MeldC Runtime/Kernel Implementation Guide

James Lee
Wenwey Hseush
Scott Meyer

Technical Report
CUCS-44-92

Columbia University
Department of Computer Science
New York, NY 10027

January 18, 1993

The Programming Systems Laboratory is supported by National Science Foundation grants CCR-9106368 and CCR-8858029, by grants and fellowships from AT&T, BNR, Bull, DEC, IBM and SRA, by the New York State Center for Advanced Technology in Computers and Information Systems and by the NSF Engineering Research Center for Telecommunications Research.

Preface

Before we start, we would like to give, the user, a general overview of this manual. First let us start off by stating that this manual is not intended for a general user who plans on using MeldC. This manual is intended for the user who plans on changing MeldC to their liking. This manual is designed in such a way to do so for such a user. It presents an in depth look at MeldC to allow for users to use this manual as a reference guide when making any detailed changes. If you are a general user, put this manual down and do not even bother reading it, it will not help you use MeldC.

The manual has been broken down into logical sections so that a user can go to a main topic and then find the catagory within that topic. The main pieces are all put into a separate chapter. The subtopics have been put into sections within that chapter. Let us give you a general layout of what each chapter contains and how it may apply to you.

In Chapter 2, we cover the material having to do with the Kernel. Here is where we discuss the material used to allow MeldC to run on a UNIX platform. This material includes:

- Starting of MeldC.
- Allocating memory, which includes memory for objects in MeldC.
- Handling signals.
- Scheduling jobs to run and be suspended.
- Linearizing messages so that they may be sent across a network to a remote source.
- Delayuntil/responses.
- Creating new threads.
- Enqueuing and dequeuing of threads from the ready queue.
- Handling stack switches.

Let us now give you an idea of where to start to look for certain material of the Kernel:

- If you want to see the scheduler for MeldC, driver, you should refer to Section 2.6. Here we describe the functions used to lock and unlock atomic locks and we also outline the main scheduler function. You would want to refer to this section if you wanted to change the scheduling mechanism used by MeldC.

- If you want to see how frames are allocated by MeldC, you should refer to Section 2.7. This section will help you if you want to allocate memory in a different way. This section also outlines the different stack methods used by MeldC. If you wanted to implement a different stack method from the ones provided, you would need to refer to this section to get an idea of how to do so. Also see Section 2.8 for changing how MeldC allocates memory from a low level.

- In Section 2.9 we outline how we linearize data to be sent across the network to some external source. See this section if you wish to modify how this is done.

- In Section 2.10 we outline how delayuntil/response calls are handled in MeldC. See this section for modifying the existing method for doing so.

- In Section 2.11 we outline how frames are enqueued and dequeued from the ready queue. To change the means of scheduling jobs, it may be necessary to modify this section as well.

- In Section 2.13 we cover the material dealing with doing stack switches. This section may be of interest if you decide to change the means of executing new threads in MeldC.

- In Section 2.15, we discuss the assembly code used by MeldC to implement the RUN, RESUME and YIELD functionality. This code is system dependent and must be modified for systems other than a **SUN4** or **MIPS** platform. Refer to these sections to see how this code was implemented.

In Chapter 3 we discuss the data structures used by MeldC. In this chapter you may want to refer to the given sections for the following reasons:

- If you want to change the means of allocating the memory for the structures used for MeldC frames, you would want to refer to Section 3.3. In this section is a detailed look at how the frames are allocated their memory. Note that the frame structure allocation defers for each stack method, so if you add of alter a stack method, modifications may also have to take place here.

- In Section 3.6, we cover the macros used to handle delayuntil/respond messages. If the method for handling delayuntil/respond messages is changed, it may be necessary to change these macros as well.
- In Section 3.8, we cover the macros used to to and unlock atomic locks. If this mechanism is changed, these macros may have to be changed as well.
- In Section 3.10, we discuss the structure used for holding the shadow object information. If modifications are performed on the implementation of shadow objects, then this structure may also require modifications to reflect those of the shadow object implementation.
- In Section 3.11, we discuss the structure used for objects. If any changes are done to how objects are accessed, locked, kept track of and so forth, this structure may needed to be changed as well.
- In Section 3.12 we discuss the structure used to hold a parameter for a thread. If more information is needed about how to keep track of parameter information for threads, this structure may need to be changed.
- In Section 3.13 we discuss the structure used to for keeping track of all of the methods of a class. Any changes to how methods are accessed may require this structure to change as well.
- If you want to add or change the structures used for MELDC frames, you would want to refer to Section 3.14. In this section is a detailed look at the components used to make up the frame structure.
- In Section 3.18 we cover the structure used by MELDC for the message passing, _msg. An changes to either binding or the passing of optional parameters may involve making changes in this structure as well.

In Chapter 4, we cover the MetaClass of MELDC. This chapter covers the MetaClass definition and it's creation. It also covers the creation of new objects and classes, as well as how they are destroyed. The chapter also discusses how shadow objects are created and attached, as well as detaching them. It also discusses how shadow object methods are executed and how a kernel trap is executed (how a message is handled by a shadow object). The chapter also contians the code needed to implement a GetObj or PutObj call, make high level function call to get an external object or give someone a local object to an external source. And finally, the chapter tells how binding is handled by MELDC. This chapter would be of use if you want to make changes to the following areas:

- If you plan on changing the actual definition of the MetaClass for MeldC, you should refer to Section 4.1. Here we discuss the class definition of the MetaClass, as well as the initialization of the MetaClass.

- If you want to change how an object is initialized after being first created in MeldC, you will have refer to Section 4.1.6. This is where we will initialize the instance variables of the new object.

- In Section 4.1.7, we cover how an object is allocated memory by MeldC when it needs to be first created. In Section 4.1.8 we discuss how the parent objects of the newly created object also have their memory allocated. These sections may need changes if you want to change how they are allocated memory.

- In Section 4.1.9, we discuss how a new class or object starts the creation process in the Create and CreateObj functions. There is also the CreateSubObj, Section 4.1.10, function used to create the inheritence hierarchy of an object. These sections will be of use to you if you decide to change how you would like object to be created.

- In Section 4.1.11, we discuss how a shadow object is created and than attached to the parent object. Then in Section 4.1.12, we discuss how a shadow object is attached to some existing object. These sections may be of use if you want to modify the attachment of shadow objects. Refer to Section 4.1.14 as well.

- In Section 4.1.13, we discuss how a shadow object is removed from another object. This section could be of use if you modify the attachment of a shadow, then this may need modifications as well. Refer to Section 4.1.14 as well.

- In Section 4.1.15, we discuss how shadow object methods are executed in MeldC. If you change the methods to have shadows execute, this may need to be changed to reflect those new methods.

- In Section 4.1.16, we discuss the steps needed to handle a message received by a shadow object. If you choose to implement the kernel trap in a different way, you will need to refer to the material in this section.

- In Section 4.1.17, we discuss how an object is destroyed. If the creation process is modified, than this section must also reflect any of these changes to properly remov the object.

- In Section 4.1.27, we discuss the two high level protocol calls. These are of importance if you want to change how external objects are accessed.

- In Section 4.2.7, we discuss the steps used to do the binding in MeldC. This section is of importance if you want to modify this task.

The final four chapters, Chapter 5 – Chapter 8, all deal with the topic of Shadow Objects and their use in MeldC. Chapter 5 we discuss them as a general topic. Then we give an introduction to the runtime code in Chapter 6. Then in Chapter 7, we discuss the implementation of the use of Persistent Objects in MeldC. Within this chapter, we have Section 7.3 – Section 7.6 covering the code needed to save a Persistent Objects state to the database, DB2. This code may have to be changed if your system does not contain this database system. The rest of the sections discuss the code needed to implement Persistent Objects and can be used to modify how we currently implement them. Finally, in Chapter 8, we discuss the implementation of Remote Objects in MeldC. The topics of this chapter are:

- In Section 8.1, we discuss the steps needed to encode/decode a message. Depending on thew system, this may need to be changed.

- In Section 8.2, we discuss the code needed to open a connection on the network using UDP. Since UDP is not as realiable as other networks, this section may want to be studied carefully so that it can be reimplemented using a better network service. This section includes sending and receiving messages from a remote source.

- In Section 8.3, we discuss the simple interface to shadow objects. This includes creating a new shadow object, as well as initializing it. Also, in Section 8.4, is the decsription of the shadow object interface but for the $sender implementation.

- In Section 8.5, we discuss the functions used to implement the shadow objects. This includes the init_point method, Section 8.5.2, the term_point, Section 8.5.3. It also includes the receiving and passing of objects as a parameter, Section 8.5.5 and Section 8.5.6 respectivily. This section also contains the entry_point method used by shadow objects, Section 8.5.9. All of these sections should be referred to if you want to modify how shadow objects are implemented in MeldC.

- In Section 8.6, we have the code that is used at the lowest level to check and see if objects are avaiable or not on remote hosts. Some of the code in this section contains code that is used for UDP and may have to be modified if you change the network service used. Since this section deals with the low level end of shadow objects, changes to the network or how shadow objects are implemented may force this code to be changed as well. For Section 8.7, this also applies, low level code used for $sender implementation of shadow objects.

- In Section 8.8, we discuss the implementation of how we check to see if Remote Objects are still alive on some remote source. This code may also need changes if the network service changes.

- In Section 8.9, we discuss the high level protocol used to communicate with the remote object. This section and Section 8.11 go hand in hand. Any modifications to either one, means that the other must also reflect these changes.

Contents

Credits

Gail Kaiser (Associate Professor, Columbia University)
Project leader

Wenwey Hseush (Doctoral Candidate)
Principal designer of language kernel
Co-designer of compiler

James Lee (Graduate Research Assistant and Staff)
Project manager
Principal designer of runtime system

Scott Meyer (Masters Degree Candidate)
Technical Writer

Chapter 1

Introduction to MeldC

The MELDC project has been one of the major foci of the Programming Systems Laboratory at Columbia University since 1987. Our goal is to develop a parallel and distributed object-oriented programming language for large-scale applications. Starting in 1990, the MELDC language was completely redesigned and reimplemented from scratch to produce MELDC, which is closer to C, has fewer but more sophisticated "features", and a cleaner architecture with many of the facilities implemented in the MELDC language itself.

1.1 The MELDC Object-Oriented Programming Model

MELDC is a C-based, concurrent, object-oriented language built on a *reflective* architecture. The core of the architecture is a micro-kernel (the MELDC kernel), which encapsulates a minimum set of entities that cannot be modeled as objects. All components outside of the kernel are implemented as objects in MELDC itself and are modularized in the MELDC libraries.

A MELDC program consists of a collection of active objects which send and receive messages to and from other objects (local or remote objects). An object is a runtime entity that has its own private data and control. The state (instance variables) and the behavior of an object is defined by a second program element known as *class* in object-oriented programming. The class of an object states how the program behaves in reaction to different messages delivered to the object. A sequence of actions corresponding to a received message is encapsulated as a *method*. In C++, classes are not considered as objects at runtime. They are abstract data types and are defined statically. The creation of objects are "declared" through C++ statements and the behaviors of objects are embedded in the code generated by the compiler. In MELDC, we consider a class is an object, which is defined by another object known as *metaclass* (the class of classes) in object-oriented programming. Similar to Smalltalk, the MELDC metaclass is exposed to the user and classes are explicitly created as the instance of the MetaClass. A default metaclass (called *Metaclass*) is created statically. Objects and classes other than the Metaclass are created at runtime by sending `create()` messages to the Metaclass[1]. A MELDC program starts with the Metaclass already created and a set of `create()` messages to be delivered. There are no senders for these initial messages. A subset of the messages are first delivered to the Metaclass to create classes and then the rest of messages are delivered to these classes to create global objects.[2]

MELDC is reflective in three dimensions: structural, computational and architectural. The structural reflection indicates that classes and metaclasses are objects, which are written in MELDC. The computational reflection means that object behaviors can be computed and extended at runtime. The architectural reflection indicates that new features/properties (*e.g.,* persistency and remoteness) can be constructed in MELDC. These properties can be attached to and removed from objects at runtime. The reflective architecture provides high flexibility to customize or extend object behaviors in an elegant way. For example, a programmer builds a simple type of persistent objects that do not survive catestrophic system failures and then builds a comprehensive version of persistent objects that survive system failures by applying redundancy to the simple ones. In MELDC, persistency is not a language primitive, but just another property that can be constructed for objects. The semantics of persistency or policies to implement it are defined in MELDC.

Since micro-kernel facilities cannot be replaced or modified by the MELDC programmer, several common choices are supported by the kernel and can be designated by the programmer using compiler switches. For example, MELDC intends to support a variety of parallel and distributed applications that have different concurrency characteristics. Some applications require a small number of long-lived threads while others need a large number of short-lived threads that are created and destroyed dynam-

[1] Similar to CLOS and ObjVlisp, it is possible to have multiple metaclasses which are the instance of root MetaClass. There are numerous complex issues which had been explored by many, and they are not within the scope of our discussion.

[2] In the current implementation, the source/object code is not part of class definition. Even though the class is dynamically created, constructing a class at runtime, without prelink the source/object will not be possible.

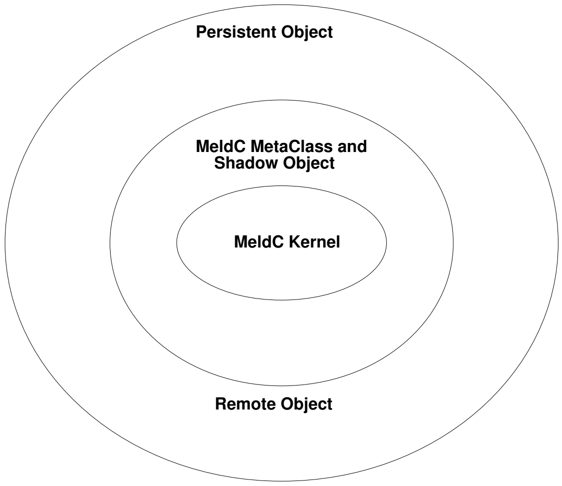

Figure 1.1: High Level MeldC Model

ically. Thus MeldC provides three different thread packages (interleaving stack, one-stack-per-thread and heap-based), which can be chosen with a compiler switch. Other compiler options enable the programmer to choose pre-emptive versus non-pre-emptive schedulers and either merging or overriding behavior for multiple inheritance.

1.2 MeldC Features

Now we would like to briefly describe the important features in MeldC to give you a better understanding of the overall MeldC language. Figure 1.2 shows the graphical interaction of the MeldC features.

1.2.1 Message Passing

In MeldC, objects communicate with each other through the message passing mechanism. There are three different types of messages:

- Symbolic
- String
- C like (fast message)

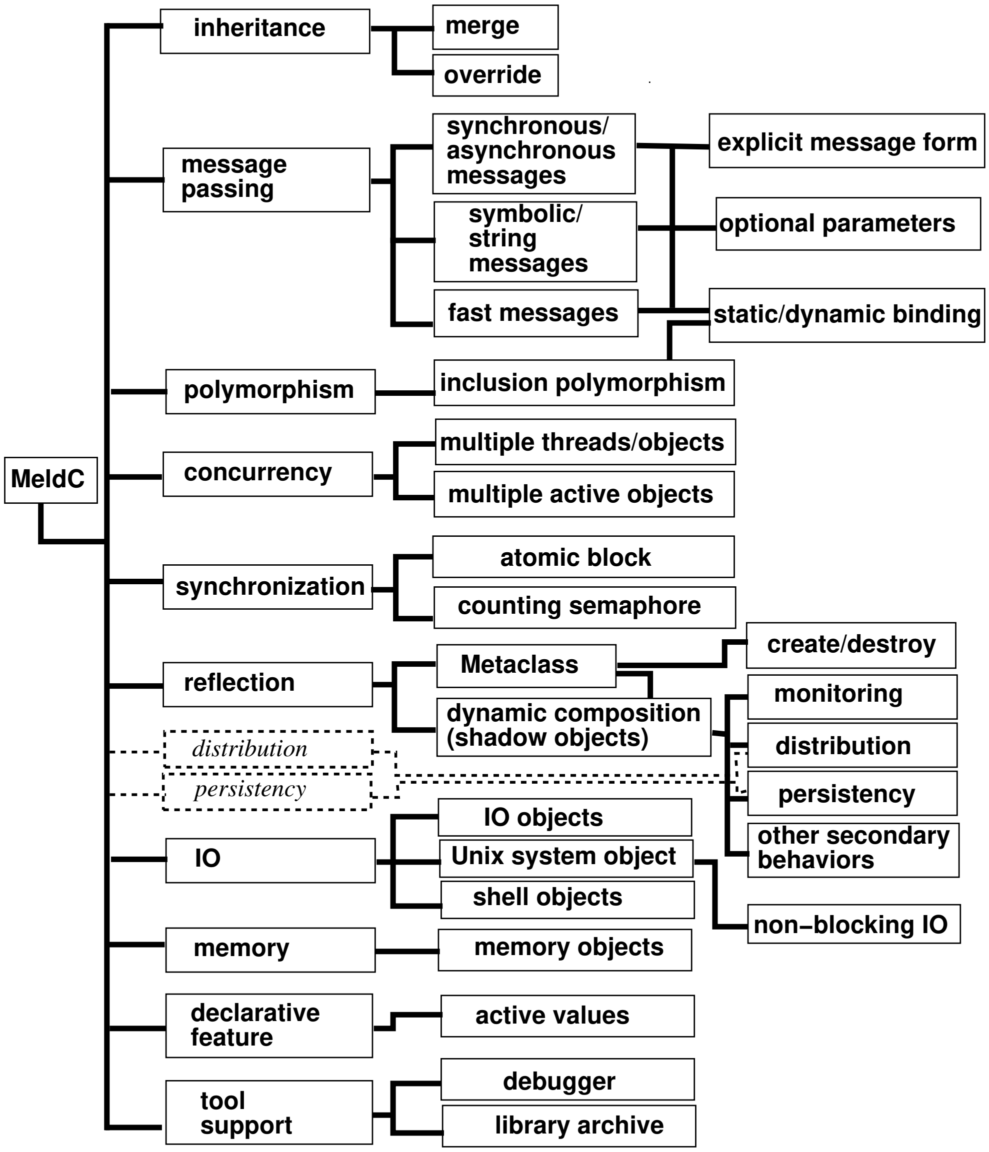

Figure 1.2: Important MELDC Features

An object can send one of these messages either synchronously or asynchronously. When a MeldC object receives a message, its default behavior is to match the message with one of its ??selector?? and fire off the method corresponding to that ??method??. This type of message is called a symbolic message. A symbolic message is similar to a function call. A simple improvement might be for an object to treat the incoming message as a regular expression. The object can then match this regular expression against the actual method name. This type of the message is called a string message. Using the MeldC messages passing schemes can entale much overhead. Sometimes, it would nice if one can fire off a C function call and by-pass the MeldC thread mechanism for the purpose of efficiency. This type of message is called C like message.

Synchronous messages in MeldC are similar to C function calls. The caller will wait for the completion of the callee before the caller continues execution. For an asynchronous message, on the other hand, the caller will not wait for the callee. Instead, the callee will be executed on a separate thread.

MeldC allows any given method to have a variable number of formal parameters. Optional parameters is a very useful feature in a programming language. It allows the programmer more flexibility by making modules more generic. One can use this mechanism to implement the parametric polymorphism, like in C++, which allows different functions to executed depending on the type of parameters.

1.2.2 Concurrency/Synchronization Mechanism

Since MeldC is claimed to be a concurrent OOPL, the concurrency obviously plays a very important role in our language. In MeldC,there are two different levels of concurrency:

- The concurrency between objects
- The concurrency within an object

Many active objects can execute simultaneously and multiple threads can be executed in an object at the same time. MeldC threads are lightweight threads, during context switching very little information is saved, hence it is less expensive than heavy-weight threads.

The synchronization mechanism is one of the crucial components in a concurrent language. In MeldC, two different types of synchronization controls are provide for the programmer:

- atomic block

- counting semaphore

The objective of *Atomic Block* is to protect instance variables from being accessed or modified by more than one thread at the time. Once a thread enters the atomic block, any other threads attempting to send messages to this object will be suspended until the atomic block is released. An atomic block provides the mechanism to synchronize the threads within the same object. MELDC also provides the counting semaphore to synchronize the execution between threads executing in different objects.

1.2.3 Inheritance

MELDC provides two different types of inheritance:

- merge
- override

The override semantic is similar to C++, which during a name conflict the child's behavior will override the parent's behavior. Unlike override, the merge semantic does not consider multiple declarations of a method in an inheritance tree a conflict. MELDC simply executes all of the methods. The merge semantic is an experimental inheritance semantic. There are few problems that have not yet been correctly resolved, such as the return value for the merge function. Nonetheless, it could be a possible alternative to the override semantics.

1.2.4 Reflection

Two very important ideas in MELDC are derived from the concept of reflection. MELDC's MetaClass is a form of *Structure reflection*, and MELDC's dynamic composition is a form if *Computation reflection*.

MELDC's MetaClass is the class of all the classes in MELDC. The MetaClass defines the behavior of all classes and is implemented in the MELDC language. Any object oriented language without the concept of a MetaClass will have to hard code an object's creation and destruction as a part of the language system. In the case of MELDC, the Create and Destroy methods are actually written in MELDC, both of which are methods of the MetaClass.

Sometimes a programmer needs to change the behavior of a MELDC program "on the fly." The power to dynamically change program behavior can be extremely useful in debugging large programs and auditing large objectbases. It can be used for more advanced uses, as well. All of MELDC's distributed programming aspects are based on the concept of dynamically modifying program behavior to form a link to other MeldC processes. In MeldC, however, this power does not come from self-modifying code. Rather, MeldC offers the dynamic extension of object behavior through the use of the reflective (Computational reflection) architecture.

1.2.5 Active Value

Active values are variables that cause side effects (typically assigning a value to another variable) to occur immediately after their value is changed or they are assigned a value. For example, if we have an equation which provides the relation between Fahrenheit-Celsius. Using the active-value, it does not matter whether F or C is changed, the MeldC will ensure the resulting values will be correct by changing F or C proportionally.

1.2.6 UNIX I/O Interface

UNIX processes can be executed concurrently inside the UNIX kernel. For example, any two UNIX processes can concurrently initiate I/O to the same device. Unfortunately, this is not the case for the MeldC light-weight threads. Even though UNIX does provide the asynchronous I/O facility, the task might be too complex for the regular programmer. MeldC provides a class which simulates the synchronous UNIX I/O for each light-weight thread. The use of a UNIX I/O interface can give the user the appearance of a blocking I/O and yet at the same time, allow other threads to continue their execution.

1.2.7 Tool Support

There are two important tools that are commonly used by the MeldC programmer:

- the debugger
- the library archive

The MeldC debugger is built on top of gdb. This gives mcgdb a means to debug MeldC code with a standard interface. It also allows the use of gdb's ability to step through and examine the C statements which compose the majority of a MeldC program's methods. The mcgdb will also allow a programmer to exam the instance variables of an object.

The library archive, mar, allows a MeldC programmer to group a set of related classes together and form a class library. The library archive also provides many facilities like revision control and the ability to determine the dependencies of a feature, which automatically places/extracts them in/from the same archive. The mar command is also a good mechanism to organize the classes in the MeldC language.

1.2.8 Memory Management

The implementation of MeldC in C implies that the global heap space is shared by all objects. This might not be an acceptable solution. Instead, we believe each individual object should have its own memory space. That is, each piece of the memory allocated should have an owner object associated with it. This means that only the owner is allowed to free the object's memory. By localizing the memory allocation, we reduce the chance of memory leak and yet at the same time, it is extremely helpful in debugging a program with a memory problem.

1.3 Final Remarks

The object of this manual is to show you, the user, the actual code used in implementing MeldC. The manual has been broken up into major components of the MeldC language. These chapters include the following topics:

- The MeldC Kernel - Chapter 2.
- The MeldC header file, Meldc_user.h - Chapter 3.
- The MeldC MetaClass - Chapter 4.
- The Shadow Object concept and their usage in MeldC - Chapter 5
- The MeldC Persistent Objects - Chapter 7
- The MeldC Remote Objects - Chapter 8

Each of these chapters covers a major piece of MeldC in an order that, we feel, will make later chapters more understandable. This does not mean the chapters must be read in this order. If you have prior knowledge of any of these topics for MeldC, then you should be able to just skip to the chapters that you are not familar with. We hope that this serves you well in your exploration of the MeldC language and we welcome any questions that you may have.

Please direct your questions to:

- MeldC@cs.columbia.edu

Chapter 2

Kernel

2.1 Introduction to Kernel

The MeldC Kernel, which provides buffer management and a control mechanism of threads for the runtime system is shown in the shadowed area in Figure 2.1.
It's main goal is to provide a mechanism for MeldC to operate properly in a UNIX environment. It is used for the following purposes:

- Starting of MeldC.
- Allocating memory, which includes memory for objects in MeldC.
- Handling signals.
- Scheduling jobs to run and be suspended.
- Linearizing messages so that they may be sent across a network to a remote source.
- Delayuntil/responses.
- Creating new threads.
- Enqueuing and dequeuing of threads from the ready queue.
- Handling stack switches.

To see how the some of the Kernel's features fits into the whole MeldC architecture, refer to Figure 2.2.

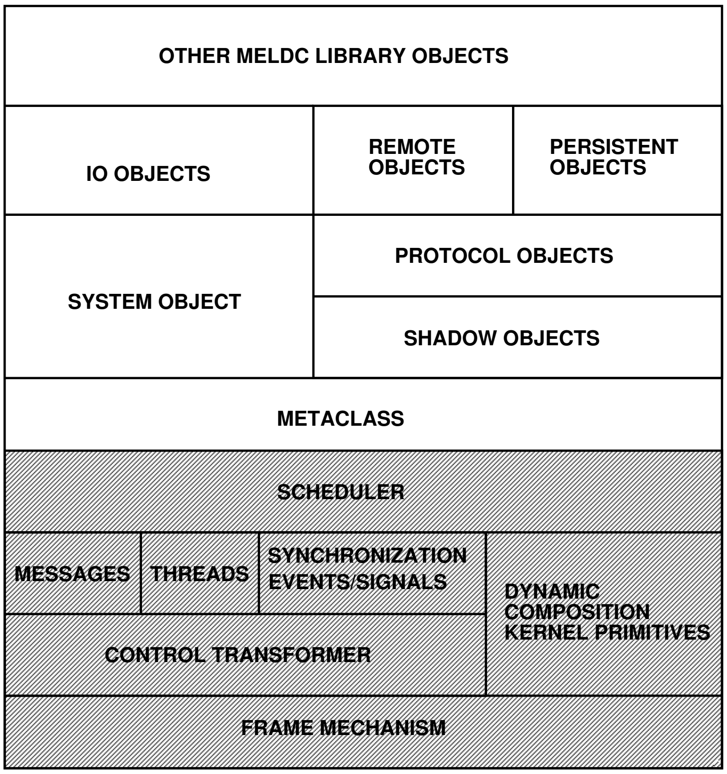

Figure 2.1: Kernel Architecture

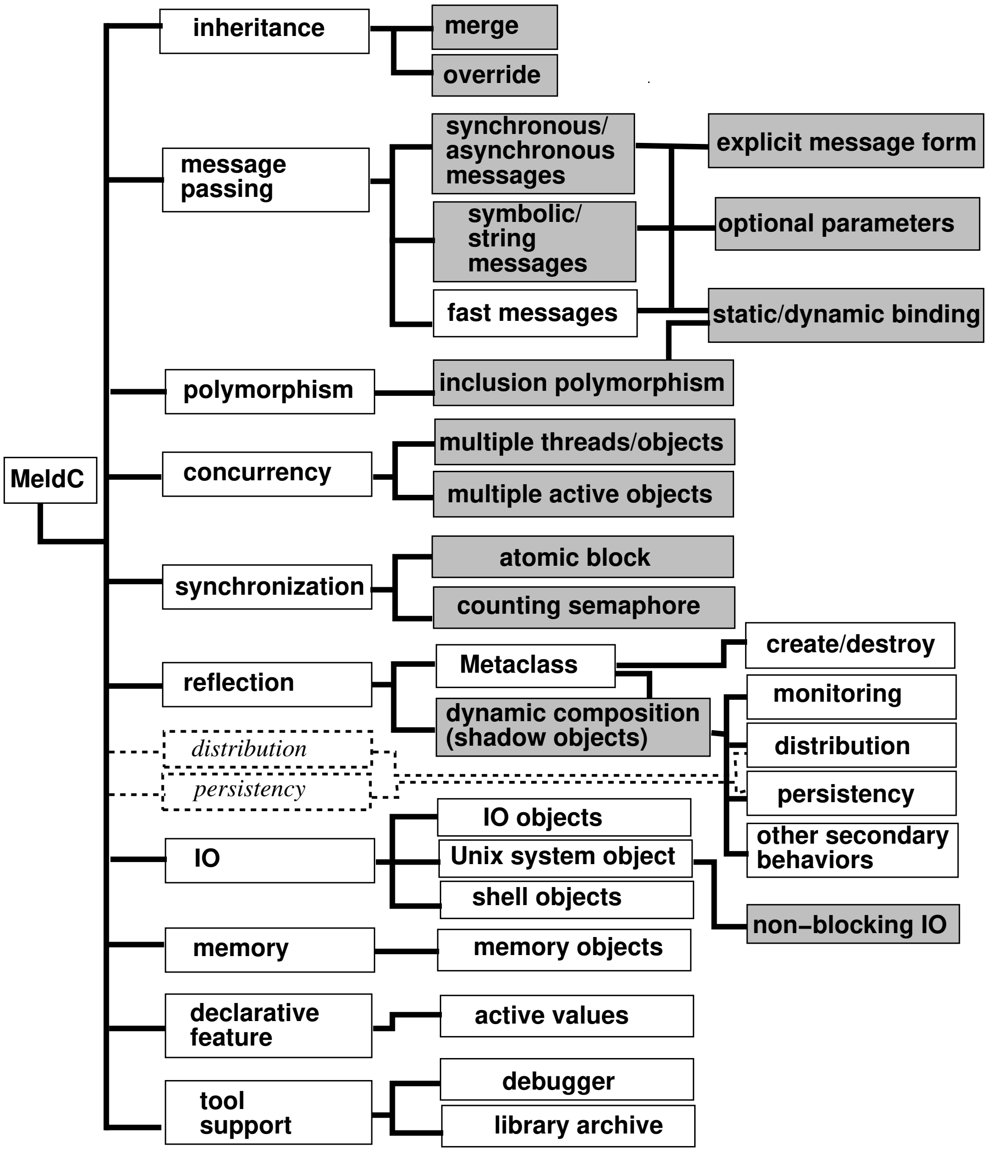

Figure 2.2: Important Kernel Features

In this chapter of the document, only the frame allocation mechanism will be discussed in much detail. All other sections will be discussed more briefly than the rest of the MeldC manual. This is due to the fact that most of the concepts here are older, well known ideas. We will give an overview of the concepts though to show how the Kernel interacts with the rest of the MeldC environment. It must also be noted, that the Kernel has been programmed with C concepts as opposed to MeldC concepts. This should make this chapter easier to understand just for that reason alone.

2.2 Message Transformation Mechanism

In our model, internal messages, serving as the control entities (function calls) in object-oriented languages, external messages, serving as the medium in interprocess communication, events and signals, the medium for intraprocess communication, and threads are **isomorphic** manifesting themselves differently at the conceptual level to programmers. At the system level, these **messages** are ultimately transformed into an internal messages which trigger invocation of methods. For example, an external message from a remote object is transformed into an internal message when it arrives at a machine. An internal message is transformed into a thread when it arrives at an object.

One of the major tasks in the MeldC Kernel is to provide this service. Our system also distinguishes threads and messages from objects. The MeldC runtime system does not model a thread as an object. An internal message **metamorphoses** into a thread, which represents the execution of the method triggered by the message arriving at an object. A function call on an object is modeled as a sequence of four steps:

1. The calling thread is transformed into a message
2. When the message arrives at the object, the message is transformed into a thread, which executes the called function
3. When the function completes, the thread is transformed into a reply message, which carries the return value
4. When the reply message arrives at the calling object, it is transformed back to a thread, which continues the code after the call statement

Threads and messages are each identified by a globally unique id. Since the transformation is handled by the MeldC Kernel transparently, it provides an easy way to implement remote procedure calls as well as remote light-weight threads.

The integration of internal messages, external messages, events, signals and threads greatly facilitates building dynamic systems in a distributed environment. A dynamic system is characterized by the dynamic feature of programs, which preserves the compile-time information (**e.g.**, function/variables names and addresses) throughout the runtime execution, such that external objects (**e.g.**, the end-users) can dynamically bind programme elements to certain properties. One notable example of dynamic binding is allowing end-users to dynamically invoke a method. At runtime, the end-user merely composes a message which carries information about the function name and the values of input parameters. The system also supports dynamic type checking since it is often necessary to ensure correct program behaviour.

While some programming paradigms provide some sense of dynamic function invocation with the use of dispatch tables, which are to be filled in by programmers explicitly, MELDC provides a transparent format of dynamic binding. The programmers are not required to provide the compile-time information as part of their programs. When an external message, in a pre-defined format arrives at a process, the system readily transforms it into an internal message which triggers the execution of a method. Our notion of transparency between external messages and internal messages creates an illusion that objects live in a network without machine and process boundaries. An object can communicate with any other object, as long as it can name the remote object. This is possible since MELDC enforces a globally unique naming convention for objects. An end-user can pretend to be an object in the network, as long as he/she can simulate and interpret messages. A dynamic system built in MELDC can treat I/O devices, the X servers or other network entities as remote MELDC objects; this feature again illustrates our principle of object uniformity.

Our architecture defines clearly a transformation mechanism between different types of control entities, which include external messages, function calls, events, and signals, and internal messages. We treat internal messages as the fundamental control entities in our system and all other forms of interaction between objects will ultimately be transformed into this format. This control uniformity greatly reduces the complexity needed in the Kernel to handle various types message passing. Moreover, this design avoids the pitfalls of incompatibility between different control mechanisms in large scale distributed systems. The behaviour of the runtime system in handling message passing can easily be modified by revising the transformation mechanism component in the Kernel. Should a new message passing entity be needed, it can be implemented fairly easily; the Kernel merely needs to define a new transformation mechanism.

2.3 Main.c

2.3.1 Description of main.c

The first piece of code to discuss is the main.c file. In this file resides the start up of the MeldC environment. The main.c file will be used to initialize part of the system through a series of system calls. The first function call will be to the minitialize function, located in the system.c file. This function will handle the initialization of the system variables. The next function call, in the main.c file, is to the thread_init function, located in the thread_init.c file, discussed in Section 2.4. This function will handle the allocation of stack space for the threads to be placed in memory. The next function call in main.c is to the signal_init function, defined in the file signal_init.c, discussed in Section 2.5. This function will be used to set up and initialize the signals that will be used by the system. The last of the functions to be called by main.c is to the Init_MsgTables function, defined in the file delayuntil.c, discussed in Section 2.10.3. It will be used to initialize the message tables used for delayuntil/response messages. More will be discussed later on these tables in the delayuntil.c file, see Section 2.10. Next we make an mfork call, located in the file event.c, discussed in Section 2.11. It is similar to a fork call in Unix, where it will create a new thread. Then we will call the driver function, the Kernel scheduler, located in the driver.c file, which will then take over from here. The driver function will be discussed later in the driver.c file, see Section 2.6.

2.3.2 Code of main.c

```
struct _class_init default_class;

struct _frame * $GLOBAL$HOPELESS$P;

int $MeldC$MeldC$Start();
struct $MeldC$MetaClass *$MeldC$MetaClass;

struct global_lock global_lock;

int X11_fd;

int atomic_lock_count = 0; /*
                            * Number of atomic lock current
                            * held by the system
```

```
                            */
extern Display * display;
extern XtAppContext context;

int global_argc;
char ** global_argv;
char ** global_envp;
struct _object *global_obj;

#ifdef sun4
#ifdef PREEMPTIVE
extern int preemptive;
#endif
#endif

main(argc,argv,envp)
char * argv[];
char * envp[];
int argc;
{

#ifdef PREEMPTIVE
  (void)enter_meldc_kernel();
#endif
  global_argc = argc;
  global_argv = argv;
  global_envp = envp;

  global_lock.g_lock.status = 0;
  global_lock.g_queue.head = 0;
  global_lock.g_queue.tail = 0;

  (void) minitialize();

  (void) thread_init();

  (void) signal_init();

  global_obj = (struct _object *)MALLOC(sizeof(struct _object));
  global_obj->$.class_ptr = global_obj;
  global_obj->$.object_priority = 3;
```

```
  Init_MsgTables();

  mfork(global_obj, $MeldC$MeldC$Start, 0, METACLASS_PRIORITY);

  driver();
}
```

2.4 Thread_init.c

2.4.1 Description of thread_init.c

The thread_init function, located in thread_init.c, has the main task of allocating stack space for the threads to be placed in memory. The size of the stack space will be determined by the environment variable, MELDC_-STACK. The MELDC_STACK variable will allow the user to alter the size of the stack to be used. The function will first check the MELDC_STACK variable value. The unit of measurement for the MELDC_STACK variable is defined in pages. If the value is greater than the MIN_STACK_SIZE variable, then the stack will be allocated as defined by MELDC_STACK variable. If the value of the MELDC_STACK variable is less then the minimum stack size allowed, then the default size will be used. This default size will depend on the stack method to be used:

- If the stack method to be used is the Heap method, then no stack space will be allocated since this method will allocate the space as is needed. This method, due to this fact, will be a lower bound on wasted memory.
- If the Circular Interleaving approach is used, then one very large stack will be allocated at the beginning.
- If the Stack Base approach is used, then the size of the stack will be determined by the size of each individual thread. This is known at the time of a thread's creation. This method, will have a lower bound on used CPU time.

2.4.2 Code of thread_init.c

```
#define SIG_STACK 10240

int max_memory = 0;
```

```
#define MIN_STACK_PAGE 3

#ifndef DEFAULT_APPROACH

int total_number_pages = MIN_STACK_PAGE;

extern char ** init_frame_stack();

extern char **sp;

#endif

struct _class_init default_class;

int stack_size;
struct _object *global_obj;

void thread_init()
{
  char * pointer, * getenv();

  pointer = getenv("MELDC_STACK");

  if (pointer != NULL)
      {
      /*
       * The min stack size for the meldc porgram is
       * equal to MIN_STACK_SIZE
       */
#ifdef  DEFAULT_APPROACH
      stack_size = atoi(pointer);
      if (stack_size < MIN_STACK_PAGE)
          stack_size = MIN_STACK_PAGE * getpagesize();
      else
          stack_size *= getpagesize();
#else
      total_number_pages = atoi(pointer);
      if (total_number_pages < MIN_STACK_PAGE)
          total_number_pages = MIN_STACK_PAGE;
      stack_size = total_number_pages * getpagesize();
#endif
      }
```

```
  else
      {
      stack_size = MIN_STACK_PAGE * getpagesize();
      }

#ifdef DEFAULT_APPROACH
  init_frame_stack(stack_size);
#endif
}
```

2.5 Signal_init.c

2.5.1 Description of signal_init.c

The next file is signal_init.c, which contains the signal_init function. As stated before, this function will be used to set up and initialize the signals that will be used by the system. These signals include:

- sigio
- sigalarm
- sigusr1 – used only for preemptive scheduling

One thing to note about the signals is that, they are handled on a different stack. That is, instead of being on the C stack, they have what is referred to as the sig_stack. The sig_stack is used only by the signals. This was done for the reason that MELDC manipulates the C stack. If a signal came in the middle of the manipulation of the C stack, which could happen, then it could possibly mess up the C stack. For this reason, the stacks remain separate.

2.5.2 Code of signal_init.c

```
#define MIN_STACK_SIZE 10000
#define SIG_STACK 10240

double sig_stack[SIG_STACK];

void (*sigio_func)();
void (*sigalarm_func)();

extern int iohandler();
```

```
extern int alarmhandler();
extern void signal_handler();

void signal_init()
{
  struct sigstack ss;
  struct sigvec vec;
  struct sigvec ovec;

  ss.ss_sp = (char *)(&sig_stack[SIG_STACK-1]);
  ss.ss_onstack = 0;

  if (sigstack(&ss,NULL) == -1)
    {
    printf("bad signal stack. \n");
    exit(-1);
    }

  vec.sv_handler =(void (*) ()) alarmhandler;

  vec.sv_flags = SV_ONSTACK;

  vec.sv_mask = sigmask(SIGALRM) | sigmask(SIGIO) |
                sigmask(SIGUSR1);

  if (sigvec(SIGALRM,&vec,&ovec) == -1)
    {
    printf("bad sigvec . \n");
    exit(-1);
    }

  sigalarm_func = ovec.sv_handler;

  vec.sv_handler =(void (*) ()) iohandler;

  vec.sv_flags = SV_ONSTACK;

  vec.sv_mask = sigmask(SIGALRM) | sigmask(SIGIO) |
                sigmask(SIGUSR1);

  if (sigvec(SIGIO,&vec,&ovec) == -1)
    {
```

```
    printf("bad sigvec . \n");
    exit(-1);
    }

#ifdef PREEMPTIVE
  sigio_func = ovec.sv_handler;

  vec.sv_handler =(void (*) ()) signal_handler;

  vec.sv_flags = SV_ONSTACK;

  vec.sv_mask = SIGUSR1;

  if (sigvec(SIGUSR1,&vec,&ovec) == -1)
    {
    printf("bad sigvec . \n");
    exit(-1);
    }
#endif
}
```

2.6 Driver.c

2.6.1 Description of driver.c

The driver.c file has the main function as the systems scheduler. That is, it will handle the running and suspending of jobs. It will do the enqueuing and dequeuing of frames as well as the RUN and RESUME functionality. Lets look at it more closely.

