail.messagebrowser
Class	org.gnome.evolution.mail.bonobomenu:1.0
Target	EMMenuTargetSelect
Defined	mail/em-message-browser.c:184

2.3. Mail Menu Targets

2.3.1. Message Selection Target

This target is used to define context for operations on a selection of messages in the view's message list.

Name	`select`
Structure	EMMenuTargetSelect

Qualifiers	`one` = `EM_MENU_SELECT_ONE`
	`many` = `EM_MENU_SELECT_MANY`
	`mark_read` = `EM_MENU_SELECT_MARK_READ`
	`mark_unread` = `EM_MENU_SELECT_MARK_UNREAD`
	`delete` = `EM_MENU_SELECT_DELETE`

	`undelete` = `EM_MENU_SELECT_UNDELETE`
	`mailing_list` = `EM_MENU_SELECT_MAILING_LIST`
	`resend` = `EM_MENU_SELECT_EDIT`
	`mark_important` = `EM_MENU_SELECT_MARK_IMPORTANT`
	`mark_unimportant` = `EM_MENU_SELECT_MARK_UNIMPORTANT`
	`flag_followup` = `EM_MENU_SELECT_FLAG_FOLLOWUP`
	`flag_completed` = `EM_MENU_SELECT_FLAG_COMPLETED`
	`flag_clear` = `EM_MENU_SELECT_FLAG_CLEAR`
	`add_sender` = `EM_MENU_SELECT_ADD_SENDER`
	`mark_junk` = `EM_MENU_SELECT_MARK_JUNK`
	`mark_nojunk` = `EM_MENU_SELECT_MARK_NOJUNK`
	`folder` = `EM_MENU_SELECT_FOLDER`

3. Config Windows and Druids

The mail config class is `org.gnome.evolution.mail.config:1.0`.

3.1. Mail Preferences Page

The main mail preferences page.

Id	org.gnome.evolution.mail.prefs
Type	E_CONFIG_BOOK
Class	org.gnome.evolution.mail.config:1.0
Target	EMConfigTargetPrefs
Defined	mail/em-mailer-prefs.c:726

3.2. Mail Account Editor

The account editor window.

Id	org.gnome.evolution.mail.config.accountEditor
Type	E_CONFIG_BOOK
Class	org.gnome.evolution.mail.config:1.0
Target	EMConfigTargetAccount
Defined	mail/em-account-editor.c:2423

3.3. New Mail Account Druid

The new mail account druid.

Id	org.gnome.evolution.mail.config.accountDruid
Type	E_CONFIG_DRUID
Class	org.gnome.evolution.mail.config:1.0
Target	EMConfigTargetAccount
Defined	mail/em-account-editor.c:2434

3.4. Folder Properties Window

The folder properties window.

Id	org.gnome.evolution.mail.folderConfig
Type	E_CONFIG_BOOK
Class	org.gnome.evolution.mail.config:1.0
Target	EMConfigTargetFolder
Defined	mail/em-folder-properties.c:283

3.5. Mail Composer Preferences

The mail composer preferences settings page.

Id	org.gnome.evolution.mail.composerPrefs
Type	E_CONFIG_BOOK
Class	org.gnome.evolution.mail.config:1.0
Target	EMConfigTargetPrefs
Defined	mail/em-composer-prefs.c:901

3.6. Mail Config Targets

3.6.1. Account Target

The account target is used for configuring accounts, and so has a pointer to the EAccount being configured. This is a copy of the actual account object, and is copied to the original once the data is ready to commit.

Name	`account`
Structure	EMConfigTargetAccount
Items	Define some of the items available and where they fit in the gui

3.6.2. Preferences Target

The preferences target is used for global preferences. As such it just contains a pointer to the global configuration store - a GConfClient.

Name	`prefs`
Structure	EMConfigTargetPrefs

3.6.3. Folder Target

Name	`folder`
Structure	EMConfigTargetFolder

4. Events

The mail event class is `org.gnome.evolution.mail.events:1.0`.

4.1. message.reading

message.reading is emitted whenever a user views a message.

Title	Viewing a message
Target	EMEventTargetMessage
Defined	mail/em-folder-view.c:2011

4.2. folder.changed

folder.changed is emitted whenever a folder changes. There is no detail on how the folder has changed.

Title	Folder changed
Target	EMEventTargetFolder
Defined	mail/mail-folder-cache.c:252

4.3. Mail Event Targets

4.3.1. Folder Target

Name	`folder`
Structure	EMEventTargetFolder
Qualifiers	List qualifiers

5. Formatters

The mail formatter hook class is `com.novell.evolution.mail.format:1.0`.

5.1. Base Formatter

The EMFormat class is the base class for all formatting types. It should only be used to define compound and complex types which do not rely on outputting any textual information, or rely on any screen or print output differences.

Name	`EMFormat`
Target	EMFormatHookTarget

5.2. HTML Formatter

The EMFormatHTML class is the base class for most formatting types which generate HTML output. It renders output to a GtkHTML object. It uses a fairly complex multi-thread approach to the formatting to ensure the user-interface is not blocked for processing. GtkHTML is used in a limited way by this class for HTML parsing and resolution of embedded objects. Embedded objects and Widgets may not be used from formatters which hook onto this entry point.

Name	`EMFormatHTML`
Target	EMFormatHookTarget

This section needs a huge amount of explanation, and/or more detail needs to be added to another section about the formatter class

5.3. HTML Display Formatter

The EMFormatHTMLDisplay class is a subclass of EMFormatHTML, and is used as a mail display widget. As such, it has access to all of the facilities of GtkHTML, such as embedded widgets. Like the EMFormatHTML class, this uses a complex multi-thread architecture.

Name	`EMFormatHTMLDisplay`
Target	EMFormatHookTarget

This section needs a huge amount of explanation, and/or more detail needs to be added to another section about the formatter class

5.4. HTML Print Formatter

The EMFormatHTMLPrint class is a subclass of EMFormatHTML, and is used as a mail printing widget. It cannot access embedded widgets. For most purposes you would normally only connect to the EMFormatHTML hook, and generate generic HTML output which could be printed or shown on-screen if it isn't overriden by the display formatter.

Name	`EMFormatHTMLPrint`
Target	EMFormatHookTarget

This section needs a huge amount of explanation, and/or more detail needs to be added to another section about the formatter class

5.5. Mail Quote Formatter

The EMFormatQuote class is a subclass of EMFormat, and is used as generator for quoted mail content and for inline-forwarding. This formatter converts message objects into a pure HTML stream, which is not parsed directly, but normally fed to the message composer.

Name	`EMFormatQuote`
Target	EMFormatHookTarget

5.6. Mail Formatter Targets

There is only one target for all mail formatters, and it is implied automatically for all formatter hooks.

Structure	EMFormatHookTarget

Flags	`inline` = `EM_FORMAT_HANDLER_INLINE`
	`inline_disposition` = `EM_FORMAT_HANDLER_INLINE_DISPOSITION`

Chapter 5. Contacts Hooks

Hooks available in the the contacts component.

1. Popup menus

The contacts popup menu class is `org.gnome.evolution.addressbook.popup:1.0`.

1.1. Calendar Popup Targets

TBD

2. Main menus

The addressbook menu class is `org.gnome.evolution.addressbook.bonobomenu:1.0`.

2.1. Contacts Menu Targets

TBD

3. Config Windows and Druids

The addressbook config class is `org.gnome.evolution.addressbook.config:1.0`.

3.1. Contacts Config Targets

TBD

4. Events

None defined.

Chapter 6. Calendar Hooks

Hooks available in the the calendar component.

1. Popup menus

The calendar popup menu class is `org.gnome.evolution.calendar.popup:1.0`.

1.1. Calendar Main View Context Menu

The context menu on the main calendar view. This menu applies to all view types.

Id	org.gnome.evolution.calendar.view.popup
Class	org.gnome.evolution.calendar.popup:1.0
Target	ECalPopupTargetSelect
Defined	calendar/gui/e-calendar-view.c:1428

1.2. Calendar Source Selector Context Menu

The context menu on the source selector in the calendar window.

Id	org.gnome.evolution.calendar.source.popup
Class	org.gnome.evolution.calendar.popup:1.0
Target	ECalPopupTargetSource
Defined	calendar/gui/calendar-component.c:409

1.3. Calendar Main View Context Menu

The context menu on the main calendar view. This menu applies to all view types.

Id	org.gnome.evolution.calendar.view.popup
Class	org.gnome.evolution.calendar.popup:1.0
Target	ECalPopupTargetSelect
Defined	calendar/gui/e-calendar-view.c:1428

1.4. Tasks Source Selector Context Menu

The context menu on the source selector in the tasks window.

Id	org.gnome.evolution.tasks.source.popup
Class	org.gnome.evolution.calendar.popup:1.0
Target	ECalPopupTargetSource
Defined	calendar/gui/tasks-component.c:354

1.5. Calendar Popup Targets

TBD

2. Main menus

The calendar menu class is `org.gnome.evolution.calendar.bonobomenu:1.0`.

2.1. Calendar Menu Targets

TBD

3. Config Windows and Druids

The calendar config class is `org.gnome.evolution.calendar.config:1.0`.

3.1. Calendar Config Targets

TBD

4. Events

None defined.

Chapter 7. Shell Hooks

1. Main menus

The mail menu class is `org.gnome.evolution.shell.bonobomenu:1.0`.

The plugin callback data will be the target matching the plugin menu itself, and the callback returns no value.

1.1. Shell Main Menu

This hook point is used to add bonobo menu's to the main evolution shell window, used for global commands not requiring a specific component.

Id	org.gnome.evolution.shell
Type	ESMenu
Target	ESMenuTargetShell
Defined	shell/e-shell-window.c:765

2. Events

The shell event class is `org.gnome.evolution.shell.events:1.0`.

2.1. Shell online state changed

This event is emitted whenever the shell online state changes.

Only the online and offline states are emitted.

Id	state.changed
Target	ESMenuTargetState
Defined	shell/e-shell.c:1083

Part III. Reference

This section of the book is a detailed API reference of the objects and methods that implement the core plugin system and hooks.

It contains the detailed information required for all uses of the plugin system. That is, implementors of new hook types, application developers providing hook points, and plugin developers.

Table of Contents

Chapter 8. EPlugin

The EPlugin object manages the loading and invocation of physical plugin definitions and plugin binaries. The base EPlugin class is an abstract class which loads plugin definitons, resolving hooks, and provides an api for invoking callbacks.

The EPluginLib object is a concrete derived class of EPlugin which handles loading shared libraries using the GModule interface.

Name

struct _EPlugin -- An EPlugin instance.

struct _EPlugin

Synopsis

```
struct _EPlugin {
  GObject object;
  char * id;
  char * path;
  GSList * hooks_pending;
  char * description;
  char * name;
  char * domain;
  GSList * hooks;
  int enabled:1;
};
```

Members

object	Superclass.
id	Unique identifier for plugin instance.
path	Filename where the xml definition resides.
hooks_pending	A list hooks which can't yet be loaded. This is the xmlNodePtr to the root node of the hook definition.
description	A description of the plugin's purpose.
name	The name of the plugin.
domain	The translation domain for this plugin.
hooks	A list of the EPluginHooks this plugin requires.
enabled	Whether the plugin is enabled or not. This is not fully implemented.

Description

The base EPlugin object is used to represent each plugin directly. All of the plugin's hooks are loaded and managed through this object.

Description

The base EPlugin object is used to represent each plugin directly. All of the plugin's hooks are loaded and managed through this object.

Name

struct _EPluginClass --

struct _EPluginClass

Synopsis

```
struct _EPluginClass {
  GObjectClass class;
  const char * type;
  int (* construct (EPlugin *, xmlNodePtr root);
  void *(* invoke (EPlugin *, const char *name, void *data);
};
```

Members

class	Superclass.
type	The plugin type. This is used by the plugin loader to determine which plugin object to instantiate to handle the plugin. This must be overriden by each subclass to provide a unique name.
construct	The construct virtual method scans the XML tree to initialise itself.
invoke	The invoke virtual method loads the plugin code, resolves the function name, and marshals a simple pointer to execute the plugin.

Description

The EPluginClass represents each plugin type. The type of each class is registered in a global table and is used to instantiate a container for each plugin.

It provides two main functions, to load the plugin definition, and to invoke a function. Each plugin class is used to handle mappings to different languages.

Description

The EPluginClass represents each plugin type. The type of each class is registered in a global table and is used to instantiate a container for each plugin.

It provides two main functions, to load the plugin definition, and to invoke a function. Each plugin class is used to handle mappings to different languages.

Name

struct _EPluginLib --

struct _EPluginLib

Synopsis

```
struct _EPluginLib {
  EPlugin plugin;
  char * location;
  GModule * module;
};
```

Members

plugin	Superclass.
location	The filename of the shared object.
module	The GModule once it is loaded.

