Flavors4C

User's Manual

Versions 1.0

by Peter Ohler

Disclaimer of warranty

THIS SOFTWARE AND MANUAL ARE PROVIDED "AS IS" AND WITHOUT WARRANTIES AS TO PERFORMANCE OR MERCHANTABILITY. THIS SOFTWARE IS PROVIDED WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES WHATSOEVER. BECAUSE OF THE DIVERSITY OF CONDITIONS AND HARDWARE UNDER WHICH THIS SOFTWARE MAY BE USED, NO WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE IS OFFERED. THE USER IS ADVISED TO TEST THE SOFTWARE THOROUGHLY BEFORE RELYING ON IT. THE USER MUST ASSUME THE ENTIRE RISK OF USING THE SOFTWARE. ANY LIABILITY OF SELLER OR MANUFACTURER WILL BE LIMITED EXCLUSIVELY TO PRODUCT REPLACEMENT OR REFUND OF THE PURCHASE PRICE.

Contents

Chapter 1: Introduction 1-1
Some Background 1-1
Document Contents 1-2

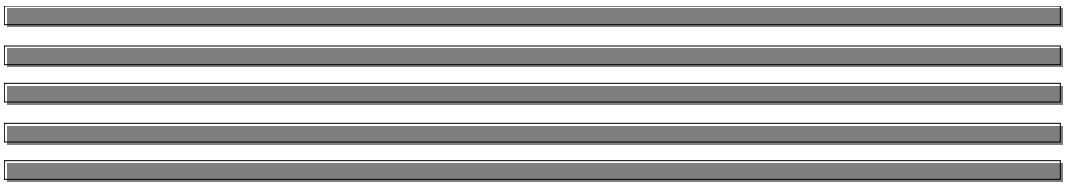
Chapter 2: Object Oriented Programming with C 2-1
What is Object Oriented Programming? 2-1
Object Oriented Techniques with C 2-3
Why Use Flavors4C? 2-4

Chapter 3: Features of Flavors4C 3-1

Chapter 4: Flavors4C Reference Manual 4-1
Flavors4C Variables and Types 4-1
Flavors4C Functions 4-3

Chapter 5: Hints for Using Flavors4C 5-1

Index Index-1



1. Introduction

There are numerous benefits to be realized by using object oriented programming techniques during software development. The advantages of using object oriented techniques have been examined in detail by many researchers and many articles have been written which expound upon the virtues of object oriented systems. Without trying to go into the detail, some of the advantages of object oriented techniques are:

- Object oriented designs are ideal for large systems that become too complex to manage reasonably with other programming techniques.
- Object oriented programs can be easier to debug due to the high degree of modularity inherent it the object oriented approach.
- Software is easier to maintain and expand in a well designed object oriented program.
- Reusable programs and modules are a natural outcome of object oriented systems.

Some Background

C has become a popular language for code development due to it's portability and it's execution speed. It does, however, have it's shortcomings. One of these shortcomings is the lack of tools that allow developers to use object oriented coding. There have been several attempts to remedy this situation but none have been entirely successful. None of these attempts to add object oriented capability to C have had all of the features found in robust object oriented systems such as Smalltalk or LISP flavors. Some of the more well known object oriented C systems such as C++ and Objective-C have added some limited degree of object oriented capability, but only at the expense of using preprocessors or by creating a new language and compiler. The lack of a fully featured object oriented system for C that is pure C prompted the development of Flavors4C.

One of the primary considerations in the design of Flavors4C was that the package should have all the features found in one of the more robust systems such as Smalltalk or LISP flavors. LISP flavors is designed to be utilized as a hybrid system. LISP Flavors allows for procedural calls when procedural code is more efficient and object oriented code when that is the best way to handle a task. C is a procedural language and any object oriented extension should provide the object oriented extensions needed to make C a hybrid system so that the synergistic effect of having both procedural and object oriented code

intermixed can be realized. The LISP flavor system, being a fully featured hybrid system, was chosen as the model for Flavors4C.

There are several ways in which the Flavors4C system could have been implemented; as a compiler, as a preprocessor, as a highly structured set of coding rules, or as a library of functions. Creating a new compiler was not considered since it would reduce the portability of system and would require a significant duplication of effort that has been put into C compilers by many other individuals. The preprocessor method would be one the fastest for run time but it would be very difficult to handle some of the more advanced features of flavors found in the LISP world. It would also be very difficult to debug with existing debugging tools. The second alternative of following some rigid set of coding rules would also be very fast at run time, but it was really only slightly different than the preprocessor solution. The third alternative, the library method would be only slightly slower but it could have every feature found in LISP flavors. The slight drop in speed anticipated by using a library instead of a preprocessor or a rigid set of coding rules is insignificant in a properly constructed application. A simple analysis of the processing utilized for each function in a properly designed object oriented application will make it clear that even a significant speed degradation in the object oriented calls will have an insignificant effect on the overall performance of an application.

In general object oriented techniques should be used only at the higher levels of the software design where calls to functions or messages sent to objects are used for control purposes. At the lower level of the design, where repetitive task occur frequently, procedural calls should be uses since the tasks are usually better suited to the use of procedural calls, which require less overhead. Experience has shown that in large hybrid systems written in LISP on LISP machines, the object oriented calls are used for control and the procedural code is used for lower level operations. With the mix of procedural and object oriented code distributed as described even drastic decreases in the speed of the object oriented calls have only negligible effects on the overall performance of a hybrid software project.