2.6.2 Headers of driver.c

```
extern int stack_size;
extern struct global_lock global_lock;

#ifdef  DEFAULT_APPROACH
extern char ** sp;
#endif

#define RUN { _obj=_frame->$.obj ; _frame->$.state = S_READY;
              run(); }
```

```
#define RESUME { _obj=_frame->$.obj ; resume(); }

#define ENQ_F(h,t,f) ((t==0)?(h=t=f):(t=t->$.next=f),
                       f->$.next=(struct _frame *) 0)
#define DEQ_F(h,t,f) ((f=h)!=0?((h=h->$.next)==0?
                      (t=(struct _frame *)0):0):0)

int _threadid = 0;
int _threadcount = 0;

struct _object *_obj;
     /* current obj (global variable for passing parameter */
struct _frame *_frame;
     /* current frame (global variable)                    */
```

2.6.3 _unblock_frame

Description of _unblock_frame

The driver.c file starts off with the atomic unlock function. This function will unlock the lock set by the atomic lock function. The lock will prevent frames within the same object from accessing the object if an atomic lock has been set. This means that the only frame allowed to reenter and execute within the object is that frame with a threadid equal to that threadid that has the lock. If more than one frame is competing to set the lock, only one will get the lock. Once this frame has obtained the lock, all other frames attempting to access this object will be blocked and put on the objects lock queue until the lock is released. At such a time, all of the frames waiting on that atomic lock just released will be put back on the ready queue. This will allow these frames to continue execution. Some will just execute and finish, while others may set the lock again.

Code of _unblock_frame

```
_unblock_frame(obj)
struct _object *obj;
{
  struct _frame *fp;

  while (obj->$.lock_queue->head)
      {
      DEQ_F(obj->$.lock_queue->head, obj->$.lock_queue->tail,
```

```
            fp);
      ENQUEUE_FRAME(fp);
      }
}
```

2.6.4 Global_unlock

Description of global_unlock

There is also a global_unlock function, which is used very infrequently. The reason is that, a global lock will not just lock an object, but all of the code. This will prevent all objects but one from executing. This means that all frames that belong to the object that has the lock will be allowed to execute. The misuse of this lock, could possibly cause the system to deadlock, see Section 3.8 for more on this topic. Not only that, but it also defeats the purpose of what the system was designed for, multiple threads. For this reason, a global lock and unlock will be used only for the attachment of shadow objects, discussed in Chapter 4, the MetaClass chapter.

Code of global_unlock

```
global_unlock()
    {
    struct _frame *fp;
    while(global_lock.g_queue.head)
        {
        DEQ_F(global_lock.g_queue.head,global_lock.g_queue.tail,
              fp);
        ENQUEUE_FRAME(fp);
        }
    }
```

2.6.5 Driver

Description of driver

Next we come to the actual driver function itself, which serves the purpose as the scheduler for MeldC. It will first check to see if a global lock has been set. If there is a lock set, then the frame will be placed on a lock queue. This means that the frame has just been blocked. Now if there is no lock, we must check the status of the frame.

If the frame status is S_NEW, it means that the frame has not yet been executed. We will first check to see if an atomic lock has been set by another frame. If so, then this frame will be put onto the lock queue until the lock is released. Otherwise, no lock has been set or this frame has the lock set. It should be noted that the lock queue for the atomic lock is part of the object structure. At this point we must then assign a thread id structure to the frame. If the frame was called asynchronously, we must then construct a new thread id structure for the frame and store it in the frame structure, _frame. If the frame was called synchronously, then the frame id will be inherited from the caller. And finally, we can then invoke the MeldC assembly RUN. The RUN will execute the frame. See Section 2.16.1 for more on the assembly RUN.

If the status of the frame is not S_NEW, then it must be S_READY. We will first check to see if the atomic lock was set. If not, we will then invoke the MeldC assembly RESUME. The RESUME will enqueue the frame to the ready queue to allow it to execute again from where it had left off before a YIELD. Note that this implies that a RESUME is done only after one case, a YIELD. If the atomic lock was set by some other thread, then the frame will be blocked and put on the objects lock_queue. Otherwise, this frame will be restarted with the RESUME call. See Section 2.16.3 for more on the assembly RESUME.

Let us briefly discuss a YIELD at this point. A YIELD is done when either:

- A job decides to hand control over to some other job in the system to allow equal time usage.
- Or a YIELD may occur when a job is suspended when waiting for a lock to be removed from an object.

Once a frame does a YIELD, it could be called from either the RUN routine or from the RESUME routine. As stated above, the YIELD will return the frame of execution to the point right after the point of where the RESUME was called. See Section 2.16.5 for more on the assembly YIELD.

When a frame has completed its execution, the frame will be returned back to the point right after the RESUME, or otherwise known as, under the control of the scheduler. At this point, the frame can have three different statuses. Each status will have a different effect on the frame. We will execute a switch statement on the status to decide what step to perform on the frame next. Here are the statuses and respective effect on the frame.

1. S_WAIT - In this case, the frame is waiting for another frame to complete it's execution. So we will do nothing and leave it suspended.
2. S_READY - Here we will simply return the frame back to the ready queue, enqueue the frame.

3. S_EXIT - This status states that the job is to be destroyed.

    - If this frame was called synchronously, then depending on the stack type we will determine the method used to free up the old frame. Based on the stack type, we will call the free_frame function, discussed in frame_alloc below, in the proper fashion to free the frame. After the old frame is freed, we will retrieve the caller of the frame and the caller frame will then be enqueued and set to READY, so that the caller frame may run again.
    - Now, if the frame was called asynchronous, we will usually not have a caller. When this is the case, we will free the frame. Not having a caller is true for all but one exception, the case of a shadow object. See Chapter 5, the shadow objects chapter, for more.

Code of driver

```
driver(){
  struct _frame *tmp_frame;
  int priority;
        /* ranged from 0 to 6, 6 is the highest priority */
  int memory;

#ifdef PREEMPTIVE
  enter_meldc_kernel();
#endif
  for(;;){
    while((priority = dequeue(&_frame)) >= 0) {
      /* process the event */

      if ( (global_lock.g_lock.status) && (
           (_frame->$.caller == 0) ||
           (memcmp((char *)&(global_lock.g_lock.threadid),
                   (char *)&(_frame->$.caller->$.threadid),
                   sizeof(struct _threadid)))))
           {
           ENQ_F(global_lock.g_queue.head,
                 global_lock.g_queue.tail,
                _frame);
           continue;
           }
```

```
      if(_frame->$.state == S_NEW) {

        if (_frame->$.obj != 0 && _frame->$.obj->$.lock &&
           _frame->$.obj->$.lock->status > 0 &&
           (_frame->$.caller == 0 ||
           (memcmp((char *)&(_frame->$.obj->$.lock->threadid),
                   (char *)&(_frame->$.caller->$.threadid),
                   sizeof(struct _threadid)))))
            {
            ENQ_F(_frame->$.obj->$.lock_queue->head,
                _frame->$.obj->$.lock_queue->tail,
                _frame);
            continue;
            }

        if (_frame->$.caller == 0)
            {
            _threadcount++;
            if (_frame->$.obj)
                _frame->$.threadid.objid = _frame->$.obj->$.objid;
            else
                {
                _frame->$.threadid.objid.pid = 0;
                _frame->$.threadid.objid.counter = 0;
                _frame->$.threadid.objid.hostid = 0;
                }
            _frame->$.threadid.counter = _threadid ++;
            }
        else
            {
            _frame->$.threadid = _frame->$.caller->$.threadid;
            }

#ifdef STACK_BASE
        if (_frame->$.ssp == NULL)
             {
             _frame->$.ssp = _frame->$.caller->$.ssp;
             _frame->$.sfp = _frame->$.caller->$.sfp;
             }
#endif
        RUN;
      }
```

```
      else {
        if (_frame->$.obj != 0 && _frame->$.obj->$.lock &&
           _frame->$.obj->$.lock->status > 0 &&
           ((memcmp((char *)&(_frame->$.obj->$.lock->threadid),
                    (char *)&(_frame->$.threadid),
                    sizeof(struct _threadid)))))
            {
            ENQ_F(_frame->$.obj->$.lock_queue->head,
                _frame->$.obj->$.lock_queue->tail,
                _frame);
            continue;
            }
        RESUME;
      }
#ifdef sun4
#ifdef PREEMPTIVE
      (void)enter_meldc_kernel();
#endif
#endif

      switch(_frame->$.state)
          {
          case S_EXIT :

              /*
               * Generate a event (destruction of the thread)
               */


              if(((tmp_frame = _frame->$.caller) != 0) &&
                  (_frame->$.sync_mode == SYNC))
                  {
                  /*
                   * Synchronous call
                   */

#ifdef DEFAULT_APPROACH
                  if (_frame->$.add_memory)
                      {
                      m_free_all(_frame->$.add_memory);
                      }
                  free_frame(_frame);
```

```
#endif

#ifdef STACK_BASE
                  if (_frame->$.add_memory)
                      {
                      m_free_all(_frame->$.add_memory);
                      }
                  free_frame(_frame);
#endif

#ifdef STACK_BASE_SLOW

                  if (_frame->$.add_memory)
                      {
                      m_free_all(_frame->$.add_memory);
                      }

                  memory = stack_size - ((char *)_frame->$.ssp -
                                         (char *)_frame->$.original_ssp);

                  /*
                   * If the caller thread is not the interrupt thread
                   * from the kerenl. (Kernel trap use by the shadow
                   * object.) we will free the frame.
                   */
                  if (_frame->$.original_ssp ==
                      _frame->$.caller->$.original_ssp)
                      free_stack_frame(_frame);
#endif

                  tmp_frame->$.state = S_READY;
                  enqueue(tmp_frame);

                  }
              else
                  {
                  /*
                   * Asynchronous call
                   */

                  /*
                   * In most of the case, asynchronous request usually do not
```

```
                  * have a caller, but with on exception.
                  *
                  * If the caller is not NULL, the caller must be a shadow
                  * object, we should enqueue the shadow object, and also
                  * free up the current frame's stack.
                  */

                 if ( (tmp_frame != NULL) &&
                      (_frame->$.sync_mode == ASYNC) &&
                      (_frame->$.obj->$.shadow_obj_struct))
                     {
                     tmp_frame->$.state = S_READY;
                     enqueue(tmp_frame);
                     }
                 /*
                  generate_notify(DEST_EVENT,_frame,URGENT);
                  */
                 _threadcount--;

#ifdef STACK_BASE
                 if (_frame->$.add_memory)
                     {
                     m_free_all(_frame->$.add_memory);
                     }
                 free_stack(_frame);
#endif

#ifdef STACK_BASE_SLOW
                 memory = stack_size - ((char *)_frame->$.ssp -
                                         (char *)_frame->$.original_ssp);
                 if (_frame->$.add_memory)
                     {
                     m_free_all(_frame->$.add_memory);
                     }
                 free_stack_frame(_frame);
#endif

#ifdef DEFAULT_APPROACH
                 if (_frame->$.add_memory)
                     {
                     m_free_all(_frame->$.add_memory);
                     }
```

```
                  free_frame(_frame);
#endif
                  }
              break;
          case S_READY :
              enqueue(_frame);
              break;
          case S_BLOCKED :
              printf("**** NOT YET IMPLEMENTED ****\n");
          case S_WAIT :
              break;
          default:
              printf("Illegal State\n");
              break;
          }
    }

    /* _signal_event(1); for wenwey's  scheduling */

    /* blocking SIGIO is very important before empty() is */
    /* called. If you cannot understand why, DO NOT TRY */
    /* TO CHANGE IT. DON'T move the critical section */
    /* (sigblock--sigsetmask) inside empty(). SIGIO (enqueue */
    /* an event) might come between empty() and sigpause() */
    BEGIN_CRITICAL_SEC;
    while(empty()){
      sigpause(sigmask(SIGUSR1));
    }
    END_CRITICAL_SEC;
  }
}
```

2.7 Frame_alloc.c

2.7.1 Description of frame_alloc.c

The next file of importance to examine is the frame_alloc.c file. The main concern of this file is to deal with the allocation of stack space for a frame. To do so though, depends on which stack method is being used since they all must be done differently. The basic steps for all stack methods is to:

1. Use a function to allocate stack space for the frame.

2. Then later we will make a call back here again to free the frame, when S_EXIT is given as the frame status.

In this section, we will briefly discuss each stack method in general, how they use the stacks to do execution and how they keep the stacks organized and then we will discuss how each also allocates their memory in general. It should be noted that most of the current stack implementations use two different stacks. One is for execution, the C stack, and the other is the allocation stack. Every time a frame is ready to be executed, we have to swap the frame from the allocation stack to the C stack. Note that this is different than the context switch idea. MeldC still performs the context switch.

2.7.2 Headers of frame_alloc.c

```
#define S_ALLOC 1
#define S_FREE 0

#ifdef FRAME_ALLOCATION_HEAP_BASE
#define DEFAULT_APPROACH
#endif

#ifdef CIRCULAR_INTERLEAVING
#define DEFAULT_APPROACH
#endif

#ifdef DEFAULT_APPROACH

extern int max_memory;
int total_memory_used = 0;
```

2.7.3 Circular Interleaving Stack Method

Description of Circular Interleaving Stack Method

The first method encountered is the Circular Interleaving Stack method. This method will use one big stack for keeping the frames on. This method will also use two pointers. This is to keep track of where the top and bottom parts of the stack are. Between the bottom pointer, bsp, and the top pointer, sp, lies the frames. The rest of the stack is considered to be free for putting more frames. When the top stack pointer gets to the upper limit of the stack, it will wrap around. That is, it will be moved all the way around to the bottom now and will continue as if it never happened. This is why it is called circular. When a frame is to be executed, the frame will be moved to the C stack. It will be moved back and forth between the two stacks until it has completed it execution. When a frame is to be released, the status of the frame is simply set to free. This will allow other frames to use this memory location in the stack. At this time, it is vital to check and see if the pointers can be moved either up, bsp, or down, sp, to compact the stack since we now have a free stack space. The only way to move the two pointers into free space is by:

- Top pointer, sp – This pointer can only move down if the top-most frame is removed. When this frame is removed, then a check is made to see if there is a hole directly below the newly removed frame. If so, then the sp pointer can be moved all the way down until a frame is encountered, or the bottom pointer is found (this means the stack is empty).
- Bottom pointer, bsp – This pointer can only be moved up if the bottom frame is freed. Here we will also check for holes directly above this newly freed frame and incorporate them into freed stack space.

In both cases, all memory freed between the upper and lower frames will become holes. These holes are not usable though for new frames to go into. This considers these holes as wasted memory. This method in theory is supposed to use similiar memory as the Heap Base method, but it is just as efficent as the Stack Base method.

The following subsections contain the functions that are needed to perform the Circular Interleaving method.

Headers of Circular Interleaving Stack Method

```
#elif CIRCULAR_INTERLEAVING

static char *stack_top  = (char *) 0;
static char *stack_bottom = (char *) 0;
static char **sp = (char **) 0;
static char **bsp = (char **) 0;
extern int max_memory;
```

Description of init_frame_stack

We will pre-allocate the stack size by making a call to init_frame_stack function. This is done by passing the stack size wanted to the function.

The stack will be allocated with malloc and the sp and bsp pointers will be set to the bottom of the stack.

Code of init_frame_stack

```
init_frame_stack(stack_size)
int stack_size;
{
  int size = 2 * sizeof(char *);

  stack_top = malloc(stack_size);
  if (stack_top == NULL)
      {
      printf("MeldC Fatal Error : init_frame_stack() malloc
              falied. \n");
      exit(-1);
      }
  stack_bottom = stack_top + stack_size;
  sp = (char **) stack_bottom;
  sp = (char **) ((char *) sp - 3 * sizeof(char *));
  bsp = sp;
  *sp = (char *) 0;
  *(sp+1) = (char *) S_ALLOC;
}
```

Description of alloc_frame

Next we have a function used for allocation of a frame, alloc_frame. This function will simply move the sp pointer up the stack, or around to the bottom and start up again if there is not enough room on top, and return the new sp. The sp pointer is moved the distance needed to put the frame into.

Code of alloc_frame

```
char *alloc_frame(size)
int size;
    {
    char * p;
#ifdef PREEMPTIVE
    char * return_sp;
```

```
    BEGIN_CRITICAL_SEC;

#endif

    p = (char *) sp;

    if ( ((sp - bsp)) > 0  && ((sp - bsp) < size))
        {
        printf("stack overflow error \n");
        exit(0);
        }

    if((sp = (char **) ( (char *) sp - size)) < (char **)
        stack_top)
        {
        sp = (char **)((char *)sp+ size);
        *((int *) sp +2) = (int)((char *) stack_bottom -
                           3 * sizeof(char *));
        if (bsp >= (char **)(((char *) stack_bottom - 3 *
                       sizeof(char *))))
            {
            printf("stack overflow error \n");
            exit(0);
            }
        sp = (char **)(((char *) stack_bottom - 3 *
             sizeof(char *)));

        if ( ((sp - bsp)) > 0  && ((sp - bsp) < size))
            {
            printf("stack overflow error \n");
            exit(0);
            }
        sp = (char **)((char *)sp - size);
        }

    *sp = p;   /*  frame pointer */
    *((int *) p +2) = (int)sp;
    *(sp+1) = (char *) S_ALLOC;  /* status */

    if (max_memory < ((int)stack_bottom - (int)sp))
        {
        max_memory  = (int)stack_bottom - (int) sp;
```

```
        }
#ifdef PREEMPTIVE

    return_sp = sp;

    END_CRITICAL_SEC;

    return((char *) (return_sp));

#else
    return((char *)sp);
#endif
    }
```

Description of free_frame

Next we have the function for freeing a frame when we are done executing it, free_frame. Here we pass a pointer to the frame in the stack. We will mark this segment in the stack as S_FREE. Then we will check to see if we can move the bsp or sp pointers up or down, respectively, in the stack.

Code of free_frame

```
free_frame(p)
char **p;
    {
#ifdef PREEMPTIVE
    BEGIN_CRITICAL_SEC;

#endif
    if(p == sp)
        {
        for(sp = (char **)(*sp); ((*(sp+1) == S_FREE) &&
                 (sp != bsp));
           (sp = (char **)(*sp)));
        if (bsp == sp)
            sp = bsp = (char **)(stack_bottom - 3 *
                        sizeof(char *));
        }
    else
        {
        *(p+1) = (char *) S_FREE;
```

```
        while ( ( *(((int *)*(bsp + 2)) +1) == S_FREE) &&
                                               (sp != bsp))
            bsp = (char **)*(bsp+2);
        if (bsp == sp)
            sp = bsp = (char **)(stack_bottom - 3 *
                       sizeof(char *));
        }
#ifdef PREEMPTIVE
     END_CRITICAL_SEC;
#endif
    }
```

2.7.4 Heap Base Method

Description of Heap Base Method

The Heap Base method has the main idea of wasting as little memory as possible. It does so, by not allocating any memory until a new frame needs it. When a new frame needs memory, then it will be created, but not until such a time. The execution of the frames takes place on the C stack. So every time a frame is to be executed, it will be moved to the C stack. Heap Base will waste very little memory and is therefore considered the lower bound on wasting memory.

Headers of Heap Base Method

```
#elif FRAME_ALLOCATION_HEAP_BASE

#include <stdio.h>
#include "meldc_user.h"
#include "sigio.h"
#include "malloc.h"

int malloc_count = 0;
int free_count = 0;

#define BLOCK_SIZE (sizeof(struct _frame) * 2)
```

Description of init_frame_stack

This function is not needed here since there is no initial stack space needed. It just gets the space for the frames on the fly in the alloc_frame function discussed next.

Code of init_frame_stack

```
init_frame_stack()
    {
    }
```

Description of alloc_frame

The alloc_frame function will be passed the size of the frame that needs memory allocated. This size will then be used to do a malloc that will allocate the memory needed. Then the pointer to that memory will be returned.

It should be noted that a large cache mechanism can be used. That is, for us to get large pieces of memory at a time and just give small sections at a time to those who need it until the cache is gone. When the cache is gone, we should get another large piece of memory. This will speed up the allocation of the frames, since a malloc is an expensive call.

Code of alloc_frame

```
struct _frame *alloc_frame(size)
int size;
{
struct _frame *fp;
int * pointer;
int i;

i = size / BLOCK_SIZE  ;

fp = MALLOC(BLOCK_SIZE * (i + 1) +4);

total_memory_used += size;

pointer = (int *) fp;
*pointer ++ = size;
fp = (struct _frame *) pointer;

if (total_memory_used > max_memory)
    max_memory = total_memory_used;

if (fp == NULL)
    {
    printf("malloc failed. \n");
```

```
    exit(-1);
    }
fp->$.caller = (struct _frame *) 0;
fp->$.size = size;

return(fp);
}
```

Description of free_frame

The free_frame function will simply do a free on the pointer to the stack space. This will release the memory to the system. This shows why space is not wasted, there is no hole created from the stack method. It just uses the stack space memory and then returns it.

If the cache mechanism discussed in the alloc_frame function is used, the free memory can just be placed back into the cache to be redistributed.

Code of free_frame

```
free_frame(fp)
struct _frame *fp;
{
int i;
int * pointer;

if (fp == NULL)
    return;

pointer = (int *) fp;
pointer --;

if (*pointer == 0)
    {
    printf("Error in Free Frame. \n");
    return;
    }

FREE(pointer);
return;
}
```

2.7.5 Stack Base Slow Method

Description of Stack Base Slow Method

The Stack Base Slow method has one stack per thread. That is, the stack is made up blocks for each frame, where the frame size is fixed at the MELDC_STACK value. The method will simply find a block in the allocated stack that is marked FREE for the frame to go into. Then when a frame is freed, the method will free the block that the frame resided in by marking the block as FREE. It will use the C stack for execution of each frame by swapping them in and out of the two stacks.

Headers of Stack Base Slow Method

```
#ifdef STACK_BASE_SLOW

#define MAX_BUFFER_STACK 200

extern int total_number_pages;
int free_stack_counter;
char * free_stack_cache[MAX_BUFFER_STACK+1];
```

Description of free_stack

For the Stack Base Slow method, we will first have the free_stack function. It will be passed the fp pointer to do so. The block will be declared as free to be used by another frame.

Code of free_stack

```
id free_stack(fp)
struct _frame * fp;
    {
    caddr_t addr;

#ifdef PREEMPTIVE
    BEGIN_CRITICAL_SEC;
#endif

    if (free_stack_counter < MAX_BUFFER_STACK)
        {
        free_stack_cache[free_stack_counter ++] =
                                  fp->$.original_ssp;
```

```
        }
    else
        {
        /*
         * Restore back to its original address.
         */
        if (munmap(fp->$.original_ssp,
            (getpagesize() * total_number_pages)) == -1
)
            {
            perror("ERROR munmap: ");
            }
        }
#ifdef PREEMPTIVE
    END_CRITICAL_SEC;
#endif
    }
```

Description of allocate_stack

The next function is for the allocation of the stack. A pointer is passed that has the value of the top of the stack. Here a pointer is returned to the piece of memory to use.

Code of allocate_stack

```
char ** allocate_stack(stack_top)
char ** stack_top;
    {
    char ** pointer;
    char * temp_stack_top;

#ifdef PREEMPTIVE
    BEGIN_CRITICAL_SEC;
#endif

    if (free_stack_counter > 0)
        {
        free_stack_counter --;
        pointer =
             (char **)free_stack_cache[free_stack_counter];
        *stack_top = (char *)pointer;
```

```
        free_stack_cache[free_stack_counter] = 0;
        pointer = (char **) ((char *)
           pointer + (getpagesize() * total_number_pages));
        pointer = ((char **) pointer - 2 * sizeof(char *));
        *pointer = (char *) 0;
        *(pointer+1) = (char *) S_ALLOC;
        /*
        return((char **)pointer);
        */
        }
    else
        {
        pointer = (char **)init_frame_stack(total_number_pages,
                  &temp_stack_top);
        *stack_top = temp_stack_top;
        }

#ifdef PREEMPTIVE
    END_CRITICAL_SEC;
#endif

    return((char **)pointer);
    }
```

Description of free_stack_frame

This function, free_stack_frame, will be used to free the frame off of the stack by making calls to both the free_frame and free_stack functions.

Code of free_stack_frame

```
void * free_stack_frame(fp)
struct _frame * fp;

    {
#ifdef PREEMPTIVE
    BEGIN_CRITICAL_SEC;
#endif

    if (fp->$.caller != NULL)
        {
        free_frame(fp,&(fp->$.caller->$.ssp));
```

```
        }
    else
        {
        free_frame(fp,&(fp->$.ssp));
        free_stack(fp);
        }
#ifdef PREEMPTIVE
    END_CRITICAL_SEC;
#endif
    }
```

Description of a_new_thread

When a new thread is created, the memory is allocated for the thread using the a_new_thread function. Here the functions allocate_stack and alloc_frame are called to do so.

Code of a_new_thread

```
struct _frame * a_new_thread(s)
int s;
    {
    char * stack;
    struct _frame * fp;
    char * stack_top;
    struct _frame * alloc_frame();

#ifdef PREEMPTIVE
    BEGIN_CRITICAL_SEC;
#endif

    stack = (char *)allocate_stack(&stack_top);

    fp = (struct _frame *) alloc_frame(s,&stack);

    fp->$.ssp = (char **)stack;
    fp->$.original_ssp = stack_top;

#ifdef PREEMPTIVE
    END_CRITICAL_SEC;
#endif
    return(fp);
```

```
  }
```

Description of alloc_frame

The alloc_frame function will allocate the memory needed for the frame using the size of the frame. The frame pointer will be set to the frame and the status of the memory block will be set to S_ALLOC.

Code of alloc_frame

```
struct _frame *alloc_frame(size,sp)
int size;
char ***sp;
{
  char *p;

  p = (char *) *sp;
  *sp = (char **) ( (char *) *sp - size) ;

  **sp = p;   /*  frame pointer */
  *(*sp+1) = (char *) S_ALLOC;  /* status */

  return((struct _frame *) (*sp));
}
```

Description of free_frame

The free_frame function will be used to free the frame. This is done by setting the status of the block in the stack to S_FREE.

Code of free_frame

```
free_frame(p,sp)
char ***sp;
char **p;
{
  if(p == *sp){
    for(*sp = (char **)(**sp); *(*sp+1) == S_FREE;
        *sp = (char **)(**sp));
  }
  else {
    *(p+1) = (char *) S_FREE;
  }
```

```
}

#endif STACK_BASE_SLOW
```

2.7.6 Stack Base Method

Description of Stack Base Method

The stack base method is system dependent and has been written for the sun4 case only. This method will have a different stack for each frame as in the Heap Base method. But this method will execute the frame on the stack that it is saved on. It will have $n + 1$ stacks for n threads. The execution of a frame using this method, that is RUN, YIELD and RESUME, will use the same stack as frame allocation. It will push the C frame on top of the stack that we allocated for the frame. When a YIELD occurs, the current sp and fp will be saved into the global frame structure. The RESUME will then use these values to start the process where it left off later.

Headers of Stack Base Method

```
#ifdef STACK_BASE

#define MAX_BUFFER_STACK 200

extern int total_number_pages;
int free_stack_counter;
char * free_stack_cache[MAX_BUFFER_STACK+1];
```

Description of free_stack

This function, free_stack, will be used to free up a stack that was allocated to some frame. The frame would have to be allocated by the allocate_stack function, discussed next.

Code of free_stack

```
void free_stack(fp)
struct _frame * fp;
    {
    caddr_t addr;

    total_allocation --;
    if (free_stack_counter < MAX_BUFFER_STACK)
```

```
        {
        free_stack_cache[free_stack_counter ++] =
                                       fp->$.original_ssp;
        }
    else
        {
        /*
         * Restore back to its original address.
         */
        if (munmap(fp->$.original_ssp,(getpagesize()*
                                total_number_pages)) == -1)
            {
            perror("ERROR munmap: ");
            }
        }
    }
```

Description of allocate_stack

The frame allocation function, frame_allocation, is done synchronous. It will allocate a new stack and the frame structure will become the first entry on the stack.

Code of allocate_stack

```
char ** allocate_stack(stack_top)
char ** stack_top;
    {
    char ** pointer;
    char * temp_stack_top;

    total_allocation ++;
    if (free_stack_counter > 0)
        {
        free_stack_counter --;
        pointer = (char **)
                  free_stack_cache[free_stack_counter];
        *stack_top = (char *)pointer;
        free_stack_cache[free_stack_counter] = 0;
        pointer = (char **)
                  ((char *)pointer + (getpagesize() *
                                      total_number_pages));
```

```
        pointer = ((char **) pointer - 2 * sizeof(char *));
        *pointer = (char *) 0;
        *(pointer+1) = (char *) S_ALLOC;
        return((char **)pointer);
        }

    pointer = (char **)init_frame_stack(total_number_pages,
                                        &temp_stack_top);
    *stack_top = temp_stack_top;
    return((char **)pointer);
    }
```

Description of free_frame

We now have the free_frame function used to free a frame when it is done executing. The sp will be restored, if the caller is not NULL. All of this information has been saved in the frame structure already. If the caller is NULL, then the entire stack will be freed.

Code of free_frame

```
void * free_frame(fp)
struct _frame * fp;
    {
    if (fp->$.caller == NULL)
        return;
    fp->$.caller->$.ssp = fp->$.fp;
    }
```

Description of a_new_thread

The asynchronous method will be done with the function a_new_thread. If it is done synchronous, then the new frame will be located at the top of the stack and the sp pointer will be moved to the top also. The old sp will also be saved into the frame structure at this time.

Code of a_new_thread

```
#ifdef sun4
struct _frame * a_new_thread(s)
int s;

    {
```

```
    char * stack;
    struct _frame * fp;
    char * stack_top;

    /*
     * In sun4, the stack has to be 8 bytes align.'
     */

    s = (s /8) * 8 + 8;
    stack = (char *)allocate_stack(&stack_top);

    fp =  (struct _frame *)(stack - s - 96);

    fp->$.original_ssp = stack_top;

    fp->$.sfp = (char **)(stack - 96);
    fp->$.ssp = (char **)(fp - 64);

    return(fp);
    }
#endif

#endif /*ifdef STACK_BASE */
```

2.8 Meldc_malloc.c

The next file in the Kernel to discuss is the file meldc_malloc. As of now, malloc is still used for low level memory allocation. This is due to the fact that, during the course of the MELDC execution, there are many small pieces of memory that get allocated. It is difficult and very complex to keep track of them. That is, it is difficult to determine when the pointer is no longer useful. It is therefore necessary to have a design that will ensure all of the allocated memory can be properly freed. These routines provide that mechanism by grouping all of this memory together and free it when finished using that memory.

2.8.1 M_create

Description of m_create

The first function to discuss is the m_create function. This function is used to create a parent for the memory hierarchy. A malloc is simply used to allocate a MeldC memory structure.

Code of m_create

```
M_MEM m_create(parent)
M_MEM parent;
    {
    M_MEM ptr;

    BEGIN_CRITICAL_SEC;
    /*
     * Allocate MeldC memory structure.
     */
    ptr = (M_MEM)MALLOC(sizeof(MEM));
    ptr->parent = parent;
    ptr->head = NULL;
    ptr->magic_number = MAGIC;
    ptr->next = (M_MEM)NULL;

    if (parent != NULL)
        {
        if (parent->next != NULL)
            ptr->next = parent->next;
        parent->next = ptr;
        }
    END_CRITICAL_SEC;
    return(ptr);
    }
```

2.8.2 M_malloc

Description of m_malloc

The next function, m_malloc, is used to allocate the memory and then save the information of this piece of memory under the parent pointer. The memory is of the form:

Magic Number
Next Pointer
size of memory
actual memory

The magic number is a fixed value that is used to check if the memory has been corrupted by checking the bits value. The next pointer points to the next memory location. The size is obviously how large this piece of memory is and then the stored information finally comes next. The memory tree would look like this:

parent node ⟶ memory ⟶ memory ⟶ memory
↕
↕
parent node ⟶ memory ⟶ memory ⟶ memory

The m_malloc function will receive two parameters. A pointer to the parent and the size of memory needed. If the parent pointer is NULL, then we will call the m_create function. This will allocate the memory needed. This memory will than be attached to the parent and then we will return. If the parent pointer is not NULL, we will still allocate the memory and then simply put the allocated space into the memory tree as shown above. We will then return.

Code of m_malloc

```
char * m_malloc(parent,size)
M_MEM *parent;
int size;
    {
    int * pointer;

    BEGIN_CRITICAL_SEC;
```

```
    if (*parent == NULL)
        *parent = m_create(NULL);

    pointer = (int *)MALLOC(size + sizeof(int) * 3);
    *pointer = MAGIC;
    *(pointer+1) = NULL;

    if ((*parent)->head == NULL)
        (*parent)->head = (char *)pointer;
    else
        {
        *(pointer + 1) = (int)((*parent)->head);
        ((*parent)->head) = (char *)pointer;
        }

    *(pointer +2) = size;
    pointer += 3;

    END_CRITICAL_SEC;

    return((char *)pointer);
    }
```

2.8.3 M_free_all

Description of m_free_all

The next function, m_free_all, is used to free up the memory of the whole hierarchal tree. This is needed when the parent ends execution, but there are still children in existence. All of the children below that parent must also be freed at this time, otherwise the memory will never be freed and memory leaks will occur. The function will search recursively down the tree below the parent and release all memory associated with that parent as it recursively unloops. At the same time, the function will make sure that the user did not corrupt the memory by checking the magic number.

Code of m_free_all

```
int m_free_all(parent)
M_MEM parent;
    {
    char *  temp_ptr;
```

```
if (parent == NULL)
    return;

BEGIN_CRITICAL_SEC;

if (parent->magic_number != MAGIC)
    {
    printf("MeldC Fatal Error: m_free_all() Invalid parent
            \n");
    m_fatal_error();
    }

if (parent->next != NULL)
    m_free_all(parent->next);

while (parent->head != NULL)
    {
    temp_ptr = parent -> head;

    if (*((int *)temp_ptr) != MAGIC)
        {
        printf("MeldC Fatal Error : m_free_all() Invalid
                pointer  \n");
        m_fatal_error();
        }
    else
        {
        parent->head = (char *) *((int *)
                                  (parent->head) + 1);
        FREE(temp_ptr);
        }
    }
/*
 * If we are not free up the root node, we want to
 * set the next pointer of our parent node to NULL.
 */
if (parent->parent)
    parent->parent->next = NULL;
FREE(parent);

END_CRITICAL_SEC;
```

```
    }
```

2.8.4 M_free

Description of m_free

This function, m_free, will also free up memory. But unlike the m_free_all function, this function will only free the memory pointer that it is given. The function will check to see if the magic number has been corrupted. If it has been, than a core dump will occur.

Code of m_free

```
int m_free(parent,pointer)
M_MEM parent;
char * pointer;
    {
    char  **  temp_ptr;

    BEGIN_CRITICAL_SEC;

    if (parent == NULL)
        return;

    if (parent->magic_number != MAGIC)
        {
        printf("MeldC Fatal Error : m_free_all() Invalid
                parent  \n");
        m_fatal_error();
        }

    temp_ptr = &(parent ->head);

    /*
     * The first 12 bytes of the pointer is reserve for
     * system information.
     */

    pointer = (char *)((int *)pointer - 3);

    if (*((int *)pointer) != MAGIC)
        {
```

```
        printf("MeldC Fatal Error : m_free_all() Invalid
                pointer  \n");
        m_fatal_error();
        }
    if (parent->head != NULL)
        {
        while ((*temp_ptr != NULL) && ((char *)(*temp_ptr)
                != pointer))
            temp_ptr = (char **)((int *)(*temp_ptr) + 1);

        if (*temp_ptr == NULL)
            {
            printf("MeldC Fatal Error : m_free() Invalid
                    pointer  \n");
            m_fatal_error();
            }
        else
            {
            *temp_ptr = (char *)*((int *)(*temp_ptr) + 1);
            FREE(pointer);
            }
        }
    END_CRITICAL_SEC;
    }
```

2.9 Linear.c

Description of linear.c

The file linear.c handles the linearization and unlinearization of frames. The objective of the linearization routines is to convert the variables internal representation to the external ascii form needed to send data over to a remote location or an external environment (i.e. disk). The variables converted will be a frame. This frame will have all of it's information changed by the functions. This will allow us to send frames back and forth across the network to remote locations. Refer to Chapter ??, the Remote Objects chapter, for more information on this exchanging of frames between remote locations.

A warning should be issued at this point. The linearize routine does not linearize the basic pointer type properly. When we pass a integer pointer, we will linearize the pointer. When a char pointer is passed, we will replace with a NULL terminated string. And when an object is passed in, we only linearize the name.

Headers of linear.c

```
#define ADVANCE(s) { s += strlen(s); }
#define ADVANCE_SIZE(s, b) { s += strlen(b); }
#define INC_SIZE(s, b) { s += (b); }
```

2.9.1 Linearize

Description of linearize

The first function, linearize, will take the given parameters from a frame and convert them into linearized form. The function will receive a pointer to the frame, the object parameter list and the message buffer. The function will first make a call to the linearize_size function, discussed next. This call will return the buffer size that is needed to contain the linearized data. The function will then allocate the memory of the size returned from linearized_size. Next, the function will put the method name into the message buffer to be sent across the network. It will also advance the buffer pointer so that the parameters can be put in next. Now the function will start to actually linearize the parameters. This is done by calling the pack function, discussed in Section 2.9.14. This function will do all of the grunt work and return to us the proper form needed. We will also do this for the optional parameter structure, if the structure is not empty. If the structure is not empty, it means that we are trying to pass some object as a parameter to the remote source, see Section 8.5.6 for more on this topic. The pack routine will linearize the object information and put the information into the obj_parm structure. Both values returned will be put into the message buffer to be sent to the remote location. Finally, the function will return the size of the buffer allocated to the caller.

Code of linearize

```
int linearize(frame, bufp,obj_param)
struct _frame *frame;
char **bufp;
struct object_as_param ** obj_param;
{
  struct _para *para;
  struct _msg *msg;
  int i, size;
```

```
  char *buf;

  if(frame ==  0 || frame->$.mp ==  0) return(0);
  if((size = linearize_size(frame)) == 0) return(0);

#ifdef PREEMPTIVE
  BEGIN_CRITICAL_SEC;
#endif

  buf = *bufp = m_malloc(&(_frame->$.add_memory),size);

  /* set method name */
  sprintf(buf, "%s ", frame->$.mp->name);
  ADVANCE(buf);

  for(para = frame->$.mp->para_table; para != 0 &&
      para->name != 0; para++){
    if(strcmp(para->name, "$RET") == 0) {
      sprintf(buf, "%s#", para->decl_type_ptr->c_type_desc);
      ADVANCE(buf);
      break;
    }
  }

#ifdef PREEMPTIVE
  END_CRITICAL_SEC;
#endif

  if(para == 0 || para->name == 0){
    return(0);
  }

#ifdef PREEMPTIVE
  BEGIN_CRITICAL_SEC;
#endif

  /* set parameters  */
  for(para = frame->$.mp->para_table; para != 0 &&
      para->name != 0; para++){
    if(strcmp(para->name, "$RET") != 0)
        {
        pack(para->decl_type_ptr->c_type_desc,
```

```
             para->decl_type_ptr,
             ((char *) frame + (para->offset)),
             &buf, 1,obj_param);
        }
  }
  /* optional para, if any */
  if((msg = frame->$.msg) != 0){
    for(i = 0; i < msg->_num_arg; i++)
        {
        pack(msg->_para[i].type->c_type_desc,
           msg->_para[i].type, (char *)
           ((int)(msg->_para[i].offset) + (int)(msg)),
           &buf, 1,obj_param);
        }
  }
#ifdef PREEMPTIVE
  END_CRITICAL_SEC;
#endif

  return(size);
}
```

2.9.2 Linearize_size

Description of linearize_size

The function linearize_size, will return the buffer size that is needed to contain the linearized data. The function will do this by first receiving the frame to be linearized. Then it will find the method name. Then the function will compute the length which includes the method name and all of the parameters, including the optional ones. It will compute the parameter sizes by making a call to the pack_size function, discussed in Section 2.9.15. The pack_size function will return the size needed to hold all of these parameters. Finally, the function will return the size value computed.

Code of linearize_size

```
int linearize_size(frame)
struct _frame *frame;
{
  struct _para *para;
```

```
  struct _msg *msg;
  int i, size = 0;

  /* method name */

  if(frame->$.mp->name == 0){
    return(0);
  }

#ifdef PREEMPTIVE
  /*
   * If we are not doing any preemptive schedule, the
   * following will be executed atomically, hence there
   * is no need to disable the signal.
   */
  BEGIN_CRITICAL_SEC;
#endif

  size = strlen(frame->$.mp->name) + 1;

  for(para = frame->$.mp->para_table; para != 0 &&
      para->name != 0; para++){
    if(strcmp(para->name, "$RET") == 0) {
      size += strlen(para->decl_type_ptr->c_type_desc) + 2;
      break;
    }
  }
  /* parameters  */
  for(para = frame->$.mp->para_table; para != 0 &&
      para->name != 0; para++){
    if(strcmp(para->name, "$RET") != 0){
      pack_size(para->decl_type_ptr->c_type_desc,
                para->decl_type_ptr,
                ((char *) frame + (para->offset)),
                &size,
                1);
    }
  }
  /* optional para, if any */
  if((msg = frame->$.msg) != 0){
    for(i = 0; i < msg->_num_arg; i++){
      pack_size(msg->_para[i].type->c_type_desc,
```

```
                msg->_para[i].type, (char *)
                ((int)(msg->_para[i].offset) + (int)msg),
                &size,
                1);
    }
  }
#ifdef PREEMPTIVE
  END_CRITICAL_SEC;
#endif

  return(size+1);
}
```

2.9.3 Linearize_obj

Description of linearize_obj

Next we have the linearize_obj function used to linearize an object. The function will be passed the pointer to the object. First it will make a call to the linearize_obj_size function, described next. This function will compute the size buffer needed to store the object in linearized format. Next, the linearize_obj function will allocate the memory needed to store the linearized object by using the returned size value from the linearize_obj_size. The function will then put the object name into the message buffer allocated in linearized form and advance the buffer pointer. Finally, the function will return the message buffer size allocated back to the caller. This function has the primary use of flushing a persistent object to disk. See Chapter 5 for more.