Description

This is a concrete EPlugin class. It loads and invokes dynamically loaded libraries using GModule. The shared object isn't loaded until the first callback is invoked.

When the plugin is loaded, and if it exists, “e_plugin_lib_enable” will be invoked to initialise the

Description

This is a concrete EPlugin class. It loads and invokes dynamically loaded libraries using GModule. The shared object isn't loaded until the first callback is invoked.

When the plugin is loaded, and if it exists, “e_plugin_lib_enable” will be invoked to initialise the

Name

struct _EPluginLibClass --

struct _EPluginLibClass

Synopsis

```
struct _EPluginLibClass {
  EPluginClass plugin_class;
};
```

Members

plugin_class	Superclass.

Description

The plugin library needs no additional class data.

Description

The plugin library needs no additional class data.

Name

struct _EPluginHookTargetKey --

struct _EPluginHookTargetKey

Synopsis

```
struct _EPluginHookTargetKey {
  const char * key;
  guint32 value;
};
```

Members

key	Enumeration value as a string.
value	Enumeration value as an integer.

Description

A multi-purpose string to id mapping structure used with various helper functions to simplify plugin hook subclassing.

Description

A multi-purpose string to id mapping structure used with various helper functions to simplify plugin hook subclassing.

Name

struct _EPluginHookTargetMap --

struct _EPluginHookTargetMap

Synopsis

```
struct _EPluginHookTargetMap {
  const char * type;
  int id;
  const struct _EPluginHookTargetKey * mask_bits;
};
```

Members

type	The string id of the target.
id	The integer id of the target. Maps directly to the type field of the various plugin type target id's.
mask_bits	A zero-fill terminated array of EPluginHookTargetKeys.

Description

Used by EPluginHook to define mappings of target type enumerations to and from strings. Also used to define the mask option names when reading the XML plugin hook definitions.

Description

Used by EPluginHook to define mappings of target type enumerations to and from strings. Also used to define the mask option names when reading the XML plugin hook definitions.

Name

struct _EPluginHook -- A plugin hook.

struct _EPluginHook

Synopsis

```
struct _EPluginHook {
  GObject object;
  struct _EPlugin * plugin;
};
```

Members

object	Superclass.
plugin	The parent object.

Description

An EPluginHook is used as a container for each hook a given plugin is listening to.

Description

An EPluginHook is used as a container for each hook a given plugin is listening to.

Name

struct _EPluginHookClass --

struct _EPluginHookClass

Synopsis

```
struct _EPluginHookClass {
  GObjectClass class;
  const char * id;
  int (* construct (EPluginHook *eph, EPlugin *ep, xmlNodePtr root);
  void (* enable (EPluginHook *eph, int state);
};
```

Members

class	Superclass.
id	The plugin hook type. This must be overriden by each subclass and is used as a key when loading hook definitions. This string
construct	Virtual method used to initialise the object when loaded.
enable	Virtual method used to enable or disable the hook.

Description

The EPluginHookClass represents each hook type. The type of the class is registered in a global table and is used to instantiate a container for each hook.

should contain a globally unique name followed by a

and a version specification. This is to ensure plugins only hook into hooks with the right API.

Description

The EPluginHookClass represents each hook type. The type of the class is registered in a global table and is used to instantiate a container for each hook.

Name

e_plugin_get_type --

e_plugin_get_type

Synopsis

```
GType e_plugin_get_type (void);
void;
```

Arguments

void no arguments

Description

Standard GObject type function. This is only an abstract class, so you can only use this to subclass EPlugin.

Return value

The type.

Name

e_plugin_add_load_path --

e_plugin_add_load_path

Synopsis

```
void e_plugin_add_load_path (path);
const char * path;
```

Arguments

path The path to add to search for plugins.

Description

Add a path to be searched when `e_plugin_load_plugins` is called. By default the system plugin directory and ~/.eplugins is used as the search path unless overriden by the environmental variable `EVOLUTION_PLUGIN_PATH`.

Name

e_plugin_load_plugins --

e_plugin_load_plugins

Synopsis

```
int e_plugin_load_plugins (void);
void;
```

Arguments

void	no arguments

Description

Scan the search path, looking for plugin definitions, and load them into memory.

Return value

Returns -1 if an error occured.

Name

e_plugin_register_type --

e_plugin_register_type

Synopsis

```
void e_plugin_register_type (type);
GType type;
```

Arguments

type	The GObject type of the plugin loader.

Description

Register a new plugin type with the plugin system. Each type must subclass EPlugin and must override the type member of the EPluginClass with a unique name.

Name

e_plugin_construct --

e_plugin_construct

Synopsis

```
int e_plugin_construct (ep, root);
EPlugin * ep;
xmlNodePtr root;
```

Arguments

ep	An EPlugin derived object.
root	The XML root node of the sub-tree containing the plugin definition.

Description

Helper to invoke the construct virtual method.

Return value

The return from the construct virtual method.

Name

e_plugin_invoke --

e_plugin_invoke

Synopsis

```
void * e_plugin_invoke (ep, name, data);
EPlugin * ep;
const char * name;
void * data;
```

Arguments

ep	
name	The name of the function to invoke. The format of this name will depend on the EPlugin type and its language conventions.
data	The argument to the function. Its actual type depends on the hook on which the function resides. It is up to the called function to get this right.

Description

Helper to invoke the invoke virtual method.

Return value

The return of the plugin invocation.

Name

e_plugin_enable --

e_plugin_enable

Synopsis

```
void e_plugin_enable (ep, state);
EPlugin * ep;
int state;
```

Arguments

ep
state

Description

Set the enable state of a plugin.

THIS IS NOT FULLY IMPLEMENTED YET

Name

e_plugin_xml_prop --

e_plugin_xml_prop

Synopsis

```
char * e_plugin_xml_prop (node, id);
xmlNodePtr node;
const char * id;
```

Arguments

node	An XML node.
id	The name of the property to retrieve.

Description

A static helper function to look up a property on an XML node, and ensure it is allocated in GLib system memory. If GLib isn't using the system malloc then it must copy the property value.

Return value

The property, allocated in GLib memory, or NULL if no such property exists.

Name

e_plugin_xml_prop_domain --

e_plugin_xml_prop_domain

Synopsis

```
char * e_plugin_xml_prop_domain (node, id, domain);
xmlNodePtr node;
const char * id;
const char * domain;
```

Arguments

node	An XML node.
id	The name of the property to retrieve.
domain	The translation domain for this string.

Description

A static helper function to look up a property on an XML node, and translate it based on *domain*.

Return value

The property, allocated in GLib memory, or NULL if no such property exists.

Name

e_plugin_xml_int --

e_plugin_xml_int

Synopsis

```
int e_plugin_xml_int (node, id, def);
xmlNodePtr node;
const char * id;
int def;
```

Arguments

node	An XML node.
id	The name of the property to retrieve.
def	A default value if the property doesn't exist. Can be used to determine if the property isn't set.

Description

A static helper function to look up a property on an XML node as an integer. If the property doesn't exist, then *def* is returned as a default value instead.

Return value

The value if set, or *def* if not.

Name

e_plugin_xml_content --

e_plugin_xml_content

Synopsis

```
char * e_plugin_xml_content (node);
xmlNodePtr node;
```

Arguments

node

Description

A static helper function to retrieve the entire textual content of an XML node, and ensure it is allocated in GLib system memory. If GLib isn't using the system malloc them it must copy the content.

Return value

The node content, allocated in GLib memory.

Name

e_plugin_xml_content_domain --

e_plugin_xml_content_domain

Synopsis

```
char * e_plugin_xml_content_domain (node, domain);
xmlNodePtr node;
const char * domain;
```

Arguments

node
domain

Description

A static helper function to retrieve the entire textual content of an XML node, and ensure it is allocated in GLib system memory. If GLib isn't using the system malloc them it must copy the content.

Return value

The node content, allocated in GLib memory.

Name

e_plugin_lib_get_type --

e_plugin_lib_get_type

Synopsis

```
GType e_plugin_lib_get_type (void);
void;
```

Arguments

void no arguments

Description

Standard GObject function to retrieve the EPluginLib type. Use to register the type with the plugin system if you want to use shared library plugins.

Return value

The EPluginLib type.

Name

e_plugin_hook_get_type --

e_plugin_hook_get_type

Synopsis

```
GType e_plugin_hook_get_type (void);
void;
```

Arguments

void no arguments

Description

Standard GObject function to retrieve the EPluginHook type. Since EPluginHook is an abstract class, this is only used to subclass it.

Return value

The EPluginHook type.

Name

e_plugin_hook_enable --

e_plugin_hook_enable

Synopsis

```
void **e_plugin_hook_enable** (eph, state);
EPluginHook * eph;
int state;
```

Arguments

eph
state

Description

Set the enabled state of the plugin hook. This is called by the plugin code.

THIS IS NOT FULY IMEPLEMENTED YET

Name

e_plugin_hook_register_type --

e_plugin_hook_register_type

Synopsis

```
void e_plugin_hook_register_type (type);
GType type;
```

Arguments

type

Description

Register a new plugin hook type with the plugin system. Each type must subclass EPluginHook and must override the id member of the EPluginHookClass with a unique identification string.

Name

e_plugin_hook_mask --

e_plugin_hook_mask

Synopsis

```
guint32 e_plugin_hook_mask (root, map, prop);
xmlNodePtr root;
const struct _EPluginHookTargetKey * map;
const char * prop;
```

Arguments

root	An XML node.
map	A zero-fill terminated array of EPluginHookTargeKeys used to map a string with a bit value.
prop	The property name.

Description

This is a static helper function which looks up a property *prop* on the XML node *root*, and then uses the *map* table to convert it into a bitmask. The property value is a comma separated list of enumeration strings which are indexed into the *map* table.

Return value

A bitmask representing the inclusive-or of all of the integer values of the corresponding string id's stored in the *map*.

Name

e_plugin_hook_id --

e_plugin_hook_id

Synopsis

```
guint32 e_plugin_hook_id (root, map, prop);
xmlNodePtr root;
const struct _EPluginHookTargetKey * map;
const char * prop;
```

Arguments

root
map
prop

Description

This is a static helper function which looks up a property *prop* on the XML node *root*, and then uses the *map* table to convert it into an integer.

This is used as a helper wherever you need to represent an enumerated value in the XML.

Return value

If the *prop* value is in *map*, then the corresponding integer value, if not, then ~0.

Chapter 9. EPopup

The EPopup object manages a single popup menu. It is used to application code as a convenience function for building dynamic popup menus based on a specific context.

The EPopupHook object is loaded by the EPlugin system, and is used to provide dynamic extension to the application context menus.

Name

enum _e_popup_t -- Popup item type enumeration.

enum _e_popup_t

Synopsis

```
enum _e_popup_t {
  E_POPUP_ITEM,
  E_POPUP_TOGGLE,
  E_POPUP_RADIO,
  E_POPUP_IMAGE,
  E_POPUP_SUBMENU,
  E_POPUP_BAR,
  E_POPUP_TYPE_MASK,
  E_POPUP_ACTIVE
};
```

Constants

E_POPUP_ITEM	A simple menu item.
E_POPUP_TOGGLE	A toggle menu item.
E_POPUP_RADIO	A radio menu item. Note that the radio group is global for the entire (sub) menu. i.e. submenu's must be used to separate radio button menu items.
E_POPUP_IMAGE	A &GtkImage menu item. In this case the *`image`* field of &struct _EPopupItem points to the &GtkImage directly.
E_POPUP_SUBMENU	A sub-menu header. It is up to the application to define the *`path`* properly so that the submenu comes before the submenu items.
E_POPUP_BAR	A menu separator bar.
E_POPUP_TYPE_MASK	Mask used to separate item type from option bits.
E_POPUP_ACTIVE	An option bit to signify that the radio button or toggle button is active.

Name

struct _EPopupItem -- A popup menu item definition.

struct _EPopupItem

Synopsis

```
struct _EPopupItem {
  enum _e_popup_t type;
  char * path;
  char * label;
  EPopupActivateFunc activate;
  void * user_data;
  void * image;
  guint32 visible;
  guint32 enable;
};
```

Members

type	The type of the popup. See the &enum _epopup_t definition for possible values.
path	An absolute path, which when sorted using a simple ASCII sort, will put the menu item in the right place in the menu heirarchy. '/' is used to separate menus from submenu items.
label	The text of the menyu item.
activate	A function conforming to &EPopupActivateFunc which will be called when the menu item is activated.
user_data	Extra per-item user-data available to the application. This is not passed to the *`data`* field of *`activate`*.
image	For most types, the name of the icon in the icon theme to display next to the menu item, if required. For the `E_POPUP_IMAGE` type, it is a pointer to the &GtkWidget instead.
visible	Visibility mask. Used together with the &EPopupTarget mask to determine if the item should be part of the menu or not.
enable	Sensitivity mask. Similar to the visibility mask, but currently unimplemented.