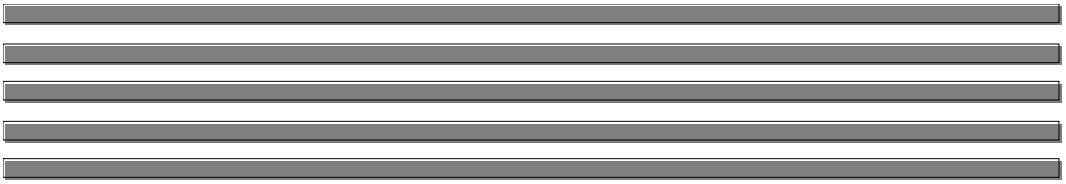
The design of Flavors4C was driven primarily by the features that would make the package as useful and as complete as possible. In some cases the LISP flavors model was enhanced to provide additional capability and in some ways the model was modified to keep the package a true C library. Execution speed was always kept as a high priority in the design and as a result the package is fairly fast as well as being complete.

Document Contents

The rest of this document is composed of several sections that:

- explain briefly how object oriented coding fits in with the C language,
- describe the features of Flavors4C,
- provide descriptions for each function and variable in Flavors4C, and
- suggest hint for efficient use of Flavors4C.

I hope you find Flavors4C useful and any comments would be appreciated.



2. Object Oriented Programming with C

There are many benefits to using object oriented techniques. Combining the advantages of object oriented techniques with the beneficial features found in C can be a real boost to software development. Flavors4C, with the variety of features included in the package, enhances the benefits of using object oriented techniques with C.

What is Object Oriented Programming?

Object oriented programming is programming with a different perspective than the conventional approach to coding. Object oriented programs can be viewed as worlds filled with creatures or objects that interact with each other to accomplish a common goal. The programmer teaches the creatures or objects how to behave and the objects interact to cause the program to be executed. It seems as if program control is lost to the objects. In actuality the control methodology has been simplified and reduced to a more modular approach. The programmers has a variety of ways in which to control the objects that are created and therefore has control over the program execution. Once this control paradigm is mastered, control of the program is easier to understand and can be abstracted to different levels to avoid confusion and to improve the organization of complex programs.

Instead of a program made up of function calls an object oriented program consists of object that interact with each other. Every object acts independently of other objects with all interactions between objects provided by a message passing facility. The flow of control in conventional programming passes from one function to another. In object oriented programming the flow of control is passed from object to object in the form of messages. The difference in the two methodologies is best illustrated by an example. In a procedural environment, if the temperature of a box is needed a function would be written to read the temperature. In an object oriented system the box would be taught to respond to a message that has requested the temperature of the box. The actual function that returns the temperature may be the same but the way that function is called is different in each system.

Each object has local variable or information that it knows about and each object knows how to respond to certain messages. The local variables are called Instance Variables and the value of Instance Variables are only known

to the object itself. Outside sources can only get at the instance variables of an object by asking the object for the value of that variable.

Objects respond to messages according to their type or class. Objects of different classes can react differently to the same message. The way a class of objects responds to a message is determined by the methods attached to the object class. Methods are basically functions that are aware of the object that the message was sent to and that have access to the Instance Variables of that object. Instance Variables as well as methods for an object are determined by the object's inherited classes.

Each object is an instance of a class of objects. In Flavors4C, as in LISP flavors these classes are called flavors. The names and types of instance variables are the same for all the objects in a specific flavor. The values of the instance variables can be different for each instance of a flavor, although initial values are included as part of each flavor. The methods for each flavor control how each instance of a flavor responds to messages. The methods and instance variables for a flavors can be defined for the flavor specifically or they can be inherited. Flavors can inherit from any number of other flavors and all the variables of the parent flavors and all the methods for the parent flavors become part of the new flavor. If two inherited flavors have the same instance variable then that variable appears only once in the new flavor, not twice. If two inherited flavors both have the same method then both methods are inherited by the new flavor. The way the methods with the same name are executed for a flavor is determined by the order of the inherited flavors and the method combination for the methods in the flavor. The modularity and reusability of object oriented systems is a result of the ability to mix flavors together to obtain the desired behavior in an object.

There are several ways that a programmer can control the inheritance of flavors. The normal order for Flavors4C is the same order that the flavors appear in the inheritance list. The order can be controlled further by specifying any required or included flavors. The order of the inherited flavors is important for method execution when messages are received. The methods for a messages are executed according to the inheritance order and the methods combination specifier.

There are four basic method types that determine the order in which the methods are executed in conjunction with the method combination. The method types are *primary*, *before*, *after*, and *whopper*. The way in which these types are executed is explained in the reference section of this manual.

Once a flavors inheritance has been set up, a flavor's method combination can still be used to control the way an object responds to a message sent to it. The default way that methods are executed for a message is the Daemon combination. This scheme causes the value of the method of the topmost flavor from the inherited flavors to be returned after the method is executed. Any *before* methods are executed before the *primary* method and any *after* methods are executed after the *primary* method. *Before* and *after* method are defined just as the *primary* methods are and can even be written to be executed when an instance variable in changed or checked. In addition *whoppers* can be defined. A *whopper* is a type of method that wraps around all the *before*, *after*, and *primary* methods. Other method combination that are useful are the *or* and *and* method combinations. In each of these all the primary methods are executed inside an *and* or an *or* clause. A good understanding of how to control

inheritance and how to use method combinations will give the object oriented programmer much more flexibility in writing programs.

Other useful features found in most object oriented systems include the ability to initialize an objects instance variables and the automatic inclusion of some default methods in a flavors. The ability to initialize an object's instance variables before th object is sent an initialization message allows “after-init” methods, which are called after an object is created to use the correct values for newly created instances of a flavor. In Flavors4C as in LISP flavors a vanilla-flavor exists and is added to each flavor as it is created. The vanilla flavor has methods that perform elementary operations for every object, such as printing out the objects name.

As in most new programming environments, the best way to learn about object oriented coding is to use it. There is an example of how to use Flavors4C in included with the Flavors4C library and include file which should be helpful in learning Flavors4C.

Object Oriented Techniques with C

In a language that is not build solely on an object oriented paradigm, such as LISP, C, and most other languages except Smalltalk, a mix of object oriented code and procedural code is used in software development. In these hybrid systems the top level design is well suited to the object oriented paradigm while the lower level routines are usually written procedurally for efficiency.

