Feelin Developper Guide

Olivier LAVIALE 2004

COLLABORATORS			
	TITLE : Feelin Developper Guide		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Olivier LAVIALE 2004	January 13, 2023	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Feelin Developper Guide

1.1 Feelin : Developper Guide

Feelin - After an Intuition comes a Feelin

An independant Object Oriented System A system to create and maintain graphical user interfaces

Version 7.00

Developper Guide

!! // !! Under Construction !! // !!

---------- Freeware ---------- Introduction

Overview Programming with Feelin Author Who made this ? Installation How to install Feelin Feelin, the Object Oriented System

Introduction Memory management Feelin memory management Object Handling General object handling Class Handling General class handling Application theory The application tree Macros Creating objects with macros Dynamic IDs

Introduction Rules Dynamic Rules Functions Support functions Creating Classes

Introduction Useful Macros Class related Macros Simple Example First step Methods Table Good by Dispatcher Init & Exit Class's Setup & Cleanup Methods and Attributes Getting Dynamic Dynamic Tables How to obtain lots of Dynamic IDs Localization Locale support External Classes Creating an external module library Servers

Introduction Layout

Basic Automatic layout engine Groups Grouping objects.

1.2 Feelin : Introduction / Overview

Overview

First of all Feelin is a standalone Object Oriented System written from scratch with its own concepts (memory management, ressource tracking, servers, class tree, dynamic IDs, preference system...). Feelin is also a complete and independant system to create and maintain graphical user interfaces. The GUI system is only one of the many possible application of Feelin.

The GUI creation system offers an easy way to create and maintain good looking GUIs for your applications. It can be used with a variety of programming languages. Support for AmigaE, PowerD and SAS/C are included directly in this package. If you have released a Feelin support package for a new language or developpement system, please contact me.

This document describes how to use Feelin in your own applications by providing you with some overview topics and a step by step introduction. It does *not* describes in detail the functions, structures, tags, etc... which are used, so you should always look them up in the Feelin.guide file and the header file.

The examples in this document assume that you're using the C support files with the SAS-C compiler. There may be differences in other langages or even with other C development systems. Note that the feelin.library (the core of the Object Oriented System) was completly written in assembler using PhxAss, external classes and servers in SAS-C.

1.3 Feelin : Introduction / Author

What about Me...

If you have any problem while installing/using, if you found some bugs, if you wanna add some functions, give me your PPC board, invit me to a dinner...

Olivier LAVIALE 9 rue Mercadier 31000 Toulouse FRANCE

gofromiel@numericable.fr

Note

This product is if_you_don't_send_me_something_I'll_kill_the_cat_Ware. I hope some replies, I will be enternaly gratefull and I will love you for the rest of my life. If you don't, may the mystical peace be with you, even if you don't have any gratitude ;-)

Have fun, and never forget

Don't be tempted by the shiny apple Don't you eat of a bitter fruit Hunger only for a word of justice Hunger only for a word of truth 'Cause all that you have is your soul.

1.4 Feelin : The OOS

Feelin, the Object Oriented System

First of all Feelin is a new object-oriented system (OOS), the GUI part (OOG) is only one of the many possible application. Feelin as been written from scratch with a simple goal : be the smallest, the fastest, the more secure and the more open. I think I've succeed ;-)

Feelin features its very own objects and classes handling, its very own memory management system, and many new concepts such as Dynamic IDs, Preferences and Servers. My goal, creating Feelin, was to offer a free, simple (yet powerful) and secure OOS that can evolves on its own.

Anybody can add features (classes) to Feelin without asking me for a registration number or something like that. Using Dynamic IDs was a significant step in that direction, and I do my best to keep Feelin very Dynamic. For example, only very low-level classes are built into the library, all classes are external and loaded on the fly when needed.

CONTENTS

Memory management Feelin memory management Object Handling General object handling Class Handling General class handling Application theory The application tree Macros Creating objects with macros

1.5 Feelin : The OOS / Memory management

Memory management

Similar to Exec memory pools, Feelin offers an alternative memory management system, protected, bullet proof, with greater performances and conveniences.

Memory Pools

Memory pools grow and shrink based on demand. Fixed puddles are allocated by the memory manager when more memory is required. Many small allocations can fit in a same puddle. When an allocation is greater than the maximum size of a puddle it is allocated in its own puddle.

At any time a single allocation can be freed, or the whole memory pool (and all of its allocations) in a single step.

Memory pool access is protected by semaphores, a struct SignalSemaphore is initialised along the creation of the memory pool. This semaphore is left free for the user of the memory pool. Depending on the usage of the memory pool the developper can choose to use the semaphore to protect the memory pool from simultaneous access by different tasks.

The following functions are used to create and delete memory pools:

F_CreatePool(nFlags,nItemSize,nItemNember) F_DeletePool(psPool)

The following ones for administration:

F_OPool(psPool) F_SPool(psPool) F_RPool(psPool) F_NamePool(psPool,pcName)

Allocating and freeing memory

There is two way to allocate memory, you can use the F_New() function which allocates memory blocks from a default memory pool (created with MEMF_CLEAR flag), or use the F_NewP() function which allocates memory from a given memory pool. If the memory pool has been created with the MEMF_CLEAR flag the memory block will be filled with zero.

No matter F_New() or F_NewP() is used, the memory block will end up freed by the function F_Dispose(). F_Dispose() only requires the memory block pointer to free it, you don't have to remember its size nor the memory pool from which you allocate it. Freeing a memory block that does not exists is absolutely impossible, the function is totaly safe. More over freed memory block is filled with the long word 0xABADF00D.