Description

The EPopupItem defines a single popup menu item, or submenu item, or menu separator based on the *`type`*. Any number of these are merged at popup display type to form the popup menu.

The application may extend this structure using simple C structure containers to add any additional fields it may require.

Name

struct _EPopupTarget -- A popup menu target definition.

struct _EPopupTarget

Synopsis

```
struct _EPopupTarget {
  struct _EPopup * popup;
  struct _GtkWidget * widget;
  guint32 type;
  guint32 mask;
};
```

Members

popup	The parent popup object, used for virtual methods on the target.
widget	The parent widget, where available. In some cases the type of this object is part of the published api for the target.
type	The target type. This will be defined by the implementation.
mask	Target mask. This is used to sensitise and show items based on their definition in EPopupItem.

Description

An EPopupTarget defines the context for a specific popup menu instance. The root target object is abstract, and it is up to sub-classes of &EPopup to define the additional fields required to make it usable.

Description

An EPopupTarget defines the context for a specific popup menu instance. The root target object is abstract, and it is up to sub-classes of &EPopup to define the additional fields required to make it usable.

Name

struct _EPopup -- A Popup menu manager.

struct _EPopup

Synopsis

```
struct _EPopup {
  GObject object;
  struct _EPopupPrivate * priv;
  char * menuid;
  EPopupTarget * target;
};
```

Members

object	Superclass, GObject.
priv	Private data.
menuid	The id of this menu instance.
target	The current target during the display of the popup menu.

Description

The EPopup manager object. Each popup menu is built using this one-off object which is created each time the popup is invoked.

Description

The EPopup manager object. Each popup menu is built using this one-off object which is created each time the popup is invoked.

Name

struct _EPopupClass --

struct _EPopupClass

Synopsis

```
struct _EPopupClass {
  GObjectClass object_class;
  EDList factories;
  void (* target_free (EPopup *ep, EPopupTarget *t);
};
```

Members

object_class	Superclass type.
factories	A list of factories for this particular class of popup menu.
target_free	Virtual method to free the popup target. The base class frees the allocation and unrefs the popup pointer structure.

Description

The EPopup class definition. This should be sub-classed for each component that wants to provide hookable popup menus. The sub-class only needs to know how to allocate and free the various target types it supports.

Description

The EPopup class definition. This should be sub-classed for each component that wants to provide hookable popup menus. The sub-class only needs to know how to allocate and free the various target types it supports.

Name

struct _EPopupHookMenu --

struct _EPopupHookMenu

Synopsis

```
struct _EPopupHookMenu {
  struct _EPopupHook * hook;
  char * id;
  int target_type;
  GSList * items;
};
```

Members

hook	Parent pointer.
id	The identifier of the menu to which these items belong.
target_type	The target number of the type of target these menu items expect. It will generally also be defined by the menu id.
items	A list of EPopupItems.

Description

The structure used to keep track of all of the items that a plugin wishes to add to a given menu. This is used internally by a factory method set on EPlugin to add the right menu items to a given menu.

Description

The structure used to keep track of all of the items that a plugin wishes to add to a given menu. This is used internally by a factory method set on EPlugin to add the right menu items to a given menu.

Name

struct _EPopupHook -- A popup menu hook.

struct _EPopupHook

Synopsis

```
struct _EPopupHook {
  EPluginHook hook;
  GSList * menus;
};
```

Members

hook	Superclass.
menus	A list of EPopupHookMenus, for all menus registered on this hook type.

Description

The EPopupHook class loads and manages the meta-data required to map plugin definitions to physical menus.

Description

The EPopupHook class loads and manages the meta-data required to map plugin definitions to physical menus.

Name

struct _EPopupHookClass --

struct _EPopupHookClass

Synopsis

```
struct _EPopupHookClass {
  EPluginHookClass hook_class;
  GHashTable * target_map;
  EPopupClass * popup_class;
};
```

Members

hook_class	Superclass.
target_map	Table of EPluginHookTargetMaps which enumerate the target types and enable bits of the implementing class.
popup_class	The EPopupClass of the corresponding popup manager for the implementing class.

Description

The EPopupHookClass is a concrete class, however it is empty on its own. It needs to be sub-classed and initialised appropriately.

The EPluginHookClass.id must be set to the name and version of the hook handler itself. The *`target_map`* must be initialised with the data required to enumerate the target types and enable flags supported by the implementing class.

Description

The EPopupHookClass is a concrete class, however it is empty on its own. It needs to be sub-classed and initialised appropriately.

The EPluginHookClass.id must be set to the name and version of the hook handler itself. The *`target_map`* must be initialised with the data required to enumerate the target types and enable flags supported by the implementing class.

Name

e_popup_get_type --

e_popup_get_type

Synopsis

```
GType e_popup_get_type (void);
void;
```

Arguments

void no arguments

Description

Standard GObject type function.

Return value

The EPopup object type.

Name

e_popup_new -- Create an targetless popup menu manager.

e_popup_new

Synopsis

```
EPopup * e_popup_new (menuid);
const char * menuid;
```

Arguments

menuid Unique ID for this menu.

Description

Create a targetless popup menu object. This can be used as a helper for creating popup menu's with no target. Such popup menu's wont be very pluggable.

Return value

A new EPopup.

Name

e_popup_construct --

e_popup_construct

Synopsis

```
EPopup * e_popup_construct (ep, menuid);
EPopup * ep;
const char * menuid;
```

Arguments

ep	An instantiated but uninitialised EPopup.
menuid	The menu identifier.

Description

Construct the base popup instance with standard parameters.

Return value

Returns *ep*.

Name

e_popup_add_items --

e_popup_add_items

Synopsis

```
void e_popup_add_items (emp, items, freefunc, data);
EPopup * emp;
GSList * items;
EPopupItemsFunc freefunc;
void * data;
```

Arguments

emp	An EPopup derived object.
items	A list of EPopupItem's to add to the current popup menu.
freefunc	A function which will be called when the items are no longer needed.
data	user-data passed to *freefunc*, and passed to all activate methods.

Description

Add new EPopupItems to the menus. Any with the same path will override previously defined menu items, at menu building time. This may be called any number of times before the menu is built to create a complex heirarchy of menus.

Name

e_popup_create_menu --

e_popup_create_menu

Synopsis

```
GtkMenu * e_popup_create_menu (emp, target, mask);
EPopup * emp;
EPopupTarget * target;
guint32 mask;
```

Arguments

`emp`	An EPopup derived object.
`target`	popup target, if set, then factories will be invoked. This is then owned by the menu.
`mask`	If supplied, overrides the target specified mask or provides a mask if no target is supplied. Used to enable or show menu items.

Description

All of the menu items registered on *`emp`* are sorted by path, and then converted into a menu heirarchy.

Return value

A GtkMenu which can be popped up when ready.

Name

e_popup_create_menu_once --

e_popup_create_menu_once

Synopsis

```
GtkMenu * e_popup_create_menu_once (emp, target, mask);
EPopup * emp;
EPopupTarget * target;
guint32 mask;
```

Arguments

emp	EPopup, once the menu is shown, this cannot be considered a valid pointer.
target	If set, the target of the selection. Static menu items will be added. The target will be freed once complete.
mask	Enable/disable and visibility mask.

Description

Like popup_create_menu, but automatically sets up the menu so that it is destroyed once a selection takes place, and the EPopup is unreffed. This is the normal entry point as it automates most memory management for popup menus.

Return value

A menu, to popup.

Name

e_popup_class_add_factory --

e_popup_class_add_factory

Synopsis

```
EPopupFactory * e_popup_class_add_factory (klass, menuid, func, data);
EPopupClass * klass;
const char * menuid;
EPopupFactoryFunc func;
void * data;
```

Arguments

klass	The EPopup derived class which you're interested in.
menuid	The identifier of the menu you're interested in, or NULL to be called for all menus on this class.
func	The factory called when the menu *menuid* is being created.
data	User-data for the factory callback.

Description

This is a class-static method used to register factory callbacks against specific menu's.

The factory method will be invoked before the menu is created. This way, the factory may add any additional menu items it wishes based on the context supplied in the *target*.

Return value

A handle to the factory which can be used to remove it later.

Name

e_popup_class_remove_factory --

e_popup_class_remove_factory

Synopsis

```
void e_popup_class_remove_factory (klass, f);
EPopupClass * klass;
EPopupFactory * f;
```

Arguments

klass	The EPopup derived class.
f	The factory handle returned by `e_popup_class_add_factory`.

Description

Remove a popup menu factory. If it has not been added, or it has already been removed, then the result is undefined (i.e. it will crash).

Generally factories are static for the life of the application, and so do not need to be removed.

Name

e_popup_target_new --

e_popup_target_new

Synopsis

```
void * e_popup_target_new (ep, type, size);
EPopup * ep;
int type;
size_t size;
```

Arguments

ep	An EPopup derived object.
type	type, defined by the implementing class.
size	The size of memory to allocate for the target. It must be equal or greater than the size of EPopupTarget.

Description

Allocate a new popup target suitable for this popup type.

Name

e_popup_target_free --

e_popup_target_free

Synopsis

```
void e_popup_target_free (ep, o);
EPopup * ep;
void * o;
```

Arguments

ep An EPopup derived object.

o The target, previously allocated by `e_popup_target_new`.

Description

Free the target against *ep*. Note that targets are automatically freed if they are passed to the menu creation functions, so this is only required if you are using the target for other purposes.

Name

e_popup_hook_get_type --

e_popup_hook_get_type

Synopsis

```
GType e_popup_hook_get_type (void);
void;
```

Arguments

void no arguments

Description

Standard GObject function to get the object type. Used to subclass EPopupHook.

Return value

The type of the popup hook class.

Name

e_popup_hook_class_add_target_map --

e_popup_hook_class_add_target_map

Synopsis

```
void e_popup_hook_class_add_target_map (klass, map);
EPopupHookClass * klass;
const EPopupHookTargetMap * map;
```

Arguments

klass	The derived EPopupHook class.
map	A map used to describe a single EPopupTarget type for this class.

Description

Add a target map to a concrete derived class of EPopup. The target map enumerates a single target type and the enable mask bit names, so that the type can be loaded automatically by the EPopup class.

Name

enum _em_popup_target_t -- A list of mail popup target types.

enum _em_popup_target_t

Synopsis

```
enum _em_popup_target_t {
  EM_POPUP_TARGET_SELECT,
  EM_POPUP_TARGET_URI,
  EM_POPUP_TARGET_PART,
  EM_POPUP_TARGET_FOLDER,
  EM_POPUP_TARGET_ATTACHMENTS
};
```

Constants

EM_POPUP_TARGET_SELECT	A selection of messages.
EM_POPUP_TARGET_URI	A URI.
EM_POPUP_TARGET_PART	A CamelMimePart message part.
EM_POPUP_TARGET_FOLDER	A folder URI.
EM_POPUP_TARGET_ATTACHMENTS	A list of attachments.

Description

Defines the value of the targetid for all EMPopup target types.

Description

Defines the value of the targetid for all EMPopup target types.

Name

enum _em_popup_target_select_t -- EMPopupTargetSelect qualifiers.

enum _em_popup_target_select_t

Synopsis

```
enum _em_popup_target_select_t {
  EM_POPUP_SELECT_ONE,
  EM_POPUP_SELECT_MANY,
  EM_POPUP_SELECT_MARK_READ,
  EM_POPUP_SELECT_MARK_UNREAD,
  EM_POPUP_SELECT_DELETE,
  EM_POPUP_SELECT_UNDELETE,
  EM_POPUP_SELECT_MAILING_LIST,
  EM_POPUP_SELECT_EDIT,
  EM_POPUP_SELECT_MARK_IMPORTANT,
  EM_POPUP_SELECT_MARK_UNIMPORTANT,
  EM_POPUP_SELECT_FLAG_FOLLOWUP,
  EM_POPUP_SELECT_FLAG_COMPLETED,
  EM_POPUP_SELECT_FLAG_CLEAR,
  EM_POPUP_SELECT_ADD_SENDER,
  EM_POPUP_SELECT_MARK_JUNK,
  EM_POPUP_SELECT_MARK_NOJUNK,
  EM_POPUP_SELECT_FOLDER,
  EM_POPUP_SELECT_LAST
};
```

Constants

EM_POPUP_SELECT_ONE	Only one item is selected.
EM_POPUP_SELECT_MANY	One ore more items are selected.
EM_POPUP_SELECT_MARK_READ	Message(s) are unseen and can be marked seen.
EM_POPUP_SELECT_MARK_UNREAD	Message(s) are seen and can be marked unseen.
EM_POPUP_SELECT_DELETE	Message(s) are undeleted and can be marked deleted.
EM_POPUP_SELECT_UNDELETE	Message(s) are deleted and can be undeleted.
EM_POPUP_SELECT_MAILING_LIST	If one message is selected, and it contains a message list tag.
EM_POPUP_SELECT_EDIT	The message can be opened for editing (the folder is a sent folder).
EM_POPUP_SELECT_MARK_IMPORTANT	Message(s) are not marked important.
EM_POPUP_SELECT_MARK_UNIMPORTANT	Message(s) are marked important.
EM_POPUP_SELECT_FLAG_FOLLOWUP	Message(s) are not flagged for followup.