This mix of using object oriented design at the higher levels and procedural calls at the lowest level simplifies complex problems by breaking them into more easily handled smaller problems. By partitioning the software in this way the code remains fast and if properly planned many modules can be reused for other projects. Code is reusable due to the modular structure of the object oriented design which utilizes message passing and internal data structures. Since C is used for the lower level functions that are called frequently the code can run very efficiently. In many large systems the use of an object oriented design may even make the system faster than one that uses only conventional software design due to better organization in the object oriented design.

Many programmers have been using some of the more basic fundamentals of object oriented programming in C for some time by using C structures and libraries created for those structures. To utilize some of the more advanced concepts of object oriented programming C must be extended. The most commonly accepted way of extending C is through libraries and header files. Programmers can then draw upon the libraries as needed to add certain selected functionality to their code. The advantages of extending C with libraries instead of some other methods is that libraries can be built with error checking, they follow the general syntax of C, code can be debugged with familiar debugging tools, and existing compilers and debuggers can be used. The ideal way to add an object oriented extension to C is to create a library that has the desired object oriented features. In addition to the standard object oriented features of

- classes,
- instances of classes,

- messages,
- inheritance,
- encapsulation (instance variables),
- polymorphism (different actions for the same message),
- communality,

the features that should be added to C so that it can be used for advanced object oriented programming include:

- class variables,
- dynamic class extension (add new classes at run time),
- late variable binding,
- multiple inheritance,
- detailed control of class inheritance (order and inclusion),
- options for control of message passing or function call combinations,
- abstract typing,
- base class methods.

Class variables allow a class of objects to all have the same variable. When the variable is changed by one instance of a class, the variable will be changed for all the instances of the class since the variable is the same (same address in memory) for each instance of the class.

The ability to extend the object classes at run time has some real advantages when working with systems that grow during program execution. Systems that require a high degree of user interactivity or learning systems are prime candidates for utilizing this capability.

Late variable binding allows variables to be used by several abstract classes that are inherited by some other class. If the variable binding occurs after the class is formed then the problem of dualism in variables can be avoided and classes become truly reusable. Dualism occurs when two inherited classes both have the same variable and the variable appears twice in the new class instead of just once.

In order to be able to build new classes with a variety of behavior it must be possible to inherited from many classes. Classes should be able to have an inheritance tree instead of a single inherited line.

Once multiple inheritance is allowed, the ability to control inheritance becomes much more important. It is useful to be able to specify which classes, methods, or variables must be included in a along with an abstract class once that class is included in the inheritance tree of a class which is used to create class instances. In addition the programmer should be given control over the order in which inherited flavors are combined in a class so that the behavior of the class can be tuned.

More options for message passing or function call combinations are necessary for more intricate behavior in object classes. A useful feature of LISP flavors is that by sending one message to an object many methods can be executed. Methods for the component flavors of an object are called in an order or combination specified by the programmer. Proper use of these techniques can simplify the creation of objects with very intricate and complex behavior by partitioning the class definitions into smaller, more workable subdivisions

and then controlling the way in which the methods are executed when a message is received by an instance of a class.

Methods are more useful if they are abstract or not type specific to only one class of object. Methods must be able to be used for a class or any of the classes that inherit from the class.

By including several default methods for each class, the programmers is relieved of having to duplicate some basic features that each object may need. These features include the ability to print the objects name, methods supported, or inherited classes. In LISP Flavors and in Flavors4C these methods are part of the "vanilla-flavor".

By taking advantage of the enhancements afforded to C by an object oriented package programmers as well as software designers should be able to explore many new possibilities in software development.

Why Use Flavors4C?

Using object oriented techniques in software design and development has many advantages over strictly conventional programming. A hybrid environment that allows procedural and object oriented techniques has all the advantages of both types of programming environments. The most appropriate technique needed for any given problem can be used for the greatest flexibility in code development. Flavors4C in conjunction with a good C compiler is an ideal development combination for many applications.

Flavors4C has some advantages over both LISP flavors and Smalltalk. Although LISP flavors and Smalltalk have the advantage of being interpreted systems and they may be slightly easier to use, they have disadvantages that make Flavors4C better for many applications. Smalltalk is strictly object oriented with no procedural capability. This make Smalltalk very slow. Both LISP and Smalltalk are slow in comparison to C and Flavors4C. Flavors4C is also much more portable than either LISP or Smalltalk since it is written in C and C is probably the most portable language in use today. Applications written in C can be put on almost any machine and hence Flavors4C is portable to those machines as well. Any project that must be developed in C is an ideal candidate for object oriented coding in C. Flavors4C is a complete object oriented package that is truly C and can be used in any C development environment.

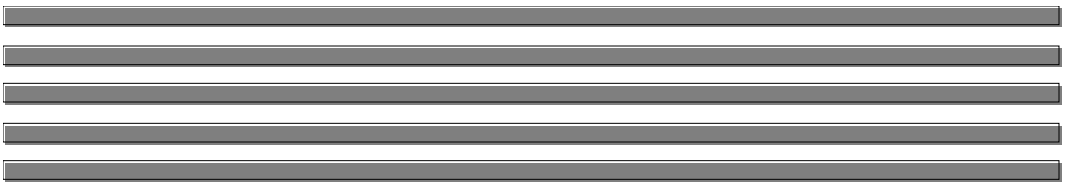
Flavors4C has numerous advantages over other object oriented packages for C. Flavors4C is easy to debug, it has a multitude of object oriented features, and it is simply a library written in C. These attributes make Flavors4C a highly competitive object oriented package for software development using C.

Flavors4C has been designed to make debugging code easier. Unlike some other packages that use a preprocessor or a custom compiler Flavors4C is a library and uses the standard C compiler or interpreter so the debugging tools that are normally used to debug C code work as expected. Since code is not rewritten or preprocessed, errors can still be traced and references to errors are not as cryptic. Since C is not known for being easy to debug some new debugging features have been added to Flavors4C. An error checking option will detect most errors that involve Flavors4C and a meaningful message will be displayed. This option can be turned off at compile time so that delivery systems will run faster. The programmer also has the option of replacing the

standard error handler with a customized error handler created by the programmer. To make inspection of objects and classes easier the vanilla-flavor has many methods that allow the programmer to inspect objects and flavors to verify that they have been created correctly.