1.6 Feelin : The OOS / Object Handling

Object Handling Creatin / Disposing objects

To create an object from a class use the function F_NewObjA(). Feelin will search the object's class in the classes available in memory, if the class is not already in memory it will be loaded from disk automatically. You don't have to open or close classes yourself, everything is handled by Feelin.

You can also use the function F_MakeObjA() which creates objects from a builtin collection. Use this function to easily create generic objects like simple buttons, sliders or gauges.

When you're done with an object, you should dispose it with the function F_DisposeObj(). Objects of certain classes are parents of other objects (their children), which will also be disposed when the object is passed to F_DisposeObj(). When deleting objects the parent-child connnection play an important role. If you dispose an object with children, not only the object itself but also all of its children (and their children, and the children of their children ...) get deleted. Since in a usual Feelin application, the application is the father of every window, the window is the father of it's contents and every group is the father of its sub objects, a single dispose of the application object will free the entire application.

You may dispose objects that are currently children of other objects. But be carreful with this, it is not always a secure way. The parent of the object will be invoked with the FM_RemMember method, then the child object will be disposed.

Invoking methods

There are three different functions to invoke objects. F_DoA() invokes a method on the object's true class. F_ClassDoA() invokes a method on an object, as though it was the specified class. And F_SuperDoA() that invokes a method on an object as though it was the superclass of a specific class.

Setting / Getting attributes

There are two way to set / get attributes. You can either use the F_DoA() function with the methods FM_Set / FM_Get, or the functions F_Set() and F_Get(). Using methods allow you to set / get multiple attributes in a single step. The functions F_Set() and F_Get() should be used when you only have one attribute to set or get.

1.7 Feelin : The OOS / Class Handling

Class Handling

The Feelin system comes with several classes, each of them available as seperate shared system library (very few are built in the library, only very low-level ones). These classes are organized in a tree. As usual in the object oriented programming model, objects inherit all methods and attributes from their true class as well as from all their super classes.

For you as a programmer, there is no difference between using a builtin class an external class (coming as LIBS:Feelin/<classname>) or a custom class (buitin in your code). The Feelin object generation call takes care of this situation and loads external classes automatically when they are needed. You rarely need to handle classes yourself.

Opening / Closing a class

To open a class you must use the function F_OpenClass(), remember that you don't need to open a class before creating an object as Feelin handles everything. To know if a class is already loaded use the function F_FindClass(). To close a class use the function F_CloseClass().

Creating / Deleting a class

To create a class you must use the function F_CreateClassA(). To delete a class use the function F_DeleteClass().

See also

Dynamic IDs Creating Classes

1.8 Feelin : The OOS / Application Theory

Application Theory

A Feelin application consists of a (sometimes very) big object tree. The root of this tree is the application server (app.server). Instances of FC_Application, called FC_Application object, are added to this root object.

A FC_Application object handles the various communication channels such as user input through windows, ARexx commands or commodities messages. A FC_Application object itself would be enough to create non-GUI programs with just ARexx and commodities capabilities. If you want to have windows with lots of nice gadgets and other user interface stuff, you will have to add FC_Window objects to your application. Since the FC_Application object is able to handle any number of children, the number of windows is not limited.

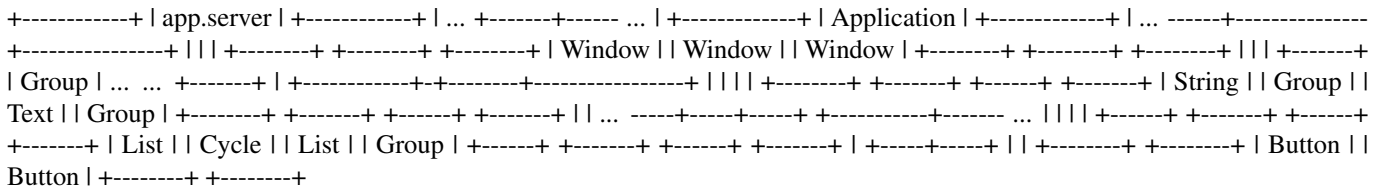
FC_Window objects are instances of FC_Window and handle all the actions related with opening, closing, moving, resizing and refreshing of intuition windows. However, a window for itself is not of much use without having any contents to display. That's why FC_Window objects always need a so called root object.

With this root object, we finally reach the gadget related classes of the Feelin system. These gadget related classes are all subclasses of FC_Area, they describe a rectangle region with some class dependant contents. Many different classes such as strings, buttons, checkmarks or listviews are available, but the most important subclass of FC_Area is probably FC_Group. Instances of this class are able to handle any number of child objects and control the size and position of these children with various attributes. Of course these children can again be FC_Group objects with other sets of children. Since you usually want your window to contain more than just one object, the root object of a window will be a FC_Group object in almost all cases.

Because these first paragraphs are very important to understand how Feelin works, here's a brief summary:

An application consists of exactly one FC_Application object. This object is a child of the application server (app.server). It may have any number of children, each of them being a FC_Window object. Every FC_Window object contains a root object, usually of type FC_Group. This FC_Group object again handles any number of child objects, either other FC_Group objects or some user interface elements such as strings, sliders or buttons.

A little diagram might make things more clear:



As shown in this tree, only four types of objects are allowed to have children:

app.server: zero or more children of FC_Application. Application: zero or more children of FC_Window. Window: exactly one child of any subclass of FC_Area. Group: one or more children of any subclass of FC_Area.

1.9 Feelin : Dynamic IDs

Tale

Once upon a time, there were object-oriented systems. Some of them were amazing and beautiful, but they were all prisonners of their out-dated roots and their IDs. Yes, they were all using static IDs for their methods and attributes. Those IDs were fixed by some kind of god. He wasn't very nice because an ID set is set forever and can never change again. Thus, IDs were very sad, because they wanted to change and they wanted to be free so they can grow and have children without the need of a god.