MPLETED	Message(s) are not flagged completed.
EM_POPUP_SELECT_FLAG_CLEAR	Message(s) are flagged for followup.
EM_POPUP_SELECT_ADD_SENDER	The message contains sender addresses which might be added to the addressbook. i.e. it isn't a message in the Sent or Drafts folders.
EM_POPUP_SELECT_MARK_JUNK	Message(s) are not marked as junk.
EM_POPUP_SELECT_MARK_NOJUNK	Message(s) are marked as junk.
EM_POPUP_SELECT_FOLDER	A folder is set on the selection.
EM_POPUP_SELECT_LAST	The last bit used, can be used to add additional types from derived application code.

Description

Name

enum _em_popup_target_uri_t -- EMPopupTargetURI qualifiers.

enum _em_popup_target_uri_t

Synopsis

```
enum _em_popup_target_uri_t {
  EM_POPUP_URI_HTTP,
  EM_POPUP_URI_MAILTO,
  EM_POPUP_URI_NOT_MAILTO
};
```

Constants

EM_POPUP_URI_HTTP	This is a HTTP or HTTPS url.
EM_POPUP_URI_MAILTO	This is a MAILTO url.
EM_POPUP_URI_NOT_MAILTO	This is not a MAILTO url.

Description

Name

enum _em_popup_target_part_t -- EMPopupTargetPart qualifiers.

enum _em_popup_target_part_t

Synopsis

```
enum _em_popup_target_part_t {
  EM_POPUP_PART_MESSAGE,
  EM_POPUP_PART_IMAGE
};
```

Constants

EM_POPUP_PART_MESSAGE	This is a message type.
EM_POPUP_PART_IMAGE	This is an image type.

Description

Name

enum _em_popup_target_folder_t -- EMPopupTargetFolder qualifiers.

enum _em_popup_target_folder_t

Synopsis

```
enum _em_popup_target_folder_t {
  EM_POPUP_FOLDER_FOLDER,
  EM_POPUP_FOLDER_STORE,
  EM_POPUP_FOLDER_INFERIORS,
  EM_POPUP_FOLDER_DELETE,
  EM_POPUP_FOLDER_SELECT
};
```

Constants

EM_POPUP_FOLDER_FOLDER	This is a normal folder.
EM_POPUP_FOLDER_STORE	This is a store.
EM_POPUP_FOLDER_INFERIORS	This folder may have child folders.
EM_POPUP_FOLDER_DELETE	This folder can be deleted or renamed.
EM_POPUP_FOLDER_SELECT	This folder exists and can be selected or opened.

Description

Name

enum _em_popup_target_attachments_t -- EMPopupTargetAttachments qualifiers.

enum _em_popup_target_attachments_t

Synopsis

```
enum _em_popup_target_attachments_t {
  EM_POPUP_ATTACHMENTS_ONE,
  EM_POPUP_ATTACHMENTS_MANY
};
```

Constants

EM_POPUP_ATTACHMENTS_ONE	There is one and only one attachment selected.
EM_POPUP_ATTACHMENTS_MANY	There is one or more attachments selected.

Description

Name

struct _EMPopupTargetURI -- An inline URI.

struct _EMPopupTargetURI

Synopsis

```
struct _EMPopupTargetURI {
  EPopupTarget target;
  char * uri;
};
```

Members

target	Superclass.
uri	The encoded URI to which this target applies.

Description

Used to represent popup-menu context on any URI object.

Description

Used to represent popup-menu context on any URI object.

Name

struct _EMPopupTargetSelect -- A list of messages.

struct _EMPopupTargetSelect

Synopsis

```
struct _EMPopupTargetSelect {
  EPopupTarget target;
  struct _CamelFolder * folder;
  char * uri;
  GPtrArray * uids;
};
```

Members

target	Superclass.
folder	The CamelFolder of the selected messages.
uri	The encoded URI representing this folder.
uids	An array of UID strings of messages within *`folder`*.

Description

Used to represent a selection of messages as context for a popup menu. All items may be NULL if the current view has no active folder selected.

Description

Used to represent a selection of messages as context for a popup menu. All items may be NULL if the current view has no active folder selected.

Name

struct _EMPopupTargetPart -- A Camel object.

struct _EMPopupTargetPart

Synopsis

```
struct _EMPopupTargetPart {
  EPopupTarget target;
  char * mime_type;
  struct _CamelMimePart * part;
};
```

Members

target	Superclass.
mime_type	MIME type of the part. This may be a calculated type not matching the *`part`*'s MIME type.
part	A CamelMimePart representing a message or attachment.

Description

Used to represent a message part as context for a popup menu. This is used for both attachments and in-line-images.

Description

Used to represent a message part as context for a popup menu. This is used for both attachments and in-line-images.

Name

struct _EMPopupTargetFolder -- A folder uri.

struct _EMPopupTargetFolder

Synopsis

```
struct _EMPopupTargetFolder {
  EPopupTarget target;
  char * uri;
};
```

Members

target	Superclass.
uri	A folder URI.

Description

This target is used to represent folder context.

Description

This target is used to represent folder context.

Name

struct _EMPopupTargetAttachments -- A list of composer attachments.

struct _EMPopupTargetAttachments

Synopsis

```
struct _EMPopupTargetAttachments {
  EPopupTarget target;
  GSList * attachments;
};
```

Members

target	Superclass.
attachments	A GSList list of EMsgComposer attachments.

Description

This target is used to represent a selected list of attachments in the message composer attachment area.

Description

This target is used to represent a selected list of attachments in the message composer attachment area.

Name

em_popup_target_new_select --

em_popup_target_new_select

Synopsis

```
EMPopupTargetSelect * em_popup_target_new_select (emp, folder,
folder_uri, uids);
EMPopup * emp;
struct _CamelFolder * folder;
const char * folder_uri;
GPtrArray * uids;
```

Arguments

emp	-- undescribed --
folder	The selection will ref this for the life of it.
folder_uri	
uids	The selection will free this when done with it.

Description

Create a new selection popup target.

Return value

Name

em_popup_target_new_attachments --

em_popup_target_new_attachments

Synopsis

```
EMPopupTargetAttachments * em_popup_target_new_attachments (emp, at-
tachments);
EMPopup * emp;
GSList * attachments;
```

Arguments

emp	
attachments	A list of EMsgComposerAttachment objects, reffed for the list. Will be unreff'd once finished with.

Description

Owns the list *attachments* and their items after they're passed in.

Return value

Chapter 10. EMenu

The EMenu object manages the menus for a given view or component. It is used by application code to allow the plugin system an entry point to current application view. It may also be used by the application as a convenience function to dynamically alter the menu system based on user context.

The EMenuHook object is loaded by the EPlugin system, and is used to provide dynamic extension to the application menus.

Name

enum _e_menu_t -- Menu item type.

enum _e_menu_t

Synopsis

```
enum _e_menu_t {
  E_MENU_ITEM,
  E_MENU_TOGGLE,
  E_MENU_RADIO,
  E_MENU_TYPE_MASK,
  E_MENU_ACTIVE
};
```

Constants

E_MENU_ITEM	Normal menu item.
E_MENU_TOGGLE	Toggle menu item.
E_MENU_RADIO	unimplemented.
E_MENU_TYPE_MASK	Mask used to separate item type from option bits.
E_MENU_ACTIVE	Whether a toggle item is active.

Description

The type of menu items which are supported by the menu system.

Description

The type of menu items which are supported by the menu system.

Name

struct _EMenuItem -- A BonoboUI menu item.

struct _EMenuItem

Synopsis

```
struct _EMenuItem {
  enum _e_menu_t type;
  char * path;
  char * verb;
  GCallback activate;
  void * user_data;
  guint32 visible;
  guint32 enable;
};
```

Members

type	Menu item type. `E_MENU_ITEM` or `E_MENU_TOGGLE`.
path	BonoboUI Path to the menu item.
verb	BonoboUI verb for the menu item.
activate	Callback when the menu item is selected. This will be a EMenuToggleActivateFunc for toggle items or EMenuActivateFunc for normal items.
user_data	User data for item.
visible	Visibility mask, unimplemented.
enable	Sensitivity mask, combined with the target mask.

Description

An EMenuItem defines a single menu item. This menu item is used to hook onto callbacks from the bonobo menus, but not to build or merge the menu itself.

Description

An EMenuItem defines a single menu item. This menu item is used to hook onto callbacks from the bonobo menus, but not to build or merge the menu itself.

Name

struct _EMenuPixmap -- A menu icon holder.

struct _EMenuPixmap

Synopsis

```
struct _EMenuPixmap {
  char * command;
  char * name;
  int size;
  char * pixmap;
};
```

Members

command	The path to the command or verb to which this pixmap belongs.
name	The name of the icon. Either an icon-theme name or the full pathname of the icon.
size	The e-icon-factory icon size.
pixmap	The pixmap converted to XML format. If not set, then EMenu will create it as required. This must be freed if set in the free function.

Description

Used to track all pixmap items used in menus. These need to be supplied separately from the menu definition.

Description

Used to track all pixmap items used in menus. These need to be supplied separately from the menu definition.

Name

struct _EMenuUIFile -- A meu UI file holder.

struct _EMenuUIFile

Synopsis

```
struct _EMenuUIFile {
  char * appdir;
  char * appname;
  char * filename;
};
```

Members

appdir	TODO; should this be handled internally.
appname	TODO; should this be handled internally.
filename	The filename of the BonoboUI XML menu definition.

Description

These values are passed directly to `bonobo_ui_util_set_ui` when the menu is activated.

Description

These values are passed directly to `bonobo_ui_util_set_ui` when the menu is activated.

Name

struct _EMenuTarget -- A BonoboUI menu target definition.

struct _EMenuTarget

Synopsis

```
struct _EMenuTarget {
  struct _EMenu * menu;
  struct _GtkWidget * widget;
  guint32 type;
  guint32 mask;
};
```

Members

menu	The parent menu object, used for virtual methods on the target.
widget	The parent widget where available. In some cases the type of this object is part of the published api for the target, in others it is merely a GtkWidget from which you can find the toplevel widget.
type	Target type. This will be defined by the implementation.
mask	Target mask. This is used to sensitise show items based on their definition in EMenuItem.

Description

An EMenuTarget defines the context for a specific view instance. It is used to enable and show menu items, and to provide contextual data to menu invocations.

Description

An EMenuTarget defines the context for a specific view instance. It is used to enable and show menu items, and to provide contextual data to menu invocations.

Name

struct _EMenu -- A BonoboUI menu manager object.

struct _EMenu

Synopsis

```
struct _EMenu {
  GObject object;
  struct _EMenuPrivate * priv;
  char * menuid;
  struct _BonoboUIComponent * uic;
  EMenuTarget * target;
};
```

Members

object	Superclass.
priv	Private data.
menuid	The id of this menu instance.
uic	The current BonoboUIComponent which stores the actual menu items this object manages.
target	The current target for the view.

Description

The EMenu manager object manages the mappings between EMenuItems and the BonoboUI menus loaded from UI files.

Description

The EMenu manager object manages the mappings between EMenuItems and the BonoboUI menus loaded from UI files.

Name

struct _EMenuClass --

struct _EMenuClass

Synopsis

```
struct _EMenuClass {
  GObjectClass object_class;
  EDList factories;
  void (* target_free (EMenu *ep, EMenuTarget *t);
};
```

Members

object_class	Superclass type.
factories	A list of factories for this particular class of main menu.
target_free	Virtual method to free the menu target. The base class free method frees the allocation and unrefs the EMenu parent pointer.

Description

The EMenu class definition. This should be sub-classed for each component that wants to provide hookable main menus. The subclass only needs to know how to allocate and free the various target types it supports.

Description

The EMenu class definition. This should be sub-classed for each component that wants to provide hookable main menus. The subclass only needs to know how to allocate and free the various target types it supports.