The advantages of Flavors4C over other C object oriented package is not limited to the debugging capability of Flavors4C. Flavors4C has many object oriented features not found in other object oriented packages for C. Unlike other object oriented packages for C, Flavors4C has a wealth of attributes previously found only in LISP flavors or Smalltalk. Flavors4C has a complete inheritance system that supports inherited methods and variables. Multiple inheritance is supported and variables are not duplicated if they exist in multiple inherited flavors. Several method combinations are supported including the *or*, *and*, and *daemon* combination. *Before*, *after* , and *whopper* daemons are supported for the *daemon* combinations in Flavors4C. Variables are determined at run time, which may be slightly slower than at compile time, but the increased flexibility afforded by run time determination easily offsets the small performance degradation. A full set of vanilla-flavor methods are included and there are several options available for defining a flavor's inheritance structure. A brief look through the reference section should make clear the extent of Flavors4C's object oriented features, all of which are C functions, external variables, or simple macros.

Flavors4C is all C. It is not a preprocessor or a new language and does not suffer from the drawbacks of using a preprocessor or the drawbacks of having to learn a new language. Since Flavors4C is a library it is as portable as any other library written in C. It can be used in any C environment including some of the new C interpreters. It works well with compiler / editor combinations and is easy to use. Flavors4C is used by including the Flavors4C header file in files that use Flavors4C and by adding the Flavors4C library to the make file. No extra steps such as a preprocessor are required. Any programmer familiar with C can use Flavors4C immediately. Flavors4C is an extensive object oriented package that should be more than adequate for even the most advanced object oriented programmer.



3. Features of Flavors4C

Flavors4C was designed to be a versatile, fully featured object oriented system modeled after the MIT flavors system found on LISP machines. Flavors4C was developed to be as portable as possible by adhering to the C standard as outlined by Kernighan and Ritchie in “The C Programming Language” as much as possible and by using a general C type syntax.

Flavors4C has almost all the features that even the most experience flavor hacker could want. Multiple inheritance is available and that inheritance can be controlled by the use of required and included flavors. Several method combinations are supported in addition to the default daemon combination which includes before and after daemons as well as whoppers. A vanilla flavor exists along with many methods for the base flavor and the programmer has access to a NULL terminated array of flavor pointers for all the flavors. Some additions have been made to LISP flavors in that class variables are supported and there is a broadcast function that sends the same message to every instance of a specified flavor. For more detailed information on each function see the reference section. To get a better understanding of flavors refer to the manuals from a LISP machine or “LISP LORE” by Hank Bromley, which is an excellent book which touches on the subject.

Although Flavors4C has many similarities to LISP flavors it is not exactly the same. After all Flavors4C is for C not LISP. Conceptually Flavors4C follows the LISP flavors model fairly closely and any experience LISP flavors programmer should be able to utilize Flavors4C quickly. The similarity between LISP flavors and Flavors4C should make code translation from LISP to C much easier.

Flavors4C is written to follow the C standard as described by Kernighan and Ritchie in “The C Programming Language” so that it can be compiled on almost any C compiler. It can be made available for almost any machine and compiler as long as that combination supports standard C. Since Flavors4C is a library of functions code is transportable to any machine and compiler that has the Flavors4C library on it.

To shorten development time and still maintain fast delivery systems many of the debugging features have been included in Flavors4C can be turned off by simply changing the value of one external variable. The error detection features in Flavors4C will detect most of the errors that could occur while generating Flavors4C code. The function that reports errors can also be replaced by a user supplied function for custom error reporting.

Whenever possible Flavors4C has been designed to make programming easier and object oriented coding as flexible as possible. An example of the flexibility

of Flavors4C is the two ways to reference a class or flavor by most functions. Classes or flavors can be referenced using either strings or pointers to reference classes or flavors during program execution. Programming is eased by adding many informative methods that can be called for the vanilla-flavor so that information can be obtained about each object that is created. As Flavors4C is updated more features will be added in response to feedback from the users of Flavors4C so that it remains a flexible and fully featured object oriented packages for C.

In order to make Flavors4C as useful as possible nearly all of the features found in LISP flavors also exist in Flavors4C. Many of the functions or macros have the same names as those used in LISP flavors, although the arguments may be different. If it is still difficult to understand how some of the functions in Flavors4C are used after ready the function descriptions and looking at the example program, then further reading on LISP flavors should help to clarify any questions.

Flavors4C Variables and Types

typedef char*Pntr;

A Pntr is a generic pointer (really a char pointer) which can be used when ever a general pointer is needed. The pointer must always be cast before it is used otherwise the compiler will expect a char type.

extern Flavors *allFlavors;

The **allFlavors** variable is a NULL terminated array of all the defined flavors. As new flavors are added the actual array pointed to by this variable may change.

extern int (*WarningFunction)();

The **WarningFunction** variable is a pointer to the function used when an error occurs. The function should take one string as an argument. The string usually contains information that will help identify the error that has occurred. The returned value of the function is not important. If **WarningFunction** is NULL then the default error function will be used. The default error function will print out the error message to standard output and then terminate the program.

---

extern int error_check;

The **error_check** variable is normally set to 1 and error checking occurs. If **error_check** is set to NULL then some error checking is disabled and some function will work faster. It is wise to leave **error_check** set to 1 while developing code and only set it to NULL when the code works properly.