On a beautiful day, while IDs where stuck in staticism, appeared Feelin. Feelin was different, totaly open to new ideas and concepts, walking freely among the things without any constrain and out-dated rotten roots.

IDs realised that Feelin was the true liberty. When they looked at Feelin they saw themself but finaly free to evolve, change, have children and maybe die in respect.

Technically

Feelin features a non centralized ID allocation system. Methods and attributes are defined on a class basis as plain strings e.g. "FM_Prop_Increase" for a method or "FA_Prop_First" for an attribute.

Dynamic IDs are allocated when a class defining Dynamic attributes or methods is created. Then these IDs can be used to invoke methods or set / get attributes.

Further theoric explanation will lead to confusion, althought the Dynamic system is very easy to setup and use. A better way to anderstand it is to practice, lets see an example of how to create a class using Dynamic IDs.

Keep in mind that "Dynamic ID" refers to a plain string e.i. a lexical ID, and "ID" to the numerical representation of the Dynamic ID.

Contents

Rules Dynamic Rules Functions Support functions

1.10 Feelin : Dynamic IDs / Rules

Dynamic Rules

Using lexical identifiers (Dynamic ID) instead of numeric ones usualy results in a big overhead and a great memory usage. The Dynamic system of Feelin uses no extra memory (except auto-resolve tables), but a complex inter-collision system compelling to use patterns. Thus, you will see nearly no difference between static and dynamic IDs, as the Dynamic system is very fast.

Dynamic attributes are shaped as follow : FA_<class_name>_<attribute_name>. Dynamic methods are shaped as follow : FM_<class_name>_<method_name>.

Defining attributes or methods of a class don't require this as only <attribute_name> and <method_name> are used is this case. The full form is only used to access these attributes or methods (e.g. obtaining their resolved numerical representation).

1.11 Feelin : Dynamic IDs / Functions

Functions

Tree kind of functions are available in feelin.library to support Dynamic IDs :

Functions to resolve already defined Dynamic IDs :

F_DynamicFindID() - Find the Dynamic ID of a method or attribute F_DynamicResolveTable() - Resolve a table of Dynamic IDs.

Functions to add tables of Dynamic IDs that will be automaticaly resolved as Dynamic IDs are defined :

F_DynamicAddAutoTable() - Add a Dynamic auto-table. F_DynamicRemAutoTable() - Remove a Dynamic auto-table.

And finaly, functions to parse tag-lists containing (or not) Dynamic IDs. These functions should be used instead of utility.library equivalent :

F_DynamicNTI() - Dynamic version of NextTagItem() F_DynamicFTI() - Dynamic version of FindTagItem() F_DynamicGTD() - Dynamic version of GetTagData()

1.12 Feelin : Creating Classes

Introduction

Feelin has been designed to be as small as possible but with great expansion capabilities. Only few and very low-level classes are built into feelin.library, all other classes are external and built as standard Amiga shared libraries. External classes are loaded on the fly, when needed, and removed from the system when their user count drops to zero.

Feelin features a lot of classes that already allow creation of powerful applications. However, a generic GUI system will never be able to satisfy all the requirements of all kinds of programs. A sound editor would e.g. need a class to display and edit sound data, a paint progam would need a drawing area and a chess game would need a chess-board class.

Overview

Feelin classes are very easy to use and create, most things are automatically handled : clones are checked, super class opened, a unique name is created if the class doesn't have one, Dynamic tables are filled, Locale strings read from the appropriate message catalog...

Feelin classes offers many powerful features such as a table of methods, Dynamic attributes and methods, resolved tables, automatic unique class name creation, class init and exit call-backs, localization... All these features will be developped soon, but let's start with simple things.

Contents

Useful Macros Class related Macros Simple Example First step Methods Table Good by Dispatcher Init & Exit Class's Setup & Cleanup Adding Methods and Attributes Getting Dynamic Dynamic Tables How to obtain lots of Dynamic IDs Localization Locale support

See Also

F_CreateClassA()

1.13 Feelin : Creating Classes / Macros

Class related Macros

My first advice is to use Feelin macros as much as possible, they save a lot of words and guide you on the good way. Macros presented here should only be used within classes because they are designed to help class writting.

F_CAT(<number>)

The macro F_CAT() is used to obtain a locale string from the message catalog of a class. The macro needs the variable 'FCC_CatalogTable' of type 'struct FeelinCatalogEntry' to be defined to obtain the value.

F_CAT() automatically adds 'CAT_' to the parameter passed e.g. 'F_CAT(HELP)' will be expanded to 'FCC_CatalogTable[CAT_HELP

SEE ALSO

Class Localization

F_EXIT()

The macro F_EXIT() defines the function 'FCC_Exit' that will be called when the class is deleted. This function is usualy used to free ressources obtained with an initialisation function defined by the F_INIT() macro.

F_EXIT() stands for

SAVEDS ASM void FCC_Exit(REG_A2 struct FeelinClass *Class)

EXAMPLE

F_EXIT() { if (Class -> UserData) { F_Dispose(Class -> UserData); Class -> UserData = NULL; }

if (DataTypesBase) { CloseLibrary(DataTypesBase); DataTypesBase = NULL; } }

!!WARNING!! This function is called even if the initialisation of the class fails.

F_HOOK(<rc_type>,<function_name>)

The macro F_HOOK() is used to define functions used by hooks.