Code of linearize_obj

```
int linearize_obj(obj,para,bufp,obj_param)
struct _para *para;
struct _object * obj;
char **bufp;
struct object_as_param ** obj_param;
{
  struct _msg *msg;
  int i, size;
  char *buf;

  if ( (obj == NULL) || para == NULL) return(0);
```

```
  if((size = linearize_obj_size(obj,para)) == 0)
    return(0);

#ifdef PREEMPTIVE
  BEGIN_CRITICAL_SEC;
#endif

  buf = *bufp = m_malloc(&(_frame->$.add_memory),size);

  /* set obj name */
  sprintf(buf, "%s ", obj->$.name);
  ADVANCE(buf);

  /* set parameters  */
  for(;para != 0 && para->name != 0; para++)
      {
      pack(para->decl_type_ptr->c_type_desc,
             para->decl_type_ptr,
             ((char *) obj + (para->offset)),
             &buf, 1,obj_param);
      }
#ifdef PREEMPTIVE
  END_CRITICAL_SEC;
#endif

  return(size);
}
```

2.9.4 Linearize_obj_size

Description of linearize_obj_size

This function, linearize_obj_size, is used by the linearize_obj function, discussed above, to find the buffer size needed to hold the linearized object. This function will first find the objects name length. This value will be put into the size variable. Then the size of the object itself will be computed. The total size will then be returned.

Code of linearize_obj_size

```
int linearize_obj_size(obj,para)
struct _object * obj;
```

```
struct _para *para;
{
  struct _msg *msg;
  int i, size = 0;

  /* method name */

  if ( (obj == NULL) || (para == NULL))
      return(0);

#ifdef PREEMPTIVE
  /*
   * If we are not doing any preemptive schedule, the
   * following will be executed atomically, hence there
   * is no need to disable the signal.
   */
  BEGIN_CRITICAL_SEC;
#endif

  size = strlen(obj->$.name) + 1;

  /* parameters  */
  for(; para != 0 && para->name != 0; para++)
      {
      pack_size(para->decl_type_ptr->c_type_desc,
                para->decl_type_ptr,
                ((char *) obj + (para->offset)),
                &size,
                1);
      }
#ifdef PREEMPTIVE
  END_CRITICAL_SEC;
#endif

  return(size+1);
}
```

2.9.5 Linearize_msg

Description of linearize_msg

This function, the linearize_msg function is just like the others discussed previously except that it is used to linearize a message only. It will take the method name, frame pointer, the buffer for the message to reside in and the object parameter list as parameters passed from the calling function. The function will first put the method name into the buffer. Next, it will put the parameters into the buffer. Finally, it will pack the message passed to it into the buffer. It will do so by using a for loop and making a call to the pack function, discussed in Section 2.9.14, for each message argument. It will then return the size of the buffer to the caller.

Code of linearize_msg

```
int linearize_msg(method_name,frame,buf,obj_param)
char * method_name;
char *buf;
struct _frame * frame;
struct object_as_param ** obj_param;
{
  int i, size;
  struct _msg *msg;
  struct _para *para;

  msg = frame->$.msg;

  if (method_name == NULL)
      return(0);

#ifdef PREEMPTIVE
  BEGIN_CRITICAL_SEC;
#endif

  /* method name */
  if (msg)
      sprintf(buf, "%s ", msg->_msg_name);
  else
      sprintf(buf, "%s ",method_name);

  ADVANCE(buf);
```

```
  for(para = frame->$.mp->para_table; para != 0 &&
      para->name != 0; para++)
    {
    if(strcmp(para->name, "$RET") == 0)
      {
      size  += strlen(method_name) +
               strlen(para->decl_type_ptr->c_type_desc) + 2;
      sprintf(buf, "%s#", para->decl_type_ptr->c_type_desc);
      ADVANCE(buf);
      break;
      }
    }
  if (msg == NULL)
      return(size);

  /* para */
  for(i = 0; i < msg->_num_arg; i++){
    pack(msg->_para[i].type->c_type_desc,
         msg->_para[i].type,
         (char *) ((int)(msg->_para[i].offset) + (int) msg),
         &buf, 1,obj_param);
  }
#ifdef PREEMPTIVE
  END_CRITICAL_SEC;
#endif

  return(size);
}
```

2.9.6 Linearize_msg_size

Description of linearize_msg_size

This function, the linearize_msg_size function, will be used to find the buffer size needed to contain the parameters and message to be linearized as in the linearize_msg function, discussed above. It will find the size of the parameters passed to be linearized first. Then it will do the same for the message arguments using the pack_size function, discussed in Section 2.9.15. When the function is done, it will return the size of the buffer needed, the value in the size variable.

Code of linearize_msg_size

```
int linearize_msg_size(method_name,frame)
char * method_name;
struct _frame * frame;
{
  int i, size = 0;
  struct _msg *msg;
  struct _para *para;

  if (method_name == NULL)
      return(0);

#ifdef PREEMPTIVE
  BEGIN_CRITICAL_SEC;
#endif

  msg = _frame->$.msg;

  for(para = frame->$.mp->para_table; para != 0 &&
      para->name != 0; para++)
    {
    if(strcmp(para->name, "$RET") == 0)
      {
      size  += strlen(method_name) +
               strlen(para->decl_type_ptr->c_type_desc)
               + 2;
      break;
      }
    }
#ifdef PREEMPTIVE
  END_CRITICAL_SEC;
#endif

  if (msg == NULL)
      return(size);

#ifdef PREEMPTIVE
  BEGIN_CRITICAL_SEC;
#endif

  /* para */
```

```
  for(i = 0; i < msg->_num_arg; i++)
    {
    pack_size(msg->_para[i].type->c_type_desc,
              msg->_para[i].type,
              (char *) ((int)(msg->_para[i].offset)+(int)msg),
              &size,
              1);
    }
#ifdef PREEMPTIVE
  END_CRITICAL_SEC;
#endif

  return(size+2);
}
```

2.9.7 Linearize_return_value

Description of linearize_return_value

This function, linearize_return_value, will be used to put the return value, to be returned to some remote location, into the linearized form. It will send the return value, or values, to the pack function, to put the return value(s) into linearized form. It will store these linearized values into the buffer variable.

Code of linearize_return_value

```
int linearize_return_value(mp, buf, addr,obj_param)
struct _method *mp;
char *buf;
char *addr;
struct object_as_param ** obj_param;
{
  struct _para *para;

  if(mp == 0) return(0);

#ifdef PREEMPTIVE
  BEGIN_CRITICAL_SEC;
#endif
  /* return value */
  for(para = mp->para_table; para != 0 &&
```

```
      para->name != 0; para++){
    if(strcmp(para->name, "$RET") == 0) {
      pack(para->decl_type_ptr->c_type_desc,
           para->decl_type_ptr,
           addr,  &buf, 1,obj_param);
      break;
    }
  }
#ifdef PREEMPTIVE
  END_CRITICAL_SEC;
#endif
}
```

2.9.8 Linearize_return_value_size

Description of linearized_return_value_size

This function, linearized_return_value_size, will simply find the size buffer needed to send the return value(s) back to the remote location. It will simply make a call to the pack_size function for each return value and find the size it needs to store it in linearized form. It will add up all of these values and return the total size to the caller.

Code of linearized_return_value_size

```
int linearize_return_value_size(mp, addr)
struct _method *mp;
char *addr;
{
  struct _para *para;
  int size = 0;

  if(mp == 0) return(0);

#ifdef PREEMPTIVE
  BEGIN_CRITICAL_SEC;
#endif

  /* return value */
  for(para = mp->para_table; para != 0 &&
      para->name != 0; para++){
    if(strcmp(para->name, "$RET") == 0) {
```

```
      pack_size(para->decl_type_ptr->c_type_desc,
                para->decl_type_ptr,
                addr,
                &size,
                1);
      break;
    }
  }
#ifdef PREEMPTIVE
  END_CRITICAL_SEC;
#endif

  return(size + 1);
}
```

2.9.9 Unlinearize

Description of unlinearize

The unlinearize function is exactly what it sounds like. It will take a message that was received from a remote location in linearized form and change this message back. That is, we will change the external ascii form needed for network messages back to the internal representation for the machine to use. It is just as easy as the linearization function, in that it will call unpack function, discussed in Section 2.9.16, to do the actually linearization. It will call unpack on both the parameters and optional parameters, if any exist. In this case, the obj_param structure will not be NULL if an object is being passed to us from some remote source. We will unlinearize this object information in the obj_param structure where it will be used to get the object from the remote location, see Section 8.5.5 for more on this topic. The value will then be stored into a buffer as each piece is unlinearized.

Code of unlinearize

```
int unlinearize(frame, buf,obj_param,where_allocate)
struct _frame *frame;
char *buf;
struct object_as_param * obj_param;
struct _object * where_allocate;
{
  struct _para *para;
  struct _msg *msg;
```

```
  int i, s;

#ifdef PREEMPTIVE
  BEGIN_CRITICAL_SEC;
#endif
  buf = strchr(buf, ' ') + 1; /* method */
  buf = strchr(buf, '#') + 1; /* return type */

  /* para */
  for(para = frame->$.mp->para_table; para != 0 &&
      para->name != 0 && buf;
        para++){
    if(strcmp(para->name, "$RET") == 0) continue;
    unpack(para->decl_type_ptr->c_type_desc,
           para->decl_type_ptr,
           ((char *) frame + (para->offset)),
           &buf,frame,obj_param,where_allocate);
  }
  /* optional para, if any */
  if((msg = frame->$.msg) != 0){
    for(i = 0; i < msg->_num_arg; i++){
      unpack(msg->_para[i].type->c_type_desc,
             msg->_para[i].type,
             (char *)((int)(msg->_para[i].offset) + (int)msg),
             &buf,frame,obj_param,where_allocate);
    }
  }
#ifdef PREEMPTIVE
  END_CRITICAL_SEC;
#endif
}
```

2.9.10 Unlinearize_obj

Description of unlinearize_obj

This function, unlinearize_obj, will unlinearize a message from a remote location that contains an object. We will be passed a pointer to an object, a parameter list, a buffer to put the unlinearized forms in and a pointer to the head parameter. The function will go down the parameter list and unlinearize one at a time with the unpack function, discussed in Section 2.9.16. Each value will be stored into the buffer as each is done.

Code of unlinearize_obj

```
int unlinearize_obj(obj,para, buf,obj_param_head,
                    where_allocate)
struct _object *obj;
struct _para * para;
char *buf;
struct object_as_param ** obj_param_head;
struct _object * where_allocate;
{
  struct _msg *msg;
  int i, s;
  struct object_as_param  static_obj_param;
  struct object_as_param * obj_param;

#ifdef PREEMPTIVE
  BEGIN_CRITICAL_SEC;
#endif
  buf = strchr(buf, ' ') + 1; /* method */
  /*
  buf = strchr(buf, '#') + 1; *//* return type */

  /* para */
  while(para != 0 && para->name != 0 && buf)
      {

      static_obj_param.next = NULL;
      static_obj_param.object_name[0] = 0;

      unpack(para->decl_type_ptr->c_type_desc,
             para->decl_type_ptr,
             ((char *) obj + (para->offset)),
             &buf,_frame,&static_obj_param,where_allocate);

      if (static_obj_param.object_name[0] != 0)
          {
          if(*obj_param_head == NULL)
              {
              *obj_param_head
                  = (struct object_as_param *)
                     m_malloc(&(_frame->$.add_memory),
                     sizeof(struct object_as_param));
```

```
                obj_param = *obj_param_head;
                }
            else
                {
                obj_param->next
                   = (struct object_as_param *)
                      m_malloc(&(_frame->$.add_memory),
                      sizeof(struct object_as_param));

                obj_param = obj_param->next;
                }
            strcpy(obj_param->object_name,
                   static_obj_param.object_name);
            obj_param->pointer = (int) para;
            obj_param->next = NULL;
            }
       para ++;
       }
#ifdef PREEMPTIVE
  END_CRITICAL_SEC;
#endif
}
```

2.9.11 Unlinearize_return_value

Description of unlinearize_return_value

This function, unlinearize_return_value, will take the linearized return value and put it into the unlinearized form. To do so, it will also use the unpack function. The function will call unpack for the return value. The unlinearized return value will than be stored into the buffer.

Code of unlinearize_return_value

```
int unlinearize_return_value(frame, buf,obj_param,
                             where_allocate)
struct _frame *frame;
char *buf;
char * obj_param;
struct _object * where_allocate;
  {
  struct _para *para;
```

```
  if(frame == 0 || frame->$.mp == 0)
      return((enum return_type)NONE);

#ifdef PREEMPTIVE
  BEGIN_CRITICAL_SEC;
#endif

#ifdef DEBUG
  printf("buffer in unlinearize %s \n",buf);
#endif
  /* return value */
  for(para = frame->$.mp->para_table; para != 0 &&
      para->name != 0; para++)
      {
      if(strcmp(para->name, "$RET") == 0)
          {
          unpack(para->decl_type_ptr->c_type_desc,
                 para->decl_type_ptr, (char *)(*((int *)
                 ((char *) frame + (para->offset)))),
                 &buf,frame,obj_param,where_allocate);
          break;
          }
      }
#ifdef PREEMPTIVE
  END_CRITICAL_SEC;
#endif
  }
```

2.9.12 Unlinearize_method_name

Description of unlinearize_method_name

The unlinearize_method_name function will convert the method name from the received linearized form to the needed unlinearized form needed. It will do so by following the same uncoding steps followed by the unpack function. Here, unlike unpack, we know that the name is in a certain form. This therefore makes this process very simple.

Code of unlinearize_method_name

```
int unlinearize_method_name(buf, nbuf, size)
```

```
char *buf, *nbuf;
int size;
{
  char *p;
  int i;
#ifdef PREEMPTIVE
  BEGIN_CRITICAL_SEC;
#endif

  p = strchr(buf, ' ');
  i = p - buf;

  if(i > size)
      i = 0;
  else
      {
      strncpy(nbuf, buf, i);
      nbuf[i] = '\0';
      }
#ifdef PREEMPTIVE
  END_CRITICAL_SEC;
#endif
  return(i);
}
```

2.9.13 Unlinearize_return_type

Description of unlinearize_return_type

The last unlinearize function is the unlinearize_return_type. This function will take the return type in the linearized form and change it to the unlinearized form needed. It will do the conversion using the same steps used by the unpack function, but on a much easier level since the types are very limited.

Code of unlinearize_return_type

```
int unlinearize_return_type(buf, nbuf, size)
char *buf, *nbuf;
int size;
{
  char *p;
```

```
  int i;
#ifdef PREEMPTIVE
  BEGIN_CRITICAL_SEC;
#endif
  buf = strchr(buf, ' ') + 1;  /* method name */
  p = strchr(buf, '#');
  i = p - buf;

  if(i > size)
      {
      i = 0;
      }
  else
      {
      strncpy(nbuf, buf, i);
      nbuf[i] = '\0';
      }
#ifdef PREEMPTIVE
  END_CRITICAL_SEC;
#endif
  return(i);
}
```

2.9.14 Pack

Description of pack

The next function is the pack function, the backbone of the linearization of a message to be sent across a network to some remote location or an external source such as disk. This function will take a variables internal representation and do the actual conversion to ascii. The function does so by getting the variable type information from the passed value in the type_str variable. It will use this variable type in a switch statement. The rest of the variables passed are:

- type_str – This is the parameter type of the parameter passed to pack to be converted to linearized form.
- type_elem – This parameter is used only for the linearization of a structure.
- addr – Contains the value of the variable.
- bufp – This is the resulting buffer.

- top – Used to identify the level of the recursion.
- obj_param – The list of the objects which are passed as parameters. Only the name of the object is linearized.

The function will actually linearize the variables passed to it one at a time and will be called recursively when it attempts to linearize a structure. Note that when we say linearize, it means to put it into the external ascii form.

Code of pack

```
char *pack(type_str, type_elem, addr, bufp, top,obj_param)
char *type_str;  /* type string */
struct _elem_tab *type_elem;  /* struct elem table */
char *addr;
char **bufp;
int top;  /* 1 : top level */
struct object_as_param ** obj_param;
{
  int s, i, k;
  char *naddr, *oaddr;
  struct _elem_tab *elem;  /* struct elem table */
  char *p;


  switch(*type_str){
  case 'i': /* int */
    sprintf(*bufp, "i%d ",  *((int *) addr));
    ADVANCE(*bufp);
    return((char *) (addr + sizeof(int)));
    break;

  case 'v': /* int */
    sprintf(*bufp, "v ");
    ADVANCE(*bufp);
    return((char *) (addr + sizeof(int)));
    break;

  case 'h': /* int */
    sprintf(*bufp, "h%d ",  *((short *) addr));
    ADVANCE(*bufp);
    return((char *) (addr + sizeof(short)));
```

```
    break;

  case 'l': /* int */
    sprintf(*bufp, "l%ld ",  *((long *) addr));
    ADVANCE(*bufp);
    return((char *) (addr + sizeof(long)));
    break;

  case 'f': /* float */
    sprintf(*bufp, "f%f ",  *((float *) addr));
    ADVANCE(*bufp);
    return((char *)(addr + sizeof(float)));
    break;

  case 'd': /* double */
    sprintf(*bufp, "d%lf ",  *((double *) addr));
    ADVANCE(*bufp);
    return((char *)(addr + sizeof(double)));
    break;

  case 'c': /* char */
    sprintf(*bufp, "c%i ",  (int) (*((char *) addr)));
    ADVANCE(*bufp);
    return((char *) (addr + sizeof(char)));
    break;

  case 'n': /* unsigned */
    {
    switch(* (type_str+1))
        {
          case 'i': /* int */
            sprintf(*bufp, "i%d ",  *((int *) addr));
            ADVANCE(*bufp);
            return((char *) (addr + sizeof(int)));
            break;

          case 'h': /* int */
            sprintf(*bufp, "h%d ",  *((short *) addr));
            ADVANCE(*bufp);
            return((char *) (addr + sizeof(short)));
            break;
```

```
        case 'l': /* int */
          sprintf(*bufp, "l%ld ",  *((long *) addr));
          ADVANCE(*bufp);
          return((char *) (addr + sizeof(long)));
          break;

        case 'c': /* char */
          sprintf(*bufp, "c%i ",  (int) (*((char *) addr)));
          ADVANCE(*bufp);
          return((char *) (addr + sizeof(char)));
          break;

      }
  break;
  }

case 's':  /* struct */
  strcpy(*bufp, "s{");
  ADVANCE(*bufp);
  for(elem = type_elem->elem_tab;
      elem != 0 && elem->c_type_desc != 0;
      elem++){
    pack(elem->c_type_desc, elem, addr + elem->offset,
         bufp, 0,obj_param);
  }
  strcpy(*bufp, "} ");
  ADVANCE(*bufp);
  return(addr + type_elem->size);
  break;

case 'u':  /* union */
  strcpy(*bufp, "u{");
  ADVANCE(*bufp);
  elem = type_elem->elem_tab;
  oaddr = addr;
  for(elem = type_elem->elem_tab;
      elem != 0 && elem->c_type_desc != 0;
      elem++){
    naddr = pack(elem->c_type_desc, elem, addr, bufp,
                 0,obj_param);

    if(naddr > oaddr) oaddr = naddr;
```

```
                                /* use largest one */
    }
    strcpy(*bufp, "} ");
    ADVANCE(*bufp);
    return(oaddr);
    break;

  case 'e':  /* enum  */
    sprintf(*bufp, "i%d ",  *((int *) addr));
    ADVANCE(*bufp);
    return((char *) (addr + sizeof(int)));
    break;

  case '*':

    if(*(type_str+1) == 'c'){
        if(*((char **) addr) != 0){
            s = strlen(*((char **) addr));
            sprintf(*bufp, "t%6d %s ", s,
                    *((char **) addr));
            ADVANCE(*bufp);
        }
        else {
            sprintf(*bufp, "t- ");
            ADVANCE(*bufp);
        }
        return((char *) (addr + sizeof(char *)));
    }
    else {

        if (*((char **) addr) != NULL)
            {
            strcpy(*bufp, "*");
            ADVANCE(*bufp);
            pack(type_str+1, type_elem, *((char **) addr),
                 bufp, 0);
            return((char *) (addr + sizeof(char *)));
            }
        else
            {
            strcpy(*bufp,"*i0");
            ADVANCE(*bufp);
```

```
          return((char *) (addr + sizeof(char *)));
          }
    }
    break;


  case '[':
    if(*(type_str+1) == ']'){
      if(*(type_str+2) == 'c'){
        if(*((char **) addr) != 0){
          s = strlen(*((char **) addr));
          sprintf(*bufp, "t%6d %s ", s, *((char **) addr));
          ADVANCE(*bufp);
        }
        else {
          sprintf(*bufp, "t- ");
          ADVANCE(*bufp);
        }
        return((char *) (addr + sizeof(char *)));
      }
      else {
        strcpy(*bufp, "*");
        ADVANCE(*bufp);
        pack(type_str+2, type_elem, *((char **)addr), bufp,
             0,obj_param);
        return((char *) (addr + sizeof(char *)));
      }
    }
    else if(isdigit(*(type_str+1))) {
      k = atoi(type_str+1);  /* ??? */
      p = strchr(type_str, ']') + 1;
      s = p - type_str;
      if(top){
        strcpy(*bufp, "*");
        ADVANCE(*bufp);
        strncpy(*bufp, type_str, s);
        *(*bufp+s) = '\0';
        ADVANCE(*bufp);
        strcpy(*bufp, "{");
        ADVANCE(*bufp);
        naddr = *((char **) addr);
        for(i = 0; i < k ; i++){
```

```
          naddr = pack(p, type_elem, naddr, bufp, 0,obj_param);
        }
        addr = (char *) (addr + sizeof(char *));
      }
      else {
        strncpy(*bufp, type_str, s);
        *(*bufp+s) = '\0';
        ADVANCE(*bufp);
        strcpy(*bufp, "{");
        ADVANCE(*bufp);
        for(i = 0; i < k ; i++){
          addr = pack(p, type_elem, addr, bufp, 0,obj_param);
        }
      }
      strcpy(*bufp, "} ");
      ADVANCE(*bufp);
      return(addr);
    }
    else {
      printf("Unknown size in linearization\n");
      return(addr);
    }
    break;

  case '(':
    return(addr);
    break;

  default:
    /*
     * is this an object ?
     */
    if (type_elem)
        {
        char * actual_pointer;
        char pointer_type;
        char obj_id[MAX_HOSTNAME];

        if ((strncmp(type_str,"object",6) == 0) ||
            (strncmp(type_elem->postfix_type,"class",5) == 0) ||
            (strncmp(type_elem->postfix_type,"* class",7) == 0))
            {
```

```
struct _object * obj;

/*
 * we are passing the object to the remote ...
 */



struct object_as_param * ptr;

ptr = (struct object_as_param *) m_malloc(
                    &(_frame->$.add_memory),
                    sizeof(struct object_as_param));

ptr->next = (struct object_as_param *) NULL;
obj = (struct _object *)*((char **)addr);

if (obj)
    {
    strncpy(ptr->object_name,obj->$.name,
            MAX_HOSTNAME);

    strncpy(ptr->class_name,obj->$.class_ptr->$.name,
            MAX_HOSTNAME);

    sprintf(*bufp, "O{ %s } ", obj->$.name);
    }
else
    {
    /*
     * We are passing the NULL object.
     */
    sprintf(*bufp, "O{ __$__NULL_OBJ } ");
    strncpy(ptr->object_name,"__$__NULL_OBJ",
            MAX_HOSTNAME);
    }


ptr->obj = obj;

if (*obj_param != NULL)
    ptr->next = *obj_param;
```

```
            *obj_param = ptr;

            ADVANCE(*bufp);

            return((char *) (addr + sizeof(char *)));
            }
        }
    return(addr);
    break;
  }
}
```

2.9.15 Pack_size

Description of pack_size

This function, pack_size, will simply return the size of the parameter passed to the linearization function to be linearized. This function will be called from the respective linearization size function. The pack_size function must take certain parameters to accomplish it's goal. Some of these parameters are:

- type_str – This is the type of the parameter passed. It is used to decide how big a parameter is since most types of variables do differ in size.
- addr – This is the value of the parameter.

The function will do a switch on the type_str variable first. Since we now know the type, we can then compute the length according to the type size and the size of the parameter value. This will be done for all types including int, float, double, short, long, char, unsigned int, unsigned short, unsigned long, unsigned char, a struct, a union, and enum.

Code of pack_size

```
char *pack_size(type_str, type_elem, addr, sizep, top)
char *type_str;  /* type string */
struct _elem_tab *type_elem;  /* struct elem table */
char *addr;
int *sizep;
int top;  /* 1 : top level */
{
```

```
int s, i, k;
char *naddr, *oaddr;
struct _elem_tab *elem;  /* struct elem table */
char cbuf[64], *p;

switch(*type_str){
case 'i': /* int */
  sprintf(cbuf, "i%d ",  *((int *) addr));
  ADVANCE_SIZE(*sizep, cbuf);
  return((char *) (addr + sizeof(int)));
  break;

case 'v': /* int */
  sprintf(cbuf, "v ");
  ADVANCE_SIZE(*sizep, cbuf);
  return((char *) (addr + sizeof(int)));
  break;

case 'h': /* int */
  sprintf(cbuf, "h%d ",  *((short *) addr));
  ADVANCE_SIZE(*sizep, cbuf);
  return((char *) (addr + sizeof(short)));
  break;

case 'l': /* int */
  sprintf(cbuf, "l%ld ",  *((long *) addr));
  ADVANCE_SIZE(*sizep, cbuf);
  return((char *) (addr + sizeof(long)));
  break;

case 'f': /* float */
  sprintf(cbuf, "f%f ",  *((float *) addr));
  ADVANCE_SIZE(*sizep, cbuf);
  return((char *)(addr + sizeof(float)));
  break;

case 'd': /* double */
  sprintf(cbuf, "d%lf ",  *((double *) addr));
  ADVANCE_SIZE(*sizep, cbuf);
  return((char *)(addr + sizeof(double)));
  break;
```

```
case 'c': /* char */
  sprintf(cbuf, "c%i ",  (int) (*((char *) addr)));
  ADVANCE_SIZE(*sizep, cbuf);
  return((char *) (addr + sizeof(char)));
  break;

case 'n': /* unsigned */
  {
  switch(* (type_str+1))
      {
        case 'i': /* int */
          sprintf(cbuf, "i%d ",  *((int *) addr));
          ADVANCE_SIZE(*sizep, cbuf);
          return((char *) (addr + sizeof(int)));
          break;

        case 'h': /* int */
          sprintf(cbuf, "h%d ",  *((short *) addr));
          ADVANCE_SIZE(*sizep, cbuf);
          return((char *) (addr + sizeof(short)));
          break;

        case 'l': /* int */
          sprintf(cbuf, "l%ld ",  *((long *) addr));
          ADVANCE_SIZE(*sizep, cbuf);
          return((char *) (addr + sizeof(long)));
          break;

        case 'c': /* char */
          sprintf(cbuf, "c%i ",  (int) (*((char *) addr)));
          ADVANCE_SIZE(*sizep, cbuf);
          return((char *) (addr + sizeof(char)));
          break;
      }
  break;
  }

case 's':  /* struct */
  INC_SIZE(*sizep, 2);
  for(elem = type_elem->elem_tab;
      elem != 0 && elem->c_type_desc != 0;
      elem++){
```

```
    pack_size(elem->c_type_desc, elem, addr + elem->offset,
         sizep, 0);
  }
  INC_SIZE(*sizep, 2);
  return(addr + type_elem->size);
  break;

case 'u':  /* union */
  {
    int size, osize;
    INC_SIZE(*sizep, 2);
    elem = type_elem->elem_tab;
    oaddr = addr;
    osize = 0;
    for(elem = type_elem->elem_tab;
        elem != 0 && elem->c_type_desc != 0;
        elem++){
      size = 0;
      naddr = pack_size(elem->c_type_desc, elem, addr,
                        &size, 0);
      if(naddr > oaddr) oaddr = naddr;
                                /* use largest one */
      if(size > osize) osize = size;

    }
    INC_SIZE(*sizep, (osize + 2));
    return(oaddr);
  }
  break;

case 'e':  /* enum  */
  sprintf(cbuf, "i%d ",  *((int *) addr));
  ADVANCE_SIZE(*sizep, cbuf);
  return((char *) (addr + sizeof(int)));
  break;

case '*':


  if(*(type_str+1) == 'c'){
      if(*((char **) addr) != 0){
          s = strlen(*((char **) addr));
```

```
            INC_SIZE(*sizep, (s+9));
        }
        else {
            INC_SIZE(*sizep, 3);  /* "t- "   */
        }
    return((char *) (addr + sizeof(char *)));
    }
    else
        {
        if (*((char **) addr) != NULL)
            {
            INC_SIZE(*sizep, 1);
            pack_size(type_str+1, type_elem,
                      *((char **) addr), sizep, 0);
            return((char *) (addr + sizeof(char *)));
            }
        else
            {
            INC_SIZE(*sizep, 5);
            return((char *) (addr + sizeof(char *)));
            }
        }
  break;


  case '[':
    if(*(type_str+1) == ']'){
      if(*(type_str+2) == 'c'){
        if(*((char **) addr) != 0){
          s = strlen(*((char **) addr));
          INC_SIZE(*sizep, (s+9));
        }
        else {
          INC_SIZE(*sizep, 3);  /* "t- "   */
        }
        return((char *) (addr + sizeof(char *)));
      }
      else {
        INC_SIZE(*sizep, 1);
        pack_size(type_str+2, type_elem,
                  *((char **)addr), sizep, 0);
        return((char *) (addr + sizeof(char *)));
```

```
      }
    }
    else if(isdigit(*(type_str+1))) {
      k = atoi(type_str+1);  /* ??? */
      p = strchr(type_str, ']') + 1;
      s = p - type_str;
      if(top){
        INC_SIZE(*sizep, (2+s));
        naddr = *((char **) addr);
        for(i = 0; i < k ; i++){
          naddr = pack_size(p, type_elem, naddr, sizep, 0);
        }
        addr = (char *) (addr + sizeof(char *));
      }
      else {
        INC_SIZE(*sizep, (s+1));
        for(i = 0; i < k ; i++){
          addr = pack_size(p, type_elem, addr, sizep, 0);
        }
      }
      INC_SIZE(*sizep, 2);
      return(addr);
    }
    else {
      printf("Unknown size in linearization\n");
      return(addr);
    }
    break;

  case '(':
    return(addr);
    break;

  default:
    /*
     * Is this an object ?
     */
    if (type_elem)
        {
        if ((strncmp(type_str,"object",6) == 0) ||
            (strncmp(type_elem->postfix_type,"class",5) == 0) ||
            (strncmp(type_elem->postfix_type,"* class",7) == 0))
```

```
            {
            struct _object * obj;
            char * pointer;
            char obj_id[MAX_HOSTNAME];

            /*
             * we are passing the object to the remote ...
             * Warning we will not do any check to see whether
             * the object is exported.
             * if the object is not exported by the remote side
             * the error will be return.
             */
            if (*((int **)addr) != NULL)
                {
                obj = (struct _object *)*((char **)addr);

                INC_SIZE(*sizep, 18 + strlen(obj->$.name) +
                         strlen(construct_object_id(obj_id,
                         obj->$.objid)));

                return((char *) (addr + sizeof(char *)));
                }
            else
                {
                /*
                 * If the object is euql to null, we will add
                 * the string O{ __$__NULL_OBJ } into the buffer.
                 */
                INC_SIZE(*sizep,20);
                }
            }
        }
    return(addr);
    break;
  }
}
```

2.9.16 Unpack

Description of unpack

The unpack function is just the reverse of the pack function. Here, the function is called by one of the unlinearize functions since a message was received in linearized form. The function will take the type of the string as a parameter passed to it and use a switch statement on it to find the right steps to use for each type of parameter. The function will return the converted value back to the calling function.

Code of unpack

```
#define TYPE_CONVERT(buftype, bufformat, addrtype) {\
          buftype c;\
          sscanf(*bufp+1, bufformat, &c);\
          *((addrtype *) addr) = c;\
          *bufp = strchr(*bufp, ' ') + 1;\
          return((char *) (addr + sizeof(addrtype)));}


char *unpack(type_str, type_elem, addr, bufp,frame,obj_param,
             where_allocate)
char *type_str;  /* type string */
struct _elem_tab *type_elem;  /* struct elem table */
char *addr;
char **bufp;
struct _frame * frame;
struct object_as_param * obj_param;
struct _object * where_allocate;
{
  int s, i, k;
  char cbuf[64];
  char *naddr, *oaddr;
  struct _elem_tab *elem;  /* struct elem table */
  M_MEM * memory;

  /*
   * M_MEM is define as a pointer to the memory structure.
   * M_MEM * mean the pointer to a pointer.
   */
  if ((where_allocate == NULL) && (_frame))
      memory =  &(_frame->$.add_memory);
```

```
else
    memory =  &(where_allocate->$.add_memory);

if (*type_str == 'n')
    type_str ++;

switch(*type_str){
case 'i': /* int */
  {
    switch(**bufp){
    case 'i' :
      TYPE_CONVERT(int, "%d", int);
      break;
    case 'v': /* int */
      *bufp = strchr(*bufp, ' ') + 1;
      return((char *) (addr + sizeof(int)));
      break;
    case 'c' :
      TYPE_CONVERT(int, "%d", int);
      break;
    case 'f':
      TYPE_CONVERT(float, "%f", int);
      break;
    case 'd':
      TYPE_CONVERT(double, "%lf", int);
      break;
    default:
      return((char *) 0);
      break;
    }
  }
  break;

case 'v': /* int */
  {
    switch(**bufp){
    case 'v': /* int */
      *bufp = strchr(*bufp, ' ') + 1;
      return((char *) (addr + sizeof(int)));
      break;
    default:
      *bufp = strchr(*bufp, ' ') + 1;
```

```
        return((char *) (addr + sizeof(int)));
        break;
      }
    }
    break;

  case 'h': /* int */
    {
      switch(**bufp){
      case 'i' :
        TYPE_CONVERT(int, "%d", short);
        break;
      case 'h' :
        TYPE_CONVERT(int, "%d", short);
        break;
      case 'c' :
        TYPE_CONVERT(int, "%d", short);
        break;
      case 'f':
        TYPE_CONVERT(float, "%f", short);
        break;
      case 'd':
        TYPE_CONVERT(double, "%lf", short);
        break;
      default:
        return((char *) 0);
        break;
      }
    }
    break;


  case 'l': /* int */
    {
      switch(**bufp){
      case 'i' :
        TYPE_CONVERT(int, "%d", long);
        break;
      case 'l' :
        TYPE_CONVERT(int, "%d", long);
        break;
      case 'c' :
```

```
        TYPE_CONVERT(int, "%d", long);
        break;
      case 'f':
        TYPE_CONVERT(float, "%f", long);
        break;
      case 'd':
        TYPE_CONVERT(double, "%lf", long);
        break;
      default:
        return((char *) 0);
        break;
      }
    }
    break;

  case 'f': /* float */
    {
      switch(**bufp){
      case 'i' :
        TYPE_CONVERT(int, "%d", float);
        break;
      case 'c' :
        TYPE_CONVERT(int, "%d", float);
        break;
      case 'f':
        TYPE_CONVERT(float, "%f", float);
        break;
      case 'd':
        TYPE_CONVERT(double, "%lf", float);
        break;
      default:
        return((char *) 0);
        break;
      }
    }
    break;

  case 'd': /* double */
    {
      switch(**bufp){
      case 'i' :
        TYPE_CONVERT(int, "%d", double);
```

```
        break;
      case 'c' :
        TYPE_CONVERT(int, "%d", double);
        break;
      case 'f':
        TYPE_CONVERT(float, "%f", double);
        break;
      case 'd':
        TYPE_CONVERT(double, "%lf", double);
        break;
      default:
        return((char *) 0);
        break;
      }
    }
    break;

  case 'c': /* char */
    {
      switch(**bufp){
      case 'i' :
      case 'c' :
        TYPE_CONVERT(int, "%d", char);
        break;
      default:
        return((char *) 0);
        break;
      }
    }
    break;

  case 's':  /* struct */
    {
      switch(**bufp){
      case 's' :
        *bufp = strchr(*bufp, '{') + 1;
        for(elem = type_elem->elem_tab;
            elem != 0 && elem->c_type_desc != 0;
            elem++){
          unpack(elem->c_type_desc, elem,
                 addr + elem->offset, bufp,
                 frame,obj_param);
```

```
        }
        *bufp = strchr(*bufp, '}') + 1;
        return(addr + type_elem->size);
        break;
      default:
        return((char *) 0);
        break;
      }
    }
    break;

  case 'u':  /* union */
    {
      switch(**bufp){
      case 'u' :
        *bufp = strchr(*bufp, '{') + 1;
        for(elem = type_elem->elem_tab;
            elem != 0 && elem->c_type_desc != 0;
            elem++){
          unpack(elem->c_type_desc, elem, addr,
                 bufp,frame,obj_param);
        }
        *bufp = strchr(*bufp, '}') + 1;
        return(addr + type_elem->size);
        break;
      default:
        return((char *) 0);
        break;
      }
    }
    break;

  case 'e':  /* enum  */
    {
      switch(**bufp){
      case 'i' :
        TYPE_CONVERT(int, "%d", int);
        break;
      case 'c' :
        TYPE_CONVERT(int, "%d", int);
        break;
      default:
```

```
      return((char *) 0);
      break;
    }
  }
  break;

case '*':

  /*
   * For current implementation, we have to use malloc
   * instead of m_malloc.  Since the remote object
   * pass a pointer to us, we have to go ahead and
   * allocate the pointer.  It is the user's responsi-
   * bility to free the pointer.
   */

  if(*(type_str+1) == 'c' && **bufp == 't'){
    int i = 0;
    if(*(*bufp + 1) == '-'){
      *((char **) addr) = (char *) 0;
      *bufp += 3;
      return((char *) (addr + sizeof(char *)));

    }
    else if(sscanf(*bufp+1, "%6d", &i) == 1){
      *((char **) addr) = m_malloc(memory,(i+1));
      strncpy(*((char **) addr), *bufp+8, i);
      *(*((char **) addr) + i) = '\0';
        *bufp += i+8+1;
      return((char *) (addr + sizeof(char *)));
    }
    else {
      return((char *) 0);
    }
  }
  else if(**bufp == '*'){
    *bufp += 1;
    if (strncmp(*bufp,"i0",3) == 0)
        {
        /*
         * It is a NULL Pointer.
         */
```

```
          *((char **)addr) = NULL;
          }
      else
          {
          /*
           * Allocate the memory ...
           */
          if (type_elem->elem_tab == NULL)
              {
              /*
               * The pointer is a simple type.
               * (i.e. int * ..)
               */
              *((char **) addr) =
               m_malloc(memory,type_elem->size);
              }
          else
              {
              int size = 0;
              int counter = 0;

              /*
               * pointer points to a structure.
               */
              while(type_elem->elem_tab[counter].c_type_desc)
                  {
                  size +=
                      type_elem->elem_tab[counter++].size;
                  }
              *((char **) addr) = m_malloc(memory,size);
              }
          unpack(type_str+1, type_elem, *((char **) addr),
                 bufp);
          }
      return((char *) (addr + sizeof(char **)));
    }
    else {
      return((char *) 0);
    }
    break;
```

```
case '[':
  {
    int idx1, idx2;
    char * type;

    idx1 = idx2 = 0;
    switch(**bufp){
    case '[' :
      sscanf(type_str+1, "%d", &idx1);
      sscanf(*bufp+1, "%d", &idx2);
      if(idx1 == idx2 || idx1 == 0 || idx2 == 0)
        {
        type = strchr(type_elem->c_type_desc, ']') + 1;

        for(*bufp = strchr(*bufp,
                           '{') + 1; **bufp != '}';){
          {
          if((addr = unpack(type, type_elem->elem_tab,
                            addr, bufp,frame,
              obj_param)) == 0) return((char *) 0);
          }
        }

        *bufp = strchr(*bufp, ' ') + 1;
        return(addr);
      }
      else {
        return((char *) 0);
      }
      break;
    case '*' :
      return((char *) 0);
      break;
    default :
      return((char *) 0);
      break;
    }
  }
  break;

case '(':
  return(addr);
```

```
    break;

  default:
    {
    char obj_name[MAX_HOSTNAME];

    if ((**bufp == 'i') && (*(*bufp +1) == 0) &&
        (*(*bufp +2) == ' '))
        {
        *((char **) addr) = 0;
        }

    if (**bufp != 'O')
        return(addr);
    /*
     * Important ..
     *
     * There are two different ways to unlinearize
     * the MeldC object.
     *
     * 1) User will pass in the description about
     *    the object (contain in obj_param), the
     *    unlinearize  routine will copy the inform-
     *    ation into the proper location.
     *
     * 2) User will not pass any information, instead
     *    user would like unlinearize routine return
     *    the information about the object in the
     *    linearize string.    In this case object_name
     *    field inside the obj_param will equal to NULL.
     */

    if (obj_param == NULL)
        {
        printf("MeldC Error : unlinearize(),Unable to
                unlinearize the object.");
        return;
        }
    else if (obj_param->object_name[0] == 0)
        {
        sscanf(*bufp+1,"{ %s }",obj_param->object_name);
```

```
        *bufp = strchr(*bufp,'}');

        if (*bufp != NULL)
            *bufp = strchr(*bufp, ' ');

        if (*bufp != NULL)
            *bufp += 1;


        return(addr);
        }


    sscanf(*bufp+1,"{ %s }",obj_name);

    *bufp = strchr(*bufp,'}');

    if (*bufp != NULL)
        *bufp = strchr(*bufp, ' ');

    if (*bufp != NULL)
        *bufp += 1;

    while (obj_param != NULL)
        {
        if (strcmp(obj_name,obj_param->object_name) == 0)
            {
            /*
             * Make sure it is the right type.
             */
            if ((strcmp(type_str,obj_param->class_name) != 0)
                && (strncmp(type_str,"object",6) != 0))
            {
                /*
                 * the class did not match.
                 */
                printf("MeldC Error : unlinearize()\n");
                printf("The objects have incompatible
                        type \n");
                break;
                }
```

```
            /*
             * Assign the object pointer.
             */
#ifdef TRACE
            printf("in the unpack routine. %x %x\n",
                    *((char **) addr), obj_param->obj);
#endif
            *((char **) addr) = (char *) obj_param->obj;
            return((char *) (addr + sizeof(char *)));
            }
        else if (strcmp(obj_name,"__$__NULL_OBJ") == 0)
            {
            /*
             * User passing NULL pointer to the remote
             * machine.
             */
            *((char **) addr) = (char *) NULL;
            return((char *) (addr + sizeof(char *)));
            }
        else
            {
            *((char **) addr) = 0;
            printf("MeldC Error : unlinearize()\n");
            printf("Invalid object name \n");
            break;
            }
        }
    return(addr);
    break;
    }
  }
}
```

2.10 Delayuntil.c

2.10.1 Description of delayuntil.c

The next file to discuss in the Kernel is delayuntil.c. The function will handle a delayuntil/response series of calls. There is more than one way to use the delayuntil/response calls. The first way, is to make the delayuntil call from object A as follows:

delayuntil X

This would make object A go to sleep until a response of X is sent back. So to wake it back up again, a response is needed such as:

respond X

This would say to respond to any object waiting on reponse X and wake it up. In our case, the object is waiting for response X, so it will wake up. If there is more then one delayuntil X calls waiting, we will only wake up the first match that we find. Now the other way to do the delayuntil/respond is more specific. Here, the delayuntil is directed at a specific object. That is, we will call for a delayuntil as above, but we must be woken up with a response from a specific object. This delayuntil is done by object A as follows:

delayuntil X from B

This means that object A does a delayuntil a response X is received from only object B. So lets say object D sends a response of X to object A. Thread A will not wake up from this response since thread A is waiting for a response from B. This response from D will be stored in the hash table and thread D will be put to sleep until a delayuntil matches. There are two ways to wake up A. They are as follows:

respond X to A or
respond X

The first respond would send a response of X to object A from object B. Since A had done a delayuntil X from B, this will again wake up object A. The second call of respond X would also wake up object A under one condition. That condition is that object B made the respond X call. So if the second response was made by object D, there would be no match. This idea of a general call matching a specific call also works for the repond side. That is, if object A issued:

respond X to B

and no delayuntil was issued yet, we could match this call by issuing a delayuntil as follows:

delayuntil X

This would match the above respond call only if the delayuntil was issued by object B. If the delayuntil was issued by object C, then object C would be forced to wait until some respond is issued that matches. Note, that the first call of:

delayuntil X
respond X

You can also use the finally format of:

delayuntil 'X' from B
respond 'X' to A

This call will enclose the actual response in the call itself that is to be received by object A to wake up. The response to receive is any string in the place of X.