#define STR_SPECIFY ...
#define Pntr_SPECIFY ...

extern int flavor_specifier;

The **flavor_specifier** variable is used to determine how flavor specifiers are read. If **flavor_specifier** is set to the default value of NULL then functions such as **DefFlavor()** will first look for a flavor pointer that matches the argument that represents a flavor. If there is no match for the specified flavor then the arguments is assumed to be a string and all the flavor names are checked for a match. If still no match is found an error occurs. If **flavor_specifier** is set to STR_SPECIFY then only the flavor names are searched. If **flavor_specifier** is set to Pntr_SPECIFY then only the pointers to the flavors are searched.

#define DAEMON ...
#define OR_COMB ...
#define AND_COMB ...
#define DAEMON_OR ...
#define DAEMON_AND ...
#define PROGN ...

These defines are used in the **MethodCombination()** function to determine how methods will be executed.

#define BASE_FLAVOR_LAST ...
#define BASE_FLAVOR_FIRST ...

These defines are used in the **MethodCombination()** function to determine the order that methods are executed in.

#define PRIMARY ...
#define BEFORE ...
#define AFTER ...
#define AROUND WHOPPER
#define WHOPPER ...
#define OR_COMB /* already defined */
#define AND_COMB /* already defined */

These defines are used in the **DefMethod()** function to specify the type of method being defined.

```
#define SETTABLE        ...
#define GETTABLE        ...
#define ALL             ...
```

These defines are used in the **AddInstanceVar()** and **AddClassVar()** functions to determine which the variable access methods will be created for the variable.

```
#define INIT            NULL
#define NO_INIT  ...
```

These defines are used in the **MakeInstance()** function to specify if the 'init' method will be called when the instance is created or if the used will call the 'init' method explicitly.

Flavors4C Functions

```
Flavor*
DefFlavor(name, parents)
        char   *name;
        Pntr   parents;
```

The function **DefFlavor()** returns a pointer to a a new flavor and is used to create new classes or flavors. The arguments are the *name* of the new flavor which is a string and the *parents* which is a Pntr. The *parents* argument can be either a NULL terminated array of pointers to the flavors that the new flavor will inherit from or a string composed of the names of the inherited flavors, separated by white space (space, newline, tab, ...). The variable **flavor_specifier** determines how the parents argument will be checked. If **flavor_specifier** is set to STR_SPECIFY then only strings will be checked for the *parents* argument. If **flavor_specifier** is set to Pntr_SPECIFY then a NULL terminated array of flavor pointers is expected. If flavor_specifier is set to (STR_SPECIFY | Pntr_SPECIFY) then either a string or a NULL terminated array will work.

> Note: The flexibility of using a string or a NULL terminated list does have a price associated with it. If the **flavor_specifier** external variable is set to (STR_SPECIFY | Pntr_SPECIFY) and strings are used to identify the flavors in the *parents* argument then there is a very remote chance that one of the first flavor name will be the same as the address of one of the other flavors. If this happens an error may occur.

Unlike LISP where many arguments are passed to the **DefFlavor()** function to create a new flavor and to set it's attributes, the **DefFlavor()** in Flavors4C only creates the basic flavor. After the basic flavor has been created, variables and methods can be added to it. Other attributes such as included flavors and required flavors can also be added after the basic flavor has been created. To close the new flavor so that it can be used the **CompileFlavor()** function should be called.

---

void
AbstractFlavor(flavor)
Pntr flavor;

AbstractFlavor() is used to assure that no instances of that flavor can be created. If an attempt is made to create an instance of the flavor an error will occur. This function is useful if the flavor is intended to only be used with other flavors. Abstract flavors are generally referred to as mixins since they are mixed in with other flavors. The single argument, flavor, can be either a string or a pointer to a flavor. The type is controlled by the external variable **flavor_specifier** just as in the **DefFlavor()** function definition.

---

void
RequiredVars(flavor, varNames)
Pntr flavor;
char *varNames;

The **RequiredVars()** function is used only on abstract flavors. The flavor will automatically become an abstract flavor if this function is called. The function **AbstractFlavor()** does not have to be called. Once a variable has been made a required variable that variable must be a part of any flavor that contains the specified flavor with the required variables. If the variable does not exist an error will occur when the function **CompileFlavor()** is called for the flavor. **RequiredVars()** is useful when creating flavor mixins that have methods that expect to find variable not contained in the mixin flavor. The first argument, *flavor*, to **RequiredVars()** is a pointer to a flavor that will have the required variables or a pointer to the flavor's name. The variable **flavor_specifier** is used to specify the type used for the *flavor* argument. The second argument is a string with the variable names separated by white space.

---

void
RequiredMethods(flavor, methods)
Pntr flavor;
char *methods;

The **RequiredMethods()** function is identical to the **RequiredVars()** function except the string of names should be the names of methods not variables. The flavor will be an abstract flavor and no instance can be created directly from the flavor.

---

void
RequiredFlavors(flavor, required)
Pntr flavor;
Pntr required;

The **RequiredFlavors()** function is similar to the **RequiredVars()** function in that the flavor specified by the *flavor* argument becomes an abstract flavor. An error will occur if an attempt is made to make an instance of the

specified flavor or any flavor that includes the specified flavor if the required flavors are not included as well. The first argument to **RequiredFlavors()** is a pointer to a flavor or the flavor name. The second argument is either a NULL terminated array of flavor pointers or a string composed of the required flavor names separated by white space. The variable **flavor_specifier** is used to determine if flavor pointers, a string pointer, or either are used for both the *flavor* and *required* arguments.

---

void
IncludedFlavors(flavor, included)
    Pntr flavor;
    Pntr included;

The **IncludedFlavors()** function assures that all the included flavors are in any flavor that uses the flavor specified by the *flavor* argument. If a flavor is formed that has the specified flavor as a mixin and all the included flavors are not in the new flavor then the included flavors is added to the new flavors inherited flavors list immediately after the specified flavor. The first argument to **IncludedFlavors()** is a pointer to a flavor or the flavor name. The second argument is either a NULL terminated array of flavor pointers or a string composed of the required flavor names separated by white space. The variable **flavor_specifier** is used to determine if flavor pointers, a string pointer, or either are used for both the *flavor* and *included* arguments.