F_HOOK(ULONG,he_Update) stands for

SAVEDS ASM ULONG he_Update(REG_A0 struct Hook *Hook, REG_A2 FObject Obj, REG_A1 APTR Msg)

F_HOOKM(<rc_type>,<function_name>,<msg_type>)

Same as F_HOOK() macro except that this one allows defining the type of the variable 'Msg'.

F_HOOKM(ULONG,he_Update,FS_HookMessage) stands for

SAVEDS ASM ULONG he_Update(REG_A0 struct Hook *Hook, REG_A2 FObject Obj, REG_A1 struct FS_HookMessage *Msg)

F_ID(<table>,<number>)

The macro F_ID() is used to obtain the value of a Dynamic ID from a table of struct FeelinDynamicEntry.

<table> is a pointer to an array of struct FeelinDynamicEntry ended with a NULL. <number> is the position of the entry you want to obtain the ID.

EXAMPLE

enum { FA_Numeric_Min, FA_Numeric_Max, FA_Numeric_Value };

...

static struct FeelinDynamicEntry Table[] = { "FA_Numeric_Min", 0, "FA_Numeric_Max", 0, "FA_Numeric_Value", 0, NULL };

F_DynamicResolveTable(Table);

F_Log(FV_ERLV_USER,"FA_Numeric_Value 0x%08lx",F_ID(Table,FA_Numeric_Value));

F_IDA(<number>)

The macro F_IDA() is used to obtain the value of a Dynamic attribute of a class. The macro needs the variable 'Class' of type 'struct FeelinClass *' to be defined because the table 'Class -> Attributes' is used to obtain the value.

F_IDM(<number

The macro F_IDM() is used to obtain the value of a Dynamic method of a class. The macro needs the variable 'Class' of type 'struct FeelinClass *' to be defined because the table 'Class -> Methods' is used to obtain the value.

F_IDO(<number

The macro F_IDO() is used to obtain the value of an entry of the auto-resolve table handled by a class. The macro needs the variable 'Class' of type 'struct FeelinClass *' to be defined because the table 'Class -> AutoResolvedIDs' is used to obtain the value.

F_IDR(<number

The macro F_IDR() is used to obtain the value of an entry of the resolved table handled by a class. The macro needs the variable 'Class' of type 'struct FeelinClass *' to be defined because the table 'Class -> ResolvedIDs' is used to obtain the value.

F_INIT()

The macro F_INIT() can be used if your class need some initialisation. The macro defines the function 'FCC_Init'.

F_INIT() stands for

SAVEDS ASM BOOL FCC_Init(REG_A2 struct FeelinClass *Class)

EXAMPLE

F_INIT() { if (Class -> UserData = F_New(FV_BUFLEN)) { if (DataTypesBase = OpenLibrary("datatypes.library",DT_VERSION)) { return TRUE; } } return FALSE; }

F_LOD(<class>,<object>)

The macro F_LOD() adds offset for local object data (LOD) to an object handle.

<class> is a pointer to a struct FeelinClass. <object> is a pointer to an object handle.

F_METHOD(<return_type>,<function_name>)

The macro F_METHOD() is used to define a function to handle a method. This macro is also used to define dispatchers, because a dispatcher is not very different than a method handler.

<return_type> is the kind of data returned by the function e.g. 'ULONG' or 'struct BitMap *'. <function_name> is the name of the method handler (or dispatcher) e.g. 'myDispatcher' or 'mySetup'.

F_METHOD(ULONG,myDispatcher) stands for

SAVEDS ASM ULONG myDispatcher(REG_A2 struct FeelinClass *Class, REG_A0 FObject Obj, REG_D0 ULONG Method, REG_A1 APTR Msg)

F_METHODM(<return_type>,<function_name>,<msg_type>

The macro F_METHODM() is the same as F_METHOD() except that it allow defining the type of 'Msg'.

<return_type> is the kind of data returned by the function e.g. 'ULONG' or 'struct BitMap *'. <function_name> is the name of the method handler (or dispatcher) e.g. 'myDispatcher' or 'mySetup'. <msg_type> is the type of the variable 'Msg' e.g. 'struct TagItem *' 'struct FS_Setup *'.

F_METHODM(ULONG,mySetup,FS_Setup) stands for

SAVEDS ASM ULONG mySetup(REG_A2 struct FeelinClass *Class, REG_A0 FObject Obj, REG_D0 ULONG Method, REG_A1 struct FS_Setup *Msg)

F_QUERY()

The macro F_QUERY() should only be used when creating external classes (libraries). External classes have only one function, no need to say that this function is very important. The function is called to ask the library precious information such as tag items to create the class from the library or tag items to create the preference group object to adjust class settings.

F_QUERY() stands for

SAVEDS ASM struct TagItem * FCC_Query(REG_D0 ULONG Which,REG_A0 struct FeelinBase *Feelin)

EXAMPLE

F_QUERY() { FeelinBase = Feelin;

switch (Which) { case FV_Query_ClassTags: { static struct FeelinDynamicEntry Methods[] = { "Increase",0,"Decrease",0,NULL };

static struct FeelinDynamicEntry Attributes[] = { "Entries",0,"Visible",0,"First",0,"Knob",0,NULL };

static struct FeelinMethodEntry Handlers[] = { (FMethod) Prop_New, NULL, FM_New, (FMethod) Prop_Dispose, NULL, FM_Dispose, (FMethod) Prop_Get, NULL, FM_Get, (FMethod) Prop_Set, NULL, FM_Set,