Now that the calls have been demonstrated, lets get into the specifics of the calls. When the delayuntil is done by a object, the object's information will be stored in a hash table. For the object to go to sleep though, a check must be done. We will go to the hash slot of the object and check for a response. This is necessary since it is possible for the response to be done before the delayuntil. If there is a response in the hash slot, we will check to see if the response is the required one needed, if any is needed at all. If there is a match, the response will be taken from the slot and the delayuntil will return immediately. In this case, the delayuntil object will never go to sleep. If there is only a specific response to wake up the object from a delayuntil and there is no match, then the delayuntil will be inserted in the slot and the object will go to sleep. In this case, the response will also be left for when it is intended to be used. If there was no response to begin with, the delayuntil object will be put into the hash slot to wait until a response is issued for it.

When a response is given, the hash table will be used to look up that object to respond to. If there is an object that matches the one to wake up and the delayuntil is waiting for this response object, then the object will be enqueued to the ready queue and will eventually be restarted again. If there is a delayuntil there waiting for a response and the delayuntil is not waiting for this response, then the response will also be put into the hash table. It will wait there until a delayuntil is done that matches this response. If there is no delayuntil there, the response will be stored into the hash table at that slot. When a delayuntil is then done that matches, the delayuntil will see the response already there and will just be started immediately, if it matches as discussed above. The functions to do all of this and some other miscellaneous functions are all discussed below in the following subsections.

2.10.2 Headers of delayuntil.c

```
struct _who {
  struct _who *next;
  struct _object *my_obj;
  struct _object *dest_obj;
  struct _frame  *frame;
};

struct _who_queue {
  struct _who *head;
  struct _who **last_next;
};

struct _h_msg_entry
{
    char           *name;
    HEntry         *next;
    struct _who_queue *q;
};

HTable *meldcNewHashTable();
HTable *$sym_msg_rec_table, *$str_msg_rec_table;
HTable *$sym_msg_wait_table, *$str_msg_wait_table;
HEntry *meldcHash();
```

2.10.3 Init_MsgTables

Description of Init_MsgTables

This function will be used later in the MetaClass chapter. It is used to initialize the tables that are used to store any excess delayuntil and response messages. Each delayuntil has to correspond to a response message. Any extra messages will be stored in this table, waiting for it's counterpart.

Code of Init_MsgTables

```
Init_MsgTables(){
  $sym_msg_rec_table = meldcNewHashTable(STRINGHASH,
                                         DEFAULTSIZE,
                                         20);
  $str_msg_rec_table = meldcNewHashTable(STRINGHASH,
                                         DEFAULTSIZE,
                                         20);
  $sym_msg_wait_table = meldcNewHashTable(STRINGHASH,
                                          DEFAULTSIZE,
                                          20);
  $str_msg_wait_table = meldcNewHashTable(STRINGHASH,
                                          DEFAULTSIZE,
                                          20);
}
```

2.10.4 New_who

Description of new_who

This function is used to fill in the destination object (my_obj) frame and next pointer values to be stored into the hash table when a new insertHash call is made, discussed in Section 2.12. The function will return the structure that was filled with the values.

Code of new_who

```
static struct _who *new_who(dest_obj, my_obj,frame)
struct _object *dest_obj;
struct _object *my_obj;
struct _frame *frame;
{
  struct _who *w;
  SALLOCATE(w, struct _who);
```

```
  w->dest_obj = dest_obj;
  w->my_obj = my_obj;
  w->frame = frame;
  w->next = (struct _who *) NULL;
  return(w);
}
```

2.10.5 Remove_who

Description of remove_who

The remove_who function will be used to check and see if an entry can be removed from the hash table. This can be done if either the delayuntil found a response match or vise verse then the entry can be removed from the hash table and the entry slot in the table will be set to FREE. This can be done with a deleteHash call. If the there is no match in the table for either a response or a delayuntil call in the table, then the value of NULL will be put into the table.

Code of remove_who

```
static struct _who *remove_who(tab, msg, obj)
HTable *tab;
char *msg;
struct _object *obj;
{
  struct _who **wp, *w = NULL;
  struct _h_msg_entry *p;

  if((p = (struct _h_msg_entry *) meldcHash(tab, msg)) == NULL
      || p->q == NULL)
      {
      w = NULL;
      }
  else
      {
      for(wp = &(p->q->head); *wp != NULL; ]
          wp = &((*wp)->next))
          {
          if(obj == 0 || obj == (*wp)->my_obj)
              {
              w = *wp;
```

```
            *wp = w->next;
            if(*wp == NULL) p->q->last_next = wp;
            if(p->q->head == NULL)
                {
                FREE(p->q);
                deleteHash(tab, msg);
                /* remove hash entry and who_queue  */
                }
            break;
            }
        }
  }
  return((struct _who *) w);
  }
```

2.10.6 Insert_who

Description of insert_who

The insert_who function will insert someone into the hash table at some slot using the hash function. It will take the entry structure passed to it, w, and check the table. If the slot is empty, then the entry structure will simply be put into the table at that slot. If there is an entry there, then we must put the entry at the end of a linked list at the slot. An entry is put into the table at some slot using the insertHash call.

Code of insert_who

```
static struct _who *insert_who(tab, msg, w)
HTable *tab;
char *msg;
struct _who *w;
{
  struct _h_msg_entry *p;

  if((p = (struct _h_msg_entry *)
                     meldcHash(tab, msg)) == NULL){
    p = (struct _h_msg_entry *) insertHash(tab, msg);
    p->q = (struct _who_queue *) NULL;
  }

  if(p->q == NULL){
```

```
    SALLOCATE(p->q, struct _who_queue);
    p->q->head = (struct _who *) NULL;
    p->q->last_next = &(p->q->head);
  }

  *(p->q->last_next) = w;
  p->q->last_next = &(w->next);
  w->next = (struct _who *) NULL;
  return(w);
}
```

2.10.7 _delayuntil

Description of _delayuntil

The _delayuntil function will handle the delayuntil call. It will be passed a message, an object that the delayuntil was from and a message type. The message type will be either string or symbol type. The function will check the slot of the table using the remove_who call. If the call returned non-NULL, then there was a match. Since we have a delayuntil, then a response was already waiting. This means we do not have to put the thread asleep. In this case, we will just set the frame status to S_READY. This will allow the thread to restart immediately by enqueuing the thread to run again.

Now, if the response is not there or there was no match, then a NULL was returned by the remove_who function. This means we must put the thread into the hash table to wait for the proper response. This is done with the insert_who function, discussed in Section 2.10.6. Then we must set the frame status to S_WAIT which will prevent the thread from running.

Code of _delayuntil

```
enum function_states _delayuntil(msg, from_obj, msg_type)
char *msg;
struct _object *from_obj;
enum msg_types msg_type;
  {
  struct _who *w;
  HTable *msg_rec_table;
  HTable *msg_wait_table;
  enum function_states return_value;

  BEGIN_CRITICAL_SEC;
```

```
  if(msg_type == STR_MSG)
      {
      msg_rec_table = $str_msg_rec_table;
      msg_wait_table = $str_msg_wait_table;
      }
  else
      {
      msg_rec_table = $sym_msg_rec_table;
      msg_wait_table = $sym_msg_wait_table;
      }

  if((w = remove_who(msg_rec_table, msg,
      from_obj)) != NULL)
      {
      /*
       * There is already a respond message.
       */

      FREE(w);

#ifdef DEBUG
      printf("Delayuntil request with a response
              already waiting.\n");
#endif

      return_value = S_READY;
      }
  else
      {

      /*
       * Insert the delayuntil message into the
       * waiting queue.
       */

#ifdef DEBUG
      printf("Delayuntil request %s\n",msg);
#endif

      insert_who(msg_wait_table, msg,
                 new_who(from_obj,_obj,_frame));
```

```
        return_value = S_WAIT;
        }
    END_CRITICAL_SEC;
    return(return_value);
    }
```

2.10.8 _respond

Description of _respond

The _respond function will handle a respond call by some thread. It will take a message, an object that the respond is to and a message type as parameters passed to it. It will call the remove_who function, like the delayuntil function. If the call returns non-NULL, then we found a thread that did a delayuntil that matched our respond. In this case, we do not want the delayuntil thread to wait anymore, so we will change it's frame status to S_READY and enqueue it to allow it to run again. We also set the frame status of the thread that did the respond to S_READY to allow it to also run after being enqueued again.

If the remove_who returned NULL, then there was either no delayuntil in the hash table slot or there was no match of the respond. In either case, it means we must wait for a delayuntil call to be made that matches this respond. So we will put the respond into the hash table at the slot with a insert_who call, discussed in Section 2.10.6.

Code of _respond

```
enum function_states _respond(msg, to_obj, msg_type)
char *msg;
struct _object *to_obj;
enum msg_types msg_type;
{
    struct _who *w;
    HTable *msg_rec_table;
    HTable *msg_wait_table;
    enum function_states return_value;

    BEGIN_CRITICAL_SEC;

    if(msg_type == STR_MSG)
        {
        msg_rec_table = $str_msg_rec_table;
```

```
        msg_wait_table = $str_msg_wait_table;
        }
    else
        {
        msg_rec_table = $sym_msg_rec_table;
        msg_wait_table = $sym_msg_wait_table;
        }

    if((w = remove_who(msg_wait_table, msg,
                       to_obj)) != NULL)
        {
        /*
         * there is a message already waiting.
         */
        if (w->frame)
            {
            w->frame->$.next = (struct _frame *) 0;
            w->frame->$.state = S_READY;
            enqueue(w->frame);
            }
        else
            {
            printf("MeldC Fatal Error: Unable to
                    response.\n");
            }
        FREE(w);

#ifdef DEBUG
        printf("Responding to a delayuntil \n");
#endif

        return_value = S_READY;
        }
    else
        {
        /*
         * Queue the response message into the queue.
         */
        insert_who(msg_rec_table, msg, new_who(to_obj,
                                               _obj,
                                               _frame));
```

```
#ifdef DEBUG
      printf("No delayuntil responding to, queue it
              up %s\n",msg);
#endif
      return_value = S_READY;
                       /* Yes, S_READY, not S_WAIT */
      }
  END_CRITICAL_SEC;
  return(return_value);
  }
```

2.11 Event.c

2.11.1 Description of event.c

This file contains some of the Kernel functions that are used to enqueue and dequeue a frame to/from the ready queue and fork off a new thread.

2.11.2 Headers of event.c

```
#define ENULL (struct _frame *) 0

#ifdef STACK_BASE
extern char **sp;
#endif

struct event_queue {
  struct _frame *front;
  struct _frame *rear;
};

/* q[6] has the highest priority  */
/* q[0] has the lowest priority   */
static struct event_queue q[7] = {
  {ENULL, ENULL},{ENULL, ENULL},{ENULL, ENULL},
  {ENULL, ENULL},{ENULL, ENULL},{ENULL, ENULL},
  {ENULL, ENULL}};

/*
 * initialize the event queue.
 */
```

```
empty(){
  if(q[0].front == NULL &&
     q[1].front == NULL &&
     q[2].front == NULL &&
     q[3].front == NULL &&
     q[4].front == NULL &&
     q[5].front == NULL &&
     q[6].front == NULL) return(1);
  else return(0);
}
```

2.11.3 Dequeue

Description of dequeue

This function is used by the Kernel to remove a frame from the ready queue. It will use the frame pointer passed in to do so.

Code of dequeue

```
dequeue(fpp) /* consider time-efficiency */
struct _frame **fpp;
{
  register int priority;

#ifdef DEBUG
  print_q6("DEQUEUE");
#endif DEBUG

  BEGIN_CRITICAL_SEC;   /* block SIGIO */
  for(priority = 6; priority >= 0 ; priority--){
    if(q[priority].front != NULL){
      *fpp = q[priority].front;
      q[priority].front = (*fpp)->$.next;
      if (q[priority].front == NULL) q[priority].rear = NULL;
      (*fpp)->$.next = (struct _frame *) 0;
      break;
    }
  }
  END_CRITICAL_SEC;
  return(priority);  /* return priority, -1 indicates empty  */
}
```

2.11.4 Enqueue

Description of enqueue

This function is used by the Kernel to put a new frame onto the ready queue to allow the frame to be executed. This function will use the frame pointer, fp, passed in to do so.

Code of enqueue

```
enqueue(fp)
register struct _frame *fp;
{
register int priority;
static int sorting = 1;

  BEGIN_CRITICAL_SEC;

  if (fp != NULL)
      {
      if (fp->$.priority != NO_PRIORITY)
          {
          priority = fp->$.priority;
          }
      else if (fp->$.sync_mode == SYNC)
          {
          /*
           * If the request is a synchronous request, we will
           * schedule the frame using its parent frame priority.
           * else we will use the object's priority.
           */

          if ((fp->$.caller != NULL))
              {
              priority = fp->$.caller->$.priority;
              fp->$.priority = priority;
              }
          else
              {
              if (fp->$.priority == NO_PRIORITY)
                  priority = fp->$.priority = OBJECT_PRIORITY;
              }
          }
```

```
      else
          {
          /*
           * ASYN call.
           */
          if (fp->$.obj)
              {
              priority = fp->$.obj->$.object_priority;
              fp->$.priority = priority;
              }
          else
              {
              if (fp->$.priority == NO_PRIORITY)
                  priority = fp->$.priority = OBJECT_PRIORITY;
              }
          }



      (q[priority].rear == NULL) ? (q[priority].front = fp) :
      (q[priority].rear->$.next = fp);
      q[priority].rear = fp;
      }

  END_CRITICAL_SEC;
#ifdef DEBUG
  print_q6("ENQUEUE");
#endif DEBUG
  return;
}
```

2.11.5 Mfork

Description of mfork

This function will fork off a new frame, which implies the creation of a new frame, and then will store the new frame into the ready queue so that the new frame will be allowed to execute. This function is used by meldC.out.c, the MeldC startup program.

It will first allocate the memory for the frame. Then fields of the frame will be filled in. These include the func, the obj, the state which equals S_NEW, the caller, the add_memory, the save_registers and the next fields of the frame structure.

Code of mfork

```
#ifdef CIRCULAR_INTERLEAVING
#define DEFAULT_APPROACH
#endif

#ifdef FRAME_ALLOCATION_HEAP_BASE
#define DEFAULT_APPROACH
#endif
/*
 * forking off a new thread, it is used by meldC.out.c.
 * (The Meldc Starup program.
 */

mfork(obj, func, size, priority)
struct _object *obj;
int (*func)();
int size;
int priority;
{
  struct _frame *fp, *alloc_frame();

#ifdef DEFAULT_APPROACH
  fp = alloc_frame((size < sizeof(struct _frame)) ?
       sizeof(struct _frame):size);
#else /* DEFAULT_APPROACH */

#ifdef STACK_BASE
      fp = (struct _frame *)
           a_new_thread((size < sizeof(struct _frame))
           ? sizeof(struct _frame):size);
#endif
#ifdef STACK_BASE_SLOW
      fp = (struct _frame *)
           a_new_thread((size < sizeof(struct _frame))
           ? sizeof(struct _frame):size);
#endif
#endif /* End of DEFAULT_APPROACH */

  fp->$.func = func;
```

```
  fp->$.obj = obj;
  fp->$.state = S_NEW;
  fp->$.caller = 0;
  fp->$.priority = priority;
  fp->$.add_memory = NULL;
  fp->$.save_registers[0] = 0;

  fp->$.next = (struct _frame *) 0;

  BEGIN_CRITICAL_SEC;
  (q[priority].rear == NULL) ? (q[priority].front = fp)
  : (q[priority].rear->$.next = fp);
  q[priority].rear = fp;
  END_CRITICAL_SEC;
#ifdef DEBUG
  print_q6("ENQUEUE-M");
#endif DEBUG
}
```

2.12 Hash.c

2.12.1 Description of hash.c

This file is similar to the description given in the hash.m file. This file basically contains all of the functions needed to do hash functions needed by the Kernel. These functions include:

- insertHash – Used to insert an entry into the hash table.
- deleteHash – Used to remove an entry from the hash table.

These two are the most used. For more detail on these two functions and the others in this file, see Section 8.12.4.

2.12.2 Code of hash.c

```
/*
 * type       == The type of the hash table.
 *
 * table_size == the size of the initial table.
 *
 * increment  == When the elements in the table exceed the
```

```
 *              upper bound , the size of this hash table
 *              will be increased by "increment" amount.
 */

HTable *
meldcNewHashTable(type, increment, table_size)
int type;
int increment;
int table_size;
{
    HTable *new_table;

    new_table = (HTable *) MALLOC(sizeof(HTable));
    new_table->type = type;
    new_table->increment = increment;
    new_table->number_of_entry = 0;
    new_table->table_size = table_size ;

    if (table_size == 0)
        new_table->table = NULL;
    else if (table_size > 0)
        {
        HTable * temp_ptr;
        int counter = table_size - 1;

        new_table->table = (HEntry **) MALLOC(sizeof(HEntry *)
                                              * table_size);

        while (counter >= 0)
            {
            new_table->table[counter] = (HEntry *)0;
            counter --;
            }
        }
    else printf("Error in table_size [%d] !\n", table_size);
    return new_table;
}

/*********************************************************/
/*                                                       */
/*********************************************************/
```

```
HEntry *
meldcHash(h_table, h_name)
HTable *h_table;
char   *h_name;
{
    HEntry *result;
    int key ;

    BEGIN_CRITICAL_SEC;

    key = getHashKey(h_table->type, h_name,
                     h_table->table_size);

    if(key == -1 || h_table->table[key] == NULL)
        {
        result = (HEntry *)NULL;
        }
    else
        {
        /* printf("Key is %d\n", key); */

        result = (h_table->table)[key];
        if (h_table->type == STRINGHASH)
            while(strcmp(result->u.name, h_name) != 0)
                {
                if (result->next == NULL)
                    {
                    result = (HEntry *) NULL;
                    break;
                    }
                else result = result->next;
                }
        else
            while(memcmp(result->u.objid, h_name,
                         sizeof(ObjID)) != 0)
                {
                if (result->next == NULL)
                    {
                    result = (HEntry *) NULL;
                    break;
                    }
                else result = result->next;
```

```
                }
        }
    END_CRITICAL_SEC;
    return result;
}

/*************************************************************/
/*                                                           */
/*************************************************************/

HEntry *
insertHash(h_table, h_name)
HTable *h_table;
char   *h_name;
{
    int key;
    HEntry *entry;

    BEGIN_CRITICAL_SEC;
    if((HASHRATE * h_table->table_size) <=
                   h_table->number_of_entry)
        reHash(h_table, POSITIVE);

    if(meldcHash(h_table, h_name) != NULL)
        {
        entry = (HEntry *) NULL;
        }
    else
        {
        entry = myInsert(h_table, h_name);
        h_table->number_of_entry++;
        }
    END_CRITICAL_SEC;
    return entry;
}

/*************************************************************/
/*                                                           */
/*************************************************************/

int
deleteHash(h_table, h_name)
```

```
HTable *h_table;
char   *h_name;
    {
    HEntry *previous, *entry;
    int key ;
    int result = -1;

    BEGIN_CRITICAL_SEC;
    key = getHashKey(h_table->type, h_name,
                     h_table->table_size);

    if ((key == -1) || (h_table->table[key] == NULL))
        {
        result =  FALSE;
        }
    else
        {
        entry = h_table->table[key];
        if (h_table->type == STRINGHASH)
            {
            if(strcmp(entry->u.name, h_name) == 0)
                {
                h_table->table[key] = entry->next;
                result =  myDelete(h_table, entry);
                }
            else
                {
                while(strcmp(entry->u.name, h_name) != 0)
                    {
                    if (entry->next == NULL)
                        {
                        result =  FALSE;
                        break;
                        }
                    else
                        {
                        previous = entry;
                        entry = entry->next;
                        }
                    }
                }
            }
```

```
        else
            {
                if(memcmp(entry->u.objid, h_name,
                          sizeof(ObjID)) == 0)
                {

                h_table->table[key] = entry->next;
                result =  myDelete(h_table, entry);
                }
            else
                {
                while(memcmp(entry->u.objid, h_name,
                             sizeof(ObjID)) != 0)
                    {
                    if (entry->next == NULL)
                        {
                        result = FALSE;
                        break;
                        }
                    else
                        {
                        previous = entry;
                        entry = entry->next;
                        }
                    }
                }
            }
        if (result == -1)
            {
            previous->next = entry->next;
            result =  myDelete(h_table, entry);
            }
        }
    END_CRITICAL_SEC;
    return result;
    }

/************************************************************/
/*                                                          */
/************************************************************/

static int
```

```
getHashKey(type, h_something, h_size)
int  type;
int  *h_something;
int  h_size;
{
    if (h_size > MAXSIZE || h_size <= 0)
    {
        fprintf(stderr, "Invalid Hash Table Size [ %d ]\n",
                h_size);
        return -1;
    }

    if(type == STRINGHASH)
        return getStringHashKey((char *) h_something,
                                h_size);
    else
        return getBinaryHashKey(h_something, h_size);
}

static int
getStringHashKey(h_str, h_size)
char *h_str;
int h_size;
{
    int look, cut, h_code;

    h_code = getStringIndex(*h_str);
    if(h_code == -1) return -1;

    if (h_size <= C_ALPHA_SIZE)
    {
        cut = C_ALPHA_SIZE / h_size;
        return normalized(h_code / cut, h_size);
    }

    look = h_size / C_ALPHA_SIZE + 1;
    return
        normalized
        (h_code + getStringHashKey(h_str + 1, look) *
         C_ALPHA_SIZE, h_size);
}
```

```
static int
getBinaryHashKey(h_number, h_size)
int *h_number;
int h_size;
{
    int look, cut, h_code;

    h_code = getBinaryIndex(*h_number);

    if (h_size <= BINARY_SIZE)
    {
        cut = BINARY_SIZE / h_size;
        return normalized(h_code / cut, h_size);
    }

    look = h_size / BINARY_SIZE + 1;
    return
        normalized
        (h_code + getBinaryHashKey(h_number + 1, look) *
         BINARY_SIZE, h_size);
}

/*******************************************************/
/*                                                     */
/*******************************************************/

static int
getStringIndex(my_c)
char my_c;
{
    if(my_c == '$' || my_c == '\0' || my_c == '#') return 0;
    if(my_c <= '9' && my_c >= '0') return my_c - '0' + 1;
    if(my_c <= 'Z' && my_c >= 'A') return my_c - 'A' + 11;
    if(my_c == '_') return 37;
    if(my_c <= 'z' && my_c >= 'a') return my_c - 'a' + 38;
    fprintf(stderr, "Invalid String [ %c ]\n", my_c);
    return -1;
}

static int
getBinaryIndex(my_i)
int my_i;
```

```
{
    return (my_i % BINARY_SIZE);
}

/*****************************************************/
/*                                                   */
/*****************************************************/

static HEntry *
myInsert(h_table, h_name)
HTable *h_table;
char *h_name;
{
    int key;
    HEntry *entry;

    key = getHashKey(h_table->type, h_name,
                     h_table->table_size);
    entry = (HEntry *) MALLOC(sizeof(HEntry));
    entry->next = h_table->table[key];
    if (h_table->type == STRINGHASH) entry->u.name =
        STRDUP(h_name);
    else
    {
        ObjID *oidp;
        oidp = (ObjID *) MALLOC(sizeof(ObjID));
        oidp = (ObjID *) memcpy(oidp, h_name,
                         sizeof(ObjID) / sizeof(char));
        entry->u.objid = oidp;
    }

    h_table->table[key] = entry;
    return entry;
}

static int
myDelete(h_table, entry)
HTable *h_table;
HEntry *entry;
{
    if (entry->u.name)
        FREE(entry->u.name);
```

```
    FREE(entry);
    h_table->number_of_entry--;
    if((HASHRATE * (h_table->table_size -
                    h_table->increment))
       > h_table->number_of_entry)
        reHash(h_table, NEGATIVE);
    return TRUE;
}

/************************************************************/
/*                                                          */
/************************************************************/

static int
normalized(h_code, h_size)
int h_code;
int h_size;
{
    while(h_code > (h_size - 1)) h_code -= h_size;
    return h_code;
}

/************************************************************/
/*                                                          */
/************************************************************/

static void
reHash(h_table, my_sign)
HTable *h_table;
int my_sign;
{
    HEntry          **new_hash_table, **old_hash_table,
                    *entry;
    int             new_table_size, old_table_size, i;

    new_table_size = h_table->table_size +
                     h_table->increment * my_sign;
    new_hash_table = (HEntry **) MALLOC(sizeof(HEntry *) *
                                        new_table_size);
    for (i = 0; i < new_table_size; i++)
        new_hash_table[i] = NULL;
```

```
    old_table_size = h_table->table_size;
    old_hash_table = h_table->table;
    h_table->table_size = new_table_size;
    h_table->table = new_hash_table;

    for (i = 0; i < old_table_size; i++)
    {
    HEntry * temp_entry;

        entry = old_hash_table[i];
        while (entry != NULL)
        {
            if((temp_entry = myInsert(h_table,
                                    entry->u.name))
                            == (HEntry *) NULL)
                printf("Error in ReHash!!!\n");
            else
                temp_entry->ptr = entry->ptr;
            entry = entry->next;
        }
        FREE(old_hash_table[i]);
    }
    FREE(old_hash_table);
}
```

2.13 Saveframe.c

This file contains two functions of interest, save_frame and restore_frame. They will be used to assist in doing a stack switch. That is, when a frame is to give up the C stack, the current frame state must be saved from the C stack and the new frame must be copied to the C stack.

2.13.1 Save_frame

Description of save_frame

This function will be concerned with saving the state of the frame on the C stack before the new frame is copied to the stack. The function will be passed the upper bound pointer, top of the frame, and the lower bound pointer, the bottom of the frame. The memory needed will be allocated using a call to malloc of size, save_frame_size. This value had been saved into the frame structure. The address of the allocated memory for the frame to be saved in will be put into the save_stack_frame variable of the frame structure. The frame will then be copied from the address of the pointers to the allocated memory.

Code of save_frame

```
save_frame(frame,func_low_sp,func_up_sp)
struct _frame *frame;
int *func_low_sp;
int *func_up_sp;
    {
    int * pointer;

     /*
      * For sun4
      */

    BEGIN_CRITICAL_SEC;

#ifdef sun4
    /*
     * In sun4 the unit of save_frame size is 4 bytes.
     */
    pointer= (int *)malloc(frame->$.save_frame_size *
             (sizeof(int)));
#elif mips
    pointer= (int *)malloc(frame->$.save_frame_size) ;
#endif

    frame->$.save_stack_frame[0] = pointer;
    for(; func_up_sp < func_low_sp ;)
        {
        *pointer++ = *func_up_sp++;
        }

    END_CRITICAL_SEC;
    }
```

2.13.2 Restore_frame

Description of restore_frame

This function will simply restore the frame back to the C stack to allow it to be executed again. The upper and lower pointers will be passed in again. The pointer to the frame beginning is saved in the save_stack_frame variable. A simply for loop will copy the frame from the saved_stack_frame address to the address passed in. Then the memory will be freed that held the saved frame.

Code of restore_frame

```
restore_frame(frame,func_low_sp,func_up_sp)
struct _frame *frame;
int *func_low_sp;
int *func_up_sp;
    {
    int * pointer;

     /*
      * For sun4
      */
    BEGIN_CRITICAL_SEC;
    pointer = frame->$.save_stack_frame[0];

    for(; func_up_sp < func_low_sp ;)
        {
        *func_up_sp++ = *pointer++;
        }

    free((frame->$.save_stack_frame[0]));
    END_CRITICAL_SEC;
    }
```

2.14 Kernel_interface.c

2.14.1 Description of kernel_interface.c

This section contains functions that are used by the MeldC upper layer to obtain and modify the internal data contained in the Kernel layer.

Headers of kernel_interface.c

```
#define MAX_ID_SIZE 100

#ifdef DEBUG
int time_stamp = 0;
#endif

typedef struct _object * object;
```

2.14.2 Set_current_obj

Description of set_current_obj

This function will simply set the global object, _obj, with the new given object value, obj, passed to the function. Then the function will return the old object value, _obj.

Code of set_current_obj

```
struct _object * set_current_obj(obj)
struct _object *obj;
    {
    struct _object * tmp_obj;
    tmp_obj = _obj;
    _obj = obj;
    return(tmp_obj);
    }
```

2.14.3 Reset_obj_and_frame

Description of reset_obj_and_frame

The function will first create a new frame using the NEW_FRAME macro, see Section 3.3. Then we will set the caller to the MeldC function, which is the value of the passed parameter global_frame. The thread id will be inherited from the MeldC function. The frame state will be set to S_READY. This frame will then become the global frame. The object passed in, obj, will then become the global object. This function will be used by a shadow object to restore it's frame.

Code of reset_obj_and_frame

```
#ifndef STACK_BASE
```

```
void reset_obj_and_frame(global_frame,global_obj,obj,size,
                         function_addr)
struct _object *obj;
int size;
int (* function_addr) ();
struct _frame ** global_frame;
struct _object ** global_obj;
    {
    struct _frame * fp, *old_fp;

    /*
     * Creating a new frame
     */

    NEW_FRAME(fp,size,obj,function_addr,NULL,NULL,ASYNC);

    fp->$.caller = *global_frame;

    if (*global_frame)
        {
        fp->$.priority = (*global_frame)->$.priority;
        fp->$.threadid = (*global_frame)->$.threadid;
        fp->$.sync_mode = (*global_frame)->$.sync_mode;
        }

    fp->$.state = S_READY;


    fp->$.size = 0;

    if (*global_frame)
        memcpy(fp->$.global_variables,
               (*global_frame)->$.global_variables,
               (sizeof(int) * 8));

    *global_frame = fp;

    if ((global_obj != NULL) && (*(global_obj) != NULL))
        *global_obj = obj;

    return;
```

```
    }


#else /* If we define STACK_BASE */


void reset_obj_and_frame(global_frame,global_obj,obj,
                         size,function_addr)
struct _object *obj;
int size;
int (* function_addr) ();
struct _frame ** global_frame;
struct _object ** global_obj;
    {
    struct _frame * fp, *old_fp;

    /*
     * Creating a new frame
     */

    NEW_DYNAMIC_FRAME(fp,size,obj,function_addr,NULL,
                      NULL,ASYNC);

    fp->$.caller = *global_frame;

    if (*global_frame)
        {
        fp->$.priority = (*global_frame)->$.priority;
        fp->$.threadid = (*global_frame)->$.threadid;
        }

    fp->$.state = S_READY;

    fp->$.size = 0;

    *global_frame = fp;

    if ((global_obj != NULL) && (*(global_obj) != NULL))
        *global_obj = obj;

    return;
    }
```

```
#endif
```

2.14.4 Restore_old_frame

Description of restore_old_frame

The function will be passed a pointer to a frame, fp. The global frame in _frame will be freed and then the new _frame will be set to the value of fp.

Code of restore_old_frame

```
#ifndef STACK_BASE


void restore_old_frame(fp)
struct _frame *fp;
    {

#ifdef STACK_BASE_SLOW
    _frame->$.caller = (struct _frame *)NULL;
    free_stack_frame(_frame);
#else
    free_frame(_frame);
#endif

    _frame = fp;
    _obj = _frame->$.obj;
    }


#else /* If we define STACK_BASE */


void restore_old_frame(fp)
struct _frame *fp;
    {
    FREE(_frame);

    _frame = fp;
```

```
    _obj = _frame->$.obj;
    }


#endif
```

2.14.5 Construct_thread_id

Description of construct_thread_id

This function will take the buffer and threadid values as parameters. Then it will put the objid.counter, objid.pid and objid.hostid values into the buffer. Then the buffer will be returned.

Code of construct_thread_id

```
char * construct_thread_id(buffer,threadid,is_sync)
struct _threadid threadid;
char buffer[MAX_ID_SIZE];
int is_sync;
    {
    sprintf(buffer,"%d_%d_%d_%d_%d_%d_%d",
                threadid.objid.counter,
                threadid.objid.time.tv_sec,
                threadid.objid.time.tv_usec,
                threadid.objid.pid,
                threadid.objid.hostid,
                threadid.counter,is_sync);

    return(buffer);
    }
```

2.14.6 Deconstruct_thread_id

Code of deconstruct_thread_id

```
struct _threadid * deconstruct_thread_id(buffer,threadid)
struct _threadid * threadid;
char buffer[MAX_ID_SIZE];

    {
    int sync_mode;
```

```
    sscanf(buffer,"%d_%d_%d_%d_%d_%d_%d",
            &(threadid->objid.counter),
            &(threadid->objid.time.tv_sec),
            &(threadid->objid.time.tv_usec),
            &(threadid->objid.pid),
            &(threadid->objid.hostid),
            &(threadid->counter),&sync_mode);

    if (sync_mode != SYNC)
        {
        return(NULL);
        }

    return threadid;
    }
```

2.14.7 Construct_object_id

Description of construct_object_id

This function will construct the objectid using the passed buffer and objid values. It will put the objid.pid and objid.hostid values into the buffer.

Code of construct_object_id

```
char * construct_object_id(buffer,objid)
struct _objid objid;
char *buffer;
    {
    sprintf(buffer,"%d_%d_%d_%d",
            objid.time.tv_sec,
            objid.time.tv_usec,
            objid.pid,
            objid.hostid);
    return(buffer);
    }
```

2.14.8 Malloc1

Description of malloc1

This function is used to implement a simple malloc, but makes sure that only one thread enters the function at a time. The function does this by closing off the whole function as a critical section. Then the malloc is called and the pointer to the memory is returned.

Code of malloc1

```
char * malloc1(c)
int c;
    {
    char * p;

    BEGIN_CRITICAL_SEC;
    p = malloc(c);
    if (p == NULL)
        {
        p = malloc(c);
        if (p == NULL)
            {
            printf("MALLOC has Failed. size request %d\n",c);
            if ((_frame != NULL) && (_frame->$.caller))
                {
                printf("malloc :calling function %x \n",
                       _frame->$.func);
                printf("malloc :Caller of calling function
                       %x \n", _frame->$.caller->$.func);
                }
            }
        }
#ifdef DEBUG
    printf("%x Allocate Malloc time = %d size == %d   ",p,
           time_stamp++,c);
    if ((_frame != NULL) && (_frame->$.caller))
        {
        printf("%x malloc :calling function %x ",p,
               _frame->$.func);
        printf("%x malloc :Caller of calling function
               %x \n",p, _frame->$.caller->$.func);
        }
    else if (_frame != NULL)
        {
        printf("%x MALLOC :calling function %x \n",p,
               _frame->$.func);
        }
```

```
#endif
    END_CRITICAL_SEC;
    return(p);
    }
```

2.14.9 Free1

Description of free1

This function is used to implement a simple free call to free up the memory allocated by a malloc, but this function also makes sure that only one thread at a time enters the function. This is also done by enclosing the function in a critical section and doing the free in the middle.

Code of free1

```
free1(p)
char * p;
    {

    BEGIN_CRITICAL_SEC;

#ifdef DEBUG
    printf("%x Allocate Freeing %d  ",p,time_stamp);
    if (_frame->$.caller)
        {
        printf("%x free:: calling function  ",p,
                _frame->$.func);
        printf("%x free:: Caller of calling function
               %x \n",p, _frame->$.caller->$.func);
        }
    else if (_frame != NULL)
        {
        printf("%x free :calling function %x \n",p,
               _frame->$.func);
        }
#endif

    free(p);
    END_CRITICAL_SEC;
    }
```

2.14.10 MSG_ADJUST_OFFSET

Description of MSG_ADJUST_OFFSET

This function will change the offset of an optional parameter. To do so, the function is passed the destination of the optional parameter structure and the starting address of the parameter.

Code of MSG_ADJUST_OFFSET

```
int MSG_ADJUST_OFFSET(param,dest_addr,starting_num)
struct _msg *param;  /* dest optional paramter structure.*/
char * dest_addr;    /* Starting addr of actual parameter.*/
int starting_num;    /* starting array subscript */
    {
    int total_param ;
    int counter;
    int offset;

    total_param = param->_num_arg;

    if (starting_num == 0)
        {
        offset = (int)dest_addr - (int)param;
        param->_para[starting_num].offset = offset;
        starting_num ++;
        }
    for (counter = starting_num; counter < total_param;
         counter ++)
        {
        if (strcmp(param->_para[counter].type->
                   postfix_type,"char") == 0)
            {
            param->_para[counter].offset =
                    param->_para[counter-1].offset + 4;
            }
        else
            param->_para[counter].offset =
                    param->_para[counter-1].offset
                    + param->_para[counter-1].type->size;
        }
    }
```

2.14.11 Disable_destroy

Description of disable_destroy

This function will take an object, obj, as a parameter, which has the status of SUSPEND. The function will take this object and make the function do a delayuntil a certain message is set back. This will allow an object that needs this object to finish before this object is destroyed.

Code of disable_destroy

```
void disable_destroy(obj)
struct _object * obj;
    {
    char buff[MAX_ID_SIZE];
    char wait_string[MAX_ID_SIZE];

    construct_object_id(buff,obj->$.objid);
    sprintf(wait_string,"wait_message%s",buff);
    DELAYUNTIL_STR(wait_string,0);
    }
```

2.14.12 Enable_obj_destroy

Description of enable_obj_destroy

This function will allow the suspended object, obj, that was passed to the function, to be destroyed now. It was delayed until some other object had finished using this object. The object had been delayed using the disable_destroy function.

Code of enable_obj_destroy

```
void enable_obj_destroy(obj)
struct _object * obj;
    {
    char buff[MAX_ID_SIZE];
    char resp_string[MAX_ID_SIZE];

    construct_object_id(buff,obj->$.objid);
    sprintf(resp_string,"wait_message%s",buff);
    RESPOND_STR(resp_string,0);
    }
```

2.14.13 Can_we_destroy_the_object

Description of can_we_destroy_the_object

This function will take some object, obj, passed to the function and check to see if the object can be destroyed or not. It can be destroyed only if there is no other object that needs this object.

If there are no messages that are pending, that is any messages waiting for this object, then the status can be set to OK and the object can just be destroyed. If the number of pending messages is less than zero, then destroy has been called a multiple number of times for this object, so we can ignore this one. If there are pending messages for this object, then this object can not be destroyed yet. If the pending message is to the object itself, then it can be destroyed. Otherwise, the status of the object will be set to SUSPEND. Then the function will call the disable_destroy function, discussed in Section 2.14.11. When the pending message is finally done, we will return back to this function and set the status to OK and allow the object to be destroyed.