---

void
NoVanillaFlavor(flavor)
    Pntr flavor;

The **NoVanillaFlavor()** function removes the vanilla-flavor from the inheritance list of the flavor specified by the *flavor* argument. Normally the vanilla flavor is included in all flavors and is put at the end of the inherited flavors list. The vanilla flavor has many useful methods that are useful in debugging. It is not very common to use this function. The *flavor* argument to **NoVanillaFlavor()** is a pointer to a flavor or the flavor name. The variable **flavor_specifier** is used to determine if flavor pointers, a string pointer, or either are used for the *flavor* argument. The methods supplied by the vanilla flavor are “init”, “printf-self”, “sprintf-self”, “describe”, “which-operations”, “kill”, “has-variable”, “has-method”, “has-flavor”, and “instance-of”.

The “init” method is used to initialize the object when it is created. It if useful for keying AFTER “init” methods which perform some operations on the new object immediately after it is created.

The “printf-self” simply prints the internal object name to the standard output device. The value returned by the method is a pointer to the a string which is the name of the object. The same character array is used every time the method is called so the name should be copied before it is used.

The “sprintf-self” prints the internal object name to the string pointed to by the pointer given as the only argument to the method. The value returned by the method is a pointer to the new string.

The “describe” method prints out a description of the object to the standard output device.

The “which-operations” method returns a pointer to a string which is a concatenation of all the method names that the object can respond to.

The “kill” method is used to remove an object from memory and then clean up to remove all Flavors4C links to the object.

The “has-variable”, “has-method”, and “has-flavor” methods return a pointer to the object that was called if the variable, method, or flavor is part of the object. If the variable, method, or flavor is not contained in the object, NULL is returned by the method. The flavor argument to the “has-flavor” method can be either a pointer to a flavor or a string which is the name of a flavor.

The “instance-of” method will return a pointer to the flavor that the object is an instance of.

```
void
MethodCombination(flavor, type, order, methods)
        Pntr   flavor;
        int            type, order;
        char   *methods;
```

The **MethodCombination()** function is used to set the method combination for one or more methods. The method combination for each flavor and method determines how the methods will be executed when an instance of the flavor is sent a message. The first argument, flavor, is a pointer to the flavor or flavor name. The second and third arguments are *type* and *order*, both are integers. The default *type* is the DAEMON method combination. The default *order* is BASE_FLAVOR_LAST. The *order* argument determines the order in which the methods for the inherited flavors are executed. The only other value for the *order* argument is BASE_FLAVOR_FIRST. The last argument to **MethodCombination()** is a string which should be all the method names separated by white space. The possible method combinations in Flavors4C are:

DAEMON

The DAEMON combination for methods is best described by a diagram.(see Figure 1) When a message is sent to an object all the whoppers for the parent are executed first in the order specified. Next the all the before methods for the parent flavor are executed in reverse order. The first primary method that is found for a flavor on the parents list of inherited flavors is executed next and then all the after methods are executed in normal order. The value returned by the Send is the value returned by the primary method unless no primary method exists and then NULL is returned.

OR_COMB

The OR_COMB combination is the same as the LISP flavors OR method combination. All the OR_COMB methods from the parents inherited flavors are executed in order until a value other than NULL is returned. If no method evaluates to a non NULL value then NULL is returned otherwise the value of the last executed method is returned. No BEFORE methods, AFTER methods, or WHOPPERS are executed.

AND_COMB

The AND_COMB combination is the same as the LISP flavors AND method combination. All the AND_COMB methods from the parents inherited flavors are executed in order until a NULL value is returned. If all the methods evaluate to a non NULL value then the last value is returned otherwise NULL is returned. No BEFORE methods, AFTER methods, or WHOPPERS are executed.

DAEMON_OR

The DAEMON_OR combination is the same as the LISP flavors DAEMON-WITH-OR method combination. All the OR_COMB methods from the parents inherited flavors are executed in order until a value other than NULL is returned. If no method evaluates to a non NULL value then the value of the primary method is returned otherwise the value of the last executed method is returned. All the BEFORE methods, AFTER methods, or WHOPPERS are executed in the same order as they would for the DAEMON combination method.

DAEMON_AND

The DAEMON_AND combination is the same as the LISP flavors DAEMON-WITH-AND method combination. All the AND_COMB methods from the parents inherited flavors are executed in order until a NULL value is returned. If all the methods evaluate to a non NULL value then the value of the primary method is returned otherwise NULL is returned. All the BEFORE methods, AFTER methods, or WHOPPERS are executed in the same order as they would for the DAEMON combination method.

PROGN

The PROGN combination is the same as the LISP flavors PROGN method combination. All the methods from the parents inherited flavors are executed in order and the value of the last method is returned. No BEFORE methods, AFTER methods, or WHOPPERS are executed.