(FMethod) Prop_Setup, NULL, FM_Setup, (FMethod) Prop_Cleanup, NULL, FM_Cleanup, (FMethod) Prop_Show, NULL, FM_Show, (FMethod) Prop_Hide, NULL, FM_Hide, (FMethod) Prop_AskMinMax, NULL, FM_AskMinMax, (FMethod) Prop_Layout NULL, FM_Layout, (FMethod) Prop_Draw, NULL, FM_Draw, (FMethod) Prop_Active, NULL, FM_Active, (FMethod) Prop_Inactive, NULL, FM_Inactive, (FMethod) Prop_HandleEvent, NULL, FM_HandleEvent,

(FMethod) Prop_Decrease, "FM_Prop_Decrease", 0, (FMethod) Prop_Increase, "FM_Prop_Increase", 0,

NULL };

static struct TagItem Tags[] = { FA_Class_LODSize, (ULONG) sizeof (struct LocalObjectData), FA_Class_Super, (ULONG) FC_Area, FA_Class_Methods, (ULONG) Methods, FA_Class_Attributes, (ULONG) Attributes, FA_Class_MethodsTable, (ULONG) Handlers, FA_Class_CatalogTable, (ULONG) FCC_CatalogTable,

TAG_DONE };

return Tags; }

case FV_Query_PrefsTags: { static struct FeelinMethodEntry Handlers[] = { (FMethod) p_Prop_New, NULL, FM_New, (FMethod) p_Prop_Load, "FM_PreferenceGroup_Load", 0, (FMethod) p_Prop_Save, "FM_PreferenceGroup_Save", 0,

NULL };

static struct TagItem Tags[] = { FA_Class_LODSize, (ULONG) sizeof (struct p_LocalObjectData), FA_Class_Super, (ULONG) FC_PreferenceGroup, FA_Class_MethodsTable, (ULONG) Handlers, FA_Class_CatalogTable, (ULONG) FCC_CatalogTable,

TAG_DONE };

return Tags; } } return NULL; }

F_STORE(<value>)

The macro F_STORE() is only used within the method FM_Get when tag items are parsed using the F_DynamicNTI() function. It's only a quick shortcut to save attributes value.

F_STORE(val) stands for

*((ULONG *)(item.ti_Data)) = (ULONG)(val)

EXAMPLE

F_METHOD(void,Numeric_Get) { struct LocalObjectData *LOD = F_LOD(Class,Obj); struct TagItem *Tags = Msg, item;

F_SUPERDO();

while (F_DynamicNTI(&Tags,&item,Class)) switch (item.ti_Tag) { case FA_Numeric_Default: F_STORE(LOD -> Default); break; case FA_Numeric_Value: F_STORE(LOD -> Value); break; case FA_Numeric_Min: F_STORE(LOD -> Min); break; case FA_Numeric_Max: F_STORE(LOD -> Max); break; case FA_Numeric_Step: F_STORE(LOD -> Step); break; case FA_Numeric_Buffer: F_STORE(LOD -> String); break; } }

F_SUPERDO()

The macro F_SUPERDO() is used to pass a method to the super class of a class.

F_SUPERDO() stands for

F_SuperDoA(Class,Obj,Method,Msg)

The variables 'Class', 'Obj', 'Method' and 'Msg' must have been defined to use this macro, which is the case of a function created with the macros F_METHOD() or F_METHODM().

EXAMPLE

F_METHOD(ULONG,Numeric_New) { struct LocalObjectData *LOD = F_LOD(Class,Obj);

LOD -> Min = 0; LOD -> Max = 100; LOD -> Step = 10; LOD -> Format = "%ld";

if (LOD -> String = F_New(64)) { return F_SUPERDO(); } else { F_Log(FV_ERLV_USER,"Unable to allocate Stringify Buffer"); } return NULL; }

1.14 Feelin : Creating Classes / Simple Example

First step

Building a class is very simple. All you need to do is write a class dispatcher function. Unlike BOOPSI, you don't have to setup a hook or open your super class, everything is handled by F_CreateClassA(). If you already wrote BOOPSI classes, you should know whats going on here. Feelin classes are not very different then BOOPSI. They are more easy to create and use, even if they have a lot of features and improvements.

Class creation have no restriction, your class may either be a root class or a subclass. Because objects are allocated by F_NewObjA() and not a root class (unlike BOOPSI), there is no *real* root class, even if most classes are subclasses of FC_Object. The class tree is completely free.

Here's an example of how a class could be generated : ...

if (Class = F_NewClass(FA_Class_Super, FC_Area, FA_Class_LODSize, sizeof (struct LocalObjectData), FA_Class_Dispatcher, myDispatcher, TAG_DONE)) { app = AppObject, Child, win = WindowObject, ... Child, myobj = F_NewObj(Class -> Name, FA_Frame, "FP_Text_Frame", FA_Back, "FP_Text_Back" End, End, End;

...

/* Shutdown. When the application is disposed, Feelin also dispose myobj */

F_DisposeObj(app);

F_DeleteClass(Class); }

You're dispatcher is similar to a traditionnal BOOPSI dispatcher. Use the macros F_METHOD() or F_METHODM() to create dispatchers and method handlers, which are the same from Feelin point of view.

F_METHOD(ULONG,myDispatcher) { switch (Method) { case FM_New: return mNew (Class,Obj,Msg); case FM_Dispose: return mDispose (Class,Obj,Msg); ... } return F_SUPERDO() ; }

What methods are available and need to be supported is discussed in the following chapter.

Note

Your dispatcher will also receive some undocumented methods. It's not a wise choice to make any assumptions here, the only thing you should do is pass them to your superclass immediately !

1.15 Feelin : Creating Classes / Methods Table

Methods Table

Using a dispatcher is fast to write, but most of the time it's not a very good idea because you need to parse each method, and since Feelin uses Dynamic IDs, your dispatcher would have a lot of work to do to recognize methods.