Name

struct _EMenuHookMenu -- A group of items targetting a specific menu.

struct _EMenuHookMenu

Synopsis

```
struct _EMenuHookMenu {
  struct _EMenuHook * hook;
  char * id;
  int target_type;
  GSList * items;
  GSList * uis;
  GSList * pixmaps;
};
```

Members

hook	Parent pointer.
id	The identifier of the menu or view to which these items belong.
target_type	The target number of the type of target these menu items expect. This will be defined by menu itself.
items	A list of EMenuItems.
uis	A list of filenames of the BonoboUI files that need to be loaded for an active view.
pixmaps	A list of EMenuHookPixmap structures for the menus.

Description

This structure is used to keep track of all of the items that a plugin wishes to add to specific menu. This is used internally by a factory method defined by the EMenuHook to add the right menu items to a given view.

Description

This structure is used to keep track of all of the items that a plugin wishes to add to specific menu. This is used internally by a factory method defined by the EMenuHook to add the right menu items to a given view.

Name

struct _EMenuHook -- A BonoboUI menu hook.

struct _EMenuHook

Synopsis

```
struct _EMenuHook {
  EPluginHook hook;
  GSList * menus;
};
```

Members

hook	Superclass.
menus	A list of EMenuHookMenus for all menus registered on this hook type.

Description

The EMenuHook class loads and manages the meta-data to required to map plugin definitions to physical menus.

Description

The EMenuHook class loads and manages the meta-data to required to map plugin definitions to physical menus.

Name

struct _EMenuHookClass -- Menu hook type.

struct _EMenuHookClass

Synopsis

```
struct _EMenuHookClass {
  EPluginHookClass hook_class;
  GHashTable * target_map;
  EMenuClass * menu_class;
};
```

Members

hook_class	Superclass type.
target_map	Table of EluginHookTargetMaps which enumerate the target types and enable bits of the implementing class.
menu_class	The EMenuClass of the corresponding popup manager for implementing the class.

Description

The EMenuHookClass is an empty concrete class. It must be subclassed and initialised appropriately to perform useful work.

The EPluginHookClass.id must be set to the name and version of the hook handler the implementation defines. The *target_map* must be initialised with the data required to enumerate the target types and enable flags supported by the implementing class.

Description

The EMenuHookClass is an empty concrete class. It must be subclassed and initialised appropriately to perform useful work.

The EPluginHookClass.id must be set to the name and version of the hook handler the implementation defines. The *target_map* must be initialised with the data required to enumerate the target types and enable flags supported by the implementing class.

Name

e_menu_get_type --

e_menu_get_type

Synopsis

```
GType e_menu_get_type (void);
void;
```

Arguments

void no arguments

Description

Standard GObject type function. Used to subclass this type only.

Return value

The EMenu object type.

Name

e_menu_construct --

e_menu_construct

Synopsis

```
EMenu * e_menu_construct (em, menuid);
EMenu * em;
const char * menuid;
```

Arguments

em	An instantiated but uninitislied EPopup.
menuid	The unique identifier for this menu.

Description

Construct the base menu instance based on the parameters.

Return value

Returns *em*.

Name

e_menu_add_items --

e_menu_add_items

Synopsis

```
void * e_menu_add_items (emp, items, uifiles, pixmaps, freefunc,
data);
EMenu * emp;
GSList * items;
GSList * uifiles;
GSList * pixmaps;
EMenuItemsFunc freefunc;
void * data;
```

Arguments

emp	An initialised EMenu.
items	A list of EMenuItems or derived structures defining a group of menu items for this menu.
uifiles	A list of EMenuUIFile objects describing all ui files associated with the items.
pixmaps	A list of EMenuPixmap objects describing all pixmaps associated with the menus.
freefunc	If supplied, called when the menu items are no longer needed.
data	user-data passed to *freefunc* and activate callbacks.

Description

Add new EMenuItems to the menu's. This may be called any number of times before the menu is first activated to hook onto any of the menu items defined for that view.

Return value

A handle that can be passed to remove_items as required.

Name

e_menu_remove_items --

e_menu_remove_items

Synopsis

```
void e_menu_remove_items (emp, handle);
EMenu * emp;
void * handle;
```

Arguments

emp
handle

Description

Remove menu items previously added.

Name

e_menu_activate --

e_menu_activate

Synopsis

```
void e_menu_activate (em, uic, act);
EMenu * em;
struct _BonoboUIComponent * uic;
int act;
```

Arguments

em	An initialised EMenu.
uic	The BonoboUI component for this views menu's.
act	If `TRUE`, then the control is being activated.

Description

This is called by the owner of the component, control, or view to pass on the activate or deactivate control signals. If the view is being activated then the callbacks and menu items are setup, otherwise they are removed.

This should always be called in the strict sequence of activate, then deactivate, repeated any number of times.

Name

e_menu_update_target --

e_menu_update_target

Synopsis

```
void e_menu_update_target (em, tp);
EMenu * em;
void * tp;
```

Arguments

em An initialised EMenu.

tp Target, after this call the menu owns the target.

Description

Change the target for the menu. Once the target is changed, the sensitivity state of the menu items managed by *em* is re-evaluated and the physical menu's updated to reflect it.

This is used by the owner of the menu and view to update the menu system based on user input or changed system state.

Name

e_menu_class_add_factory --

e_menu_class_add_factory

Synopsis

```
EMenuFactory * e_menu_class_add_factory (klass, menuid, func, data);
EMenuClass * klass;
const char * menuid;
EMenuFactoryFunc func;
void * data;
```

Arguments

klass	An EMenuClass type to which this factory applies.
menuid	The identifier of the menu for this factory, or NULL to be called on all menus.
func	An EMenuFactoryFunc callback.
data	Callback data for *func*.

Description

Add a menu factory which will be called when the menu *menuid* is created. The factory is free to add new items as it wishes to the menu provided in the callback.

TODO

Make the menuid a pattern?

Return value

A handle to the factory.

Name

e_menu_class_remove_factory --

e_menu_class_remove_factory

Synopsis

```
void e_menu_class_remove_factory (klass, f);
EMenuClass * klass;
EMenuFactory * f;
```

Arguments

klass	Class on which the factory was originally added.
f	Factory handle.

Description

Remove a popup factory. This must only be called once, and must only be called using a valid factory handle *f*. After this call, *f* is undefined.

Name

e_menu_target_new --

e_menu_target_new

Synopsis

```
void * e_menu_target_new (ep, type, size);
EMenu * ep;
int type;
size_t size;
```

Arguments

ep	An EMenu to which this target applies.
type	Target type, up to implementation.
size	Size of memory to allocate. Must be >= sizeof(EMenuTarget).

Description

Allocate a new menu target suitable for this class. *size* is used to specify the actual target size, which may vary depending on the implementing class.

Name

e_menu_target_free --

e_menu_target_free

Synopsis

```
void e_menu_target_free (ep, o);
EMenu * ep;
void * o;
```

Arguments

ep	EMenu on which the target was allocated.
o	Tareget to free.

Description

Free a target.

Name

e_menu_hook_get_type --

e_menu_hook_get_type

Synopsis

```
GType e_menu_hook_get_type (void);
void;
```

Arguments

void	no arguments

Description

Standard GObject function to get the object type. Used to subclass EMenuHook.

Return value

The type of the menu hook class.

Name

e_menu_hook_class_add_target_map --

e_menu_hook_class_add_target_map

Synopsis

```
void e_menu_hook_class_add_target_map (klass, map);
EMenuHookClass * klass;
const EMenuHookTargetMap * map;
```

Arguments

klass	The derived EMenuHook class.
map	A map used to describe a single EMenuTarget for this class.

Description

Adds a target map to a concrete derived class of EMenu. The target map enumerates a single target type, and the enable mask bit names, so that the type can be loaded automatically by the EMenu class.

Chapter 11. EConfig

The EConfig object manages the building of dynamic configuration pages to configure specific application objects. The same basic object can be used to fully drive a wizard-like druid object, or to drive a note-book of configuration options. It is used by application code to provide the core controller in a model-view-controller implementation of a UI window.

The EConfigHook object is loaded by the EPlugin system, and is used hook in additional configuration items into configuration windows or druids dynamically.

Name

enum _e_config_target_change_t -- Target changed mode.

enum _e_config_target_change_t

Synopsis

```
enum _e_config_target_change_t {
  E_CONFIG_TARGET_CHANGED_STATE,
  E_CONFIG_TARGET_CHANGED_REBUILD
};
```

Constants

E_CONFIG_TARGET_CHANGED_STATE	A state of the target has changed.
E_CONFIG_TARGET_CHANGED_REBUILD	A state of the target has changed, and the UI must be reconfigured as a result.

Description

How the target has changed. If *`E_CONFIG_TARGET_CHANGED_REBUILD`* then a widget reconfigure is necessary, otherwise it is used to check if the widget is complete yet.

Description

How the target has changed. If *`E_CONFIG_TARGET_CHANGED_REBUILD`* then a widget reconfigure is necessary, otherwise it is used to check if the widget is complete yet.

Name

enum _e_config_t -- configuration item type.

enum _e_config_t

Synopsis

```
enum _e_config_t {
  E_CONFIG_BOOK,
  E_CONFIG_DRUID,
  E_CONFIG_PAGE,
  E_CONFIG_PAGE_START,
  E_CONFIG_PAGE_FINISH,
  E_CONFIG_SECTION,
  E_CONFIG_SECTION_TABLE,
  E_CONFIG_ITEM,
  E_CONFIG_ITEM_TABLE
};
```

Constants

E_CONFIG_BOOK	A notebook item. Only one of this or *`E_CONFIG_DRUID`* may be included in the item list for the entire configuration description.
E_CONFIG_DRUID	A druid item. Only one of this or *`E_CONFIG_BOOK`* may be included in the item list for the entire configutation description.
E_CONFIG_PAGE	A configuration page. The item *`label`* will be either the notebook tab label or the druid page title if no factory is supplied.
E_CONFIG_PAGE_START	A druid start page. Only one of these may be supplied for a druid and it should be the first page in the druid.
E_CONFIG_PAGE_FINISH	A druid finish page. Only one of these may be supplied for a druid and it should be the last page of the druid.
E_CONFIG_SECTION	A section in the configuration page. A page for this section must have already been defined. The item *`label`* if supplied will be setup as a borderless hig-compliant frame title. The content of the section will be a GtkVBox. If a factory is used then it is up to the factory method to create the section and add it to the parent page, and return a GtkVBox for following sections.
E_CONFIG_SECTION_TABLE	A table section. The same as an *`E_CONFIG_SECTION`* but the content object is a GtkTable instead.
E_CONFIG_ITEM	A configuration item. It must have a parent section defined in the configuration system.
E_CONFIG_ITEM_TABLE	A configuration item with a parent *`E_CONFIG_SECTION_TABLE`*.

Description

A configuration item type for each configuration item added to the EConfig object. These are merged from all contributors to the configuration window, and then processed to form the combined display.

Description

A configuration item type for each configuration item added to the EConfig object. These are merged from all contributors to the configuration window, and then processed to form the combined display.

Name

struct _EConfigItem -- A configuration item.

struct _EConfigItem

Synopsis

```
struct _EConfigItem {
  enum _e_config_t type;
  char * path;
  char * label;
  EConfigItemFactoryFunc factory;
  void * user_data;
};
```

Members

type	The configuration item type.
path	An absolute path positioning this item in the configuration window. This will be used as a sort key for an ASCII sort to position the item in the layout tree.
label	A label or section title string which is used if no factory is supplied to title the page or section.
factory	If supplied, this will be invoked instead to create the appropriate item.
user_data	User data for the factory.

Description

The basic descriptor of a configuration item. This may be subclassed to store extra context information for each item.

Description

The basic descriptor of a configuration item. This may be subclassed to store extra context information for each item.

Name

struct _EConfigTarget -- configuration context.

struct _EConfigTarget

Synopsis

```
struct _EConfigTarget {
  struct _EConfig * config;
  struct _GtkWidget * widget;
  guint32 type;
};
```

Members

config	The parent object.
widget	A target-specific parent widget.
type	The type of target, defined by implementing classes.

Description

The base target object is used as the parent and placeholder for configuration context for a given configuration window. It is subclassed by implementing classes to provide domain-specific context.

Description

The base target object is used as the parent and placeholder for configuration context for a given configuration window. It is subclassed by implementing classes to provide domain-specific context.

Name

struct _EConfig -- A configuration management object.

struct _EConfig

Synopsis

```
struct _EConfig {
  GObject object;
  struct _EConfigPrivate * priv;
  int type;
  char * id;
  EConfigTarget * target;
  struct _GtkWidget * widget;
  struct _GtkWidget * window;
};
```

Members

object	Superclass.
priv	Private data.
type	Either *`E_CONFIG_BOOK`* or *`E_CONFIG_DRIUD`*, describing the root window type.
id	The globally unique identifider for this configuration window, used for hooking into it.
target	The current target.
widget	The GtkNoteBook or GnomeDruid created after
window	If :`create_window` is called, then the containing toplevel GtkDialog or GtkWindow appropriate for the *`type`* of configuration window created.