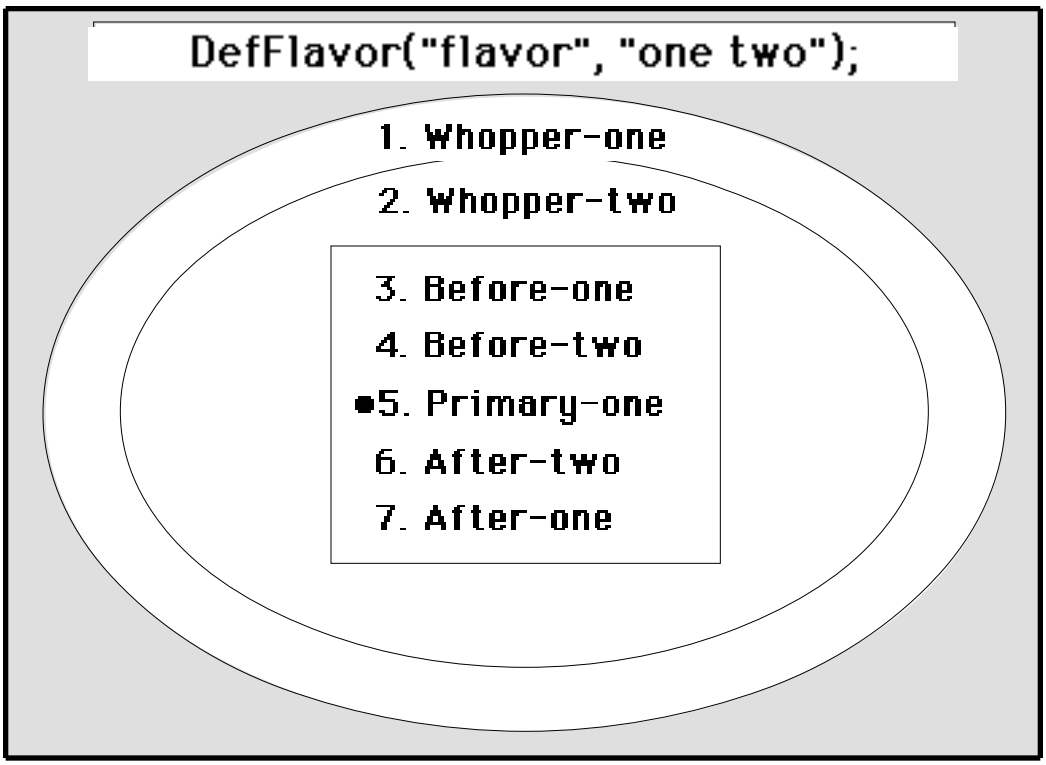


Figure 1. Method execution order

---

void
DefaultHandler(flavor, function)
Pntr flavor;
Pntr (*function)();

The **DefaultHandler()** function will replace, for the flavor specified by the *flavor* argument, the default error function that is executed if a method is called that does not exist. The function pointed to by the *function* argument will receive the same arguments as the method that was called, including the first 'self' argument.

---

void
DefMethod(flavor, type, name, function)
Pntr flavor;
char type, *name;
Pntr (*function)();

The **DefMethod()** function is used to attach a function to a flavor as a method. The first argument, *flavor*, is a pointer to a flavor or a flavor name. The second argument, *type*, is the type of method. The type can be either PRIMARY, BEFORE, AFTER, (AROUND or WHOPPER), OR_COMB, or AND_COMB and is used by the method combination for that method and flavor. The *name* is a string which is the name of the method. The last argument is a pointer to the function for the method.

The functions that can be used for methods have some restrictions. All method functions must return a pointer (Pntr). For all of the method type except the WHOPPER or AROUND methods the function's first argument must be a pointer (Pntr) which will be set to the object being sent the message. Usually this argument is called 'self'. The first argument of a WHOPPER's function must be a Pntr that is usually called 'methods' and the second argument must be a Pntr usually called 'self'. The 'methods' argument is only used when the **ContinueWhopper()** function is called. The 'self' argument is refers to the object sent the message.

Pntr
AddInstanceVar(flavor, name, flags, type, initValue)
Pntr flavor;
char *name;
int flags;
.... type, initValue;

The **AddInstanceVar()** is really a macro which is used to add an instance variable to a flavor. **AddInstanceVar()** calls the function **AddIVar()**. The **AddIVar()** function uses the sizeof the *type* argument and returns a pointer which is in turn used to set the initial value of the variable being added to the flavor. The initial value and type can be any any type supported by the compiler for direct assignments. Any variable such as a structure, double, int, etc. that can be assigned a value using the '=' operator can be given an initial value. If the variable already exists for the flavor, possibly from some inherited flavor, only the initial value of the variable for the specified flavor is changed. The parent flavor's initial value does not change. The first arguments can be either a pointer to a flavor or a flavor name. The second argument is the name of the variable, the third is a flag or a logical or { | } of more than one flag. The possible flag values are SETTABLE, GETTABLE, or ALL. The flags set up the "set-*varName*" and "pset-*varName*" methods and the "*varName*" methods which are used to set the value of a variable or to get the pointer to a variable. These methods can not be modified or changed.

The "set-*varName*" and the "pset-*varName*" methods are generated if the flag value included the SETTABLE value. (Note: The "set-*varName*" method may not work for complex variables on some machines and with some compilers.) These methods are used to set the value of the variable, *varName*, for the object. An example of each is shown below:

```
/*      In each case the value of the instance variable "var" for the object is changed to the value of i.          */

Instance      *object;
int           i = 33;

Send(object, "set-var", i);
Send(object, "pset-var", &i);
```

The "*varName*" method will return a pointer to the instance variable, varName. The actual value of the variable can be derived by casting the pointer returned to the correct type for the variable, *varName*.

Pntr
AddClassVar(flavor, name, flags, type, initValue)
Pntr flavor;
char *name;
int flags;
.... type, initValue;

The **AddClassVar()** macro is identical to the **AddInstanceVar()** macro except that the variable added to the flavor is a class variable. The value of the variable is the same for every instance of the flavor. If one instance of the flavor changes the class variable the variable changes for all the instances. The variable is actually stored in one place so a pointer to the variable can be used to change the value for all instances.

---

void
CompileFlavor(flavor)
Pntr flavor;

The **CompileFlavor()** function initializes the flavor specified by the *flavor* argument so that it can be used by other flavors and instances can be made of it. All methods should be attached to the flavor, all variables should be added to the flavor, and all required, included, method combination, and other setup function for the flavor should be complete before calling this function.

Pntr
ContinueWhopper(methods, object, the_rest)
Pntr methods;
Instance *object;
.... the_rest;

The **ContinueWhopper()** function requires at least two arguments and should only be called inside a whopper. The first arguments is taken directly from the first argument, usually called ‘methods’, of the whopper function. The second argument to **ContinueWhopper()** is the second argument of the whopper function or ‘self’. The rest of the arguments correspond to the arguments normally supplied to the method. The are usually the same arguments as were passed to the method or else some alteration to the original arguments. (such as 2*arg, 3+arg, etc.) **ContinueWhopper()** will return a pointer (Pntr) just as any other method does.