Code of can_we_destroy_the_object

```
#define OK 0
#define SUSPEND 1
#define DUP 2

int can_we_destroy_the_object (obj)
struct _object * obj;
    {
    int status;

    BEGIN_CRITICAL_SEC;

    if (obj->$.pending_message == 0)
        status = OK;
    if (obj->$.pending_message > 0)
        {
        /*
         * There are message waiting for this object, we
         * can not destroy this object, until those messages
         * finished.
         */

        if (obj->$.pending_message == 1)
```

```
        {
        struct _frame * fp = _frame;
        /*
         * Do not suspend itself.
         */

        while ((fp) && (fp->$.obj != obj))
            {
            fp = fp->$.caller;
            }
        if (fp)
            {
            status = OK;
            }
        else
            {
            obj->$.pending_message *= -1;
            status = SUSPEND;
            }
        }
    else
        {
        obj->$.pending_message *= -1;
        status = SUSPEND;
        }
    }

else if (obj->$.pending_message < 0)
    {
    /*
     * mulitple destroy has been issued at the same
     * object, we will ignore this one.
     */
    status = DUP;
    }
END_CRITICAL_SEC;

if (status == SUSPEND)
    {
    disable_destroy(obj);
    status = OK;
    }
```

```
return status;
}
```

2.15 Description of Eval.sun4.s

This section discusses and presents the assembly code needed to implement context switching for the four stack methods:

- Circular Interleaving
- Heap Base
- Stack Base
- Stack Base Slow

This assembly code is written especially for use on SPARC systems. It should be noted at this point that this part of code for MeldC is the only part that is not portable. The assembly code must be written for each system that MeldC is to be run on. A limitation should also be noted, that the MeldC context switching does not save the floating point registers. Due to this, any floating point operations that are performed and interrupted in the middle may not return the proper value due to register corruption.

This code will be used, as stated already, to do the context switching for MeldC. In otherwords, this code will handle the RUN, RESUME and YIELD operations for MeldC. To outline this process, we first start with the RUN operation. When a function is first called, it will be put on a run queue from where it will be choosen to execute. After the function has been choosen to run, it has two choices:

1. The function will finish it's execution.
2. The function will do a YIELD.

When the function is done executing, execution will be given back to the scheduler, where a new fucntion will be choosen. If a YIELD is done, the scheduler will also be given execution again. In this case though, the old function which did the YIELD will be suspended and will be waiting to RESUME. When the new functions are done executing, the old function will do a RESUME to continue it's execution. Let's now look at the detail to how this is all done.

2.16 Headers of Eval.sun4.s

```
#ifdef CIRCULAR_INTERLEAVING
```

```
#define FRAME_SIZE 0x4

#define FRAME_STATE 0xc

#define FUNCTION_PTR 0x14

#define SAVE_FRAME_SIZE 0x1c

#define REENTRY_ADDR 0x18

#define STACK_PTR 0x28

#define GLOBAL_VARIABLE 0x128

#define SAVE_REGISTERS 0x148

#define _TMP_SP %g2

#define _RETURN_TO_SCHEDULER %g3

#define _FP_OF_RUN_OR_RESUME %g4

#define _FUNC_LOW_SP %g5

#define _FUNC_UP_SP %g6


#else /* Everything else. */

#define FRAME_SIZE 0x4

#define FRAME_STATE  0x8

#define FUNCTION_PTR 0x10

#define REENTRY_ADDR 0x14

#define SAVE_FRAME_SIZE 0x18

#define STACK_PTR 0x24

#define FRAME_PTR 0x28
```

```
#ifdef STACK_BASE

#define GLOBAL_VARIABLE 0x30

#define SAVE_REGISTERS 0x50

#elif STACK_BASE_SLOW

#define GLOBAL_VARIABLE 0x12c

#define SAVE_REGISTERS 0x14c

#else

#define GLOBAL_VARIABLE 0x124

#define SAVE_REGISTERS 0x144

#endif

#define _TMP_SP %g2

#define _RETURN_TO_SCHEDULER %g3

#define _FP_OF_RUN_OR_RESUME %g4

#define _FUNC_LOW_SP %g5

#define _FUNC_UP_SP %g6

#endif



#ifdef FRAME_ALLOCATION_HEAP_BASE
#define DEFAULT_APPROACH
#endif

#ifdef CIRCULAR_INTERLEAVING
#define DEFAULT_APPROACH
#endif
```

```
#ifndef STACK_BASE

/*
 * Except the stack base implementation, rest of the
 * implementation will be the following assembly code.
 */

#define KERNEL

#include <sun4/asm_linkage.h>


!
! eval.sun4.s :
! Containing function "run" "resume" "yield"
!
! Last modifier : James
!
! Date          : 4/17/91
!
! Few changes from its orginal code (which generate from eval.c)
!
!     1) Run,resume and yield are no longer function calls.
!        A program that calls any of these three routines, the cost of
!        this operation will be a a jump statement vs a function call.
!
!     2) In Sun4 Sparc arch, saving the register set is not as trival
!        as in other machine.  It is necessary to flush all the register
!        set back to the kernel memory before we update our fp or sp.
!        (please refer to the documemnt README.SPARC in this directory
!         for more detail)


LL0:
        .seg    "data"
        .seg    "bss"
        .align  4
        .global _signal_mask
_signal_mask:
        .word   0x0
```

```
        .align  4
        .global _previous_mask
_previous_mask:
        .word   0x0

        .seg    "text"
        .proc 0110
        .global _run
```

2.16.1 Description of RUN for Eval.sun4.s

Let us first look at the RUN operation. When a RUN is done, we first will save the return address to the scheduler. This address will be used when a YIELD is done. At this point, we must also save the current fp, frame pointer, so that it will appear we returned from this function when YIELD returns. We now can load the function into memory to be run. The information we need to run the function is the beginning address of the function. After getting the needed information about the function and then properly saving it, we will execute the function. This is done with the *"call %gl, 0"* line. When the function terminates or YIELD's, we will always return to the point just after this call.

When the execution returns to this point, the scheduler address must be restored. This address is the return address that we saved when the RUN first started. After we restore the scheduler's address, we will call the scheduler to choose the next function to execute and start this process all over again.

There are a few things that should be noted about the assembly code. The line *"mov 0x3, %o1"* is the frame's state, which in this case equals S_EXIT. The following lines should also be pointed out:

```
#ifdef PREEMPTIVE
        call    _exit_meldc_kernel,0
        nop
#endif
```

These lines will enable preemptive scheduling.

2.16.2 Code of RUN for Eval.sun4.s

```
_run:
        !#PROLOGUE# 1
        !.seg   "data1"
```

```
L20:
        !.ascii "* RUN * \n"
        !.seg   "text"
        !set    L20,%o0
        !call   _printf,1
        !nop

!
! Save the return address, that can be used by yield.
! Yield will use this address as return address to the
! scheduler.
!
        mov     %o7,_RETURN_TO_SCHEDULER

!
! We also have to save the current fp, so that when
! yield returns it will appear that it was returned from
! this routine.
!
        mov     %fp,_FP_OF_RUN_OR_RESUME
        mov     %sp,_FUNC_LOW_SP
!
! Making the call to the procedure that is schedule
! to run by the scheduler.
!

        sethi   %hi(__frame),%o0
        ld      [%o0+%lo(__frame)],%o0

        st      _RETURN_TO_SCHEDULER,[%o0 + (3*4+GLOBAL_VARIABLE)]
        st      _FP_OF_RUN_OR_RESUME,[%o0 + (4*4+GLOBAL_VARIABLE)]
        st      _FUNC_LOW_SP,[%o0 + (5*4+GLOBAL_VARIABLE)]

#ifdef PREEMPTIVE
        call    _exit_meldc_kernel,0
        nop
#endif
        sethi   %hi(__frame),%o0
        ld      [%o0+%lo(__frame)],%o0
        ld      [%o0+FUNCTION_PTR],%g1
        call    %g1,0
        nop
```

```
        sethi   %hi(__frame),%o0
        ld      [%o0+%lo(__frame)],%o0
        ld [%o0 + (3*4)+GLOBAL_VARIABLE], _RETURN_TO_SCHEDULER

! The function which "run" calls, did not issue any yield,
! then When it finished, it will set its status to S_EXIT
! and return.
!
! If the function issued a yield, it will restart later by
! a resume call.  This process can repeat several times,
! until the function execute return statement, than it will
! return back to this routine and execute the following code.
!
        sethi   %hi(__frame),%o0
        ld      [%o0+%lo(__frame)],%o0
        mov     0x3,%o1
        st      %o1,[%o0+FRAME_STATE]
        mov     _RETURN_TO_SCHEDULER,%i7
LE48:
        ret
        nop
```

2.16.3 Description of RESUME for Eval.sun4.s

As stated before, we will do a RESUME when a YIELD has been done. What must first be done, is for us to save the return address of the function that made the YIEld call. The YIELD operation will later use this address to make it look like it has returned from RESUME. Along with saving the return address, it is important to save the scheduler's state, sp and fp pointers. This RESUME is obviously going to restore some old function. All of the information needed by this function had been saved by YIELD. The RESUME will simply restore all of this imformation saved. First we will get the pointers, fp and sp. Now we must get the stack, but this method depends on the size of the stack before the YIELD. If the stack excedes 256 bytes, we must find the spot it was saved at. If the stack was less than 256 bytes, it is a simple matter of getting the stack from the reserved spot inside of the frame structure. After we restore the stack, we must than move the fp and sp pointers to the proper addresses. Once we compute the addresses, we can than restore the i and l registers. Note that when preemptive scheduling is used, we must also restore the o and g registers. Now we can allow the function to execute.

2.16.4 Code of RESUME for Eval.sun4.s

```
_resume:
        !#PROLOGUE# 1
        !.seg   "data1"
L30:
        !.ascii "* RESUME *\n"
        !.seg   "text"
        !set    L30,%o0
        !call   _printf,1
        !nop
        !Restore all the registers.


!
! Save the return address, (located in o7), in case, the
! function issued a yield.  Yield can act like it has return
! from resume by using the return address and fp.
!
        mov     %sp,_FUNC_LOW_SP

        sub     %sp,400,%sp
        t       3
        mov     %o7,_RETURN_TO_SCHEDULER
        mov     %fp,_FP_OF_RUN_OR_RESUME


!
! when the function is suspended by yield, yield save all the
! relvent information into a memory (__frame). The following
! code is going to restore these information. (restore them
! on top of the stack.
!



!
! We also want to make sure that, the number of bytes is not equal
! to zero. We will also set up the index that will laster be used
! to restore our data.
!
```

```
        sethi   %hi(__frame),%o1
        ld      [%o1+%lo(__frame)],%o1
        ld      [%o1+SAVE_FRAME_SIZE],%o0
        sll     %o0,0x2,%o1
        sub     _FUNC_LOW_SP,%o1,_FUNC_UP_SP
        sethi   %hi(__frame),%o5
        ld      [%o5+%lo(__frame)],%o5
        add     %o5,STACK_PTR,_TMP_SP
        cmp     _FUNC_UP_SP,_FUNC_LOW_SP
        bgu     L54
        nop
!
! The loop that restore  the data.
!


!
! First compute the number of bytes that we saved in the memory.
!
        sethi   %hi(__frame),%o1
        ld      [%o1+%lo(__frame)],%o1
        ld      [%o1+SAVE_FRAME_SIZE],%o0
! Reserve the space, that will be needed, to retore the satck.
!
        sll     %o0,0x2,%o1
        sub     %sp,%o1,%sp
        t       3


! If the C stack is greater than 64 wrods (4 bytes =a word), than
! call the special restore routine, to restore the satck, else the satck
! information is located in the frame structure.
!
        cmp     %o0,0x40
        bge     L56
        nop
L55:
        ld      [_TMP_SP],%o2
        st      %o2,[_FUNC_UP_SP]
        add     _TMP_SP,0x4,_TMP_SP
        add     _FUNC_UP_SP,0x4,_FUNC_UP_SP
L53:
        cmp     _FUNC_UP_SP,_FUNC_LOW_SP
```

```
        bleu    L55
        nop

        b       L54
        nop

!
!Call the restore the frame when the C stack is greater than 256 bytes.
!

L56:    sethi   %hi(__frame),%o0
        ld      [%o0+%lo(__frame)],%o0
        mov     _FUNC_LOW_SP,%o1
        mov     _FUNC_UP_SP,%o2
        call    _restore_frame,3
        nop



!
! After we complete restore the stack, we want to move the frame
! pointer and stack pointer into the proper location, and start
! execute the previous suspend function.
!
L54:
        sethi   %hi(__frame),%o5
        ld      [%o5+%lo(__frame)],%o5
        ld      [%o5+SAVE_FRAME_SIZE],%o0
        sll     %o0,0x2,%o1
!
! Compute the address that our stack pointer should point at when
! we restart the function.
!
        sub     _FUNC_LOW_SP,%o1,%o1

!
! The following code might be a little be strange, what we try
! to do is to restore the sp of the function.
!
! We first copy the address into the fp, and than issue a
! restore command, which will in turn copy the fp into the sp
! restore all the registers l0-l7, i0-i7.
```

```
!
! The resason that we did this is because, the first 64 bytes that
! sp point at is reserve for kernel memory. We cannot directly assign
! the new address to sp, otherwise The first 64 bytes will be corrupted.
! The following code are another way to move the address into sp without
! destory its contain.
!


        mov     %o1,%fp

        sethi   %hi(__frame),%o2
        ld      [%o2+%lo(__frame)],%o2
        st      _TMP_SP,[%o2 + (2*4+GLOBAL_VARIABLE)]
        st      _RETURN_TO_SCHEDULER,[%o2 + (3*4+GLOBAL_VARIABLE)]
        st      _FP_OF_RUN_OR_RESUME,[%o2 + (4*4+GLOBAL_VARIABLE)]
        st      _FUNC_LOW_SP,[%o2 + (5*4+GLOBAL_VARIABLE)]
        st      _FUNC_UP_SP,[%o2 + (6*4+GLOBAL_VARIABLE)]
        restore


        sethi   %hi(__frame),%o2
        ld      [%o2+%lo(__frame)],%o2
        ld      [%o2+REENTRY_ADDR],%o7


!
! If the first file of SAVE_REGISTERS is equal to zero, this is not
! a preemptive thread, and we do not  have to restore the global and
! output register.
!


#ifdef PREEMPTIVE
        sethi   %hi(__frame),%g7
        ld      [%g7+%lo(__frame)],%g7
        ld      [%g7 + (0*4+SAVE_REGISTERS)],%g6
        cmp     %g6,0
        beq     LE51
        nop
!
! Restore all the register.
```

```
!
        ld      [%g7 + (8*4+SAVE_REGISTERS)],%o0
        ld      [%g7 + (9*4+SAVE_REGISTERS)],%o1
        ld      [%g7 + (10*4+SAVE_REGISTERS)],%o2
        ld      [%g7 + (11*4+SAVE_REGISTERS)],%o3
        ld      [%g7 + (12*4+SAVE_REGISTERS)],%o4
        ld      [%g7 + (13*4+SAVE_REGISTERS)],%o5
        ld      [%g7 + (15*4+SAVE_REGISTERS)],%o7

        st      %g0,[%g7 + (0*4+SAVE_REGISTERS)]

        save    %sp,-72,%sp

        sethi   %hi(__frame),%o0
        ld      [%o0+%lo(__frame)],%o0
        ld      [%o0 + (1*4+SAVE_REGISTERS)],%g1
        ld      [%o0 + (2*4+SAVE_REGISTERS)],%g2
        ld      [%o0 + (3*4+SAVE_REGISTERS)],%g3
        ld      [%o0 + (4*4+SAVE_REGISTERS)],%g4
        ld      [%o0 + (5*4+SAVE_REGISTERS)],%g5
        ld      [%o0 + (6*4+SAVE_REGISTERS)],%g6
        ld      [%o0 + (7*4+SAVE_REGISTERS)],%g7
        ld      [%o0 + (17*4+SAVE_REGISTERS)],%o2
        mov     %o2,%y

        call    _exit_meldc_kernel,0
        nop

        restore
#endif
```

2.16.5 Description of YIELD for Eval.sun4.s

When a YIELD is done, we must save all of the information about the function doing the YIELD. This information includes the registers and stack. The first thing done by YIELD is to save the i and l registers, save o and g if preemptive scheduling as well. Then mark the frame as S_READY. Then we will move the sp and fp pointers so that we will point to the frames stack that needs to be saved. We must than calculate the size of this stack. If the stack size is under 256 bytes, we will save the stack inside of the frame structure. If the stack is greater than 256 bytes, than we must instead, save the stack in some other memory location and save the pointer to this location in the frame structure. After this, all of the frames information will be saved and we can then restore the information of the scheduler, saved by frame when RUN executed.

2.16.6 Code of YIELD for Eval.sun4.s

```
!
! yield : Stop the execution of the current function, save
!         its state and return back to the scheduler.

_yield:
        !#PROLOGUE# 1
        !.seg   "data1"
L40:
        !.ascii "* YIELD *\n"
        !.seg   "text"
        !set    L40,%o0
        !call   _printf,1
        !Restore all the registers.


        b       L58
        nop

#ifdef PREEMPTIVE
L57:
!
! if we receive a SIGUSR1, SIGUSR1 handler will directly jump
! to this location.
!
! We need to save global and output registers.
!
! Since we are not allow to modify any thing, and yet at the
! same time we need to use a register to save the information.
!
! We will move up the stack pointer, and use the register %l0.
! The value of register %l0 should already be saved in the kernel
! save area (the previous 64 bytes).   After we done we can
! restore the %l0 register.
!
        t       3
```

```
        sub     %sp,64,%sp
        mov     %g1,%l0
        sethi   %hi(__frame),%g1
        ld      [%g1+%lo(__frame)],%g1

!
! Save the global and local register.
!
        st      %l0,[%g1 + (1*4+SAVE_REGISTERS)]
        st      %g2,[%g1 + (2*4+SAVE_REGISTERS)]
        st      %g3,[%g1 + (3*4+SAVE_REGISTERS)]
        st      %g4,[%g1 + (4*4+SAVE_REGISTERS)]
        st      %g5,[%g1 + (5*4+SAVE_REGISTERS)]
        st      %g6,[%g1 + (6*4+SAVE_REGISTERS)]
        st      %g7,[%g1 + (7*4+SAVE_REGISTERS)]

        st      %o0,[%g1 + (8*4+SAVE_REGISTERS)]
        st      %o1,[%g1 + (9*4+SAVE_REGISTERS)]
        st      %o2,[%g1 + (10*4+SAVE_REGISTERS)]
        st      %o3,[%g1 + (11*4+SAVE_REGISTERS)]
        st      %o4,[%g1 + (12*4+SAVE_REGISTERS)]
        st      %o5,[%g1 + (13*4+SAVE_REGISTERS)]
!       mov     %psr,%o0
!       st      %o0,[%g1 + (16*4+SAVE_REGISTERS)]
        mov     %y,%o1
        st      %o1,[%g1 + (17*4+SAVE_REGISTERS)]

!
! Small hack : if the first location of SAVE_REGISTERS is not NULL,
! than it is implied that this particular thread has been preemptive
! and we should restore the global and output register before we
! resume.
!
        st      %sp,[%g1 + (1*0+SAVE_REGISTERS)]
!
! Mark the frame ready. S_READY = 1;
!
        mov     0x1,%o0
        st      %o0,[%g1 + FRAME_STATE]


!
```

```
! Restore the the g1,sp,l0 register
!
        mov     %l0,%g1
        ld      [%sp+64],%l0
        add     %sp,64,%sp
        b       LE58
        nop

#endif

L58:

        sethi   %hi(__frame),%o1
        ld      [%o1+%lo(__frame)],%o1
        st      %o7,[%o1+REENTRY_ADDR]
#ifdef PREEMPTIVE
        call    _enter_meldc_kernel,0
        nop
#endif

LE58:

        sethi   %hi(__frame),%o0
        ld      [%o0+%lo(__frame)],%o0
        ld      [%o0 + (3*4)+GLOBAL_VARIABLE], _RETURN_TO_SCHEDULER
        ld      [%o0 + (4*4)+GLOBAL_VARIABLE], _FP_OF_RUN_OR_RESUME
        ld      [%o0 + (5*4)+GLOBAL_VARIABLE], _FUNC_LOW_SP
        ld      [%o0 + (6*4)+GLOBAL_VARIABLE], _FUNC_UP_SP




!
! trap 3, flush all the window registers into the kernel memeory, so
! that move fp, and sp will not cause any strange behavior.
!
! THE MAGIC TRAP !!!!


        mov     %sp,_FUNC_UP_SP
        t       3
```

```
        sub     %sp,64,%sp
!
! Compute the nunmber bytes that we have to save into the frame
! structure.
! Make sure the number of bytes we have to save is not equal to
! zero.
!

        sethi   %hi(__frame),%o4
        ld      [%o4+%lo(__frame)],%o4
        sub     _FUNC_LOW_SP,_FUNC_UP_SP,%o0
        sra     %o0,0x2,%o1
        st      %o1,[%o4+SAVE_FRAME_SIZE]
!
! If the C Stack is greater than 64 words, than call a special routine
! to save the C stack, else save in the frame structure.
!
        cmp     %o1,0x40
        bge     L70



        sethi   %hi(__frame),%o2
        ld      [%o2+%lo(__frame)],%o2
        add     %o2,STACK_PTR,_TMP_SP
        cmp     _FUNC_UP_SP,_FUNC_LOW_SP
        bgu     L60
        nop

!
! Save the state of the function that issued the yield.
!

L61:
        ld      [_FUNC_UP_SP],%o2
        st      %o2,[_TMP_SP]
        add     _FUNC_UP_SP,0x4,_FUNC_UP_SP
        add     _TMP_SP,0x4,_TMP_SP
L59:
        cmp     _FUNC_UP_SP,_FUNC_LOW_SP
        bleu    L61
        nop
```

```
        b       L60
        nop

L70:
        sethi   %hi(__frame),%o0
        ld      [%o0+%lo(__frame)],%o0
        mov      _FUNC_LOW_SP,%o1
        mov      _FUNC_UP_SP,%o2
        call    _save_frame,3
        nop
L60:

!
! Restore the sp and fp, so that returning from yield will appear
! to be it is return from resume or run.
!

!
! Retore the stack pointer.
!
!
        sethi   %hi(__frame),%o0
        ld      [%o0+%lo(__frame)],%o0
        st      _TMP_SP,[%o0 + (2*4+GLOBAL_VARIABLE)]
        st      _RETURN_TO_SCHEDULER,[%o0 + (3*4+GLOBAL_VARIABLE)]
        st      _FP_OF_RUN_OR_RESUME,[%o0 + (4*4+GLOBAL_VARIABLE)]
        st      _FUNC_LOW_SP,[%o0 + (5*4+GLOBAL_VARIABLE)]
        st      _FUNC_UP_SP,[%o0 + (6*4+GLOBAL_VARIABLE)]

        mov     _FUNC_LOW_SP,%o2
        t       3
        ldd [%o2 + (0*4)], %l0; ldd [%o2 + (2*4)], %l2;
        ldd [%o2 + (4*4)], %l4;
        ldd [%o2 + (6*4)], %l6; ldd [%o2 + (8*4)], %i0;
        ldd [%o2 + (10*4)], %i2;ldd [%o2 + (12*4)], %i4;
        ldd [%o2 + (14*4)], %i6;

        mov     %o2,%sp

!
! Restore the fp.
```

```
!
        t       3
        mov     _FP_OF_RUN_OR_RESUME,%fp
        mov     _RETURN_TO_SCHEDULER,%i7


LE57:
        ret
        nop
```

2.16.7 Preemptive Scheduling

A couple of things to note about preemptive scheduling. It is necessary to disable preemptive scheduling except when in a function. If this was not done, we could possibly enter a continuous loop. The following line will enable preemptive scheduling in the above assembly code:

```
#ifdef PREEMPTIVE
        call    _exit_meldc_kernel,0
        nop
#endif
```

To disable preemptive scheduling, we will do the following:

```
#ifdef PREEMPTIVE
        call    _enter_meldc_kernel,0
        nop
#endif
```

Below we have included the code used to enable and disable the preemptive scheduling.

2.16.8 Code of Preemptive Scheduling

```
#ifdef PREEMPTIVE

!
! enter_meldc_kernel and exit meldc_kernel will only modify
! two registers, o1 and o2.  This will allow the calling
! function to store some information into the global register.
!
        .proc 0110
        .global _enter_meldc_kernel
```

```
_enter_meldc_kernel:

        mov     0x1,%o1
        sethi   %hi(_signal_mask),%o2
        swap    [%o2+%lo(_signal_mask)],%o1

        retl
        nop

        .proc 0110
        .global _exit_meldc_kernel
_exit_meldc_kernel:

        sethi   %hi(_signal_mask),%o2
        swap    [%o2+%lo(_signal_mask)],%g0

        retl
        nop

        .proc 020
        .global _return_current_sp
_return_current_sp:
        mov     %sp,%o0
        retl
        nop
#endif
```

2.17 Eval.mips.s

2.17.1 Description of Eval.mips.s

The mips assembly code implementation of the context switching is much easier than the sun4 case. This is mainly due to the fact that mips does not support preemptive scheduling. Since the sun4 case has been described in much detail, we will not go into this case in as much detail.

2.17.2 Headers of Eval.mips.s

```
#ifdef CIRCULAR_INTERLEAVING
#define FRAME_SIZE 0x4
```

```
#define FRAME_STATE 0xc

#define FUNCTION_PTR 0x14

#define SAVE_FRAME_SIZE 0x1c

#define REENTRY_ADDR 0x18

#define STACK_PTR 0x28

#else

#define FRAME_SIZE 0x4

#define FRAME_STATE  0x8

#define FUNCTION_PTR 0x10

#define REENTRY_ADDR 0x14

#define SAVE_FRAME_SIZE 0x18

#define STACK_PTR 0x24

#define FRAME_PTR 0x28

#endif
         .verstamp       2 10
        .extern _obj 4
        .extern _frame 4
        .lcomm  $$5 4
        .sdata
        .align  2
        .align  0
$$6:
        .word   0 : 1
        .sdata
        .align  2
        .align  0
$$7:
        .word   0 : 1
```

```
        .sdata
        .align  2
        .align  0
$$8:
        .word   0 : 1
        .lcomm  $$9 4
        .globl  _SP
        .sdata
        .align  2
        .align  0
_SP:
        .word   0 : 1
        .comm   _RET_REGISTER 4
        .text
        .align  2
        .file   2 "eval.c"
        .globl  run
        .loc    2 98
 #  98  {
        .ent    run 2
```

2.17.3 Description of RUN for Eval.mips.s

In the RUN case, we must first store the return address of the scheduler. Then we will get the sp pointer address. Now we can execute the function. Once the function is done, we will return to this point. We must then set the frame state equal to S_EXIT. Then we must reset the scheduler's address so that we may return to it. Once we have the address, we will return back to that address.

2.17.4 Code of RUN for Eval.mips.s

```
run:
        .option O1
        .mask   0x80000000, -4
        .frame  $sp, 24, $31
        .loc    2 104
 # Recore the return-addr to the scheduler, so later yield()
 # can use this address to jump back to the scheduler.
 # For mips machines, RET_REGISTER --> $31 for mips machine
 # For other machines, it might be necessary to pop the stack
 # return_to_scheduler = _RET_REGISTER;
```

```
        sw      $31, $$5
        .loc    2 109
 #
 # change _SP  to the  real stack pointer ($sp)
 # So we know how big for func()'s stack frame, (see yield())
 # func_low_sp = _SP;

        sw      $sp, $$9
        .loc    2 111
 #
 #  return_value = (*(_frame->func))();

        lw      $24, _frame
        lw      $25, FUNCTION_PTR($24)
        jal     $25
$32:
 #  _frame->state = S_EXIT;

 #   move 8 bytes
        lw      $8,_frame
        li      $9, 3
        sw      $9, FRAME_STATE($8)

        .loc    2 125
 #  _RET_REGISTER = return_to_scheduler;
        lw      $31, $$5
        .loc    2 127
 #
 #}
        j       $31
        .end    run
        .text
        .align  2
        .file   2 "eval.c"
        .globl  resume
        .loc    2 146
 #   {
        .ent    resume 2
```

2.17.5 Description of RESUME for Eval.mips.s

When a resume is done, we want to allow some other function to execute. To do this, we must first save the scheduler's address so that the YIELD can later return to the scheduler. We then must set the fp and sp pointers to the correct addresses. Then we must check the size of the saved stack frame. If it is greater than 256 bytes, we must go to the reserved memory spot where the stack has been saved and restore the stack. If the stack size is under 256 bytes, we can get the stack from the frame structure, where the stack had been saved. Once this has been done, the function can resum it's execution.

2.17.6 Code of RESUME for Eval.mips.s

```
resume:
        .option O1
        .frame  $sp, 0, $31
        .loc    2 151
 # record the return address to the scheduler, so later yield()
 # can use this address to return (jump back) to the scheduler
 # For mips machines, change  RET_REGISTER to  $31.
 # return_to_scheduler = _RET_REGISTER;
        sw      $31, $$5

        .loc    2 158
 #  func_low_sp = _SP - 6;

        sw      $sp, $$9
        move    $2, $sp
        .loc    2 160
 #
 # func_up_sp = func_low_sp - _frame->save_frame_size;
        lw      $15, _frame

        lw      $24, SAVE_FRAME_SIZE($15)


        subu    $3, $2, $24
        .loc    2 165

 # adjust the stack  pointer.
 # Change the assembly code which do  $sp = func_up_sp
```

```
 # _SP = func_up_sp;
        move    $sp, $3

 # RESTORE func()'s stack frame by moving from the save area
 # in frame
 # If necessary, change asm code for fast memory-to-memory
 # transfering
 # if (_frame->$.save_frame_size > 64)
   lw      $12, SAVE_FRAME_SIZE($15)
   ble     $12, 255, $39
   .loc    2 193
 # restore_frame(_frame,func_low_sp,func_up_sp);
   move    $4, $15
   move    $5, $2
   move    $6, $3
   jal     restore_frame
   .loc    2 194
    b       $34

$39:
 # for(tmp_sp = _frame->save_stack_frame ;
 #     func_up_sp <= func_low_sp ;){
 # move 8 bytes

        addu    $3, $3, 16
        .loc    2 170

        addu    $4, $15, STACK_PTR

$33:
        .loc    2 171
 # 171      *func_up_sp++ = *tmp_sp++;
        ld      $10, 0($4)
        sd      $10, 0($3)
        addu    $4, $4, 8
        addu    $3, $3, 8
        .loc    2 172
 # 172    }
        .loc    2 172
        bltu    $3, $2, $33
$34:
        .loc    2 178
```

```
 #
 # set the return address to _frame->reentry_addr,
 # so the control will
 # return to the func()'s at the reentry_addr.
 # $31 = _frame->reentry_addr;
 # _RET_REGISTER = _frame->reentry_addr;
        lw      $10, _frame
 # move 8 bytes
        lw      $31, REENTRY_ADDR($10)
        .loc    2 185
 #
        j       $31
        .end    resume
        .text
        .align  2
        .file   2 "eval.c"
        .globl  yield
        .loc    2 212
 #
        .ent    yield 2
```

2.17.7 Description of YIELD for Eval.mips.s

When a YIELD is done, we must first set the return address to the address saved on the stack. Then we must change the sp and fp pointers. This will allow us to figure out the size of the stack so that it can be saved. If it is greater than 256 bytes, it will be saved in some reserved memory location. If it is less than 256 bytes, we can than save the stack in the frame structure. Once we have saved the stack, we can then return to the scheduler.

2.17.8 Code of YIELD for Eval.mips.s

```
yield:
        .option O1
        .frame  $sp, 0, $31
 # move func_low_sp to $3
        lw      $3, $$9
        .loc    2 218
 #
 # Change _RET_REGISTER to the return  register or the
 # return address on the stack.
```

```
#
# _frame->reentry_addr = _RET_REGISTER;
       lw      $15, _frame
# move 8 bytes
       sw      $31, REENTRY_ADDR($15)
       .loc    2 224
#
# set the func()' upper bound sp to $sp
# change _SP  to the  real stack pointer ($sp)
# So we know how big for func()'s stack frame.
# func_up_sp = _SP;
       move    $24, $sp

       addu    $2, $24, 16
       .loc    2 226
#
# frame->save_frame_size = func_low_sp - func_up_sp;

       subu    $8, $3, $24
# remove division since no multiplication

       lw      $9, _frame

# move 8 bytes
       sw      $8, SAVE_FRAME_SIZE($9)
       .loc    2 230
#
# SAVE func()'s stack frame by moving from the save
# area in frame
# If necessary, change asm code for fast memory-to-memory
# transfering


# if (_frame->$.save_frame_size > 64)
  lw      $11, _frame
  lw      $12, SAVE_FRAME_SIZE($11)
  ble     $12, 255, $35
  .loc    2 248
# {
# save_frame(_frame,func_low_sp,func_up_sp);
  subu    $2, $2, 16
  move    $4, $11
```

```
  move    $5, $3
  move    $6, $2
  jal     save_frame
#  .loc    2 249
## 249        }
   b       $37

$35:


 # for(tmp_sp = _frame->save_stack_frame ;
 #     func_up_sp <= func_low_sp ;){
        lw      $4, _frame
 # move 8 bytes
        addu    $4, $4, STACK_PTR

$36:
        .loc    2 231
 # *tmp_sp++ = *func_up_sp++;
        ld      $10, 0($2)
        sd      $10, 0($4)
        addu    $2, $2, 8
        addu    $4, $4, 8
        .loc    2 232
 # }
        .loc    2 232


        bltu    $2, $3, $36
$37:
        .loc    2 239
 # _SP =  func_low_sp + 6;
 #      addu    $sp, $3, 24

        lw      $3, $$9
        move    $sp, $3
        .loc    2 246
 #  _RET_REGISTER = return_to_scheduler;
        lw      $31, $$5
        .loc    2 248
 #
 #  }
```

```
        j       $31
        .end    yield
```

Chapter 3

Meldc_user.h

3.1 Introduction of meldc_user.h

In this section we will discuss the file meldc_user.h which contains many definitions for the MeldC language. That is, this file defines the data structures for MeldC to use. Since any programs data structure is important, we have devoted an entire chapter to covering this material, since it obviously even more important for something on a scale as large as this. Meldc_user.h defines such things as:

- The frame structure for each frame allocation mechanism
- The object structure prefix
- The method structure
- The basic frame structure used for the MeldC function call

We will discuss each of these parts separately and as in much detail as needed to understand the rest of MeldC later.

3.2 Headers of meldc_user.h

```
#ifndef MELDC_USER_H
#define MELDC_USER_H

#include <sys/time.h>
#include "malloc.h"
#include <signal.h>
```

```
#ifndef mips
#define TRUE        1
#define FALSE       0
#else
#define TRUE        1
#define FALSE       0
#endif

#ifdef PREEMPTIVE
extern int signal_mask;
#endif

#define _OBJECT  _obj
#define _FRAME _frame

#define MAX_HOSTNAME 100
#define MAX_METHOD_NAME 256

#define  MAX_STACK_FRAME_SIZE 64

/*
 * It is used by the persistent object.
 */
#define MAX_FLUSH_RATE 1024

#define REQUIRED 0
#define OPTIONAL 1

#define _VIRTUAL_OBJ  _obj->$.virtual_self

struct _frame;
struct _object;
struct _elem_tab;
struct _method;
struct global_lock;

/*
 *  States for function invocation (each frame)
 */
enum function_states {
  S_NEW = 0,           /* will be executed by eval()    */
```

```
  S_READY = 1,          /* will be executed by re_eval() */
  S_WAIT = 2,           /* wait for func call return */
  S_EXIT = 3,           /* IF CHANGED, NEED TO CHANGE */
                        /* ASSEMBLY CODE BY HAND */
  S_BLOCKED = 4,        /* wait for I/O  */
};

/*
 * The class instance.
 */
enum instance {
  C_FEATURE_NAME = 0,
  C_NAME         = 1,
  C_PRE_SYNC     = 2,
  C_POST_SYNC    = 3,
  C_STR_METHOD_TABLE = 4,
  C_SYM_METHOD_TABLE = 5,
  C_PARENT_OBJECTS = 6,
  C_INSTANCE_VAR_TABLE = 7,
};


/***************************************************/
/*                                                 */
/*  MACROs used in code generation MeldC compiler */
/*  generates code using these macros, so the     */
/*  meldC runtime designers can modify the code   */
/*  by changing macro definition and without      */
/*  modify the compiler directly.                 */
/*                                                 */
/***************************************************/

/*
 * Macros for the C code generated for the send statement.
 */

#define URGENT 6
#define METACLASS_PRIORITY 3
#define CLASS_PRIORITY 2
#define OBJECT_PRIORITY 1
#define FUNC_PRIORITY 1
#define NO_PRIORITY -1
```

```
#define SYNC 1
#define ASYNC 2


#define ATTACH 1
#define DETACH 2
```

3.3 FRAME_BODY & NEW_FRAME

3.3.1 Description of FRAME_BODY & NEW_FRAME

The first part of interest is in the defining of the FRAME_BODY and NEW_FRAME. Both the FRAME_BODY and NEW_FRAME are macros used by MeldC. The NEW_FRAME macro will allocate the FRAME_BODY structure, depending on which frame allocation mechanism is to be used, which depends on the stack type in use. The frame allocation mechanisms are defined for the following stack methods:

- Heap Base
- Circular Interleaving
- Stack Base
- Stack Base Slow

The Heap Base and Circular Interleaving stack methods will have the NEW_FRAME allocated by the DEFAULT_APPROACH. The default approach will allocate the NEW_FRAME with the alloc_frame call. Then it will call the FRAME_BODY macro to initialize the frame structure.

The Stack Base method checks for the call type for the frame, SYNC or ASYNC, then depending on which type of call was made, it calls either alloc_frame or a_new_frame. The Stack Base Slow method also checks the call type, but then uses either the alloc_frame or a_new_thread calls, each of which differs in the way the alloc_frame call is made. Both methods will also call the FRAME_BODY macro when done allocating the memory to fill in the structure.

The FRAME_BODY macro is used to simply fill in the generic frame structure that is used by all of the allocation methods. The frame structure fields are as follows:

- obj is a pointer to the object it belongs to.

- func is a pointer to a function that the frame will execute.
- state is the status of the frame and will be set to S_NEW when first initialized.
- size is the size of the frame.
- save_frame_size is the size of the frame that was saved when a context switch was executed and the frame was copied from the C stack to the frame structure.
- save_registers is used to save the state of the registers also when a context switch is done.
- mp is a pointer to the method structure, which provides for us, one level of indirection.
- sync_mode is used to tell how the frame was called, SYNC or ASYNCH.
- caller is used to hold the value of who called the frame.
- next is used to point to the next frame.
- add_memory is used to state if more memory is needed by the frame.
- msg and tmp_msg.

It should be noted that the NEW_FRAME macro will be called every time a MeldC object sends a message to another object. The compiler will translate the message to NEW_FRAME, if it is being statically bound, else the compiler will translate the message to BINDING, if it is being dynamically bound.

The NEW_DYNAMIC_FRAME should also be mentioned. This macro is used since, we can not allow a C function to allocate a MeldC frame in the Stack Base Slow implementation. If this was allowed, then when the C function returns, the newly allocated MeldC frame would be lost.

3.3.2 Code of FRAME_BODY & NEW_FRAME

```
#ifdef FRAME_ALLOCATION_HEAP_BASE

#ifndef DEFAULT_APPROACH
#define DEFAULT_APPROACH
#endif
```

```
#endif

#ifdef CIRCULAR_INTERLEAVING

#ifndef DEFAULT_APPROACH
#define DEFAULT_APPROACH
#endif

#endif


#define FRAME_BODY(p, s, o, f, m,method_ptr,type)       \
                   (p)->$.P = $GLOBAL$HOPELESS$P,       \
                   (p)->$.obj = (struct _object *) o, \
                   (p)->$.func = (int (*)()) f,  \
                   (p)->$.state = S_NEW, \
                   (p)->$.priority = NO_PRIORITY,\
                   (p)->$.errno = 0, \
                   (p)->$.size = s, \
                   (p)->$.save_frame_size = 0, \
                   (p)->$.save_registers[0] = 0, \
                   (p)->$.mp = method_ptr, \
                   (p)->$.sync_mode = type, \
                   (p)->$.next = (struct _frame *)NULL, \
                   (p)->$.caller = m,\
                   (p)->$.str_or_sym = SYM_MSG,\
                   (p)->$.msg = NULL,\
                   (p)->$.add_memory = NULL, \
                   (p)->$.tmp_msg = NULL


#ifdef DEFAULT_APPROACH

#define NEW_FRAME(p, s, o, f, m,method_ptr,type) \
                 $GLOBAL$HOPELESS$P = p,\
                 p = (struct _frame *)alloc_frame(s),    \
                 FRAME_BODY(p, s, o, f, m,method_ptr,type)


#define FREE_FRAME(p) free_frame(p)
#endif
```

```
#ifdef STACK_BASE
extern int a_new_thread();

#define NEW_FRAME(p, s, o, f, m,method_ptr,type) \
  $GLOBAL$HOPELESS$P = p,\
  ((type == SYNC) ? \
  ((p = (struct _frame *)alloc_frame((s / 8 ) * 8 + 8)), \
  (struct _frame *)0) :                                  \
  (p = (struct _frame *) a_new_thread(s))),              \
  FRAME_BODY(p, s, o, f, m,method_ptr,type)


#define NEW_DYNAMIC_FRAME(p, s, o, f, m,method_ptr,type) \
  $GLOBAL$HOPELESS$P = p,\
  ((type == SYNC) ? \
  ((p = (struct _frame *)MALLOC((s / 8 ) * 8 + 8)), \
  (struct _frame *)0) :                             \
  (p = (struct _frame *) a_new_thread(s))),         \
  FRAME_BODY(p, s, o, f, m,method_ptr,type)

#endif

#ifdef STACK_BASE_SLOW

extern struct _frame * a_new_thread();

#define NEW_FRAME(p, s, o, f, m,method_ptr,type) \
  $GLOBAL$HOPELESS$P = p,\
  ((type == SYNC) ? \
  ((p = (struct _frame *)alloc_frame((s/8)*8+8,i
  &(_frame->$.ssp))), \
  (((struct _frame *)p)->$.ssp=_frame->$.ssp), \
  (((struct _frame *)p)->$.original_ssp =
         ((struct _frame *)m)->$.original_ssp),\
  (struct _frame *)0) :                        \
  (p = (struct _frame *) a_new_thread((s/8)*8+8))), \
  FRAME_BODY(p, s, o, f, m,method_ptr,type)

#endif
```

3.4 ENQUEUE_FRAME

3.4.1 Description of ENQUEUE_FRAME

The macro definition for the enqueuing of a frame, ENQUEUE_FRAME, is done next. The enqueueing of a frame is the process of putting a frame that is ready to be executed on to the ready queue. Here, the frame will wait in the queue until the scheduler, the driver function (discussed in Section 2.6), picks the frame to execute. At such a time it will then be executed until it is blocked, does a delayuntil or completes it's execution.