Description

`create_widget` is called that represents the merged and combined configuration window.

Name

struct _EConfigClass -- Configuration management abstract class.

struct _EConfigClass

Synopsis

```
struct _EConfigClass {
  GObjectClass object_class;
  EDList factories;
  void (* set_target (EConfig *ep, EConfigTarget *t);
  void (* target_free (EConfig *ep, EConfigTarget *t);
};
```

Members

object_class	Superclass.
factories	A list of factories registered on this type of configuration manager.
set_target	A virtual method used to set the target on the configuration manager. This is used by subclasses so they may hook into changes on the target to propery drive the manager.
target_free	A virtual method used to free the target in an implementation-defined way.

Description

Name

struct _EConfigHookItemFactoryData -- Factory marshalling structure.

struct _EConfigHookItemFactoryData

Synopsis

```
struct _EConfigHookItemFactoryData {
  EConfig * config;
  EConfigItem * item;
  EConfigTarget * target;
  struct _GtkWidget * parent;
  struct _GtkWidget * old;
};
```

Members

config	The parent EConfig. This is also available in *`target`*->config but is here as a convenience. (TODO: do we need this).
item	The corresponding configuration item.
target	The current configuration target. This is also available on *`config`*->target.
parent	The parent widget for this item. Depends on the item type.
old	The last widget created by this factory. The factory is only re-invoked if a reconfigure request is invoked on the EConfig.

Description

Used to marshal the callback data for the EConfigItemFactory method to a single pointer for the EPlugin system.

Description

Used to marshal the callback data for the EConfigItemFactory method to a single pointer for the EPlugin system.

Name

struct _EConfigHookPageCheckData -- Check callback data.

struct _EConfigHookPageCheckData

Synopsis

```
struct _EConfigHookPageCheckData {
  EConfig * config;
  EConfigTarget * target;
  const char * pageid;
};
```

Members

config	
target	The current configuration target. This is also available on *`config`*->target.
pageid	Name of page to validate, or "" means check all configuration.

Description

Name

struct _EConfigHookGroup -- A group of configuration items.

struct _EConfigHookGroup

Synopsis

```
struct _EConfigHookGroup {
  struct _EConfigHook * hook;
  char * id;
  int target_type;
  GSList * items;
  char * check;
  char * commit;
  char * abort;
};
```

Members

hook	Parent object.
id	The configuration window to which these items apply.
target_type	The target type expected by the items. This is defined by implementing classes.
items	A list of EConfigHookItem's for this group.
check	A validate page handler.
commit	The name of the commit function for this group of items, or NULL for instant-apply configuration windows. Its format is plugin-type defined.
abort	Similar to the *`commit`* function but for aborting or cancelling a configuration edit.

Description

Each plugin that hooks into a given configuration page will define all of the tiems for that page in a single group.

Description

Each plugin that hooks into a given configuration page will define all of the tiems for that page in a single group.

Name

struct _EConfigHook -- Plugin hook for configuration windows.

struct _EConfigHook

Synopsis

```
struct _EConfigHook {
  EPluginHook hook;
  GSList * groups;
};
```

Members

hook	Superclass.
groups	A list of EConfigHookGroup's of all configuration windows this plugin hooks into.

Description

Name

struct _EConfigHookClass -- Abstract class for configuration window

struct _EConfigHookClass

Synopsis

```
struct _EConfigHookClass {
  EPluginHookClass hook_class;
  GHashTable * target_map;
  EConfigClass * config_class;
};
```

Members

hook_class	Superclass.
target_map	A table of EConfigHookTargetMap structures describing the possible target types supported by this class.
config_class	The EConfig derived class that this hook implementation drives.

Description

This is an abstract class defining the plugin hook point for configuration windows.

Description

This is an abstract class defining the plugin hook point for configuration windows.

Name

e_config_get_type --

e_config_get_type

Synopsis

```
GType e_config_get_type (void);
void;
```

Arguments

void no arguments

Description

Standard GObject method. Used to subclass for the concrete implementations.

Return value

EConfig type.

Name

e_config_construct --

e_config_construct

Synopsis

```
EConfig * e_config_construct (ep, type, id);
EConfig * ep;
int type;
const char * id;
```

Arguments

ep	The instance to initialise.
type	The type of configuration manager, *E_CONFIG_BOOK* or *E_CONFIG_DRUID*.
id	The name of the configuration window this manager drives.

Description

Used by implementing classes to initialise base parameters.

Return value

ep is returned.

Name

e_config_add_items --

e_config_add_items

Synopsis

```
void e_config_add_items (ec, items, commitfunc, abortfunc, freefunc,
data);
EConfig * ec;
GSList * items;
EConfigItemsFunc commitfunc;
EConfigItemsFunc abortfunc;
EConfigItemsFunc freefunc;
void * data;
```

Arguments

ec	An initialised implementing instance of EConfig.
items	A list of EConfigItem's to add to the configuration manager *ec*.
commitfunc	If supplied, called to commit the configuration items to persistent storage.
abortfunc	If supplied, called to abort/undo the storage of these items permanently.
freefunc	If supplied, called to free the item list (and/or items) once they are no longer needed.
data	Data for the callback methods.

Description

Add new EConfigItems to the configuration window. Nothing will be done with them until the widget is built.

TODO

perhaps commit and abort should just be signals.

Name

e_config_add_page_check --

e_config_add_page_check

Synopsis

```
void e_config_add_page_check (ec, pageid, check, data);
EConfig * ec;
const char * pageid;
EConfigCheckFunc check;
void * data;
```

Arguments

ec	Initialised implemeting instance of EConfig.
pageid	pageid to check.
check	checking callback.
data	user-data for the callback.

Description

Add a page-checking function callback. It will be called to validate the data in the given page or pages. If *pageid* is NULL then it will be called to validate every page, or the whole configuration window.

In the latter case, the pageid in the callback will be either the specific page being checked, or NULL when the whole config window is being checked.

The page check function is used to validate input before allowing the druid to continue or the notebook to close.

Name

e_config_set_target --

e_config_set_target

Synopsis

```
void e_config_set_target (emp, target);
EConfig * emp;
EConfigTarget * target;
```

Arguments

emp	An initialised EConfig.
target	A target allocated from *emp*.

Description

Sets the target object for the config window. Generally the target is set only once, and will supply its own “changed” signal which can be used to drive the modal. This is a virtual method so that the implementing class can connect to the changed signal and initiate a `e_config_target_changed` call where appropriate.

Name

e_config_create_widget --

e_config_create_widget

Synopsis

```
GtkWidget * e_config_create_widget (emp);
EConfig * emp;
```

Arguments

emp	An initialised EConfig object.

Description

Create the widget described by *emp*. Only the core widget appropriate for the given type is created, i.e. a GtkNotebook for the E_CONFIG_BOOK type and a GnomeDruid for the E_CONFIG_DRUID type.

This object will be self-driving, but will not close itself once complete.

Unless reffed otherwise, the management object *emp* will be finalised when the widget is.

Return value

The widget, also available in *emp*.widget

Name

e_config_create_window --

e_config_create_window

Synopsis

```
GtkWidget * e_config_create_window (emp, parent, title);
EConfig * emp;
struct _GtkWindow * parent;
const char * title;
```

Arguments

emp	Initialised and configured EMConfig derived instance.
parent	Parent window or NULL.
title	Title of window or dialog.

Description

Create a managed GtkWindow object from *emp*. This window will be fully driven by the EConfig *emp*. If *emp*.type is *E_CONFIG_DRUID*, then this will be a toplevel GtkWindow containing a GnomeDruid. If it is *E_CONFIG_BOOK* then it will be a GtkDialog containing a Nnotebook.

Unless reffed otherwise, the management object *emp* will be finalised when the widget is.

Return value

The window widget. This is also stored in *emp*.window.

Name

e_config_target_changed --

e_config_target_changed

Synopsis

```
void e_config_target_changed (emp, how);
EConfig * emp;
e_config_target_change_t how;
```

Arguments

emp
how

Description

Indicate that the target has changed. This may be called by the self-aware target itself, or by the driving code. If *how* is `E_CONFIG_TARGET_CHANGED_REBUILD`, then the entire configuration widget may be recreated based on the changed target.

This is used to sensitise Druid next/back buttons and the Apply button for the Notebook mode.

Name

e_config_abort --

e_config_abort

Synopsis

```
void e_config_abort (ec);
EConfig * ec;
```

Arguments

ec

Description

Signify that the stateful configuration changes must be discarded to all listeners. This is used by self-driven druid or notebook, or may be used by code using the widget directly.

Name

e_config_commit --

e_config_commit

Synopsis

```
void e_config_commit (ec);
EConfig * ec;
```

Arguments

ec

Description

Signify that the stateful configuration changes should be saved. This is used by the self-driven druid or notebook, or may be used by code driving the widget directly.

Name

e_config_page_check --

e_config_page_check

Synopsis

```
gboolean e_config_page_check (ec, pageid);
EConfig * ec;
const char * pageid;
```

Arguments

ec	
pageid	The path of the page item.

Description

Check that a given page is complete. If *pageid* is NULL, then check the whole config. No check is made that the page actually exists.

Return value

FALSE if the data is inconsistent/incomplete.

Name

e_config_page_get --

e_config_page_get

Synopsis

```
GtkWidget * e_config_page_get (ec, pageid);
EConfig * ec;
const char * pageid;
```

Arguments

ec	
pageid	The path of the page item.

Description

Retrieve the page widget corresponding to *pageid*.

Return value

The page widget. It will be the root GtkNotebook container or the GnomeDruidPage object.

Name

e_config_page_next --

e_config_page_next

Synopsis

```
const char * e_config_page_next (ec, pageid);
EConfig * ec;
const char * pageid;
```

Arguments

ec
pageid The path of the page item.

Description

Find the path of the next visible page after *pageid*. If *pageid* is NULL then find the first visible page.

Return value

The path of the next page, or *NULL* if *pageid* was the last configured and visible page.

Name

e_config_page_prev --

e_config_page_prev

Synopsis

```
const char * e_config_page_prev (ec, pageid);
EConfig * ec;
const char * pageid;
```

Arguments

ec	
pageid	The path of the page item.

Description

Find the path of the previous visible page before *pageid*. If *pageid* is NULL then find the last visible page.

Return value

The path of the previous page, or *NULL* if *pageid* was the first configured and visible page.

Name

e_config_class_add_factory --

e_config_class_add_factory

Synopsis

```
EConfigFactory * e_config_class_add_factory (klass, id, func, data);
EConfigClass * klass;
const char * id;
EConfigFactoryFunc func;
void * data;
```

Arguments

klass	Implementing class pointer.
id	The name of the configuration window you're interested in. This may be NULL to be called for all windows.
func	An EConfigFactoryFunc to call when the window *id* is being created.
data	Callback data.

Description

Add a config factory which will be called to `add_items` any extra items's if wants to, to the current Config window.

TODO

Make the id a pattern?

Return value

A handle to the factory.

Name

e_config_class_remove_factory --

e_config_class_remove_factory

Synopsis

```
void e_config_class_remove_factory (klass, f);
EConfigClass * klass;
EConfigFactory * f;
```

Arguments

klass	-- undescribed --
f	Handle from :`class_add_factory` call.

Description

Remove a config factory. The handle *f* may only be removed once.

Name

e_config_target_new --

e_config_target_new

Synopsis

```
void * e_config_target_new (ep, type, size);
EConfig * ep;
int type;
size_t size;
```

Arguments

ep	Parent EConfig object.
type	type, up to implementor
size	Size of object to allocate.

Description

Allocate a new config target suitable for this class. Implementing classes will define the actual content of the target.

Name

e_config_target_free --

e_config_target_free

Synopsis

```
void e_config_target_free (ep, o);
EConfig * ep;
void * o;
```

Arguments

ep	Parent EConfig object.
o	The target to fre.

Description

Free a target. The implementing class can override this method to free custom targets.

Name

e_config_hook_get_type --

e_config_hook_get_type

Synopsis

```
GType e_config_hook_get_type (void);
void;
```

Arguments

void no arguments

Description

Standard GObject function to get the object type.

Return value

The EConfigHook class type.

Name

e_config_hook_class_add_target_map --

e_config_hook_class_add_target_map

Synopsis

```
void e_config_hook_class_add_target_map (klass, map);
EConfigHookClass * klass;
const EConfigHookTargetMap * map;
```

Arguments

klass	The dervied EconfigHook class.
map	A map used to describe a single EConfigTarget type for this class.

Description

Add a targe tmap to a concrete derived class of EConfig. The target map enumates the target types available for the implenting class.

Description

Add a targe tmap to a concrete derived class of EConfig. The target map enumates the target types available for the implenting class.