Flavor*
StringToFlavor(name)
char *name;

The **StringToFlavor()** function simply returns a pointer to the flavor that corresponds to the *name* which is the first argument.

```
char*
FlavorToString(flavor)
        Flavor      *flavor;
```

The **FlavorToString()** function simply returns a pointer to the name of the flavor specified by the *flavor* argument. Change the name at your own risk. The name character array is only as long as the name so don't try to add characters after the end of the name.

```
Instance*
MakeInstance(flavor)
        Pntr   flavor;
```

The **MakeInstance()** function is used to create an instance of a flavor. The first argument is the flavor which can be either a pointer to a flavor or the name of a flavor. The function returns a pointer (Pntr) to the newly created Instance which is of the flavor specified.

```
Pntr
Send(object, message, the_rest)
        Instance    *object;
        char        *message;
        ....                the_rest;
```

The **Send()** function is used more than any other in flavors and in Flavors4C. Send is used to send messages to objects or Instances. The object executes the functions which are attached to it as methods and whoppers. The order in which these methods are executed and the value returned is determined by the method combination for the method and the flavor of the object. The first argument to send is a pointer (Pntr) to an Instance. The second argument is a string which is the name of the method being called. The rest of the arguments can be of any type. The only restriction on the *the_rest* arguments is that the space they take up is less than or equal to 10 times the size of a double for the compiler being used. It is unlikely that problems will be encountered in this area unless an exceptional number of arguments are being passed to the method or complete structures are being passed instead of pointers.

```
Pntr
BroadCast(flavor, message, the_rest)
        Pntr        flavor;
        char        *message;
        ....                the_rest;
```

The **BroadCast()** function is similar to the **Send()** function. The difference between the two is that the **BroadCast()** function send the message to every

instance of the specified flavor. The flavor argument can be either a pointer to a flavor or the name of a flavor. Until sometime in the future the value returned by the function will always be NULL.

```
Pntr
GetVar(object, varName)
        Instance    *object;
        char        *varName;
```

The **GetVar()** function is usually only used inside a method but it can be used anywhere. The function returns a pointer to the instance or class variable that is represented by the *varName*. The value of the variable can the be used or edited by the program. When **GetVar()** is called no before or after daemons are called. The pointer returned should always be cast before it is used to get the proper results.

5. Hints for Using Flavors4C

Flavors4C will work if used as described in the reference section. It can however be used more efficiently if certain guidelines are followed. There are some guidelines that will improve the performance of Flavors4C code and other suggestions are just useful bit of knowledge.

To improve the speed of variable access in methods using the function **GetVar()** be used only once. When the function **GetVar()** is used no before or after daemons will be called. It is usually only used in methods for a access to instance or class variables. Since it takes a finite amount of time to find the variable address that is returned by the **GetVar()** function the variable address should be saved as a variable in the function (not a static variable since different objects will use the same method). Once the address is known the value stored in the address can be changed or used as desired without having to call the **GetVar()** function.

Some performance increase may be realized by using shorter variable and method names that do not begin with the same sequence of letters. It is better to use names that begin differently than ones that begin the same. As an example it is better to use x-pos and y-pos than position-x and position-y as variable names.

While code is being developed the **error_check** variable should be set to 1 which is the default value. Since error checking does take time it can be turned off after code development is complete. Some increase in speed will be realized when the **error_check** variable is set to 0.

Code modules will be more reusable if they are broken into the smallest components. There is no penalty for using many inherited flavors instead of only a few inherited flavors when creating a new flavor except for a slight increase in the time required to define the classes.

Index

(*WarningFunction)()....4-1
AbstractFlavor(flavor)....4-3
AddClassVar(flavor, name, flags, type, initValue)....4-9
AddInstanceVar(flavor, name, flags, type, initValue)....4-8
    ALL....4-2
    GETTABLE....4-2
    SETTABLE....4-2
allFlavors....4-1
BroadCast(flavor, message, the_rest)....4-11
CompileFlavor(flavor)....4-9
ContinueWhopper(methods, object, the_rest)....4-9
DefaultHandler(flavor, function)....4-7
DefFlavor(name, parents)....4-3
DefMethod(flavor, type, name, function)....4-7
    AFTER....4-2
    AND_COMB....4-2
    BEFORE....4-2
    OR_COMB....4-2
    PRIMARY....4-2
    WHOPPER or AROUND....4-2
error_check....4-2
FlavorToString(flavor)....4-10
flavor_specifier....4-2
    DAEMON....4-2
    DAEMON_AND....4-2
GetVar(object, varName)....4-11
IncludedFlavors(flavor, included)....4-5
MakeInstance(flavor)....4-10
MethodCombination(flavor, type, order, methods)....4-6
    AND_COMB....4-2, 4-6
    DAEMON....4-2, 4-6
    DAEMON_AND....4-2, 4-7
    DAEMON_OR....4-2, 4-6
    OR_COMB....4-2, 4-6
    PROGN....4-2, 4-7
NoVanillaFlavor(flavor)....4-5
Pntr....4-1
RequiredFlavors(flavor, required)....4-4
RequiredMethods(flavor, methods)....4-4
RequiredVars(flavor, varNames)....4-4
Send(object, message, the_rest)....4-10
StringToFlavor(name)....4-10

Order Form
for Flavors4C

Name_____

Company_____

Address_____

City_____State_____Zip_____

Phone Home:_____Work:_____

Where did you here about Flavors4C_____

_____

To receive the most recent version Flavors4C (without the initial evaluation notice and time limitation) and printed documentation, please send this order form and a check for $59.00 to:

Pete Ohler
460 Monti Court
Pleasant Hill, CA 94523

(415) 686-1317

Please write and call for more information or comments.