To enqueue the frame is just a matter of putting the newly defined frame on the end of the ready queue using the next pointer. This call will mostly be used by the compiler generated code.

3.4.2 Code of ENQUEUE_FRAME

```
/*
 * since we allocate the frame fisrt, if any of the formal
 * paramtere require of making a funciton call, we need to
 * save our original frame before it got destroy.  The last
 * statement in the following macro restore the frame.
 */
#define ENQUEUE_FRAME(p) (p != 0 ?
                     ((p->$.next = (struct _frame *) 0), \
                     (enqueue(p)), \
                     (p = p->$.P)): 0)
```

3.5 YIELD & YIELD_NO_WAIT

3.5.1 Description of YIELD & YIELD_NO_WAIT

The YIELD and YIELD_NO_WAIT macros come next. The YIELD will give up the control of a thread and then suspend the execution of that thread. The thread's status at this time will then be set to S_WAIT. The caller frame will then wait for the completion of the callee frame, at which time the caller will then resume execution. This mechanism will be used to simulate the mechanism for a synchronous call.

The YIELD_NO_WAIT macro gives the ability of a thread to give up the control of execution for one scheduling cycle. The thread will not be suspended, but is simply moved to the end of the ready queue.

3.5.2 Code of YIELD & YIELD_NO_WAIT

```
#define YIELD (((_frame->$.state != S_BLOCKED) ?
                (_frame->$.state = S_WAIT) :
                (_frame->$.state = S_BLOCKED)) , yield())

#define YIELD_NO_WAIT (((_frame->$.state != S_BLOCKED) ?
                         (_frame->$.state = S_READY) :
                         (_frame->$.state = S_BLOCKED)),
                         yield())
```

3.6 DELAYUNTIL & RESPOND

3.6.1 Description of DELAYUNTIL & RESPOND

Next we have the DELAYUNTIL macro. When a user specifies a delayuntil in their MeldC program, that delayuntil will get translated to DELAYUNTIL. The macro will then execute the _delayuntil call. If _delayuntil returns the status S_WAIT, then the DELAYUNTIL macro will issue a YIELD. This will then suspend the execution as discussed in Section 3.5.

The RESPOND macro will be called from the translation of the respond call in a user's MeldC program. This macro will then execute the _respond call.

For more on either the delayuntil or respond calls, see Chapter 2, the Kernel chapter.

3.6.2 Code of DELAYUNTIL & RESPOND

```
enum msg_types {
 SYM_MSG = 0,
 VIRTUAL_SYM_MSG = 1,
 STR_MSG = 2,
 VIRTUAL_STR_MSG = 3,
 SYNC_MSG = 4,
 ASYNC_MSG = 5,
 EITHER_MSG = 6,
};

/*
 * we need to disable to preemptive scheduling, because,
 * _delayuntil and yield has to be issued atomically.
 */
```

```
#ifdef PREEMPTIVE

#define DELAYUNTIL_STR(msg, obj)
          {int enable; enable = signal_mask; \
           enter_meldc_kernel() ; \
           if (_delayuntil (msg, obj, STR_MSG) == S_WAIT)\
           ( YIELD );  else if (!enable)
           exit_meldc_kernel();}

#define DELAYUNTIL_SYM(msg, obj)
          {int enable; enable = signal_mask; \
           enter_meldc_kernel() ; \
           if (_delayuntil (msg, obj, SYM_MSG) == S_WAIT)\
           ( YIELD );  else if (!enable)
           exit_meldc_kernel();}
#else

#define DELAYUNTIL_STR(msg, obj)
          if (_delayuntil (msg, obj, STR_MSG) == S_WAIT)\
          ( YIELD )
#define DELAYUNTIL_SYM(msg, obj)
          if (_delayuntil (msg, obj, SYM_MSG) == S_WAIT)\
          ( YIELD )
#endif

#define RESPOND_STR(msg, obj) _respond (msg, obj, STR_MSG)
#define RESPOND_SYM(msg, obj) _respond (msg, obj, SYM_MSG)
```

3.7 BINDING & ALLOCTMP_STR Macros

3.7.1 Code of BINDING & ALLOC_TMP_STR Macros

```
#define TMP_STR_SIZE 1024

#define ALLOC_TMP_STR(s) s = (char *) malloc (TMP_STR_SIZE)
/* PLEASE DON'T USE malloc(), should have a fast and
 * non-preemptive one
 */

/*
```

```
 * usr_ret_type = the left hand side user return type.
 */
#ifndef STACK_BASE
#define BINDING(p, obj, ret, usr_ret_type, msg_type, flag) \
        p = $MeldC$DynamicBinding$internal_binding(NULL,p,
                         obj,ret,usr_ret_type,msg_type,flag)
#else

#define BINDING(p, obj, ret, usr_ret_type, msg_type, flag) \
        p = $MeldC$DynamicBinding$internal_binding(NULL,p,
                         obj,ret,usr_ret_type,msg_type,flag),\
        ((flag == SYNC_MSG) ? \
        p = reallocate_on_stack(p) : p)

#endif
```

3.8 Locking Macros

3.8.1 Description of Locking Macros

The LOCK and UNLOCK macros are defined next for the MELDC atomic lock implementation. When an atomic lock is applied to a piece of code, it will protect the instance variables within an object. The atomic lock will ensure only synchronous access to those instance variables. The UNLOCK macro will remove this lock set.

There is also the GLOBAL_LOCK and GLOBAL_UNLOCK macros. This global lock will allow only one object at a time to execute when the global lock has been set. This means that all of the frames of the same thread, equal thread id's, are only allowed to execute. This has a major drawback though. This global lock could possibly cause a deadlock in the system. This will happen when a frame, within the thread that has the global lock, tries to call another frame that does not have the same thread id. The frame with the lock will be suspended until the called frame is done. The called frame will never execute though since the lock has been set. There is also the drawback of only one frame at a time being allowed to execute. This defeats the purpose of a multiple thread environment. That is, allowing numerous frames to execute at one time.

3.8.2 Code of Locking Macros

```
extern int atomic_lock_count ;
```

```
#define LOCK (ENTER_CLIB _obj->$.lock->status++,
             atomic_lock_count++,
             _obj->$.lock->threadid =
             _frame->$.threadid EXIT_CLIB)

#define UNLOCK (atomic_lock_count--,
               (--_obj->$.lock->status == 0 ?
               (ENTER_CLIB _unblock_frame(_obj)
               EXIT_CLIB) : 0))

/*
 * User is not allow to grab the global lock, until
 * all the atomic lock is released.  Otherwise,
 * there could be a deadlock.
 */

#define GLOBAL_LOCK while( (atomic_lock_count) && \
       ( (_obj->$.lock->status) && \
       (atomic_lock_count != 1) && \
       (memcmp((char *)&(_obj->$.lock->threadid), \
       (char *)&(_frame->$.threadid), \
       sizeof(struct _threadid))))) \
       YIELD_NO_WAIT;\
       (ENTER_CLIB global_lock.g_lock.status++, \
       global_lock.g_lock.threadid =
                   _frame->$.threadid EXIT_CLIB)

#define GLOBAL_UNLOCK
        (--global_lock.g_lock.status == 0 ?
        (ENTER_CLIB global_unlock() EXIT_CLIB):0)
```

3.9 Object Id, Thread Id & Lock Queue Structures

3.9.1 Description of Object Id, Thread Id & Lock Queue Structures

Next, we define the objects id, _objid, and the threads id, _threadid. The variables in the object id:

- time
- pid
- hostid

are shared by all objects within the same process. For this reason, the counter variable was needed to make objects unique. The thread id of a frame also has a counter for uniqueness among threads of the same object.

Notice that the lock queue, _lock, and the ready queue, _queue, are also defined here at this point. The lock queue is used to store all of the frames that have been blocked by the lock. It should be noted that the lock queue, _lock, is shared among all objects in the same inheritance hierarchy. The ready queue is used to hold all of the frames that have S_READY status. That is, the frames that are ready to be executed and are waiting their turn.

3.9.2 Code of Object Id, Thread Id & Lock Queue Structures

```
struct _object;

struct _objid
{
    long            counter;
    struct timeval  time;
    int             pid;
    int             hostid;
};

/*
 * We need to generate a unique thread id.
 */
struct _threadid
{
    struct _objid objid;
    long          counter;
};

/*
#define UNLOCK_S 0
#define LOCK_S 1
*/
```

```
struct _lock
{
    int              status;
    struct _threadid threadid;
             /* ptr or the whole thing */
};

/*
 * queue struct
 */

struct _queue {
  struct _frame *head;
  struct _frame *tail;
};

struct global_lock
{
struct _lock g_lock;
struct _queue g_queue;
};
```

3.10 _shadow_obj_st

3.10.1 Description of _shadow_obj_st

This next structure, _shadow_obj_st, is the basic structure used by shadow objects, discussed in Chapter 5. The structure contains a pointer to the shadow object, which is just a regular object, so that it can by referenced by the MeldC system. The structure also contains the entry_point and exit_point, both of which are used for message redirection. The structure then has the init_point field which is used during the attachment of the shadow to initialize it. The term_point is used to clean up the shadow object when the shadow is to be detached from the object. And finally, the structure contains the next shadow object structure, which is used to point to another shadow object attached to the same object. It should be noted that this structure is also shared among all objects within the same inheritance hierarchy.

3.10.2 Code of _shadow_obj_st

```
struct _shadow_obj_st {
    struct _object * shadow_obj;
    char entry_point[_NAME_SIZE];
    char exit_point[_NAME_SIZE];
    char init_point[_NAME_SIZE];  /* During the attachment,
                                     this routine
                                     will be called. */
    char term_point[_NAME_SIZE];  /* During the detachment,
                                     this routine
                                     will be called. */
    struct _shadow_obj_st  *next;
};
```

3.11 OBJ_STRUCT_PREFIX

3.11.1 Description of OBJ_STRUCT_PREFIX

The next major component is the OBJ_STRUCT_PREFIX structure. It is used to hold the object information needed for numerous tasks on the objects. The fields of this structure are as follows:

- The objid is used to give the object an identity
- The class_ptr points to the class object
- The virtual_self is used by the override semantics of inheritance. (i.e. send foo() to $vself). If the programmer decides to use $vself, instead of $self, the system will bind the message at runtime, using the inheritance hierarchy.
- The next pointer points to the next object in the linked list that is used for the object's lock queue.
- The lock is used to determine whether the object has a lock on some piece of code
- The lock_queue is used to keep track of who gets the lock next
- The name is used to identify the object. It will not necessarily be unique though. The user does have the option of assigning a name to the object. If the user does not assign a name, the system will create a name. The name will be created in the following way:

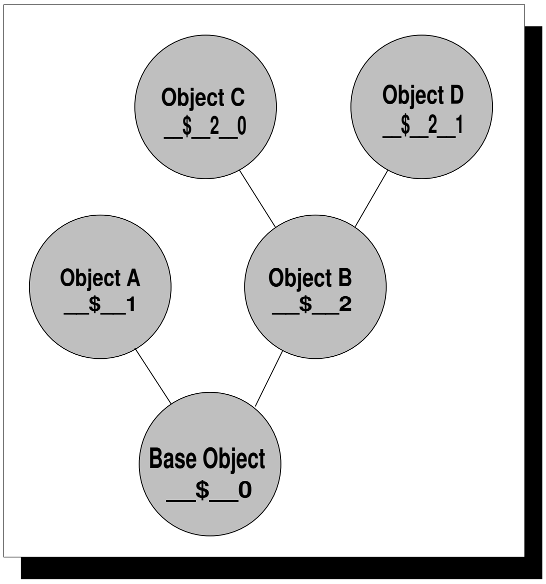

Figure 3.1: Automatic MeldC naming scheme for objects

 - If the object is the base object, the parent object, then the name will be classname.number.
 - If the object is not the parent object, not the base object, but is in the next level up, then it will be named "__$__x". The value of x will be the number assigned according to the order in which it was created on this level, starting with 1. For the next level up, the name "__$__x__y" will be used. The value of x is the parent of the object and the value of y is the number assigned according to the order of the objects created on this level. This naming scheme will continue in this fashion.

   See Figure 3.1 for an example of this scheme. In this case, object A was created off of the base object and was created first. Then object B was created next. Objects C and D were both created above object B, so inherited the basic name and then had the order in which they were created added onto the end of the name. Note that the base object can take on the name "__$__0".

- The shadow_obj_struct is a pointer to the shadows of the object. The shadows are basically used for the handling of messages for an object without the object's knowledge.

3.11.2 Code of OBJ_STRUCT_PREFIX

```
#define _NAME_SIZE 64

enum obj_field {
  OBJ_ID            = 0,
  OBJ_SHADOW_OBJ    = 1,
  OBJ_PROTOCOL_OBJ  = 2,
  OBJ_CLASS_PTR     = 3,
  OBJ_NAME          = 4
};

#define OBJ_STRUCT_PREFIX  struct {\
    struct _objid          objid;\
    struct _shadow_obj_st  *shadow_obj_struct; \
    struct _object         *protocol_obj;\
    struct _object         *class_ptr; \
    struct _object         *virtual_self; \
    struct _lock           *lock;\
    struct _queue          *lock_queue;\
    struct _object         *next;\
    int                    object_priority;\
    int                    ref_count; \
    int                    pending_message; \
    M_MEM                  add_memory;  \
    char                   name[_NAME_SIZE];\
    } $;

#define OBJ_STRUCT_PREFIX_STR "OBJ_STRUCT_PREFIX"

struct _object
{
  OBJ_STRUCT_PREFIX
  /* point to the next object in the same class */
};
```

3.12 _Para

3.12.1 Description of _para

Next we come to the parameter structure, _para, which stores all of the information regarding a parameter. This table is used to hold all of the parameters of a function. The parameter table also has the return type within it's structure. It should also be noted that the parameter table is terminated by an entry that conatins all zeros. Here are the parameter structure fields:

- The name is the parameters name assigned to it.
- The decl_type is the type of the variable in postfix char string format. That is, if the type of the parameter is char *, it will be seen as * char.
- The decl_type_ptr is a pointer to the _elem_tab structure, see Section 3.16. This table is used to store the type description that the user used within the program. It also stores the size of the variable.
- The offset tells where the parameter is inside of the frame. The value will actually be added to the address of the frame to find the beginning of the parameter.

3.12.2 Code of _para

```
struct _para
{
    char                *name;
    struct _elem_tab    *decl_type_ptr;  /* COME BACK */
    char                *decl_type;
    int                 offset;       /* offset in its
                                         frame struct */
    int                 is_optional;
};
```

3.13 _Method

3.13.1 Description of _method

The method table is also defined here. For each class, there is a method table which contains all of the methods that are defined in that class. The table is terminated by an entry that contains all zeros. From this table, one can find the method description about a particular method in that class. The method table is most useful during dynamic binding. At that time the runtime system will search through the method table, identify the method, perform the type checking on the parameter and finally allocate the frame and make the function call. Note that this structure is very useful for obtaining one level of indirection. The fields within the structure are:

- The method name is in the name field.
- The return_type is the basically the return type of the value to be returned when done.
- The func pointer points to the function to execute.
- The frame_size is the size of the frame.
- The para_no is the number of parameters that the function takes.
- The func_disp is the displacement of the function into the frame. The value will actually be added to the frame address to find the beginning of the function.
- The para_table is a pointer to the _para structure containing the description of the parameters, see the previous section, Section 3.12.

3.13.2 Code of _method

```
struct _method
{
    char                    *name;
    char                    *return_type;
    int                     (*func)();
    int                     frame_size;
    int                     para_no;
    struct _para            *para_table; /* changed from '**'
                                             - tlw 10/30/90 */
    int                     func_disp; /* The displacement
                                           into the object
                                           struct that point
                                           to the method
                                           table.*/
    int                     opt;
};
```

3.14 FRAME_STRUCT_PREFIX

3.14.1 Description of FRAME_STRUCT_PREFIX

The next major component is the FRAME_STRUCT_PREFIX used to define the basic set of needed variables required for a frame. The variables are as follows:

- The fs field is the frame status to show if a frame is free or if it has already allocated.
- The BACK_POINTER macro is used for the circular interleaving stack method, which will give the bsp pointer value.
- The state field can be either:
  - EXIT – The frame has finished running
  - READY – The frame is ready to run and is on the queue waiting for the scheduler to pick it to allow it to execute
  - WAIT – The frame has paused for a signal
  - NEW – The frame has just been created and has not run yet
- The obj pointer points to the object.
- The func pointer points to the function to execute.
- The reentry_addr is the address of where to pick up execution again when the frame is restarted if suspended and not yet finished.
- The save_frame_size is the size of the C stack that has been saved in the frame. This is done when a context switch is done.
- The caller is a pointer to the frame that called this one. If it has a value of NULL, it means that the frame was called asynchronously.
- The STACK_POINTER macro is used in both the Stack Base and Stack Base Slow methods. It will give the value for the sp and fp pointers for the Stack Base method and will give the sp value for the Stack Base Slow method.
- The save_registers field is used to save the registers of the frame when a context switch is done.
- The size field is the size allocated for the frame which is a fixed amount of 256 bytes as of now.

- The mp field is a pointer to the method structure.
- The threadid field is a pointer to the thread information.
- The P field is used as the temporary pointer, used usually by the compiler. If this frame makes a call to another function, the new function's frame will be allocated in P.
- The sender pointer points to the caller.
- The sync_mode field contains a value to show if it is sync or async.
- The next value is a pointer to the next frame in the linked list.
- The str_or_sym is used to determine if a message is a string or symbol message. A symbol message is just a regular message. For example:

  obj.hello();

  The symbol hello, is considered a symbol message. The reason this is so, is for the reason that we do not use the method name hello, but instead use the address which is contained in the variable hello.

  The string message is a character string message. The format is usually:

  obj.<char_string>

  where char_string can be "hello" or a variable that is char *. An example of this is:

  char * foo = "method1";

  obj.foo;

This will send a string message, which is in the variable foo, to the obj.

obj."hello"

This will send a string message "hello" to the obj. Note that you can not pass parameters into a string message function. The function declaration for the string message can look like the following:

```
/*
 * when the string hello arrives at this object, invoke
 * this method.
 */
int "hello"()
    {
    }

/*
 * String selectors also take regular expressions.
 *
 * The following method will be invoked if the string
 * message has "foo" as its last three characters.
 */

int ".*foo"()
    {
    }
```

- The add_memory field is a MeldC internal memory management structure.

- The msg field is a pointer to the message received by the frame. The message contains the parameters that we will pass into the function, including optional parameters.

- The tmp_msg is temporary storage, used by the compiler. This is useful for a function call such as:

obj.bar(int x, int y, optional)

Let us assume, that the formal parameters of bar looks like bar(optional). Let us also assume, that the message is dynamically binded. If the above message is statically binded, then we do not have any problem, since we can store the new optional parameters into the new frame. For dynamic binding, we do not know the frame until runtime. The compiler can not wait until runtime to generate the code. The compiler will then need to construct a new optional structure by adding more information to the old structure. The compiler can not destroy the old structure, because some other function might need it later. The compiler needs a place to store the new optional parameter structure. So, the compiler will create the optional structure in the tmp_msg. In the metaClass binding code, see Section 4.1.28, the information in the tmp_msg will be copied and transferred into the new frame's msg field.

3.14.2 Code of FRAME_STRUCT_PREFIX

```
struct _msg;

#ifdef STACK_BASE

#define STACK_POINTER \
  char ** ssp;\
  char ** sfp; \
  char *original_ssp; \

#elif STACK_BASE_SLOW

#define STACK_POINTER \
  int *save_stack_frame[MAX_STACK_FRAME_SIZE]; \
  char ** ssp;\
  char *original_ssp; \

#endif

#ifndef STACK_POINTER

#define STACK_POINTER \
```

```
int *save_stack_frame[MAX_STACK_FRAME_SIZE]; \

#endif/*ifdef STACK_BASE */

/*
 * Circular interleaving stack require one
 * additional pointer.
 */

#ifdef CIRCULAR_INTERLEAVING

#define BACK_POINTER \
  char ** b_fp; \

#endif

#ifndef BACK_POINTER
#define BACK_POINTER
#endif


#define FRAME_STRUCT_PREFIX  struct {  char **fp; \
  int  fs; \
  BACK_POINTER \
  enum function_states state; \
  struct _object *obj; \
  int (*func)(); \
  int (*reentry_addr)(); \
  int  save_frame_size; \
  struct _frame *caller; \
  int return_value; \
  STACK_POINTER \
  int global_variables[8];\
  int save_registers[18]; \
  int size; \
  struct _method *mp;\
  struct _threadid threadid; \
  int priority; \
  int errno; \
  struct _frame * P; \
  struct _object *sender; \
  int sync_mode; \
```

```
  struct _frame *next; \
  enum msg_types str_or_sym; \
  M_MEM add_memory;     \
  struct _msg *msg; \
  struct _msg *tmp_msg; \
  } $;

/* Used in code generation   */

#define FRAME_STRUCT_PREFIX_STR  "FRAME_STRUCT_PREFIX"


struct _frame {
  FRAME_STRUCT_PREFIX
};
```

3.15 Miscellaneous structures

3.15.1 Code of miscellaneous structures

```
struct _class_init
{
    char                   *name;
    char                   *feature_name;
    int                    persist_flag;
    int                    (*init)();  /* init code */
    int                    obj_size;
    int                    method_no;
    int                    sym_method_no;
    struct _method         *sym_method_table;
    int                    str_method_no;
    struct _method         *str_method_table;
    struct _para           *instance_var_table;
    void                   (*pre_sync_func)();
    void                   (*post_sync_func)();
    char                   *parents;
};

struct _class_info
{
    char                   *class_name;
```

```
    void                   (*pre_sync_func)();
    void                   (*post_sync_func)();
    char                   *parents;
};

struct _selector
{
    char           *_selector;
    char           **_str;
    struct _object *_sender;
};

struct _object_decl
{
    char           *classname;
    char           *objname;
    struct _object **obj;
    int            flag;
    char           *hostname;
};

struct _sender
{
    int            typ;
    struct _object *obj;
};
```

3.16 _Elem_tab

3.16.1 Description of _elem_tab

This structure is used to describe an element. The fields of the elem_tab structure are:

- The size variable contains the size of the element.
- The offset variable is used to tell where the element is.
- The postfix_type points to the element in postfix form.
- The c_type_desc tells what type the element is, (integer, char, ...).

- There is also the elem_tab structure as a field of the structure itself. This is used when a structure contains a substructure within it. We will the let the elem_tab be used to hold the description of all of the elements in the substructure.

3.16.2 Code of _elem_tab

```
/* Function Declaration */

struct _frame *binding();

/* Extern Variables */

extern struct _object *_obj;
extern struct _frame  *_frame;
extern struct _frame  *$P;  /* not used, In malloc.c */
enum function_states _delayuntil();

struct _elem_tab
    {
    char * c_type_desc;
    char * postfix_type;
    int  size;
    int  offset;
    struct _elem_tab *elem_tab;
    };
```

3.17 _Arg_desc

3.17.1 Description of _arg_desc

This structure is used to describe an argument. This is done using the elem_tab structure, just discussed, and the offset variable. The offset variable contains a value that is used to find the argument in the message. This value is from the beginning of the message structure.

3.17.2 Code of _arg_desc

```
struct _arg_desc {
   struct _elem_tab *type;
   int dereference;
   int offset;
```

```
};
```

3.18 _Msg

3.18.1 Description of _msg

The final structure of importance defined within this file is the message, _msg, structure. This structure is used by dynamic binding, as well as for the use of passing optional parameters. The fields are as follows:

- The type field is used to determine if a message is from an internal or an external source.
- The size of the message is stored in _size.
- The message type, _msg_type, is either a symbol message or a string message.
- A pointer, _msg_name, points to the name given to the message.
- The _return_type field is the return type to the caller.
- The _return_addr is a pointer to the return address.
- The number of arguments is saved in the _num_arg field.
- The para field contains the number of parameters stored in the structure.

3.18.2 Code of _msg

```
struct _msg {
   int _type;           /* 0 for internal, 1 for external */
   int _size;
   int _msg_type;       /* SYM_MSG or STR_MSG */
   char *_msg_name;
   struct _elem_tab *_return_type;
   char *_return_addr;
   int _num_arg;
   struct _arg_desc _para[1];
};
```

3.19 Message & Argument Macros

3.19.1 Description of Message & Argument Macros

The last thing of importance in this chapter are the macros used for returning values contained in the message and for returning the argument values. The message macros are:

- MSG_TYPE – This macro will return the message type.
- MSG_SIZE – This macro will return the size of the message.
- MSG_NAME – This macro will return the name given to the message.
- MSG_RETURN_TYPE – This will return the type of the return value.
- MSG_RETURN_ADDR – This will return the address of where the return value is to be sent.
- MSG_NUM_ARG – This will return the number of arguments that are contained within the message.
- MSG_ARG_DESC – This will give the _para structure, discussed above, so that the argument descriptions can be found.

The argument macros take an integer, passed as a parameter, that represents the arguments slot number in the list of arguments. The slot number of the argument is a well known number, since we know the order of the arguments. The macros for the arguments are:

- ARG_SIZE – This macro will return the argument size of the desired argument in slot i.
- ARG_POSTFIX_TYPE – This will return the postfix type for the argument in slot i.
- ARG_TYPE – This will give us the argument type, (int, char, ...)
- ARG_ADDR – This will give us the address of the variable. This is done using the address of the message + the offset of the argument.

3.19.2 Code of Message & Argument Macros

```
/* Macros to use  the packed arguments */

#define NEW_MSG(i)  (i<BUFSIZ?(struct _msg *)
                    m_malloc(&(_frame->$.add_memory),\
                    BUFSIZ) : (struct _msg *)
                    m_malloc(&(_frame->$.add_memory),i))

#define NEW_MSG_ASYNC(fp,i)  \
            (i<BUFSIZ?(struct _msg *)
            m_malloc(&(fp->$.add_memory),\
            BUFSIZ) : (struct _msg *)
            m_malloc(&(fp->$.add_memory),i))

#define MSG _frame->$.msg

#define MSG_NUM_ARG     (MSG ? MSG->_num_arg : 0)
#define MSG_TYPE        (MSG ? MSG->_type : 0)
#define MSG_SIZE       ((MSG && MSG_NUM_ARG) ? MSG->_size :
                         sizeof(struct _msg))
#define MSG_MSG_TYPE    (MSG ? MSG->_msg_type : SYM_MSG)
#define MSG_NAME        (MSG ? MSG->_msg_name : "")
#define MSG_RETURN_TYPE (MSG ? MSG->_return_type :
                         (struct _elem_tab *) NULL)
#define MSG_RETURN_ADDR (MSG ? MSG->_return_addr :
                         (char *) NULL)
#define MSG_ARG_DESC    (MSG ? MSG->_para :
                         (struct _arg_desc *) NULL)

#define ARG_SIZE(i)                               \
  ( (MSG != NULL) && ((int) MSG->_num_arg) < i ? \
    0:                                            \
    MSG->_para[i].type->size )

#define ARG_POSTFIX_TYPE(i)                       \
  ( (MSG != NULL) && ((int) MSG->_num_arg) < i ? \
    (char  *) NULL :                              \
    MSG->_para[i].type->postfix_type )

#define ARG_TYPE(i)                               \
  ( (MSG != NULL) && ((int) MSG->_num_arg) < i ? \
```

```
    (char  *) NULL :                              \
    MSG->_para[i].type->c_type_desc )

#define ARG_TYPE_PTR(i)                           \
  ( (MSG == NULL) || MSG->_num_arg < i ?          \
    (struct _elem_tab  *) NULL :                  \
    (struct _elem_tab *) (MSG->_para[i].type) )

#define ARG_ADDR(i)                               \
 ( MSG == NULL || MSG->_num_arg <= i ?            \
    (printf("MeldC Error : Insufficent optional
             parameters \n"),\
     kill(getpid(),SIGQUIT),                      \
    (int) NULL)  :               \
    (int *)((int)(MSG->_para[i].offset) +
           ((int) MSG)) )                         \


#endif /* MELDC_H */

/* End of meldc.h */
```

Chapter 4

MetaClass.m

4.1 The MetaClass

4.1.1 Introduction of the MetaClass

The MetaClass is the class of all classes for MELDC. This class is where all other classes begin their inheritance from. That is, all other class objects are instances of the MetaClass. This will allow all of the children of the MetaClass, all other class objects, to execute some of the most important methods. These include:

- Create - used to create an new object or class.
- Destroy - used for the sole purpose of removing either an object or a class.
- GetObj - used to get a remote object from a remote source.
- PutObj - used to put a local object to a remote source.

To see how some of the MetaClass features interact with the rest of MELDC, refer to Figure 4.1.

The MetaClass is also responsible for creating the inheritance hierarchy so that it is known which variables belong to whom and who the parent of every object is. This is important for the purpose of creating a new instance and also for the purpose of the termination of an object. Since this is the back bone of MELDC, this helps demonstrate that MELDC is indeed a thoroughly object-oriented language. This class helps make MELDC easy to use and consistent.

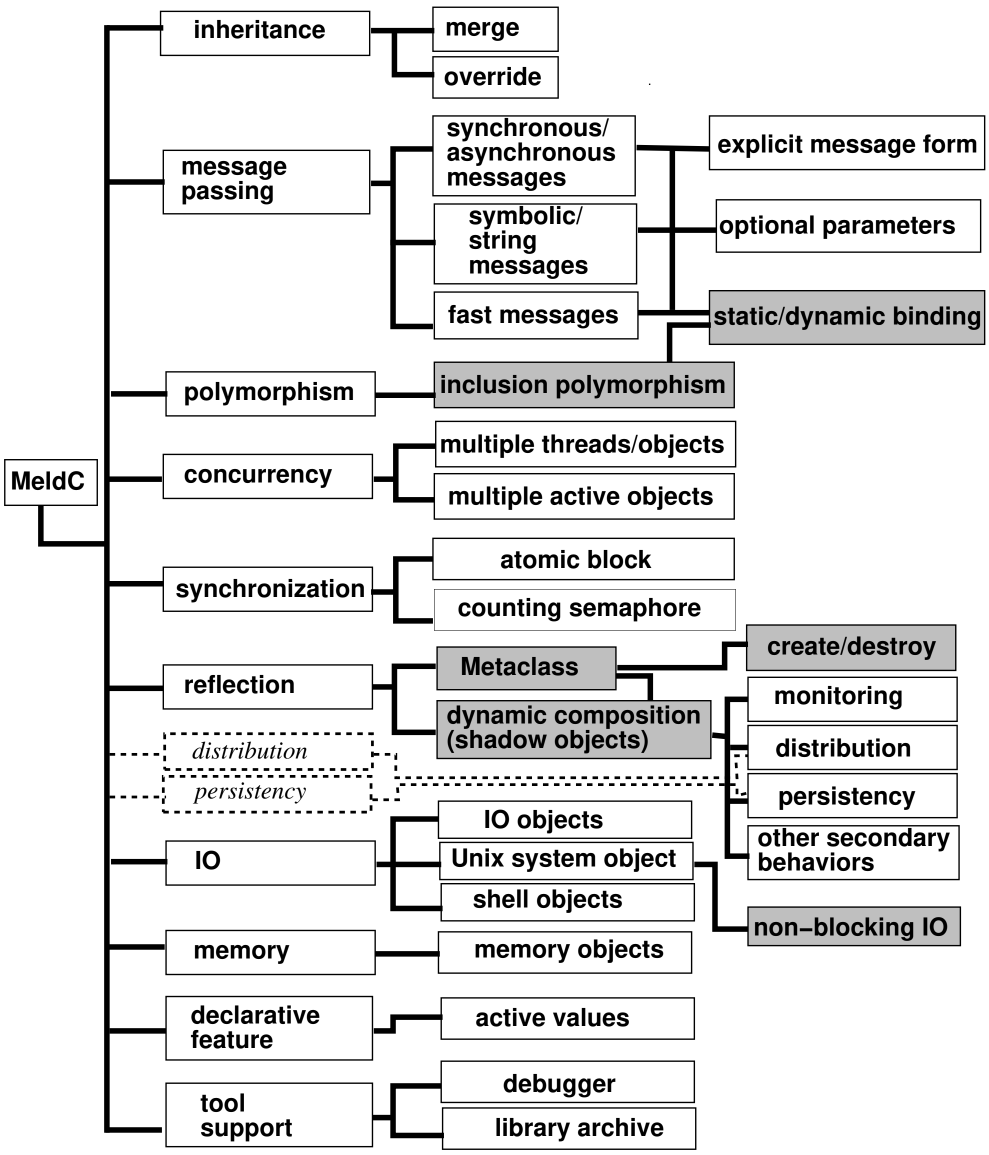

Figure 4.1: Important MetaClass Features

Since the MetaClass is itself a class and since the MetaClass creates all classes, it also means that it creates itself. This causes a problem. What we are actually asking the MetaClass to do is for the MetaClass to send a message to itself and create itself. Clearly it is not possible to do this. So for this reason, there exists in the beginning of a MeldC program a staticly created MetaClass and a few staticly created messages.

This is also why two functions have been created to initialize the MetaClass. These two functions, meta_class_init and init_method_table, are both called by the startup program, meldC.out.c, discussed next. It will statically create the MeldC MetaClass and generate the MeldC messages for the MeldC classes and global objects as well. Note that all objects in classes other than the MetaClass are created at runtime by sending a message to Create, discussed in Section 4.1.9.

4.1.2 Headers & structure of the MetaClass

```
extern int enqueue();
extern int yield();
extern int unlinearize_method_name();
extern int unlinearize_return_type();
extern int linearize_msg_size();
extern int linearize_msg();
extern int unlinearize();
extern int free_stack_frame();
extern int reset_obj_and_frame();
extern struct _frame * alloc_frame();
extern int global_unlock();
extern char * m_malloc();
extern M_MEM m_create();
extern void m_free();
extern void restore_old_frame();
extern int _unblock_frame();
extern int MSG_ADJUST_OFFSET();
extern int can_we_destroy_the_object();

feature {\sc MeldC}

interface:

exports MetaClass
exports binding
```

```
implementation:

extern struct _object * set_current_obj();
extern void Init_MsgTables();
extern struct _class_init $MeldC$MetaClass_class_init;
extern int total_number_pages;
extern void wm_create_obj();
extern struct global_lock global_lock;

DynamicBinding binding = DynamicBinding.Create();

#define SKIP_SPACE(s) for(; *s == ' '; s++);
#define NSIZE 256
#define INITIALIZE 1
#define DO_NOT_INITIALIZE 0


char *strchr();

struct _objid _objid;

struct OBJ_INSTANCE
    {
    object obj;
    struct OBJ_INSTANCE * next;
    };


class MetaClass ::=
    char                  *name;
    char                  *feature_name;
    int                   persist_flag;
    int                   (*init)();  /* init code */
    int                   obj_size;
    int                   method_no;
    int                   sym_method_no;
    struct _method        *sym_method_table;
    int                   str_method_no;
    struct _method        *str_method_table;
    struct _object        *front;
    struct _object        *rear;
    char                  *instance_var_info;
```

```
                             /* was: *para_info; */
    char                  *source_filename;
    int                   source_filesize;
    void                  (*pre_sync_func)();
    void                  (*post_sync_func)();
    struct OBJ_INSTANCE   *obj_instance = NULL ;
                             /* all the object that
                                belong to this class */
    struct OBJ_INSTANCE   *parent_objects = NULL;
                             /* class objects of
                                parent classes */
    struct _para          *instance_var_table;
```

4.1.3 MeldC.out.c

Description of MeldC.out.c

The MeldC system does not have the standard C main.c program. Instead it has the meldC.out.c startup program used to initialize the MeldC environment. This initialization, as mentioned above, also includes the calling of the MetaClass initialization function meta_class_init. This meldC.out.c program is not created by the user, but is created dynamically by the MeldC linker. Below is an example of what a meldC.out.c program will look like.

Example of a MeldC.out.c file

```
/*****************************************************/
/* Programming System Group                          */
/* Columbia Unversity                                */
/* Message-Based Thread Implementation               */
/* Version Wenwey-0.1                                */
/* generated by MeldC Linker                         */
/*****************************************************/


#include "MeldC.h"

#include "Producer_and_Consumer.h"

#include "MeldC.h"
```

```
#include "Unix.h"

extern int total_number_pages;
extern struct $MeldC$MetaClass *$MeldC$MetaClass;
#ifdef mips
extern int $MeldC$MetaClass$init();
#elif sun4
extern void $MeldC$MetaClass$init();
#endif

void
$MeldC$MeldC$Start(){
extern void $MeldC$$class_init();
extern void $Unix$$class_init();
extern void $Producer_and_Consumer$$class_init();
extern void $MeldC$$obj_init();
extern void $Unix$$obj_init();
extern void $Producer_and_Consumer$$obj_init();
$MeldC$MetaClass = (struct $MeldC$MetaClass *)
          MALLOC(sizeof(struct $MeldC$MetaClass));
$MeldC$MetaClass->$.class_ptr = $MeldC$MetaClass;
#ifndef DEFAULT_APPROACH
#endif
$MeldC$MetaClass->$.object_priority = METACLASS_PRIORITY;
mfork($MeldC$MetaClass,$MeldC$MetaClass$meta_class_init,
      sizeof(struct $MeldC$MetaClass$meta_class_init),
      METACLASS_PRIORITY);
yield();
mfork($MeldC$MetaClass,$MeldC$MetaClass$init_method_table,
      sizeof(struct $MeldC$MetaClass$init_method_table),
      METACLASS_PRIORITY);
yield();
mfork($MeldC$MetaClass,$MeldC$$class_init,
      sizeof(struct $MeldC$MetaClass$Create),
      METACLASS_PRIORITY);

DELAYUNTIL_STR("CLASS_INITIALIZED",0);
mfork($MeldC$MetaClass,$Unix$$class_init,
      sizeof(struct $MeldC$MetaClass$Create),
      METACLASS_PRIORITY);
```

```
DELAYUNTIL_STR("CLASS_INITIALIZED",0);
mfork($MeldC$MetaClass,$Producer_and_Consumer$$class_init,
      sizeof(struct $MeldC$MetaClass$Create),
      METACLASS_PRIORITY);

DELAYUNTIL_STR("CLASS_INITIALIZED",0);
yield();
_frame->$.priority = CLASS_PRIORITY;
mfork($MeldC$MetaClass,$Unix$$obj_init,
      sizeof(struct $MeldC$MetaClass$Create),
      CLASS_PRIORITY);

DELAYUNTIL_STR("OBJECT_INITIALIZED",0);
mfork($MeldC$MetaClass,$Producer_and_Consumer$$obj_init,
      sizeof(struct $MeldC$MetaClass$Create),
      CLASS_PRIORITY);

DELAYUNTIL_STR("OBJECT_INITIALIZED",0);

}
```

4.1.4 Meta_class_init

Description of Meta_class_init

The meta_class_init function is called first by the meldC.out.c program. Here we will set all of the instance variables defined in the MetaClass and the MetaClass will actually be created. These include:

- the variable name – which is the name of the class.
- the feature_name_obj_size – which is the size of the object being created.
- the method_no – which is the the total number of methods.
- the sym_method_no and the sym_method_table – which is the number in the symbol method table and a pointer to the symbol method table respectivily.
- the str_method_no and the str_method_table – which is the same as the symbol variables just stated.
- the pre_sync_func and post_sync_func – which are the pre_ and post_sync functions. These two functions are used to set instance variables in

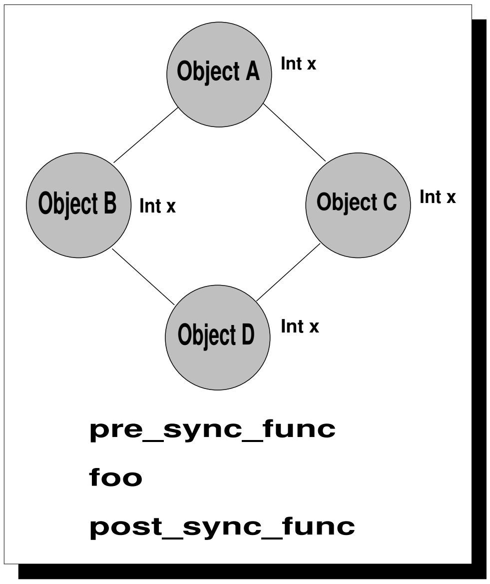

Figure 4.2: Use of the post_ and pre_sync functions for Merge inheritance

different objects to the same value when the value is changed by one of these objects. The objects will share the variables do to a Merge of the objects. The pre_sync_func will set the variables equal before a new function call. The post_sync_func will set them equal after the function call.

It should be noted that the pre_sync and post_sync functions are used only in the MERGE semantics. In the OVERRIDE semantics, both functions are set to equal to NULL.

For an example of this, see Figure 4.2. Here you can see that objects A, B, C and D all share the instance variable int x. This variable exists in the function foo. So every time function foo is called, we will do a pre_sync_func call previous to foo's execution. When foo has then completed it's execution, we will call the post_sync_func.