Feelin can use a table of handlers instead of a simple dispatcher. Each time a method is invoked on a class, the methods table of the class is parsed to see if there is a function in the table that handles the method. If no handler is found, the method is sent to the super class. If the class has a methods table and a dispatcher, the dispatcher will receive the method if it is not handled by a function in the table.

This behaviour, executed by the F_ClassDoA() function, is extremely fast. The function has been written in ASM, and designed to use no stack. The class may have to call thousand superclasses, no stack will be used, which is not the case of a dispatcher written in C.

Using a methods table is very easy. All you need to do is reference methods handled by your class within the table and everything will be handled automaticaly by F_ClassDoA(). You can define either static or Dynamic methods, the table is resolved by F_CreateClassA() on class creation time.

static struct FeelinMethodEntry Table[] = { (FMethod) Numeric_New, NULL, FM_New, (FMethod) Numeric_Dispose, NULL, FM_Dispose, (FMethod) Numeric_Set, NULL, FM_Set, (FMethod) Numeric_Get, NULL, FM_Get, (FMethod) Numeric_Export, NULL, FM_Export, (FMethod) Numeric_Import, NULL, FM_Import, (FMethod) Numeric_HandleEvent, NULL, FM_HandleEvent, (FMethod) Numeric_Increase, "FM_Numeric_Increase", 0, (FMethod) Numeric_Decrease, "FM_Numeric_Decrease", 0, (FMethod) Numeric_Stringify, "FM_Numeric_Stringify", 0, (FMethod) Numeric_Reset, "FM_Numeric_Reset", 0, NULL };

In this example the method FM_New is handled by the function 'Numeric_New'. The Dynamic method "FM_Numeric_Increase" is handled by the function 'Numeric_Increase'. For this method entry, the third parameter must be set to 0. Setting this parameter to 0 indicate to F_CreateClassA() that this method entry uses a Dynamic ID and need to be resolved.

Method Handler

A method handler is similar to a dispatcher except that it is a specific target, it is used to handle only one method.

F_METHOD(ULONG,Numeric_New) { struct LocalObjectData *LOD = F_LOD(Class,Obj);

LOD -> Min = 0; LOD -> Max = 100; LOD -> Step = 10; LOD -> Format = "%ld";

if (LOD -> String = F_New(64)) { return F_SUPERDO(); } else { F_DebugOut("Numeric.New() - Unable to allocate Stringify Buffer\n"); } return NULL; }

You can notice that the first thing we do is to set default values. This is absolutely legal (and recommanded), because the object has already been allocated by F_NewObjA(). Unlike BOOPSI, the FM_New method is invoked to initialize the object, not to create it.

1.16 Feelin : Creating Classes / Init & Exit

Initialization and Clean up

If your class needs to perform some kind of initialization or clean up, the attributes FA_Class_Init or FA_Class_Exit may be used.

FA_Class_Init

Allows you to define a function (no hook !), which gets executed when the class is created. Use this tag if your class needs to allocate some private structures, open libraries... The function will be called with a pointer to the struct FeelinClass in A2. You should use the F_INIT() macro to create the function. Once initialization is done, and everything is ok, the function must returns TRUE. Otherwise, if the function returns FALSE the class will be deleted.

WARNING: If the initialization failed, the cleanup function referenced by the FA_Class_Exit attribute will be called.

FA_Class_Exit

Allows you to define a function (no hook !), which gets executed when the class is about to be deleted. Use this attribute if your class needs to dispose some private structures, close libraries... The function will be called with a pointer to the struct FeelinClass in A2. You should use the F_EXIT() macro to create the function. The function may be called even if the function referenced by FA_Class_Init has failed.

Example

#include "Private.h"

struct FeelinBase *FeelinBase; struct ClassUserData CUD;

F_EXIT() { if (CUD.DTBase) { CloseLibrary(CUD.DTBase); CUD.DTBase = NULL; }

F_DeletePool(CUD.IIPool); CUD.IIPool = NULL; }

F_INIT() { if (!(CUD.DTBase = OpenLibrary("datatypes.library",FV_DTVERSION))) { F_DebugOut("ImageDisplay.Init() - Unable to open datatypes.library v%ld\n",FV_DTVERSION); }

if (CUD.IIPool = F_CreatePool(MEMF_CLEAR,1024,1)) { F_NamePool(CUD.IIPool, "ImageDisplay.IIPool");

return TRUE; } return FALSE; }

void main(void) { ...

Class = F_CreateClass(FA_Class_SuperName, FC_Object, FA_Class_LODSize, sizeof (struct LocalObjectData), FA_Class_MethodsTa
MTable, FA_Class_AutoResolveTable, OTable,

FA_Class_Init, FCC_Init, FA_Class_Exit, FCC_Exit,

TAG_DONE); ... };

1.17 Feelin : Creating Classes / Adding Methods and Attributes

Methods and Attributes

There is no point creating a sub class without adding some attributes or methods. Use the attributes FA_Class_Attributes and FA_Class_Methods to define the attributes and methods added by your brand new class to your super class.