Chapter 12. EEvent

The EEvent object manages broadcast of events for a given component or application. It is used by application code to provide the plugin system with an entry point for user and system state events.

The EEventHook object is loaded by the EPlugin system, and is used hook event listeners into dynamically loaded event handlers.

Name

enum _e_event_t -- Event type.

enum _e_event_t

Synopsis

```
enum _e_event_t {
  E_EVENT_PASS,
  E_EVENT_SINK
};
```

Constants

E_EVENT_PASS	A passthrough event handler which only receives the event.
E_EVENT_SINK	A sink event handler swallows all events it processes.

Description

The event type defines what type of event listener this is.

Events should normally be *`E_EVENT_PASS`*.

Description

The event type defines what type of event listener this is.

Events should normally be *`E_EVENT_PASS`*.

Name

struct _EEventItem -- An event listener item.

struct _EEventItem

Synopsis

```
struct _EEventItem {
  enum _e_event_t type;
  int priority;
  const char * id;
  int target_type;
  EEventFunc handle;
  void * user_data;
  guint32 enable;
};
```

Members

type	The type of the event listener.
priority	A signed number signifying the priority of the event listener. 0 should be used normally. This is used to order event receipt when multiple listners are present.
id	The name of the event to listen to. By convention events are of the form “component.subcomponent”. The target mask provides further sub-event type qualification.
target_type	Target type for this event. This is implementation specific.
handle	Event handler callback.
user_data	Callback data.
enable	Target-specific mask to qualify the receipt of events. This is target and implementation specific.

Description

An EEventItem defines a specific event listening point on a given EEvent object. When an event is broadcast onto an EEvent handler, any matching EEventItems will be invoked in priority order.

Description

An EEventItem defines a specific event listening point on a given EEvent object. When an event is broadcast onto an EEvent handler, any matching EEventItems will be invoked in priority order.

Name

struct _EEventTarget -- Base EventTarget.

struct _EEventTarget

Synopsis

```
struct _EEventTarget {
  struct _EEvent * event;
  guint32 type;
  guint32 mask;
};
```

Members

event	Parent object.
type	Target type. Defined by the implementation.
mask	Mask of this target. This is defined by the implementation, the type, and the actual content of the target.

Description

This defined a base EventTarget. This must be subclassed by implementations to provide contextual data for events, and define the enablement qualifiers.

Description

This defined a base EventTarget. This must be subclassed by implementations to provide contextual data for events, and define the enablement qualifiers.

Name

struct _EEvent -- An Event Manager.

struct _EEvent

Synopsis

```
struct _EEvent {
  GObject object;
  struct _EEventPrivate * priv;
  char * id;
  EEventTarget * target;
};
```

Members

object	Superclass.
priv	Private data.
id	Id of this event manager.
target	The current target, only set during event emission.

Description

The EEvent manager object. Each component which defines event types supplies a single EEvent manager object. This manager routes all events invoked on this object to all registered listeners based on their qualifiers.

Description

The EEvent manager object. Each component which defines event types supplies a single EEvent manager object. This manager routes all events invoked on this object to all registered listeners based on their qualifiers.

Name

struct _EEventClass -- Event management type.

struct _EEventClass

Synopsis

```
struct _EEventClass {
  GObjectClass object_class;
  void (* target_free (EEvent *ep, EEventTarget *t);
};
```

Members

object_class	Superclass.
target_free	Virtual method to free the target.

Description

The EEvent class definition. This must be sub-classed for each component that wishes to provide hookable events. The subclass only needs to know how to allocate and free each target type it supports.

Description

The EEvent class definition. This must be sub-classed for each component that wishes to provide hookable events. The subclass only needs to know how to allocate and free each target type it supports.

Name

struct _EEventHook -- An event hook.

struct _EEventHook

Synopsis

```
struct _EEventHook {
  EPluginHook hook;
};
```

Members

hook	Superclass.

Description

The EEventHook class loads and manages the meta-data required to track event listeners. Unlike other hook types, there is a 1:1 match between an EEventHook instance class and its EEvent instance.

When the hook is loaded, all of its event hooks are stored directly on the corresponding EEvent which is stored in its class static area.

Description

The EEventHook class loads and manages the meta-data required to track event listeners. Unlike other hook types, there is a 1:1 match between an EEventHook instance class and its EEvent instance.

When the hook is loaded, all of its event hooks are stored directly on the corresponding EEvent which is stored in its class static area.

Name

struct _EEventHookClass --

struct _EEventHookClass

Synopsis

```
struct _EEventHookClass {
  EPluginHookClass hook_class;
  GHashTable * target_map;
  EEvent * event;
};
```

Members

hook_class	
target_map	Table of EPluginHookTargetMaps which enumerate the target types and enable bits of the implementing class.
event	The EEvent instance on which all loaded events must be registered.

Description

The EEventHookClass is an empty event hooking class, which must be subclassed and initialised before use.

The EPluginHookClass.id must be set to the name and version of the hook handler itself, and then the type must be registered with the EPlugin hook list before any plugins are loaded.

Description

The EEventHookClass is an empty event hooking class, which must be subclassed and initialised before use.

The EPluginHookClass.id must be set to the name and version of the hook handler itself, and then the type must be registered with the EPlugin hook list before any plugins are loaded.

Name

e_event_get_type --

e_event_get_type

Synopsis

```
GType e_event_get_type (void);
void;
```

Arguments

void	no arguments

Description

Standard GObject type function. Used to subclass EEvent.

Return value

The EEvent type.

Name

e_event_construct --

e_event_construct

Synopsis

```
EEvent * e_event_construct (ep, id);
EEvent * ep;
const char * id;
```

Arguments

ep	An instantiated but uninitialised EEvent.
id	Event manager id.

Description

Construct the base event instance with standard parameters.

Return value

Returns *ep*.

Name

e_event_add_items --

e_event_add_items

Synopsis

```
void * e_event_add_items (emp, items, freefunc, data);
EEvent * emp;
GSList * items;
EEventItemsFunc freefunc;
void * data;
```

Arguments

emp	An initialised EEvent structure.
items	A list of EEventItems event listeners to register on this event manager.
freefunc	A function called when the *items* list is no longer needed.
data	callback data for *freefunc* and for item event handlers.

Description

Adds *items* to the list of events listened to on the event manager *emp*.

Return value

An opaque key which can later be passed to remove_items.

Name

e_event_remove_items --

e_event_remove_items

Synopsis

```
void e_event_remove_items (emp, handle);
EEvent * emp;
void * handle;
```

Arguments

emp
handle

Description

Remove items previously added. They MUST have been previously added, and may only be removed once.

Name

e_event_emit --

e_event_emit

Synopsis

```
void e_event_emit (emp, id, target);
EEvent * emp;
const char * id;
EEventTarget * target;
```

Arguments

emp	-- undescribed --
id	Event name. This will be compared against EEventItem.id.
target	The target describing the event context. This will be implementation defined.

Description

Emit an event. *target* will automatically be freed once its emission is complete.

Name

e_event_target_new --

e_event_target_new

Synopsis

```
void * e_event_target_new (ep, type, size);
EEvent * ep;
int type;
size_t size;
```

Arguments

ep	An initialised EEvent instance.
type	type, up to implementor
size	The size of memory to allocate. This must be >= sizeof(EEventTarget).

Description

Allocate a new event target suitable for this class. It is up to the implementation to define the available target types and their structure.

Name

e_event_target_free --

e_event_target_free

Synopsis

```
void e_event_target_free (ep, o);
EEvent * ep;
void * o;
```

Arguments

ep	An initialised EEvent instance on which this target was allocated.
o	The target to free.

Description

Free a target. This invokes the virtual free method on the EEventClass.

Name

e_event_hook_get_type --

e_event_hook_get_type

Synopsis

```
GType e_event_hook_get_type (void);
void;
```

Arguments

void no arguments

Description

Standard GObject function to get the EEvent object type. Used to subclass EEventHook.

Return value

The type of the event hook class.

Name

e_event_hook_class_add_target_map --

e_event_hook_class_add_target_map

Synopsis

```
void e_event_hook_class_add_target_map (klass, map);
EEventHookClass * klass;
const EEventHookTargetMap * map;
```

Arguments

klass	The derived EEventHook class.
map	A map used to describe a single EEventTarget type for this class.

Description

Add a target map to a concrete derived class of EEvent. The target map enumerates a single target type and th eenable mask bit names, so that the type can be loaded automatically by the base EEvent class.

Chapter 13. EMFormat

The EMFormat object drives the formatting of MIME message content for display, print, and replying. EMFormatHTML is an implementation of EMFormat which writes its output to a GtkHTML instance.

The EMFormatHook object is loaded by the EPlugin system, and is used hook event listeners into dynamically loaded event handlers.

Name

struct _EMFormatHook -- Mail formatter hook.

struct _EMFormatHook

Synopsis

```
struct _EMFormatHook {
  EPluginHook hook;
  GSList * groups;
};
```

Members

hook
groups

Description

The Mail formatter hook links all of the plugin formatter hooks into the relevent formatter classes.

Description

The Mail formatter hook links all of the plugin formatter hooks into the relevent formatter classes.

Name

struct _EMFormatHandler -- MIME type handler.

struct _EMFormatHandler

Synopsis

```
struct _EMFormatHandler {
  char * mime_type;
  EMFormatFunc handler;
  guint32 flags;
  struct _EMFormatHandler * old;
};
```

Members

mime_type	Type this handler handles.
handler	The handler callback.
flags	Handling flags, see enum _em_format_handler_t.
old	The last handler set on this type. Allows overrides to fallback to previous implementation.

Description

Name

enum _em_format_handler_t -- Format handler flags.

enum _em_format_handler_t

Synopsis

```
enum _em_format_handler_t {
  EM_FORMAT_HANDLER_INLINE,
  EM_FORMAT_HANDLER_INLINE_DISPOSITION
};
```

Constants

EM_FORMAT_HANDLER_INLINE	This type should be shown expanded inline by default.
EM_FORMAT_HANDLER_INLINE_DISPOSITION	This type should always be shown inline, despite what the Content-Disposition suggests.

Description

Name

struct _EMFormatPURI -- Pending URI object.

struct _EMFormatPURI

Synopsis

```
struct _EMFormatPURI {
  struct _EMFormatPURI * next;
  struct _EMFormatPURI * prev;
  void (* free (struct _EMFormatPURI *p);
  struct _EMFormat * format;
  char * uri;
  char * cid;
  char * part_id;
  EMFormatPURIFunc func;
  struct _CamelMimePart * part;
  unsigned int use_count;
};
```

Members

next	Double-linked list header.
prev	Double-linked list header.
free	May be set by allocator and will be called when no longer needed.
format	
uri	Calculated URI of the part, if the part has one in its Content-Location field.
cid	The RFC2046 Content-Id of the part. If none is present, a unique value is calculated from *`part_id`*.
part_id	A unique identifier for each part.
func	Callback for when the URI is requested. The callback writes its data to the supplied stream.
part	
use_count	

Description

This is used for multipart/related, and other formatters which may need to include a reference to out-of-band data in the content stream.

This object may be subclassed as a struct.

Description

This is used for multipart/related, and other formatters which may need to include a reference to out-of-band data in the content stream.

This object may be subclassed as a struct.

Name

struct _EMFormatPURITree -- Pending URI visibility tree.

struct _EMFormatPURITree

Synopsis

```
struct _EMFormatPURITree {
  struct _EMFormatPURITree * next;
  struct _EMFormatPURITree * prev;
  struct _EMFormatPURITree * parent;
  EDList uri_list;
  EDList children;
};
```

Members

next	Double-linked list header.
prev	Double-linked list header.
parent	Parent in tree.
uri_list	List of EMFormatPURI objects at this level.
children	Child nodes of EMFormatPURITree.

Description

This structure is used internally to form a visibility tree of parts in the current formatting stream. This is to implement the part resolution rules for RFC2387 to implement multipart/related.

Description

This structure is used internally to form a visibility tree of parts in the current formatting stream. This is to implement the part resolution rules for RFC2387 to implement multipart/related.

Name

struct _EMFormat -- Mail formatter object.

struct _EMFormat

Synopsis

```
struct _EMFormat {
  GObject parent;
  struct _EMFormatPrivate * priv;
  struct _CamelMimeMessage * message;
  struct _CamelFolder * folder;
  char * uid;
  GString * part_id;
  EDList header_list;
  struct _CamelSession * session;
  struct _CamelURL * base;
  const char * snoop_mime_type;
  struct _CamelCipherValidity * valid;
  struct _CamelCipherValidity * valid_parent;
  GHashTable * inline_table;
  GHashTable * pending_uri_table;
  struct _EMFormatPURITree * pending_uri_tree;
  struct _EMFormatPURITree * pending_uri_level;
  em_format_mode_t mode;
  char * charset;
  char * default_charset;
};
```

Members

parent
priv
message
folder
uid
part_id
header_list
session
base
snoop_mime_type
valid
valid_parent
inline_table
pending_uri_table
pending_uri_tree
pending_uri_level
mode
charset
default_charset

Description

Most fields are private or read-only.