- the instance_var_table – the instance variable table.

Note that the variables are not unique to the MetaClass. These are the MetaClass instance variables. These variables will be set for now on using the init function, discussed in Section 4.1.6, from here on out.

Code of Meta_class_init

```
methods:

/*
 * The follwoing two functions will only be called during the
 * initalization.  They are used to initialize the metaClass.
 */

/*
 * Initialize the metaClass
 */

void meta_class_init()
    {
    struct timezone tmp;
    _objid.counter = (long)0;
    _objid.pid = getpid();
    gettimeofday(&(_objid.time), &tmp);
    _objid.hostid = gethostid();

    Init_MsgTables();

    name = $MeldC$MetaClass_class_init.name;
    feature_name = $MeldC$MetaClass_class_init.feature_name;
    persist_flag = $MeldC$MetaClass_class_init.persist_flag;
    init = $MeldC$MetaClass_class_init.init;
    obj_size = $MeldC$MetaClass_class_init.obj_size;
    method_no = $MeldC$MetaClass_class_init.method_no;
    sym_method_no =
                $MeldC$MetaClass_class_init.sym_method_no;
    sym_method_table =
             $MeldC$MetaClass_class_init.sym_method_table;
    str_method_no =
                $MeldC$MetaClass_class_init.str_method_no;
    str_method_table = $
              MeldC$MetaClass_class_init.str_method_table;
    pre_sync_func    =
                $MeldC$MetaClass_class_init.pre_sync_func;
    post_sync_func   =
               $MeldC$MetaClass_class_init.post_sync_func;
    instance_var_table =
```

```
            $MeldC$MetaClass_class_init.instance_var_table;
    }
```

4.1.5 Init_method_table

Description of Init_method_table

The other initialization function is the init_method_table. It is used for initializing the method table for the MetaClass. This must be done for both the symbol and string method tables separately. All that needs to be done, is put the method function address into the symbol table and string table one at a time with a simple while loop. It should be noted that these function addresses in MeldC use one level of indirection. This function is the same as the meta_class_init function, in that we are statically creating the method tables for the MetaClass. For all other objects besides the MetaClass, we will perform this same functionality using the Create function, discussed in Section 4.1.9, to initialize the method tables.

Code of Init_method_table

```
/*
 * Initialize the metaClass method table.
 */
void init_method_table()
    {
    int counter;
    struct _method * method;

    method = sym_method_table;

    for (counter = 0; counter < sym_method_no; counter ++)
        {
        /*
         * sym_method_table->func_disp is the displacement
         * into the object structure that will be used to
         * store the func addrsss.
         */
        *((int *)(((char *)$self)+method->func_disp))=
                                    (int)method->func;
        method ++;
        }
    method = str_method_table;
```

```
    for (counter = 0; counter < str_method_no; counter ++)
        {
        /*
         * str_method_table->func_disp is the displacement
         * into the object structure that will be used to
         * store the func addrsss.
         */
        *((int *)(((char *)$self)+method->func_disp))=
                                    (int)method->func;
        method ++;
        }
    }
```

4.1.6 Init

Description of Init

This function is used in conjunction with the Create function. It is used to initialize the instance variables of any new class or object created when the Create function is called. Note that this function does this dynamically, unlike the meta_class_init function which was done statically.

The function will set the following object fields:

- name – object name
- feature_name
- obj_size
- method_no – the total number of methods
- sym_method_no and sym_method_table
- str_method_no and str_method_table
- the pre_sync_func and the post_sync_func
- the instnce_var_table

After setting these fields, the function will then call the init_parents function, see Section 4.1.20. Here we will build the instance hierarchy.

Note that there are two different init calls, init and $init. The $init call is generated by the compiler. It will usually contain the code that initializes the instance variables. For example:

```
class foo::=

int x = 1;
int y = 2;
```

In the the $init file of foo you would see the following:

```
$<feature_name>$foo$x = 1;
$<feature_name>$foo$y = 2;
```

The init function is defined instead by the user and would be called after a successful Create. The $init will always be called. On the other hand, if the user does not define an init function, then the Create function will not call init.

Code of Init

```
/*
 * Initialize the instance varibale of the class.
 * This routine will be call when creating a class object.
 */
void init(char * i_name, char * i_feature_name,
          int i_persist_flag, int (*i_init)(),
          int i_obj_size, int i_method_no,
          int i_sym_method_no,
          struct _method *i_sym_method_table,
          int i_str_method_no,
          struct _method *i_str_method_table,
          struct _para *i_instance_var_table,
          int (*i_pre_sync_func)(),
          int (*i_post_sync_func)(), char *i_parents)
    {
    <
    name             = i_name;
    feature_name     = i_feature_name;
    persist_flag     = i_persist_flag;
    init             = i_init;
    obj_size         = i_obj_size;
    method_no        = i_method_no;
```

```
    sym_method_no    = i_sym_method_no;
    sym_method_table = i_sym_method_table;
    str_method_no    = i_str_method_no;
    str_method_table = i_str_method_table;

    pre_sync_func    = (void (*)())i_pre_sync_func;
    post_sync_func   = (void (*)())i_post_sync_func;
    instance_var_table = i_instance_var_table;

    if (i_parents)
       {
       $self.init_parents(i_parents);
       }
    >
    }
```

4.1.7 Allocate_obj

Description of Allocate_obj

The allocate_obj function is used to allocate the needed memory for an object, in order for it to be created. This is done using the MALLOC call, which is a macro to malloc1 which is another macro to:

disable signals
        malloc
enable signals

This will then return the address of where the object starts in memory. The disabling and enabling of the signals prevents the malloc from being interrupted in the middle of execution. This needs to be done since malloc is not reentry code. Within this function, we will also assign a generic object name if none was given to us by the user. It will also initialize some of the objects variables. These include:

- the protocol_obj variables – set to NULL
- the shadow_obj_struct, also set to NULL
- the next variable – also set to NULL
- the class_ptr – inherit from parent

- the ref_count variable is set to 1. This variable is used for the inheritence to keep track of the number of subobjects that has used this particular object as the parent.

Code of Allocate_obj

```
object allocate_obj(char * obj_name)
    {
    struct _object *obj;
    struct _object *tmp_obj;
    static int num;
    M_MEM object_memory = (M_MEM)NULL;

    obj = (struct _object *)
          m_malloc(&(object_memory),obj_size);
    /*
     * Initialize the object's instance variables to zero.
     */
    bzero(((char *)obj) + sizeof(struct _object),
          obj_size - sizeof(struct _object));

    obj->$.add_memory = object_memory;
    /*
     * If user did not specified the obj_name, then
     * we will use the classname as an object.
     */
    if (obj_name != NULL)
        sprintf(obj->$.name, "%s", obj_name);
    else
        sprintf(obj->$.name, "%s.%d", name, num++);

    obj->$.class_ptr = (struct _object *)$self;

    /*
     * Initialize the object.
     */
    obj->$.protocol_obj = (struct _object *)NULL;
    obj->$.shadow_obj_struct =
                  (struct _shadow_obj_st *) NULL;
    /*
     * number of locks
     */
```

```
obj->$.next = (struct _object *) 0;
obj->$.ref_count = 1;
obj->$.object_priority =
                _obj->$.object_priority -1 ;

obj->$.pending_message = 0;

return(obj);
}
```

4.1.8 Allocate_parent_obj

Description of Allocate_parent_obj

The allocate_parent_obj function will be used to create the instance of a class. The compiler will generate the inheritance information for MELDC and store it as a char string. This string will be used to construct the inheritance hierarchy of the object when the object is created. The hierarchy will then be saved for the object in the parent_objects structure. The allocate_parent_obj function, will look through this structure and create all of the objects and then using the instance_variable_table, find the addresses of where to store them. The calling sequence to create the entire hierarchy will be:

Create $\longrightarrow$ allocate_parent_obj $\longrightarrow$ CreateSubObj $\hookleftarrow$
$\hookrightarrow$ allocate_parent_obj $\longleftarrow$ CreateSubObj $\longleftarrow$ allocate_parent_obj
CreateSubObj $\longrightarrow$ ......

It will continue in this fashion until all of the objects have been created. As the process is being done, we will make sure that we have not already created the parent object. This is to prevent us from creating the same parent twice. This is necessary since, MELDC uses a virtual parent scheme. To demonstrate this idea, refer to Figure 4.3. Here you see that object D was inherited from both object B and object C. Objects C and D were inherited from object A. This would cause us to create object A twice if object A was created for both objects C and D. This is what we will prevent from

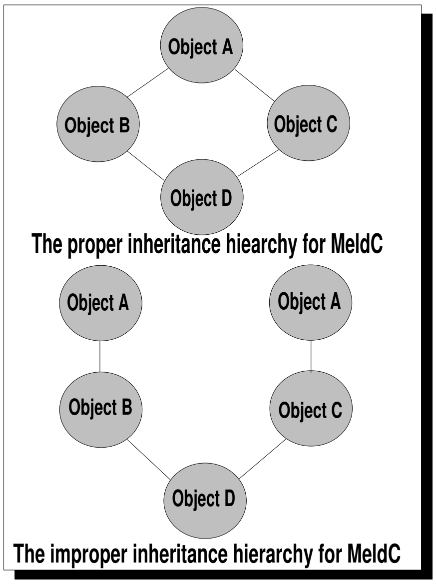

Figure 4.3: Demostrate how parents will be created.

happening by checking to see if the parent object was previously created.

Before we get into how the function works, we should point out the parameters passed to this function. The function is passed:

- name – this is the name of the object
- big_obj – this is the child object of object
- allocated_object
- optional

The optional parameter is the one we need to discuss. This parameter is really three parameters:

- initialize – should we initialize the parent objects
- new_func – the shadow_object_entry_interrupt value
- shadow_obj_struct – the shadow object structure of the object

These parameters are not used in this function, but are passed on to the CreateSubObj function. See Section 4.1.10 for how they are used.

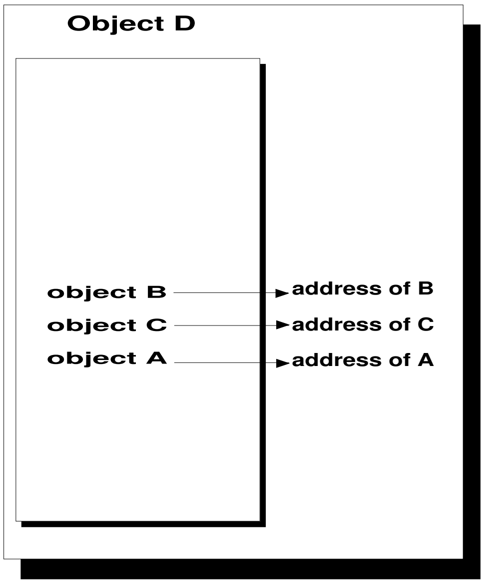

Figure 4.4: Object D's instance table.

The function will first create a unique name for each inherited object using the naming scheme mentioned in Section 3.11. Then it must make sure that there are more parent_objects, more parents to create. Then we will check to see if the parent object has already been created. If it was created, then return. If it was not created, then we must create the parent using the CreateSubObj function. This new parent will be created and then any of it's parents will also be created. At this time, it should be noted that the init function, see Section 4.1.6, will also be called to initialize the instance variables for the new parent object. We will eventually get the whole hierarchial tree of this parent back in the form of the instance_variable_table structure saved in the returned object. We will take this object and save it into the current objects instance_variable_table. To do this, we need the object name and feature name. Then we will determine the offset needed to store the parent object into the table. Object D's instance table will look like that of Figure 4.4. Here object B and C are the parents of object D. Object A is the parent of both object B and C.

Code of Allocate_parent_obj

```
/*
```

```
 * return and increment the object id.
 */

struct _objid * ret_inc_obj_id()

    {
    _objid.counter ++;
    return(&_objid);
    }

/*
 * Use for inheritance :
 *
 * This routine will create the parents of a object.
 *
 * big_obj => the child object.
 * name    => the name of the object.
 */

void allocate_parent_obj(char * name,object big_obj,
                 struct _object ** allocated_object,
                 optional)
    {
    struct OBJ_INSTANCE * pointer;
    int counter = 0;

    pointer = parent_objects;

    while (pointer != NULL)
        {
        object obj = (object)NULL;
        char new_name[20];

        /*
         * Create a unique name for each inherit object.
         */
        sprintf(new_name,"%s__%d",name,counter++);
        if (*allocated_object)
            {
            struct _object *o;

            o = *allocated_object;
```

```
        while (o != (struct _object *)NULL)
            {
            if (o->$.class_ptr ==
                  (struct _object *)(pointer->obj))
                {
                /*
                 * The object is already created.
                 */
                obj = (object) o;
                break;
                }
            o = o->$.next;
            }
        }
    if (obj != (object)NULL)
        {
        ((struct _object *)obj) ->$.ref_count ++;
        }
    else
        {
        obj = (object)(pointer->obj).
                                 CreateSubObj(new_name,
               ((struct _object *)big_obj)->$.lock,
               ((struct _object *)big_obj)->
                                   $.lock_queue,
               allocated_object,
               ((struct _object *)big_obj)->
                                        $.virtual_self,
               ((struct _objid *)MetaClass.
                           obtain_obj_info(big_obj,
                           (enum obj_field)OBJ_ID)),
               ((struct _object *)big_obj)->
                                      $.add_memory,
               $option);

        if (*allocated_object ==
                             (struct _object *)NULL)
            {
            *allocated_object = (struct _object *)obj;
            }
        else
```

```
                {
                ((struct _object *)obj)->$.next =
                                            *allocated_object;
                *allocated_object =
                                 (struct _object *)obj;
                }
            }
        /*
         * construct the name.
         */
        Struct _para * inst_table;
        char obj_name[_NAME_SIZE];

        inst_table = instance_var_table;

        sprintf(obj_name,"__$%s$%s", pointer->obj.
                class_instance_variable(C_FEATURE_NAME), u
                pointer->
                obj.class_instance_variable(C_NAME));

#ifdef DEBUG
        printf("%s \n",obj_name);
#endif

        while (inst_table ->name != NULL)
            {
            if (strcmp(inst_table->name,obj_name) == 0)
                {
                *((int *)
                (((char *)big_obj)+inst_table->offset))=
                (int)obj;
                break;
                }
            inst_table ++;
            }
        pointer = pointer -> next;
        }
    }
```

4.1.9 Create & CreateObj

Description of Create & CreateObj

Now that we have the MetaClass defined, we can define and create new classes and objects dynamically. To create a new class object or object, we can use the Create function using the syntax:

```
classname.Create();
```

The classname variable will determine what it is we are trying to create, an object or a class object. If classname is a MetaClass, then by calling Create, we are implying that we want to create a class. This class will be an instance of the MetaClass. If the classname is an ordinary class, then we are implying we want an object created. This object will be an instance of the class we used to call Create.

Every time the Create function is called, we call the CreateObj function. Here is where the real work gets done. We must first allocate the memory for the object/class object by calling allocate_obj, discussed in Section 4.1.7. We next set the reference counter, ref_count, that is used for destroying an object. It will state the number of children that the object has. We then initialize some of the object fields. These include:

- lock – the lock structure will be allocated
- lock->status – the current status of the lock, 0 for no lock set
- lock_queue – the queue will be allocated
- lock_queue->head and lock_queue->tail – both pointers will be set to NULL
- virtual_self
- shadow_obj_struct – the structure for the shadow objects

If the value of the initialize parameter, passed in through the optional parameter, is set to INITIALIZE, true, then we must set the lock->status to 1 and set the lock->threadid value to the current _frame->$.threadid.

We set the lock to lock the object until the object is created. The init function will therefore not be called until the creation is complete. We also initialize the string and symbol tables, as is done in the init_method_table function, discussed in Section 4.1.5. Then we must store the object into the object table. It should be noted that each class will keep track of the objects that belongs to it. After this we make a call to the init function. Then we must create the inheritance hierarchy, if any exists. This is done by making a call to the allocate_parent_obj, discussed in Section 4.1.8. Now we check to see if the initialize parameter was set to INITIALIZE, true. If so, then we had locked the object previouisly to prevent the init function from being called until the object was totally created, which is now, so we may now remove the lock. This is done by making the _unblock_frame(obj) call.

Note that if a user defines a class statically, then the class will be created by the meldC.out.c routine. The compiler will directly generate the C code which is equivalent to the function call MetaClass.Create with some parameters passed.

Code of Create & CreateObj

```
object Create(optional)
    {
    return($self.CreateObj(INITIALIZE,$option));
    }

object CreateNoInit(optional)
    {
    return($self.CreateObj(DO_NOT_INITIALIZE,$option));
    }

object CreateObj(int initialize, optional)
    {
    int init_exist = 0;
    int counter;
    struct _method * method;
    object o;
    struct _object * obj;
    static int num;

    o = $self.allocate_obj((char * )NULL);

    obj = (struct _object *)o ;
```

```
bcopy($self.ret_inc_obj_id(), &(obj->$.objid),
      sizeof(_objid));
/*
 * This object does not have any child, we will set its
 * reference count to 0. Any other object will have the
 * reference greater or equal to zero.
 */
obj->$.ref_count = 0;

obj->$.lock =
         (struct _lock *)m_malloc(&(obj->$.add_memory),
         sizeof(struct _lock));
obj->$.lock->status = 0;
obj->$.lock_queue =
        (struct _queue *)m_malloc(&(obj->$.add_memory),
        sizeof(struct _queue));
if (initialize == INITIALIZE)
    {
    /*
     * Lock the object, until the object is created.
     * The init function will not be called, until the
     * creation is completed.
     */

    obj->$.lock->status = 1;
    obj->$.lock->threadid = _frame->$.threadid;
    }

obj->$.lock_queue->head = (struct _frame *)NULL;
obj->$.lock_queue->tail = (struct _frame *)NULL;
obj->$.virtual_self = (struct _object *)o;
obj->$.shadow_obj_struct =
                  (struct _shadow_obj_st *)NULL;

/*
 * Initialize the object table, and store the methods
 * address into the object table.
 */

method = sym_method_table;
```

```
for (counter = 0;counter < sym_method_no;counter ++)
    {
    /*
     * sym_method_table->func_disp is the
     * displacement into the * object structure
     * that will be used to store the func address.
     */
    if (strcmp(method->name,"init") == 0)
        {
        init_exist = 1;
        }
    *((int *)(((char *)obj)+method->func_disp))=
                                  (int)method->func;
    method ++;
    }

method = str_method_table;

for (counter=0;counter < str_method_no;counter ++)
    {
    /*
     * str_method_table->func_disp is the
     * displacement into the object structure
     * that will be used to store the func address.
     */
    *((int *)(((char *)obj)+method->func_disp))=
                                 (int)method->func;
    method ++;
    }
/*
 * Store the object into the object table.
 */

/*
 * Each class will keep track those objects that
 * belong to it.
 */
if (obj_instance == NULL)
    {
    obj_instance
        = (struct OBJ_INSTANCE *)
          m_malloc(&(_obj->$.add_memory),
```

```
            sizeof(struct OBJ_INSTANCE));
        obj_instance->obj = o;
        obj_instance->next = (struct OBJ_INSTANCE *)NULL;
        }
    else
        {
        struct OBJ_INSTANCE * temp;

        temp
            = (struct OBJ_INSTANCE *)
              m_malloc(&(_obj->$.add_memory),
              sizeof(struct OBJ_INSTANCE));
        temp ->next = obj_instance;
        obj_instance = temp;
        obj_instance->obj = o;
        }

    /* do user initialization for this object */

    /*
     * We need to set the object eqaul to the object we
     * just created.
     */

    o.$init();
    /*
     * Create inheritance hierarchy (if any).
     */
    if (parent_objects)
        {
        struct _object *allocated_object = NULL;
        $self.allocate_parent_obj ("__$__1",o,
                                    &allocated_object,
                                    initialize,
                                    0,(struct
                                    _shadow_obj_st *)0,
                                    $option);
        }
    if ((initialize == INITIALIZE) && (init_exist))
        send init($option) to o;

    if (initialize == INITIALIZE)
```

```
        {
        /*
         * Unlock the object and return.
         */
        if ((obj->$.lock->status>0) &&
            (--obj->$.lock->status == 0))
            {
            /*
             * A Language kernel called, request to
             * unlock all the objects.
             */
            _unblock_frame(obj);
            }
        }
    /*
     * We need to restore the old object.
     */
    return(o);
    }
```

4.1.10 CreateSubObj

Description of CreateSubObj

We now encounter the CreateSubObj function, which plays a big part in creating the objects that belong to the same inheritance hierarchy. When there is an inheritance hierarchy to be created in the Create function, we must call this function to create the hierarchial tree. To do this, the function is passed the following parameters:

- name – the objects name
- lock – the lock structure of the object
- lock_queue – the queue structure of the object
- allocated_object
- virtual_self
- objid
- add_memory
- initialize – this is used to determine if the parent object should be initialized or not

- new_func – this is the shadow_object_entry_interrupt address
- shadow_obj_struct – the shadow object structure of the object
- optional

First, we must allocate the memory needed for the object by calling the allocate_obj, discussed in Section 4.1.7. We must then assign values to some of the objects variables. Since the objects are all in the same inheritance tree, they should all share the same lock information, have the same shadow_obj_struct and have the same thread id. We must then initialize the symbol method table. If the initialize parameter is true, INITIALIZE, then we will check to see if the method_name is "init" or "term". If the method_name is init, then set init_exist to 1, else set the new_func value to 0. If the new_func parameter contains the shadow_object_entry_interrupt value, not 0, then set the method table addresses to the new_func. See the shadow_object_entry_interrupt section for more details on why this is done, Section 8.5.9. If the new_func parameter is 0, then set the method table addresses to the functions. Now we must also initialize the string method table. Here we also check to see if the new_func parameter is 0 or not. If it is not 0, new_func equals the shadow_object_entry_interrupt address, then set the method addresses to the shadow_object_entry_interrupt address. Else, set the method addresses to the functions. We then store the object itself into it's class object table. Once again, each class will keep track of the objects that belong to it. We must then set the original object equal to this newly created object. Now we must check the initialize parameter to see if we should initialize the object or not. See the init section for more on $init. Then the call to the allocate_parent_obj is made to continue building the hierarchy tree if need be. Note that the function call is made by passing the initialize, new_func, shadow_obj_struct and $option parameters. When this call returns, we will determine if the init function should be called or not. The user has the option to call it or not with the use of the initialize parameter passed to the CreateSubObj function. Finally, we will then restore the old object.

Code of CreateSubObj

```
/*
 * Create the child object, for the inheritance hierarchy.
 */

object CreateSubObj(char * name,
                    struct _lock * lock,
```

```
                struct _queue * lock_queue,
                struct _object ** allocated_object,
                struct _object *virtual_self,
                struct _objid * objid,
                M_MEM  add_memory,
                int initialize, int new_func,
                struct _shadow_obj_st * shadow_obj_struct,
                optional)
{
int init_exist = 0;
int counter;
struct _method * method;
object o;
struct _object *obj;
static int num;

o = $self.allocate_obj(name);
/*
 * We will use the sequence number as name for the
 * parent object.
 */
obj = (struct _object *)o;

/*
 * In theory, all the obejct that in the inheritance
 * hierarchy are the same object.  They should also
 * share the same lock queue.
 *
 * Synchrnous the atomic lock queue.
 */

obj->$.lock  = lock;
obj->$.lock_queue = lock_queue;
obj->$.add_memory = add_memory;
obj->$.shadow_obj_struct = shadow_obj_struct;
obj->$.virtual_self = virtual_self;

bcopy(objid, &(obj->$.objid), sizeof(_objid));

/*
 * Initialize the object table, and store the
 * methods address into the object table.
```

```
     */
    method = sym_method_table;

    for (counter=0;counter < sym_method_no;counter ++)
        {
        /*
         * sym_method_table->func_disp is the
         * displacement into the object structure that
         * will be used to store the func addrsss.
         */
        if (initialize)
            {
            if (strcmp(method->name,"init") == 0)
                {
                init_exist = 1;
                }
            else if (strcmp(method->name,"$term") == 0)
                {
                new_func = 0;
                }
            }
        if (new_func)
            {
            *((int *)
            (((char *)obj)+method->func_disp))=
            new_func;
            }
        else
            *((int *)
            (((char *)obj)+method->func_disp))=
            (int)method->func;

        method ++;
        }
    method = str_method_table;

    for (counter=0;counter < str_method_no;counter ++)
        {
        /*
         * str_method_table->func_disp is the
         * displacement into the object structure that
         * will be used to store the func addrsss.
```

```
         */
        if (new_func)
            *((int *)
            (((char *)obj)+method->func_disp))=
            new_func;
        else
            *((int *)
            (((char *)obj)+method->func_disp))=
            (int)method->func;
        method ++;
        }
    /*
     * Store the object into the object table.
     */

    /*
     * Each class will keep track those objects that
     * belong to it.
     */

    if (obj_instance == NULL)
        {
        obj_instance
            = (struct OBJ_INSTANCE *)
              m_malloc(&(_obj->$.add_memory),
              sizeof(struct OBJ_INSTANCE));
        obj_instance->obj = o;
        obj_instance->next = (struct OBJ_INSTANCE *)NULL;
        }
    else
        {
        struct OBJ_INSTANCE * temp;

        temp
            = (struct OBJ_INSTANCE *)
              m_malloc(&(_obj->$.add_memory),
              sizeof(struct OBJ_INSTANCE));
        temp ->next = obj_instance;
        obj_instance = temp;
        obj_instance->obj = o;
        }
```

```
/* do user initialization for this object */

/*
 * We need to set the object eqaul to the
 * object we just created.
 */

if (initialize)
     o.$init();

if (parent_objects)
    $self.allocate_parent_obj(name,o,
                                allocated_object,
                            initialize,
                            new_func,
                            shadow_obj_struct,
                                $option);

if ((initialize) && (init_exist))
    send init($option) to o;
/*
 * Create inheritance hierarchy (if any).
 */

/*
 * We need to restore the old object.
 */
return(o);
}
```

4.1.11 CreateAndAttachObj

Description of CreateAndAttachObj

The CreateAndAttachObj function, is a special function used to create an object and then attach a shadow object to it right away. This is done for the reason of efficency, since it is much faster and easier to do this in one straight step. This is almost equivalent to doing both the Create and AttachObject function calls. To do this, the function is passed:

- name – the name of the object
- shadow_obj

- the entry_point and exit_point – both used by the shadow object, see Sections for more on these two values usage.
- the init_point and term_point – the init_point is used to initialize the shadow object when "attached" to an object and the term_point is used to cleanup the shadow object when a detach is executed.
- optional

Note that the entry_point, exit_point, init_point and term_point are all provided by the user.

We will first get the shadow_object_entry_interrupt value, discussed in Section 4.1.16. This will return to us the address of the shadow_object_entry_interrupt method, see Section 4.1.16. Then we will allocate the memory needed for the object by calling the allocate_obj function, discussed in Section 4.1.7. We will then assign values to some of the object fields, which include:

- the ref_count – equal to zero
- lock – allocate the lock structure
- the lock->status – the status of the lock, 0 = not locked
- lock_queue – allocate the lock queue structure
- lock_queue->head and tail are both set to NULL
- shadow_obj_struct – the structure is allocated
- shadow_obj_struct->next – set to NULL
- shadow_obj_struct->shadow_obj – this is where this function differs from the Create function in that the shadow_obj will be attached to this object at this point.
- shadow_obj_struct->entry_point – if the entry_point has been defined for the shadow object, then set this value.
- shadow_obj_struct->exit_point – if the exit_point for the shadow object has been defined, then set this value.
- shadow_obj_struct->init_point – if the init_point has been defined by the user.
- shadow_obj_struct->term_point – if the term_point has been defined by the user.

Next we will initialize the object's method table. This is where the value of the shadow_obj_entry_interrupt comes into play. This is also the place where the regular Create and this function differ. For a user to attach a shadow object, it is necessary for all of the object's method addresses to be set to some fixed location, the shadow_obj_entry_interrupt address, see Section 4.1.16 for more on why this is necessary. If the shadow_obj_entry_interrupt is equal to zero, then it implies that the user does not want to change the method table addresses. We must do this for both the string and symbol method tables. Next we will set up the class's instance table. Then we make a call to the allocate_parent_obj function, see Section 4.1.8. Note that we make the allocate_parent_obj call by specifying DO_NOT_INITIALIZE. We also pass the shadow_object_entry_interrupt value and the shadow_obj_struct. The reason for this, is that the user considers the hierarchy as one object, so we pass the shadow object information so that the hierarchy can inherit it. If the init_point does not equal NULL, the user defined some init_point, then we must call the execute_shadow_object_method. For this call, we must pass the shadow_obj, the init_point, the obj and $option. If the call returns FALSE, then the initialization has failed, so destroy the object with the Destroy call. Finally, we will restore the caller by doing a return.

Code of CreateAndAttachObj

```
/*
 * This is a special routine that will create an object and
 * attach a shadow object to it right away. For the reason
 * of efficiency.
 *
 * This routine is virtual equal to Create + AttachObject.
 */

object CreateAndAttachObj(char * name, object shadow_obj,
                          char * entry_point,
                          char * exit_point,
                          char * init_point,
                          char * term_point,
                          optional)
    {
    int init_exist = 0;
    int counter;
    struct _method * method;
    object o;
```

```
struct _object * obj;
int new_func;
static int num;

new_func =
               (int)MetaClass.shadow_object_entry_interrupt;

o = $self.allocate_obj((char * )name);

obj = (struct _object *)o ;

bcopy($self.ret_inc_obj_id(), &(obj->$.objid),
      sizeof(_objid));
/*
 * The object that does not have any child, we will set
 * its reference count to 0. Any other object will have
 * the reference greater or equal to zero.
 */
obj->$.ref_count = 0;

obj->$.lock =
         (struct _lock *)m_malloc(&(obj->$.add_memory),
         sizeof(struct _lock));
obj->$.lock->status = 0;
obj->$.lock_queue =
        (struct _queue *)m_malloc(&(obj->$.add_memory),
        sizeof(struct _queue));
obj->$.lock_queue->head = (struct _frame *)NULL;
obj->$.lock_queue->tail = (struct _frame *)NULL;

obj->$.shadow_obj_struct =
              (struct _shadow_obj_st *)
              m_malloc(&(obj->$.add_memory),
              sizeof(struct _shadow_obj_st));

obj->$.shadow_obj_struct->next =
                        (struct _shadow_obj_st *)NULL;
obj->$.shadow_obj_struct->shadow_obj =
                         (struct _object *)shadow_obj;

if ((int)entry_point)
    strncpy(obj->$.shadow_obj_struct->entry_point,
```

```
            entry_point,_NAME_SIZE);
else
    obj->$.shadow_obj_struct->
         entry_point[0] = (char) 0;

if ( ((int)exit_point))
    strncpy(obj->$.shadow_obj_struct->exit_point,
            exit_point,_NAME_SIZE);
else
    obj->$.shadow_obj_struct->
         exit_point[0] = (char) 0;

if ((int)init_point)
    strncpy(obj->$.shadow_obj_struct->init_point,
            init_point,_NAME_SIZE);
else
    obj->$.shadow_obj_struct->
         init_point[0] = (char) 0;

if ((int)term_point)
    strncpy(obj->$.shadow_obj_struct->term_point,u
            term_point,_NAME_SIZE);
else
    obj->$.shadow_obj_struct->
         term_point[0] = (char) 0;

/*
 * Initialize the object table, and store the methods
 * address into the object table.
 */

method = sym_method_table;

for (counter=0;counter < sym_method_no;counter ++)
    {
    /*
     * sym_method_table->func_disp is the displacement
     * into the object structure that is used to store
     * the func addrsss.
     */

    /*
```

```
         * If new_func address == 0, it implied, user do
         * not wish to change the methods' address to a
         * common address.  Remote object will require
         * the change of all the method's * address to a
         * common address.
         */
        *((int *)(((char *)obj)+method->func_disp))=
                                        (int)new_func;

        method ++;
        }
    method = str_method_table;

    for (counter=0;counter < str_method_no;counter ++)
        {
        /*
         * str_method_table->func_disp is the displacement
         * into the object structure that is used to store
         * the func addrsss.
         */
        *((int *)(((char *)obj)+method->func_disp))=
                                                   new_func;

        method ++;
        }
    /*
     * Each class will keep track those objects that
     * belong to it.
     */

    if (obj_instance == NULL)
        {
        obj_instance =
                    (struct OBJ_INSTANCE *)
                    m_malloc(&(_obj->$.add_memory),
                    sizeof(struct OBJ_INSTANCE));
        obj_instance->obj = o;
        obj_instance->next = (struct OBJ_INSTANCE *)NULL;
        }
    else
        {
        struct OBJ_INSTANCE * temp;
```

```
    temp = (struct OBJ_INSTANCE *)
           m_malloc(&(_obj->$.add_memory),
           sizeof(struct OBJ_INSTANCE));
    temp ->next = obj_instance;
    obj_instance = temp;
    obj_instance->obj = o;
    }
/*
 * Store the object into the object table.
 */

if (parent_objects)
    {
    struct _object *allocated_object = NULL;
    $self.allocate_parent_obj("__$__1",o,
                              &allocated_object,
                              DO_NOT_INITIALIZE,
                              new_func,
                               obj->
                              $.shadow_obj_struct);
    }

if (obj->$.shadow_obj_struct->init_point[0] != NULL)
    {
    if ($self.execute_shadow_object_method(
        obj->$.shadow_obj_struct->shadow_obj,
        obj->$.shadow_obj_struct->init_point,
             (object)obj,$option) == FALSE)
        {
        /*
         * Initialization has failed.
         */
        $self.Destroy(o);
        return((object)NULL);
        }
    }
/*
 * We need to restore the old object.
 */
return(o);
}
```

4.1.12 AttachObject

Description of AttachObject

The AttachObject function will handle the special steps needed to "attach" an object to a master object. This object to be attached to a master object is know as a shadow object. See Section 5 for more on Shadow Objects in general. Note that more than one shadow object at a time can be attached to an object. This function will receive the following parameters needed to "attach" a shadow object:

- o – the object that the shadow object will be "attached" to.
- shadow_obj – the shadow object to "attach"
- entry_point
- exit_point
- init_point
- term_point
- optional

The first thing to notice is that a global lock must be used here. The reason for this, is to prevent a message from being sent to the master object having the shadow object attached. You see, the shadow object is designed to intercept the incoming messages. So if a message is sent to the master object while the "attach" is taking place, some inconsistency within the master object may occur. If a global lock is set, we do not have to worry about a message coming in. After we set the global lock, we will then allocate the memory needed for the shadow_obj_struct. Next, we will set some of the variables within the shadow_obj_struct. These include:

- shadow_obj – the shadow object that was passed as a parameter.
- next – a pointer to the next shadow object.
- entry_point
- exit_point
- init_point – used to initialize the new attached shadow object.
- term_point – used to cleanup the shadow object when a "detach" is executed.

Then we can make a call to the AttachDetachObject, discussed in Section 4.1.14. Notice that we pass a status of ATTACH to the function. This will tell the function that we want to attach the given shadow object to the master object, since this function can also detach a shadow object. Basically, the function will assist us in setting up the object table to inform the system that a shadow object exists for the master object. Then we will check to see if the init_point has been defined by the user, init_point will not equal NULL. If it has been defined, then we will call the execute_shadow_object_method, passing the shadow_obj, the init_point, the object and $option as parameters. The $option parameter will contain any of the user defined parameters needed. If the call to execute_shadow_object_method returns FALSE, then the initialization failed. In this case, we must call the Destroy function to destroy the object. If the initialization succeeds, then the global lock can be removed with the GLOBAL_UNLOCK to allow other threads to execute again.

Note what happens when there are multiple shadow objects connected to one master object, refer to Figure 4.5. Shadow object A was connected first, then shadow object B was attached, until we reached shadow object G, which was the last to be attached. When a message arrives for the master object, shadow object G will get it first. Shadow object G will then pass the message to F and shadow object F will pass the message to E. This continues until shadow object B passes the message to shadow object A. When shadow object A is done, then A will pass the message to the master object.

Code of AttachObject

```
object AttachObject(object o,
                    object shadow_obj,
                    char * entry_point,
                    char * exit_point,
                    char * init_point,
                    char * term_point,
                    optional)

    {
    GLOBAL_LOCK;

    struct _shadow_obj_st * shadow_obj_struct;

    if (shadow_obj == (object)NULL)
        {
```

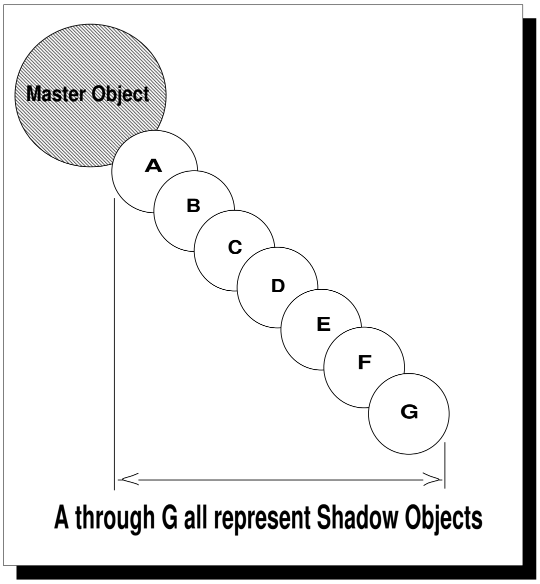

Figure 4.5: Multiple shadows attached to one object, using vertical attachment.

```
        printf("MeldC Error: AttachObject() Shadow object");
        printf("is equal to NULL \n");
        return((object)NULL);
        }

    _frame->$.priority = 6;

    shadow_obj_struct = (struct _shadow_obj_st *)
                        m_malloc(&(((struct _object *)o)->
                                                $.add_memory),
                        sizeof(struct _shadow_obj_st));

    shadow_obj_struct->shadow_obj =
                             (struct _object *)shadow_obj;

    shadow_obj_struct ->next =
                           (struct _shadow_obj_st *) NULL;

    if ((int)entry_point)
```

```
        strncpy(shadow_obj_struct->entry_point,
                entry_point,_NAME_SIZE);
    else
        shadow_obj_struct->entry_point[0] = (char) 0;

    if ((int)exit_point)
        strncpy(shadow_obj_struct->exit_point,
                exit_point,_NAME_SIZE);
    else
        shadow_obj_struct->exit_point[0] = (char) 0;

    if ((int)init_point)
        strncpy(shadow_obj_struct->init_point,
                init_point,_NAME_SIZE);
    else
        shadow_obj_struct->init_point[0] = (char) 0;

    if ((int)term_point)
        strncpy(shadow_obj_struct->term_point,
                term_point,_NAME_SIZE);
    else
        shadow_obj_struct->term_point[0] = (char) 0;

    $self.AttachDetachObject(o,shadow_obj_struct,
                ATTACH,$option);

    if (shadow_obj_struct->init_point[0] != NULL)
        {
        if ($self.execute_shadow_object_method(
                         shadow_obj_struct->shadow_obj,
                         shadow_obj_struct->init_point,
                         (object)o,$option)
                         == FALSE)
            {
            /*
             * Initialization has failed.
             */
            $self.Destroy(o);
            return((object)NULL);
            }
        }
    /*
```

```
 * Allocate the shadow object structure....
 */

GLOBAL_UNLOCK;
}
```

4.1.13 DetachObject

Description of DetachObject

The DetachObject function will allow a user to detach one or all of the shadow objects attached to some master object. The function will get a pointer to the master object and to the shadow object as passed parameters. If the shadow object value is NULL, then this informs the function that the user wants all of the shadows removed from the master object. It should be noted, that the master object can remove shadow objects that are only attached to him. See Figure 4.6. In this case, object A can only remove object B. Object C can only remove object D. Object E on the other hand can remove objects F, G and H if it desires to do so. Objects A, B, C and D make up what is known as a vertical attachment. Object E, F, G and H make up what is known as a horizontal attachment.

As is the case of the AttachObject function, we must set a global lock. This is for the same reason stated before. That is, we want to prevent any messages arriving at the master or shadow objects while in the process of doing a "detach". We first make a check to see if the shadow_obj pointer is NULL, for the reason stated above. When the user wants only one shadow removed we will have to check how many shadow objects the master object has. If there is more then one shadow object, then we must free only that shadow object in the list of the shadows. This is done by searching down the list of the shadows objects until we find a match for the one we are looking to detach. When we find the shadow object, we must check to see if the user provided a term_point function, term_point does not equal NULL. If the user provided a term_point function, we will execute it by calling the execute_shadow_object_method function. For this function call, we will pass the shadow_obj, the term_point, the object and the $option. The $option are the parameters provided by the user to execute the term_point. After the term_point function call is made, we will free the pointer to the shdow object. Then we will set the pointers in the shadow list to point around the detached shadow object. Finally, we can then do a free on the shadow objects memory. If the master object has only one shadow object, then we just have to simply detach the shadow object and restore the master object to it's original state. This is where the AttachDetachObject function comes

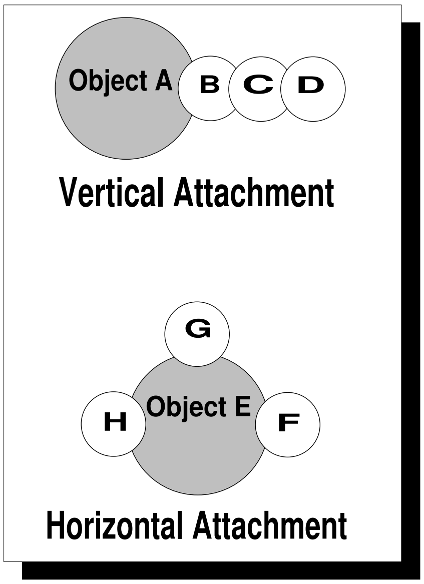

Figure 4.6: Two methods for attaching shadow objects.

in again. We will call it for two cases:

1. One is if we want to remove only one shadow object and the master object has only one shadow object.

2. The other case is when we want to remove all of the shadow objects that the master object has.

See the AttachDetachObject function for more details, Section 4.1.14. Basically though, it will remove the shadow object(s) from the master object and set the master object back to a state where the system recognizes that it no longer has any shadows. This will then mean that all messages will go directly to the master object. Note that in this call to the function, we send a status of DETACH this time. If the user wanted all of the shadow objects removed from the object, we must then free up the shadow_obj_struct. To do this, we must call the term_point function for each shadow object. This is done by doing a while loop for the entire shadow_obj_struct and calling the execute_shadow_object_method function on each iteration. Once again, the execute_shadow_object_method function will be passed the shadow object, the object, the $option and the term_point. After this, we can then finally remove the global lock on the system by doing a GLOBAL_UNLOCK.