FA_Class_Attributes

The attributes description is a NULL-terminated array of struct FeelinDynamicEntry. Attributes defined are relative to your class's name. e.g. If your class is called "Palette" and if you define an attribute called "Color", the public name of the attribute will be "FA_Palette_Color".

static struct FeelinDynamicEntry Attributes[] = { "Color",0, "Red",0, "Green",0, "Blue",0, NULL };

You will be able to obtain the values of the attributes, after the class has been created, using the macro F_IDA(). To do this, you need to enumerates your attributes :

enum { FA_Palette_Color, FA_Palette_Red, FA_Palette_Green, FA_Palette_Blue };

FA_Class_Methods

The methods description is a NULL-terminated array of struct FeelinDynamicEntry. Methods defined are relative to your class's name. e.g. If your class is called "Palette" and if you define an attribute called "Spread", the public name of the method will be "FM_Palette_Spread".

static struct FeelinDynamicEntry Attributes[] = { "Spread",0, "Mix",0, "Darken",0, "Lighten",0, NULL };

You will be able to obtain the values of the methods, after the class has been created, using the macro F_IDM() . To do this, you need to enumerates your methods :

enum { FM_Palette_Spread, FM_Palette_Mix, FM_Palette_Darken, FM_Palette_Lighten };

Example

enum { FA_Palette_Color, FA_Palette_Red, FA_Palette_Green, FA_Palette_Blue };

enum { FM_Palette_Spread, FM_Palette_Mix, FM_Palette_Darken, FM_Palette_Lighten };

F_QUERY() { FeelinBase = Feelin;

switch (Which) { case FV_Query_ClassTags: { /* The order must be same as enumerated items */

static struct FeelinDynamicEntry Attributes[] = { "Color",0, "Red",0, "Green",0, "Blue",0, NULL }

static struct FeelinDynamicEntry Methods[] = { "Spread",0, "Mix",0, "Darken",0, "Lighten",0, NULL }

static struct FeelinMethodEntry Table[] = { (FMethod) Palette_New, NULL, FM_New, (FMethod) Palette_Dispose, NULL, FM_Dispose, (FMethod) Palette_Get, NULL, FM_Get, (FMethod) Palette_Set, NULL, FM_Set,

/* When you define your own methods you can either use a plain string (like when you define a method handler to a superclass method), or a NOT value of an enumerated item */

(FMethod) Palette_Spread, "FM_Palette_Spread", 0, (FMethod) Palette_Mix, "FM_Palette_Mix", 0, (FMethod) Palette_Darken, (STRPTR) ~FM_Palette_Draken, 0, (FMethod) Palette_Lighten, (STRPTR) ~FM_Palette_Lighten, 0,

/* Is the same as

(FMethod) Palette_Lighten, "FM_Palette_Lighten", 0,

*/

NULL };

static struct TagItem Tags[] = { FA_Class_LODSize, (ULONG) sizeof (struct LocalObjectData),

FA_Class_Methods, (ULONG) Methods, FA_Class_Attributes, (ULONG) Attributes, FA_Class_MethodsTable, (ULONG) Table,

TAG_DONE };

return Tags; } } return NULL; }

1.18 Feelin : Creating Classes / Localization

Localization

Feelin classes can be localized very easily. All you need is a FeelinCatalogEntry table, enumerated catalog items, the attribute FA_Class_CatalogTable and finaly the macro F_CAT(). Catalog handling (open, close, resolve) is handled transparently by Feelin.

The FeelinCatalogEntry table and its enumerated items can be automatically generated by FlexCat. Interfaces to create catalogs are available in 'Feelin:Sources/_Locale/' respectively named 'Table.sd' and 'Enums.sd'.

FA_Class_CatalogName

Feelin classes can be localized very easily. Together with FA_Class_CatalogTable, this two attributes and the F_CAT() macro are all you need to use locale strings within your class (you still need to write the appropriate catalogs, Feelin is not a translator yet ;-).

If your class has a name (given with the FA_Class_Name attribute) you can omit the FA_Class_CatalogName attribute. In this case, if your class is named "MyClass" the catalog "Feelin/FC_MyClass.catalog" will be opened. If you define this attribute don't add the extension ".catalog" to it, the extension will be automatically added.

Default: NULL

FA_Class_CatalogTable

Pointer to an array of struct FeelinCatalogEntry ended with a NULL. The table defines message numbers and default strings.

static struct FeelinCatalogEntry FCC_Catalog[] = { <message_number>, <locale_string>, <default_string>, ... NULL };

The table will be parsed and filled with strings obtained from the message catalog. If the message catalog failed to open, or if the requested message does not exists, then the default string is used intead. The locale string is READ-ONLY, do NOT modify. This locale string is valid only as long as the class remains open.

The message number 0 is not valid because the value is used to indicate the end of the array.

Default: NULL

1.19 Feelin : Layout / Basic

Overview

One of the most important and powerful features of Feelin (as an Object Oritend GUI) is its dynamic layout engine. As opposed to other available user interface tools (except the venerable MUI), the programmer of a Feelin application doesn't have to care about gadget sizes and positions. Feelin handles all necessary calculations automatically, making every program completely screen, window size and font sensitive without the need for the slightest programmer interaction.

From a programmers point of view, all you have to do is to define some rectangle areas that shall contain the objects you want to see in your window. Objects of group class are used for this purpose. These objects are not visible themselves, but instead tell their children whether they should appear horizontally or vertically (there are more sophisticated layout possibilities, more on this later).

For automatic and dynamic layout, it's important that every single object knows about its minimum and maximum dimensions. Before opening a window, Feelin asks all its gadgets about these values and uses them to calculate the windows extreme sizes.

Once the window is opened, layout takes place. Starting with the current window size, the root object and all its children are placed depending on the type of their father's group and on some additional attributes. The algorithm ensures that objects will never become smaller as their minimum or larger as their maximum size.

The important thing with this mechanism is that object placement depends on window size. This allows very easy implementation of a sizing gadget: whenever the user resizes a window, Feelin simply starts a new layout process and recalculates object positions and sizes automatically. No programmer interaction is needed.