This is the base MIME formatter class. It provides no formatting itself, but drives most of the basic types, including multipart / * types.

Description

Most fields are private or read-only.

This is the base MIME formatter class. It provides no formatting itself, but drives most of the basic types, including multipart / * types.

Name

em_format_class_add_handler --

em_format_class_add_handler

Synopsis

```
void em_format_class_add_handler (emfc, info);
EMFormatClass * emfc;
EMFormatHandler * info;
```

Arguments

emfc	EMFormatClass
info	Callback information.

Description

Add a mime type handler to this class. This is only used by implementing classes. The *info*.old pointer will automatically be setup to point to the old hanlder if one was already set. This can be used for over-rides a fallback.

When a mime type described by *info* is encountered, the callback will be invoked. Note that *info* may be extended by sub-classes if they require additional context information.

Use a mime type of “foo/ *” to insert a fallback handler for type “foo”.

Name

em_format_class_remove_handler --

em_format_class_remove_handler

Synopsis

```
void **em_format_class_remove_handler** (emfc, info);
EMFormatClass * emfc;
EMFormatHandler * info;
```

Arguments

emfc
info

Description

Remove a handler. *info* must be a value which was previously added.

Name

emf_find_handler --

emf_find_handler

Synopsis

```
const EMFormatHandler * emf_find_handler (emf, mime_type);
EMFormat * emf;
const char * mime_type;
```

Arguments

emf
mime_type

Description

Find a format handler by *mime_type*.

Return value

NULL if no handler is available.

Name

em_format_fallback_handler --

em_format_fallback_handler

Synopsis

```
const EMFormatHandler * em_format_fallback_handler (emf, mime_type);
EMFormat * emf;
const char * mime_type;
```

Arguments

emf
mime_type

Description

Try to find a format handler based on the major type of the *mime_type*.

The subtype is replaced with “*” and a lookup performed.

Return value

Name

em_format_add_puri --

em_format_add_puri

Synopsis

```
EMFormatPURI * em_format_add_puri (emf, size, cid, part, func);
EMFormat * emf;
size_t size;
const char * cid;
CamelMimePart * part;
EMFormatPURIFunc func;
```

Arguments

emf	
size	
cid	Override the autogenerated content id.
part	
func	

Description

Add a pending-uri handler. When formatting parts that reference other parts, a pending-uri (PURI) can be used to track the reference.

size is used to allocate the structure, so that it can be directly subclassed by implementors.

cid can be used to override the key used to retreive the PURI, if NULL, then the content-location and the content-id of the *part* are stored as lookup keys for the part.

FIXME

This may need a free callback.

Return value

A new PURI, with a referenced copy of *part*, and the cid always set. The uri will be set if one is available. Clashes are resolved by forgetting the old PURI in the global index.

Name

em_format_push_level --

em_format_push_level

Synopsis

```
void em_format_push_level (emf);
EMFormat * emf;
```

Arguments

emf

Description

This is used to build a heirarchy of visible PURI objects based on the structure of the message. Used by multipart/alternative formatter.

FIXME

This could probably also take a uri so it can automaticall update the base location.

Name

em_format_pull_level --

em_format_pull_level

Synopsis

```
void em_format_pull_level (emf);
EMFormat * emf;
```

Arguments

emf

Description

Drop a level of visibility back to the parent. Note that no PURI values are actually freed.

Name

em_format_find_visible_puri --

em_format_find_visible_puri

Synopsis

```
EMFormatPURI * em_format_find_visible_puri (emf, uri);
EMFormat * emf;
const char * uri;
```

Arguments

emf
uri

Search for a PURI based on the visibility defined by

`push_level`

and

`pull_level`.

Return value

Name

em_format_find_puri --

em_format_find_puri

Synopsis

```
EMFormatPURI * em_format_find_puri (emf, uri);
EMFormat * emf;
const char * uri;
```

Arguments

emf
uri

Description

Search for a PURI based on a uri. Both the content-id and content-location are checked.

Return value

Name

em_format_clear_puri_tree --

em_format_clear_puri_tree

Synopsis

```
void em_format_clear_puri_tree (emf);
EMFormat * emf;
```

Arguments

emf

Description

For use by implementors to clear out the message structure data.

Name

em_format_set_session --

em_format_set_session

Synopsis

```
void em_format_set_session (emf, s);
EMFormat * emf;
struct _CamelSession * s;
```

Arguments

emf	Mail formatter.
s	-- undescribed --

Description

Format a message *msg*. If *emfsource* is non NULL, then the status of inlined expansion and so forth is copied direction from *emfsource*.

By passing the same value for *emf* and *emfsource*, you can perform a display refresh, or it can be used to generate an identical layout, e.g. to print what the user has shown inline.

Name

em_format_set_mode --

em_format_set_mode

Synopsis

```
void em_format_set_mode (emf, type);
EMFormat * emf;
em_format_mode_t type;
```

Arguments

emf
type

Description

Set display mode, EM_FORMAT_SOURCE, EM_FORMAT_ALLHEADERS, or EM_FORMAT_NORMAL.

Name

em_format_set_charset --

em_format_set_charset

Synopsis

```
void em_format_set_charset (emf, charset);
EMFormat * emf;
const char * charset;
```

Arguments

emf
charset

Description

set override charset on formatter. message will be redisplayed if required.

Name

em_format_set_default_charset --

em_format_set_default_charset

Synopsis

```
void em_format_set_default_charset (emf, charset);
EMFormat * emf;
const char * charset;
```

Arguments

emf
charset

Description

Set the fallback, default system charset to use when no other charsets are present. Message will be redisplayed if required (and sometimes redisplayed when it isn't).

Name

em_format_clear_headers --

em_format_clear_headers

Synopsis

```
void em_format_clear_headers (emf);
EMFormat * emf;
```

Arguments

emf

Description

Clear the list of headers to be displayed. This will force all headers to be shown.

Name

em_format_default_headers --

em_format_default_headers

Synopsis

```
void em_format_default_headers (emf);
EMFormat * emf;
```

Arguments

emf

Description

Set the headers to show to the default list.

From, Reply-To, To, Cc, Bcc, Subject and Date.

Name

em_format_add_header --

em_format_add_header

Synopsis

```
void em_format_add_header (emf, name, flags);
EMFormat * emf;
const char * name;
guint32 flags;
```

Arguments

emf	
name	The name of the header, as it will appear during output.
flags	EM_FORMAT_HEAD_* defines to control display attributes.

Description

Add a specific header to show. If any headers are set, they will be displayed in the order set by this function. Certain known headers included in this list will be shown using special formatting routines.

Name

em_format_is_attachment --

em_format_is_attachment

Synopsis

```
int em_format_is_attachment (emf, part);
EMFormat * emf;
CamelMimePart * part;
```

Arguments

emf	
part	Part to check.

Description

Returns true if the part is an attachment.

A part is not considered an attachment if it is a multipart, or a text part with no filename. It is used to determine if an attachment header should be displayed for the part.

Content-Disposition is not checked.

Return value

TRUE/FALSE

Name

em_format_is_inline --

em_format_is_inline

Synopsis

```
int em_format_is_inline (emf, partid, part, handle);
EMFormat * emf;
const char * partid;
CamelMimePart * part;
const EMFormatHandler * handle;
```

Arguments

emf	
partid	format->part_id part id of this part.
part	
handle	handler for this part

Description

Returns true if the part should be displayed inline. Any part with a Content-Disposition of inline, or if the *handle* has a default inline set, will be shown inline.

`set_inline` called on the same part will override any calculated value.

Return value

Name

em_format_set_inline --

em_format_set_inline

Synopsis

```
void em_format_set_inline (emf, partid, state);
EMFormat * emf;
const char * partid;
int state;
```

Arguments

emf

partid id of part

state

Description

Force the attachment *part* to be expanded or hidden explictly to match *state*. This is used only to record the change for a redraw or cloned layout render and does not force a redraw.

Name

em_format_format_text --

em_format_format_text

Synopsis

```
void em_format_format_text (emf, stream, dw);
EMFormat * emf;
CamelStream * stream;
CamelDataWrapper * dw;
```

Arguments

emf	
stream	Where to write the converted text
dw	-- undescribed --

Description

Decode/output a part's content to *stream*.

Name

em_format_describe_part --

em_format_describe_part

Synopsis

```
char * em_format_describe_part (part, mime_type);
CamelMimePart * part;
const char * mime_type;
```

Arguments

part

mime_type -- undescribed --

Description

Generate a simple textual description of a part, *mime_type* represents the the content.

Return value

Name

struct _EMFormatHTMLJob -- A formatting job.

struct _EMFormatHTMLJob

Synopsis

```
struct _EMFormatHTMLJob {
  struct _EMFormatHTMLJob * next;
  struct _EMFormatHTMLJob * prev;
  EMFormatHTML * format;
  struct _CamelStream * stream;
  /* We need to track the state of the visibility tree atthe point this uri was ge
  struct _CamelURL * base;
  void (* callback (struct _EMFormatHTMLJob *job, int cancelled);
  union u;
};
```

Members

next	Double linked list header.
prev	Double linked list header.
format	Set by allocation function.
stream	Free for use by caller.
puri_level	Set by allocation function.
base	Set by allocation function, used to save state.
callback	This callback will always be invoked, only once, even if the user cancelled the display. So the callback should free any extra data it allocated every time it is called.
u	Union data, free for caller to use.

Description

This object is used to queue a long-running-task which cannot be processed in the primary thread. When its turn comes, the job will be de-queued and the *`callback`* invoked to perform its processing, restoring various state to match the original state. This is used for image loading and other internal tasks.

This object is struct-subclassable. Only `em_format_html_job_new` may be used to allocate these.

Description

This object is used to queue a long-running-task which cannot be processed in the primary thread. When its turn comes, the job will be de-queued and the *`callback`* invoked to perform its processing, restoring various state to match the original state. This is used for image loading and other internal tasks.

This object is struct-subclassable. Only `em_format_html_job_new` may be used to allocate these.

Name

struct _EMFormatHTMLPObject -- Pending object.

struct _EMFormatHTMLPObject

Synopsis

```
struct _EMFormatHTMLPObject {
  struct _EMFormatHTMLPObject * next;
  struct _EMFormatHTMLPObject * prev;
  void (* free (struct _EMFormatHTMLPObject *);
  struct _EMFormatHTML * format;
  char * classid;
  EMFormatHTMLPObjectFunc func;
  struct _CamelMimePart * part;
};
```

Members

next	Double linked list header.
prev	Double linked list header.
free	Invoked when the object is no longer needed.
format	The parent formatter.
classid	The assigned class id as passed to `add_pobject`.
func	Callback function.
part	The part as passed to `add_pobject`.

Description

This structure is used to track OBJECT tags which have been inserted into the HTML stream. When GtkHTML requests them the *`func`* will be invoked to create the embedded widget.

This object is struct-subclassable. Only `em_format_html_add_pobject` may be used to allocate these.

Description

This structure is used to track OBJECT tags which have been inserted into the HTML stream. When GtkHTML requests them the *`func`* will be invoked to create the embedded widget.

This object is struct-subclassable. Only `em_format_html_add_pobject` may be used to allocate these.

Name

struct _EMFormatHTML -- HTML formatter object.

struct _EMFormatHTML

Synopsis

```
struct _EMFormatHTML {
  EMFormat format;
  struct _EMFormatHTMLPrivate * priv;
  struct _GtkHTML * html;
  EDList pending_object_list;
  GSList * headers;
  guint32 text_html_flags;
  guint32 body_colour;
  guint32 text_colour;
  guint32 frame_colour;
  guint32 content_colour;
  guint32 citation_colour;
  unsigned int load_http:2;
  unsigned int load_http_now:1;
  unsigned int mark_citations:1;
  unsigned int simple_headers:1;
  unsigned int hide_headers:1;
  unsigned int show_rupert:1;
};
```

Members

format	
priv	
html	
pending_object_list	
headers	
text_html_flags	
body_colour	
text_colour	
frame_colour	
content_colour	
citation_colour	
load_http	2:
load_http_now	1:
mark_citations	1:
simple_headers	1:
hide_headers	1:
show_rupert	1:

Description

Most of these fields are private or read-only.

The base HTML formatter object. This object drives HTML generation into a GtkHTML parser. It also handles text to HTML conversion, multipart/related objects and inline images.

Description

Most of these fields are private or read-only.

The base HTML formatter object. This object drives HTML generation into a GtkHTML parser. It also handles text to HTML conversion, multipart/related objects and inline images.