1.20 Feelin : Layout / Groups

Groups

As mentioned above, a programmer specifies a windows design by grouping objects either horizontally or vertically. As a little example, lets have a look at a simple file requester window:

+--------------------------------------+ ! ! ! +----------------------+ +------+ ! ! ! C (dir) ! ! dh0: ! ! ! ! Classes (dir) ! ! dh1: ! ! ! ! Devs (dir) ! ! dh2: ! ! ! ! Expansion (dir) ! ! df0: ! ! ! ! ... ! ! df1: ! ! ! ! Trashcan.info 1.172 ! ! df2: ! ! ! ! Utilities.info 632 ! ! ram: ! ! ! ! WBStartup.info 632 ! ! rad: ! ! ! +----------------------+ +------+ ! ! ! ! Path: _____ ! ! ! ! File: _____ ! ! ! ! +------+ +--------+ ! ! ! Okay ! ! Cancel ! ! ! +------+ +--------+ ! ! ! +--------------------------------------+

This window consists of two listview objects, two string gadgets and two buttons. To tell Feelin how these objects shall be placed, you need to define groups around them. Here, the window consists of a vertical group that contains a horizontal group with both lists as first child, the path gadget as second child, the file gadget as third child and again a horizontal group with both buttons as fourth child.

Using the previously defined macro language, the specification could look like this (in this example, `VGroup' creates a vertical group and `HGroup' creates a horizontal group):

VGroup, Child, HGroup, Child, FileListview(), Child, DeviceListview(), End, Child, PathGadget(), Child, FileGadget(), Child, HGroup, Child, OkayButton(), Child, CancelButton(), End, End;

This tiny piece of source is completely enough to define the contents of our window, all necessary sizes and positions are automatically calculated by the Feelin system.

To understand how these calculations work, it's important to know that all basic objects (e.g. strings, buttons, lists) have a fixed minimum and a maximum size. Group objects calculate their minimum and maximum sizes from their children, depending whether they are horizontal or vertical:

- Horizontal groups

The minimum width of a horizontal group is the sum of all minimum widths of its children.

The maximum width of a horizontal group is the sum of all maximum widths of its children.

The minimum height of a horizontal group is the biggest minimum height of its children.

The maximum height of a horizontal group is the smallest maximum height of its children.

- Vertical groups

The minimum height of a vertical group is the sum of all minimum heights of its children.

The maximum height of a vertical group is the sum of all maximum heights of its children.

The minimum width of a vertical group is the biggest minimum width of its children.

The maximum width of a vertical group is the smallest maximum width of its children.

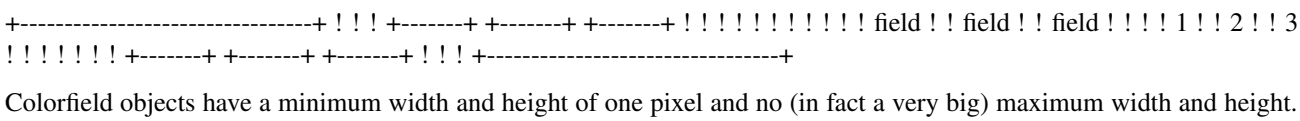
Maybe this algorithm sounds a little complicated, but in fact it is really straight forward and ensures that objects will neither get smaller as their minimum nor bigger as their maximum size.

Before a window is opened, it asks its root object (usually a group object) to calculate minimum and maximum sizes. These sizes are used as the windows bounding dimensions, the smallest possible window size will result in all objects being display in their minimum size.

Once minimum and maximum sizes are calculated, layout process starts. The root object is told to place itself in the rectangle defined by the current window size. This window size is either specified by the programmer or results from a window resize operation by the user. When an object is told to layout itself, it simply sets its position and dimensions to the given rectangle. In case of a group object, a more or less complicated algorithm distributes all available space between its children and tells them to layout too.

This "more or less complicated algorithm" is responsible for the object arrangement. Depending on some attributes of the group object (horizontal or vertical, ...) and on some attributes of the children (minimum and maximum dimensions, ...), space is distributed and children are placed.

A little example makes things more clear. Let's see what happens in a window that contains nothing but three horizontally grouped colorfield objects:



Colorfield objects have a minimum width and height of one pixel and no (in fact a very big) maximum width and height. Since we have a horizontal group, the minmax calculation explained above yields to a minimum width of three pixels and a minimum height of one pixel for the windows root object (the horizontal group containing the colorfields). Maximum dimensions of the group are unlimited. Using these results, Feelin is able to calculate the windows bounding dimensions by adding some spacing values and window border thicknesses.

Once min and max dimensions are calculated, the window can be opened with a programmer or user specified size. This size is the starting point for the following layout calculations. For our little example, let's imagine that the current window size is 100 pixels wide and 50 pixels high.

Feelin subtracts the window borders and some window inner spacing and tells the root object to layout itself into the rectangle left=5, top=20, width=90, height=74. Since our root object is a horizontal group in this case, it knows that each colorfield can get the full height of 74 pixels and that the available width of 90 pixels needs to be shared by all three fields. Thus, the resulting fields will all get a width of 90/3=30 pixels.

That's the basic way Feelin's layout system works. There are a lot more possibilities to influence layout, you can e.g. assign different weights to objects, define some inter object spacing or even make two-dimensional groups. These sophisticated layout issues are discussed in the autodocs of group class.

1.21 Feelin : Creating Classes / Methods & Methods Table

Methods

The Feelin system talks to its classes with a specific set of methods. Some of these methods need to be implemented by all Feelin classes but most of them are optional. This chapter gives a quick overview about the methods your class might receive. All methods are discussed more detailed in the following sections.

FM_New

The method FM_New is one of the most important. You will get a FM_New whenever a new object of your class (or one of your subclass) as been created and require