Code of DetachObject

```
object DetachObject(object o, object shadow_obj,optional)

    {
    GLOBAL_LOCK;

    struct _shadow_obj_st * shadow_obj_struct, *ptr;

    ptr = shadow_obj_struct =
          ((struct _object *)o)->$.shadow_obj_struct;
    _frame->$.priority = 6;

    if (shadow_obj != (object) NULL)
        {
        /*
         * Detach one shadow object.
         */
        if (shadow_obj_struct ->next ==
                             (struct _shadow_obj_st *) NULL)
            {
            /*
             * There is only one shadow object to detach, we
             * will detach the shadow object, and restore the
             * orginal object.
             */
            if (shadow_obj !=
                    (object)(shadow_obj_struct->shadow_obj))
                {
                printf("MeldC Error: The shadow object is
                        not attached ");
                printf("to the object. \n");
                return (object)NULL;
                }
            }
        else
            {
            struct _shadow_obj_st **ptr;
            /*
```

```
            * There are multiple shadow object attached to
            * the object.  We will free the shadow object,
            * and return.
            */
           ptr =
             &(((struct _object *)o)->$.shadow_obj_struct);
           while (*ptr != NULL)
               {
               if ((*ptr)->shadow_obj ==
                             (struct _object *)shadow_obj)
                   {
                   struct _shadow_obj_st * free_ptr;

                   if ((*ptr)->term_point[0] != NULL)
                       {
                       (void)$self.
                             execute_shadow_object_method(
                                        (*ptr)->shadow_obj,
                                        (*ptr)->term_point,
                                        (object)o,$option);
                       }
                   free_ptr = *ptr;
                   *ptr = (*ptr)->next;
                   m_free(((struct _object *)o)->
                                              $.add_memory,
                          (char *)free_ptr);
                   return ((object)NULL);
                   }
               }
           if (*ptr == (struct _shadow_obj_st *)NULL)
               {
               printf("MeldC Error: The shadow object is
                       not attached ");
               printf("to the object. \n");
               return ((object)NULL);
               }
           }
       }
   /*
    * We will detach the shadow objects, and restore the
    * object back to normal.
    */
```

```
$self.AttachDetachObject(o,shadow_obj_struct,DETACH,
                          $option);

GLOBAL_UNLOCK;

/*
 * Free up the shadow_object_entry.
 */

while(shadow_obj_struct !=
                       (struct _shadow_obj_st *)NULL)
    {
    if (shadow_obj_struct->term_point[0] != NULL)
        {
        (void)$self.execute_shadow_object_method(
                     shadow_obj_struct->shadow_obj,
                     shadow_obj_struct->term_point,
                     (object)o,$option);
        }

    shadow_obj_struct = shadow_obj_struct ->next;
    m_free(((struct _object *)o)->$.add_memory,
           (char *)ptr);
    ptr = shadow_obj_struct;
    }
((struct _object *)o)->$.shadow_obj_struct =
                  (struct _shadow_obj_st * ) NULL;
}
```

4.1.14 AttachDetachObject

Description of AttachDetachObject

This next function, AttachDetachObject, has already been mentioned a couple of times. It can either remove, detach, or connect, attach, a shadow object to the given master object. It does not do this whole process by itself though. For the attachment of a shadow to a master object, the AttachObject function will be excuted first, then this function will be called. For the detachment of a shadow object, we will first call the DetachObject function and then call this function. The function can then be seen as doing only a piece of the process. Let us examine what it does then.

This function will first check the status passed to it. This status is either

ATTACH or DETACH. If it is ATTACH, then we will check to see if there is more than one shadow attached to the master object. In this case, it will imply that horizontal attachment was used. See Figure 4.6. If there is already some shadows in existence for the master object, we will put the new shadow at the front of the shadow pointer list. We can do this by assigning the new shadow's next pointer to the already existing shadow_obj_struct and then reassign the structure pointer to the new shadow, or the front of the list again. Then we can simply return back to the callee. If there is no other shadow objects for the master object passed to us, we will assign the address of the shadow object to the shadow_obj_struct. But here we cannot simply return. We must now change the master object so that the system will recognize that the master object has a shadow object now. To do this, we must get the shadow_object_entry_interrupt value. This will basically give us a pointer to a new object, or known as a new shadow object. We will store the value into the new_func variable. Since we want to do an ATTACH, we will first assign the method function displacement variable for both the string and symbol method tables. Finally we will store the object into the object table.

Now going all the way back, if the status passed to the function was DETACH, then we must first assign the shadow_obj_struct a value of NULL. This will make it look like there is no shadows attached to this master object. Next we will restore the method address of the object. This is done by actual assigning the method function displacement variable the original function address in both the symbol and string tables. Finally we will store the object into the object table.

Code of AttachDetachObject

```
object AttachDetachObject(object o,
                    struct _shadow_obj_st * shadow_obj_struct,
                    int status,optional)
    {
    int init_exist = 0;
    int counter;
    struct _method * method;
    static int num;
    int new_func;
    struct _object * obj;

    /*
     * Let's raise the priority to the highest priority
     */
```

```
obj = (struct _object *)o;
if (status == ATTACH)
    {
    if (obj->$.shadow_obj_struct !=
                           (struct _shadow_obj_st *)NULL)
        {
        shadow_obj_struct->next=obj->$.shadow_obj_struct;

        obj->$.shadow_obj_struct
            = (struct _shadow_obj_st *)shadow_obj_struct;

        return ((object)obj);
        }
    else
        {
        obj->$.shadow_obj_struct = shadow_obj_struct;
        }
    new_func =
            (int)MetaClass.shadow_object_entry_interrupt;
    }
/*
 * If the operation is ATTACH, we will initialize the
 * object table, and store the methods address into the
 * object table.  else we will restore the method
 * address of the object.
 */

method = sym_method_table;

for (counter = 0; counter < sym_method_no; counter ++)
    {
    /*
     * sym_method_table->func_disp is the displacement
     * into the object structure that is used to store
     * the func addrsss.
     */

    /*
     * If new_func address == 0, it implied, user do
     * not wish to change the methods' address to a
     * common address.  Remote object will require the
```

```
         * change of all the method's address to a common
         * address.
         */
        if (status == ATTACH)
            {
            /*
             * Mark this special object, later when we
             * destroy this object, we do not call
             * "$term" function.
             */
            /*
            if (strcmp(method->name,"$term") == 0)
                *((int *)
                (((char *)obj)+method->func_disp))=(int)0;
            else
            */
            *((int *)(((char *)obj)+method->func_disp))=
                                            (int)new_func;
            }
        else
            *((int *)(((char *)obj)+method->func_disp))=
                                        (int)method->func;
        method ++;
        }

    method = str_method_table;

    for (counter=0;counter < str_method_no;counter ++)
        {
        /*
         * str_method_table->func_disp is the displacement
         * into the object structure that is used to store
         * the func addrsss.
         */
        if (status == ATTACH)
            *((int *)(((char *)obj)+method->func_disp))=
                                                 new_func;
        else
            *((int *)(((char *)obj)+method->func_disp))=
                                        (int)method->func;
        method ++;
        }
```

```
/*
 * Store the object into the object table.
 */

struct OBJ_INSTANCE * pointer;
char obj_name[MAX_METHOD_NAME];

pointer = parent_objects;

if (status == DETACH)
    ((struct _object *)o)->$.shadow_obj_struct
                   = (struct _shadow_obj_st * ) NULL;
while (pointer != NULL)
    {
    struct _para * inst_table;
    inst_table = instance_var_table;
    sprintf(obj_name,"__$%s$%s",
            pointer->obj.
            class_instance_variable(C_FEATURE_NAME),
            pointer->
            obj.class_instance_variable(C_NAME));

    while(inst_table ->name != NULL)
        {
        struct _object * next_obj;
        if (strcmp(inst_table->name,obj_name) == 0)
            {
            next_obj = (struct _object *) (*((int *)
               (((char *)obj)+inst_table->offset)));

            ((MetaClass)(next_obj->$.class_ptr)).
                                 AttachDetachObject(
                                  (object)next_obj,
                                  shadow_obj_struct,
                                  status,$option);
            break;
            }
        inst_table ++;
        }
    pointer = pointer -> next;
    }
return((object)obj);
```

```
    }
```

4.1.15 Execute_shadow_object_method

Description of Execute_shadow_object_method

This function has the purpose of executing the given shadow object method. The methods can include:

- the init_point
- the term_point
- the entry_point
- the exit_point

All of these methods are provided by the user. It will except the shadow_object, a method_name, the method name to execute (any from the above list) and the optional parameter. The optional parameter will contain the parameters provided by the user to execute the given method. Optional will also contain the object that the shadow is attached to.

To execute the methods, we must first set some of the frames fields. These include:

- tmp_msg – the frame message
- tmp_msg->_msg_name – the method_name

Then we will call the internal binding function and save the return value into the fp pointer. After the internal binding call, we will set the frame sender field to the object. Then we will enqueue the fp frame by making the ENQUEUE_FRAME macro call. Then make the YIELD macro call. After the method has completed execution, the execution will return to this point where we will return.

Code of Execute_shadow_object_method

```
int execute_shadow_object_method(struct _object * sh,
                                 char * method_name,
                                 optional)
    {
    int ret_fp = 0;
    struct _frame * fp;
```

```
struct _elem_tab return_type;
/*
 * The optional parameters are pass into the calling
 * frame.
 */
_frame->$.tmp_msg = _frame->$.msg;
_frame->$.tmp_msg->_msg_name = method_name;

fp = binding.internal_binding(fp,(object)sh, &ret_fp,
                              NULL,SYM_MSG,SYNC_MSG);
/*
 * the exit method is undefined.
 */

if ((fp == (struct _frame *)NULL) ||
    (strcmp((fp)->$.mp->name,"meldc_default")==0))
    {
    printf("MeldC Error : Invalid shadow object
            entry/exit methods.");
    printf(" Method name => %s \n",method_name);
    return(0);
    }

(fp)->$.sender = _obj;
(ENTER_CLIB 0);
ENQUEUE_FRAME(fp);
YIELD;

return(*((int *)ret_fp));
}
```

4.1.16 Shadow_object_entry_interrupt

Description of Shadow_object_entry_interrupt

The shadow_object_entry_interrupt function is one of the most important functions to the MetaClass. Without this function, the shadow object concept would not exist. It is used to handle the shadow object trap. It will handle the trap, intercept the message and then forward the message to the appropraite shadow object. An example of this is when some foo makes a call such as:

```
A.hello()
```

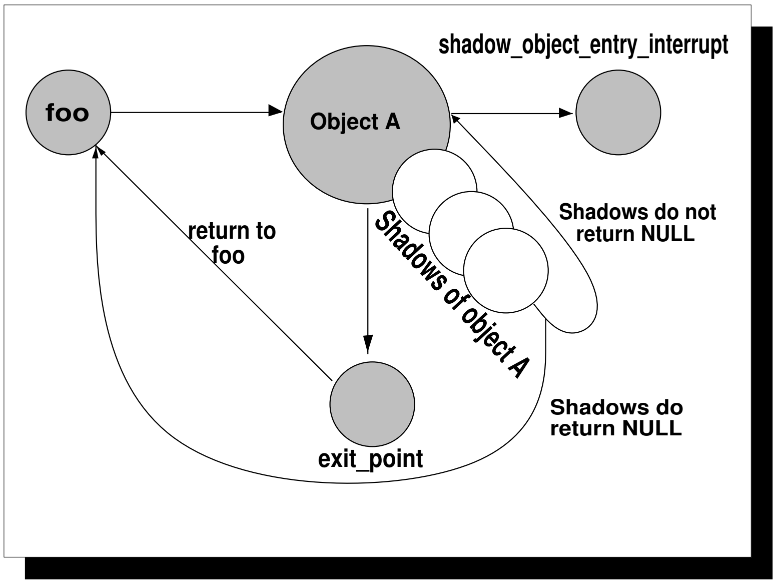

Figure 4.7: A Venn diagram of an inheritance scheme.

This foo is hidden from everyone. This call will send a message hello to object A. Object A will look in it's own method table and find as fixed address that points to a method in the MetaClass. This method is the shadow_object_entry_interrupt. The shadow_object_entry_interrupt method will then save the original frame from foo in object A first. This is necessary to save the frame since we may want to resume object A later. Then we will alter the frame sent by foo in object A to look like the frame structure of the shadow_object_entry_interrupt. Then we will call the shadow objects that are attached to object A to execute the frame. The shadows are found in object A's shadow structure. Then the shadows will then finish their execution of the frame. If the shadow objects return NULL, then return the value to some place in memory. This memory location was stated by foo as the reserved spot for the return value so that foo can recover the value. If the return value is not NULL, then object A will execute the frame hello. When object A is done, the shadow object exit point will be executed. Then the return value will then be put into that reserved memory location. To see the whole picture of what is going on see Figure 4.7. Let us now take a look at the specific steps taken by this method.

The function will save the current frame structure, since we might need to resume the execution of the frame later. We will then reset the environment, so that it will appear that a shadow_object_entry_interrupt was called the proper way. The environment is the _frame, _obj and method pointers. Next we will execute the shadow objects entry point. We must then repeat the execution of the message for all of the other shadows that exist for the master object. Then we will return a status. The status will be either a NULL or a frame. If a frame is returned, we will execute the frame. Otherwise, we will just return and do not resume the original frame. To the caller, it will appear as if the callee has returned. If there are multiple shadow objects, all of them must succeed in order for the return frame to be executed. Only the last frame returned by the shadow object entry method will be executed. Once the execution of the frame is complete, then the user provided shadow object exit point will be invoked. We will then resume the execution of the original frame.

It should be noted that, using the STACK_BASE or STACK_BASE_SLOW methods to implement shadow objects is very difficult. To do so, usually requires the insertion of a new call between the caller and callee. Some difficult cleanup for the frame is also needed.

Code of Shadow_object_entry_interrupt

```
void shadow_object_entry_interrupt()
    {
    reset_obj_and_frame(&_frame,&_obj,MetaClass,
            sizeof(struct $MeldC$MetaClass$
                           shadow_object_entry_interrupt),
            MetaClass.shadow_object_entry_interrupt);

    struct _method *mp;
    int counter;

    mp = (struct _method*)MetaClass.
              class_instance_variable(C_SYM_METHOD_TABLE);

    for (counter = 0; counter < sym_method_no; counter ++)
        {
        if(binding.msg_compare(mp->name,
                          "shadow_object_entry_interrupt",
                          (enum msg_types)SYM_MSG) == 0)
            {
            _frame->$.mp = mp;
            break;
            }
```

```
        mp ++;
        }

    object obj;
    struct _frame * frame_to_be_executed;
    struct _frame * fp;
    struct _threadid threadid;

    /*
     * The caller is the orginal frame, _frame is the
     * shadow object's frame.
     *
     * We need to preserve the orginal frame, so later
     * we might want to resume the execution of the
     * orginal frame.
     */

    frame_to_be_executed = _frame->$.caller;
    _frame->$.caller = (struct _frame *)NULL;

    struct _shadow_obj_st * shadow_obj_struct;
    struct _shadow_obj_st * head_shadow_obj_struct;

    head_shadow_obj_struct = shadow_obj_struct
        = (struct _shadow_obj_st *)
          (frame_to_be_executed->$.obj->
                              $.shadow_obj_struct);

    obj = (object)(frame_to_be_executed->$.obj);

    frame_to_be_executed->$.state = S_NEW;
    frame_to_be_executed->$.func  =
                  frame_to_be_executed->$.mp->func;
    fp = frame_to_be_executed;

    while (shadow_obj_struct !=
                    (struct _shadow_obj_st *) NULL)
        {
        int return_fp;

        /*
         * Invoke the entry point method of the
```

```
         * shadow object.
         */
        return_fp = $self.execute_shadow_object_method(
                          shadow_obj_struct->shadow_obj,
                          shadow_obj_struct->entry_point,
                          fp);
        /*
         * If the result return from the shadow object
         * is NULL, then do not execute any frame.
         */
        if ((return_fp) && (fp))
            fp = (struct _frame *)return_fp;
        else
            fp = (struct _frame *)NULL;

        shadow_obj_struct = shadow_obj_struct->next;
        }

    if (fp != (struct _frame *)NULL)
        {
        struct _frame * caller;
        int sp;

        /*
         * We will set up the exist point for the object.
         *
         * We will add a shadow frame between the caller
         * and callee's frame.  After the callee's
         * complete its execution, it will return the
         * control back to this routine.  We can then
         * invoke the exit routine for the shadow object.
         *
         * The object for this shadow frame will equal
         * to the callee's object.  This shadow frame
         * is allocated as if its object is the MetaClass.
         *
         */

        caller = fp->$.caller;
        threadid =  fp->$.threadid;

        /*
```

```
         * Even though we change the caller, the return
         * address reamin unchange.  After the execution
         * we will still save the return value in the
         * original caller.
         */

        fp->$.caller = _frame;

        /*
         * We will enqueue the frame.
         */

        /*
         * Preserve the fp value.
         */
        fp->$.P = fp;

        ENQUEUE_FRAME(fp);
        YIELD;

        /*
         * After the yield returned, fp became invalid,
         * and we can not use it anymore except in the
         * case of STACK_BASE_SLOW implementation.  In
         * STACK_BASE_SLOW implementation the driver
         * can not free the fp, becuase the caller and
         * callee are located at a different stack.
         */

        /*
         * We should call exist at this point.
         */

        shadow_obj_struct = head_shadow_obj_struct;

        while (shadow_obj_struct !=
                       (struct _shadow_obj_st *) NULL)
            {
            if(shadow_obj_struct->exit_point[0] !=
                                              (char )0)
                {
                int return_fp;
```

```
                return_fp = $self.
                         execute_shadow_object_method(
                         shadow_obj_struct->shadow_obj,
                         shadow_obj_struct->exit_point,
                         (object)obj,threadid);
                }
            /*
             * If the result return from the shadow
             * object is NULL, than do not execute any
             * frame.
             */
            shadow_obj_struct = shadow_obj_struct->next;
            }
        /*
         * Restore the frame as if it were return by the
         * object.  Ask the kernel to restore the frame.
         */

        /*
         * After the return the caller will enqueue into
         * the ready queue.
         */
        fp->$.caller = caller;

#ifdef STACK_BASE_SLOW
        /*
         * HACK :
         * In the STACK_BASE_SLOW implementation, the
         * driver will not free the fp.  Because the
         * caller and callee are not allocated at the
         * same stack.
         *
         * We will restore the old fp, so it now can be
         * freed by the driver.
         */
        (void) restore_old_frame(fp);
#endif
        return;
        }
    else
        {
        /*
```

```
         * Shadow object return NULL pointer.  We will
         * not execute the frame, we will just return,
         * as if the frame complete its execution.
         */
        (void) restore_old_frame(frame_to_be_executed);
        return;
        }
    }
```

4.1.17 Destroy

Description of Destroy

The Destroy function has the task of removing an object when it no longer exists. The interesting part about this function is that it must not only destroy the object we asked it to destroy, the object we passed to the function, but shadows associated with the object and any other objects created through inheritance. See Figure 4.8 to see an example of what we mean here. When object A is passed to the Destroy function to be destroyed, then the whole tree of objects, object B through object D, must also be destroyed. We can tell if there is more than one object by the use of the reference count variable, ref_count. If the ref_count variable of an object is not 1, then other objects are in the same tree with the object. This object should not be destroyed until it does equal 1. The tree will be traversed recursively to find the other objects and then it will destroy the objects while it descends back down. One way to look at this destruction method is, it will search out the leaves, objects with no objects above it, and destroy them. Then it will back up and destroy any other leaves. Let us look at this function in more detail.

We will first ask the Kernel for permission to destroy this object by calling the can_we_destroy_the_object. If the permission is denied, then this thread will be suspended. Permission will be denied if there are any other threads still using this object. See Section 2.14.13 for more on this function and this topic of getting permission to destroy the object. After we are granted permission to destroy the object, we will call the DetachObject function, see Section 4.1.13, if there are any shadow objects associated with the object. This call may detach and destroy the shadow object, depending on the termination function provided by the user. Then we will go through the symbol method table and determine which method names are 'term'. Those that are 'term' will have the destroy_exist variable set. Next we will destroy the subobject from the inheritence. This is done with that recursive call back to the Destroy. We decide if we should call it or not by using the

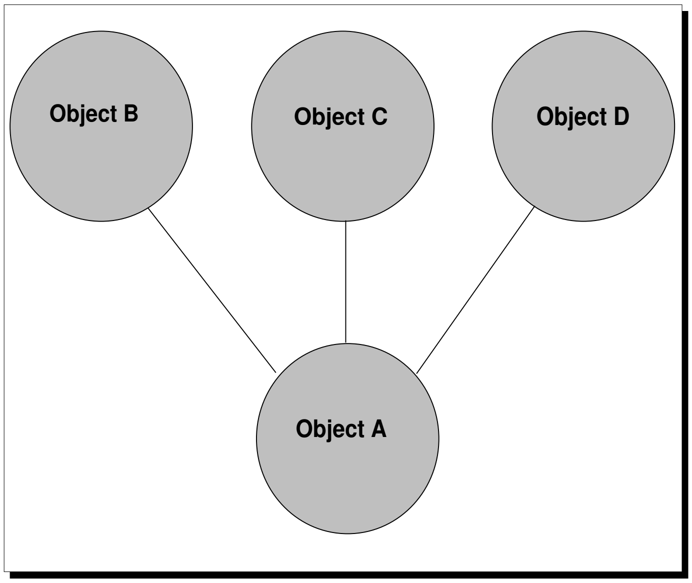

Figure 4.8: An example of when to destroy an entire hierarchial tree.

parent_objects variable. If it is not NULL, then there are objects that exist above this object. We will then call the Destroy function on one of these objects. When the parent_objects value is finally NULL, we then have to check if the ref_count is greater than one. If it is, then this implies that multiple subobjects used this object as the parent, so subtract the ref_count by one. If the ref_count is equal to one, we will call the term function. It will be called though on which variable was set when the method table was searched. If the destroy_exist variable was set, then we call the termination function by the 'o.term($option)' call. If the ref_count at this point equals 0, then it implies that this object is the last object we will destroy in the inheritance tree. It also is the only object with the ref_count equal to 0 and also means that it has no children. This means we should call the m_free_all function, to release all of the memory in the inheritance tree. If the $self variable does not equal 0, then we will call the delete_obj_instance function, described in Section 4.1.19. Note that if an object has a shadow object associated with it, then we will first detach the shadow object and then destroy the object.

Code of Destroy

```
/*
 * Destroy an object.
 */

void Destroy(object o, optional)
    {
    int destroy_exist = 0;
    int is_a_dummy_object = 0;
    struct _method * method;
    int counter;
    struct _object * obj;

    if (o == (object) NULL)
        {
        printf("MeldC Error:Can not destroy a NULL object.\n");
        return;
        }
    _frame->$.priority = 6;

    obj = (struct _object *)o;

    /*
     * Ask the meldc kernel for permission to
     * destroy this object.  If the permission is
     * deny, this thread will be suspended.    If any thread
     * that are still using this object, the kernel will deny
     * the destroy the request.
     */

    if (can_we_destroy_the_object(obj))
        {
        /*
         * If the kernel return the positive integer, it imply
         * multiple destroy is issued on the same object.
         * we will ignore this request.
         */
        return;
        }

    if (obj->$.shadow_obj_struct)
```

```
    {
    /*
     * Detach and maybe destroy the shadow object(Depend
     * on the shadow object termination function.
     */
    $self.DetachObject(o,(object)NULL);
    }

method = sym_method_table;

for (counter = 0; counter < sym_method_no; counter ++)
    {
    /*
     * sym_method_table->func_disp is the displacement
     * into the object structure that will be used to
     * store the func addrsss.
     */
    if (strcmp(method->name,"term") == 0)
        {
        destroy_exist = 1;
        }
    if (strcmp(method->name,"$term") == 0)
        {
        if (*((int *)
              (((char *)obj)+method->func_disp)) == 0)
            is_a_dummy_object = 1;
        }
    method ++;
    }

/*
 * Destroy the subobject from the inheritance.
 */

struct OBJ_INSTANCE * pointer;

pointer = parent_objects;

while (pointer != NULL)
    {
    object obj;
    char obj_name[_NAME_SIZE];
```

```
        struct _para * inst_table;

        sprintf(obj_name,"__$%s$%s",
             pointer->
               obj.class_instance_variable(C_FEATURE_NAME),
             pointer->o
               obj.class_instance_variable(C_NAME));

        inst_table = instance_var_table;

        while (inst_table ->name != NULL)
            {
            if (strcmp(inst_table->name,obj_name) == 0)
                {
                obj = (object)(*((int *)
                      (((char *)o)+inst_table->offset)));
                (void)pointer->obj.Destroy(obj,$option);
                break;
                }
            inst_table ++;
            }
        pointer = pointer -> next;
        }

    if (((struct _object *)o)->$.ref_count > 1)
        {
        /*
         * If the object reference count is not equal to 1,
         * it implied multiple subobject use this object
         * as the parent.
         * We should not free this object, until the
         * reference count reach one.
         */
        ((struct _object *)o)->$.ref_count --;
        }
    else
        {
        if (! is_a_dummy_object)
            o.$term();

        if (destroy_exist)
            o.term($option);
```

```
        if (((struct _object *)o)->$.ref_count == 0)
            {
            /*
             * In the inheritance hierarchy, there is only
             * one object which its reference count is
             * equal to zero.
             *
             * This object do not have any child and it is
             * the last object we will destroy in the
             * inheritance tree.
             */
            m_free_all(((struct _object *)o)->$.add_memory);
            }
        if ( $self != o)
            {
            (void)$self.delete_obj_instance((object) o);
            }
        }
    }
```

4.1.18 Term

Description of Term

The term function is used for the sole purpose of terminating all of the objects of a class. Before this is done, we must clean up the remains of the class. The function will do this for us. It has the purpose of actually destroying all of the objects associated with this class before we terminate. It will do so by continousily calling the Destroy function, discussed in Section 4.1.17. When all of the objects have been destroyed in this manner, we will then be allowed to terminate.

Code of Term

```
/*
 * Termination of the class, We will destroy all the
 * objects before we terminated.
 */
void term()
    {
    struct OBJ_INSTANCE * temp;
```

```
    object class_ptr;

    while(obj_instance)
        {
        temp  = obj_instance;
        obj_instance = obj_instance->next;
        class_ptr = (object)MetaClass.
             obtain_obj_info(temp->obj,OBJ_CLASS_PTR);
        class_ptr.Destroy(temp->obj);

        printf("complete %s %s \n",feature_name,name);
        }
    }
```

4.1.19 Delete_obj_instance

Description of Delete_obj_instance

The delete_obj_instance function is used in conjunction with the Destroy function. Here we will take an object as a pointer and search the instance table of the class until we find the object given. When we locate the object on the table, we will then free up the pointer and remove the object from the table.

Code of Delete_obj_instance

```
void delete_obj_instance(object obj)
    {
    struct OBJ_INSTANCE ** ptr;

    ptr = &obj_instance;

    while ( *ptr != NULL)
        {
        if ((*ptr)->obj == obj)
            {
            struct OBJ_INSTANCE * free_ptr;
            free_ptr = *ptr;
            *ptr = (*ptr)->next;
            m_free(_obj->$.add_memory,free_ptr);
            break;
            }
```

```
        ptr = &((*ptr)->next);
        }
    }
```

4.1.20 Init_parents

Description of Init_parents

The compiler will generate the inheritance information for an object as a char string during the initialization process of the object, object creation. During the initialization process, this function, init_parents, will be called. The function will look through the char string and construct the inheritance hierarchy and store it into the parent_objects structure.

The char string of parents will be passed into the function in the parents variable. It will then do a while loop until the parents string is empty. In each iteration of the while loop it will:

1. Call find_class_obj_from_name – This will find the parent_class_object using the parent_name.
2. Create an obj_instance node for a linked list.
3. Put the parent_class_object into the node.
4. Put the node into the hierarchial tree, the parent_objects structure.

Code of Init_parents

```
/*
 * builds a linked list of its parents
 */
void init_parents(char *parents)
{
    object              parent_class_object;
    char                parent_name[BUFSIZ];
    struct OBJ_INSTANCE *obj_instance;

#ifdef TRACE
fprintf(stderr,"MetaClass init_parents called by %s\n",name);
#endif

    while (*parents != '\0') {

       bzero(parent_name, BUFSIZ);
```

```
        sscanf(parents, "%[a-zA-Z0-9_$]", parent_name);
        parents += strlen (parent_name) +1;

        parent_class_object =
              (object)
              MetaClass.find_class_obj_from_name(parent_name);

        if (parent_class_object == (object)NULL) {
           fprintf(stderr,"MeldC runtime error: non-existent
                   class object -%s-\n", parent_name);
           continue;
        }

#ifdef TRACE
fprintf(stderr,"adding parent %s to %s\n",parent_name,name);
#endif

        /* add the parent class object */
        obj_instance = (struct OBJ_INSTANCE *)
                       m_malloc(&(_obj->$.add_memory),
                                sizeof(struct OBJ_INSTANCE));
        obj_instance->obj = (object) parent_class_object;
        obj_instance->next = NULL;

        if (parent_objects == NULL) {
           parent_objects = obj_instance;
        }
        else {
           obj_instance->next = parent_objects;
           parent_objects = obj_instance;
        }
    }
}
```

4.1.21 Find_class_obj_from_name

Description of Find_class_obj_from_name

The next function, find_class_obj_from_name, performs the simple task of finding the class object given the class name. This is done by searching the MetaClass table looking for a match to the class_name that was passed in. Note that since we start at the MetaClass, this can then only be called by the MetaClass.

Code of Find_class_obj_from_name

```
/*
 * Find the class object from the class name.
 *
 * IMPORTANT::
 *
 * This method can only be called by the metaClass.
 */

object find_class_obj_from_name(char * class_name)
    {
    char buf[MAX_METHOD_NAME * 2];
    struct OBJ_INSTANCE * temp;
    object obj;

    obj = MetaClass;

    if (obj != $self)
        return((object) NULL);

    temp = obj_instance;
    while (temp != ((struct OBJ_INSTANCE *)NULL))
        {
        char tmp[BUFSIZ];
        sprintf(tmp,
                "$%s$%s",
               ((MetaClass)(temp->obj))->feature_name,
               ((MetaClass)(temp->obj))->name);

        if (strcmp(tmp,class_name) == 0)
            return(temp->obj);
        else
            temp = temp->next;
        }
        return NULL;
    }
```

4.1.22 Find_obj_from_name

Description of Find_obj_from_name

This function, find_obj_from_name, will find an object given the object's name. It will be called by the object's class. The function will search the class's table until a name for some object is returned from the obtain_obj_info that matches the object name that we are looking for. When this occurs, we will return the object. If we don't find a match, NULL will be returned.

Code of Find_obj_from_name

```
/*
 * This routine will be called by object's class.
 */

object find_obj_from_name(char * obj_name)
    {
    char buf[MAX_METHOD_NAME * 2];
    struct OBJ_INSTANCE * temp;
    object obj;

    temp = obj_instance;

    while (temp != ((struct OBJ_INSTANCE *)NULL))
        {
        char * name;
        name = $self.obtain_obj_info((object)(temp->obj),u
                                              OBJ_NAME);

        if (strcmp(name,obj_name) == 0)
            return(temp->obj);
        else
            temp = temp->next;
        }
        return (object)NULL;
    }
```

4.1.23 Class_instance_variable

Description of Class_instance_variable

The class_instance_variable function is used to return the instance variables of a class depending on the parameter passed. The object can get the name of the object by passing C_NAME in the call to the function. It can also get the feature name of the object by passing C_FEATURE. It can find the size of the object by passing C_SIZE. It can also get the pre_sync_func, post_sync_func, string and symbol method tables, the parent objects or the instance variable table, all by passing a specified parameter.

Code of Class_instance_variable

```
char * class_instance_variable(enum instance type)
    {
    switch(type)
        {
        case C_FEATURE_NAME :
            return((char *) feature_name);
        case C_NAME:
            return((char *)name);
        case C_PRE_SYNC :
            return((char *)pre_sync_func);
        case C_POST_SYNC :
            return((char *) post_sync_func);
        case C_STR_METHOD_TABLE:
            return((char *) str_method_table);
        case C_SYM_METHOD_TABLE:
            return((char *) sym_method_table);
        case C_PARENT_OBJECTS :
            return((char *) parent_objects);
        case C_INSTANCE_VAR_TABLE :
            return((char *) instance_var_table);
        }
    }
```

4.1.24 Obtain_obj_info

Description of Obtain_obj_info

To obtain internal object information, the obtain_obj_info function can be used. The information you desire and the object is needed to do so. Then by using a switch statement, the information is pulled out of the object structure. The information that is available is:

- the object id
- the object's shadow object structure
- the object's class pointer
- the object name

Code of Obtain_obj_info

```
char * obtain_obj_info(object o, enum obj_field field)
    {
    struct _object * obj;

    obj = (struct _object *) o;

    switch (field)
        {
        case OBJ_ID :
            {
            return((char *)(&(obj->$.objid)));
            }
        case OBJ_SHADOW_OBJ:
            {
            return((char *)(obj->$.shadow_obj_struct->
                                          shadow_obj));
            }
        case OBJ_CLASS_PTR:
            {
            return((char *)(obj->$.class_ptr));
            }
        case OBJ_NAME:
            {
            return((char *)(obj->$.name));
            }
        default :
            {
            return((char *) NULL);
            }
        }
    }
```

4.1.25 Set_obj_info

Description of Set_obj_info

We can also set the object information using the set_obj_info. The object, field and value are passed and using the switch statement again, we can set the desired variable to the given value. The variables that we can set include the object id, the object name and the objects class pointer.

Code of Set_obj_info

```
void set_obj_info(object o,enum obj_field field,
                  char * value)
    {
    struct _object * obj;

    /*
     * obj : could be individual object, or the leaf
     *       object in the inheritance hierarchy.
     */

    obj = (struct  _object*) o;

    switch (field)
        {
        case OBJ_ID :
            {
            obj->$.objid = *((struct _objid *) value);
            break;
            }
        case OBJ_CLASS_PTR:
            {
            obj->$.class_ptr  = (struct _object *) value;
            break;
            }
        case OBJ_NAME:
            {
            /*
             * we need to set all the objects in the
             * inheritance hierarchy to the same name.
             */

            if (obj)
```

```
                {
                strncpy(obj->$.name,value,_NAME_SIZE);
                obj = obj->$.next;
                }
            break;
            }
        }
    }
```

4.1.26 Rest_current_frame_thread_id

Code of Rest_current_frame_thread_id

```
void reset_current_frame_thread_id(struct _threadid threadid)
    {
    if (_frame->$.caller != NULL)
        {
        _frame->$.caller->$.threadid = threadid;
        }
    }
```

4.1.27 GetObj & PutObj

Description of GetObj & PutObj

The GetObj function is used to get external sources. The most common, is for the retrieval of remote and persistent objects. What makes this function so special is that, the user can use this function at runtime to get an external object and not know the details of how this is done. In otherwords, this function is a high level protocol used by the user to mimic the importing of an object from some external source.

The PutObj function has also been designed for external objects. But this function will do the opposite of the GetObj. PutObj will allow a user to access an object on some external source. Basically, this function is a high level protocol used to mimic the exporting of an object to some external source.

Now let us look at both functions a little more closely. If a user does a GetObj before the PutObj was done the GetObj will fail. The reason for this is the way the GetObj works. The GetObj will send a message to the external source asking the external source if we have been granted access to the external object in question. This is done by checking the export table on the external source. This table will have an entry with the address of the external object. This entry will exist ONLY IF a PutObj has been done. The PutObj will simply find the object that will be requested and put the address of this object into the export table where a GetObj will find it. The GetObj will use the obj_init function to accomplish it's task, while the PutObj will use the register_obj function.

So to summarize these two very important functions:

1. They are used to mimic the importing and exporting of external objects at runtime, since we need some way to access the external objects at runtime, as compared to the import and export functions at compiler time.

2. They also provide the user with a high level protocol to do so.

Code of GetObj & PutObj

```
/*
 * GetObj ::
 * Importing the object...
 */

object GetObj(char * obj_name,object protocol_obj,optional)
    {
    object obj;
    obj = (object)protocol_obj.obj_init($self,obj_name,
                                        $option);
    if (obj != (object)NULL)
        {
        ((struct _object *)obj)->$.protocol_obj=
                            (struct _object *)protocol_obj;
        }
    return(obj);
    }

/*
 * Exporting an object, so that a remote meldc program can
 * access this object.
 *
 * Syntax : classname.Export(obj);
 */

int PutObj(object obj,char * name, object  protocol_obj)
    {
    return(protocol_obj.register_obj($self,obj,name));
```

```
    }
```

4.1.28 Binding

Description of Binding

The next function is the binding function for remote objects. It should first be mentioned the basic steps to follow for binding. The first step is to look through the method table and find the method name. Then compare each "type" of all the parameters. Then you can enqueue the frame to be processed.

We must get a frame pointer first, which is done by calling the external binding function. If the input frame does not contain the message structure and all of the parameters are in the linearized form, which means that the parameters are in ascii form so that they can be sent across the network as a message, then this function can be called. It will then allocate memory for the frame and call the unlinearized routine. This will then decode the buffer and store the input parameters into the frame. Note that external binding is for remote objects only. Then finally, the frame will be enqueued to be processed.

Code of Binding

```
void binding(object sender,object obj, char *input_parameter,
             char *ret, char * method,struct _method ** mp,
             enum msg_types msg_type, char * obj_param)
    {
    struct _frame * fp;
    char * ret_value;

    fp = binding.external_binding(fp,obj,input_parameter,
                                  &ret_value, method,
                                  msg_type,obj_param);
    if (fp == (struct _frame*) NULL)
        {
        printf("MeldC Fatal Error MetaClass binding():\n");
        printf("Unable to dynamic bind the message. \n");
        return;
        }

    fp->$.sender = (struct _object *)sender;
```

```
#ifdef STACK_BASE
    fp = reallocate_on_stack(fp);
#endif

    *mp = fp->$.mp;

    ENQUEUE_FRAME(fp);
    YIELD;
    /*
     * The dynamic  binding will allocate the memory for the
     * return value.  It will register the memory as the
     * current frame which is this binding routine.
     *
     * since we like the memory to be valid after the
     * destruction of this frame, we need to transter the
     * return value to the caller's frame.
     */

    if (ret_value != NULL)
        {
        struct _para *pp;

        pp =  (*mp)->para_table;

        for (;pp->name != 0 &&
             strcmp("$RET", pp->name) != 0; pp++);

        if (pp == NULL)
            {
            *ret = NULL;
            return;
            }
        /*
         * Critical Section can not be preemptived.
         */

        (*((int *)ret))  =
                 m_malloc(&(_frame->$.caller->$.add_memory),
                 pp->decl_type_ptr->size);

        memcpy((char *)(*((int *)ret)),(char *)ret_value,
               pp->decl_type_ptr->size);
```

```
        /*
         * End Critical Section.
         */
        }
    else
        *ret = NULL;
    }


end class  MetaClass
```

4.2 Class DynamicBinding

4.2.1 Description of DynamicBinding

We now have the DynamicBinding class. The most important thing to notice about this class is that it is written using ctype. This means that this part of the code will not be translated into MELDC, but directly into C. This is done for one and only one reason, for speed.

Headers for Class DynamicBinding

```
class ctype DynamicBinding ::=


methods :
```

4.2.2 Class_instance_variable

Description of Class_instance_variable

The class_instance_variable function is the same as the one discussed in Section 4.1.23, except that it is defined for the DynamicBinding class for use by the binding functions. Given a type and an object you will get the 'type' value returned. The values are feature name, object name, pre_sync function, post_sync function, string method table, symbol method table, parent objects and instance variable table.

Code of Class_instance_variable

```
char * class_instance_variable(enum instance type,
                               MetaClass obj)
```

```
    {
    switch(type)
        {
        case C_FEATURE_NAME :
            return((char *) obj->feature_name);
        case C_NAME:
            return((char *) obj->name);
        case C_PRE_SYNC :
            return((char *) obj->pre_sync_func);
        case C_POST_SYNC :
            return((char *) obj->post_sync_func);
        case C_STR_METHOD_TABLE:
            return((char *) obj->str_method_table);
        case C_SYM_METHOD_TABLE:
            return((char *) obj->sym_method_table);
        case C_PARENT_OBJECTS :
            return((char *) obj->parent_objects);
        case C_INSTANCE_VAR_TABLE :
            return((char *) obj->instance_var_table);
        }
    }
```

4.2.3 Msg_compare

Description of Msg_compare

The first function is the msg_compare. It will check a message if it is either symbol or string type. Given the type, it will do a simple compare check against the passed two strings it was given.

Code of Msg_compare

```
int msg_compare(char * s1, char * s2,
                enum msg_types msg_type)
{
  int flag1 = 0;
  int flag2 = 0;
  if(msg_type == (enum msg_types)STR_MSG) {

#ifdef NO
    if (strlen(s2) == 0)
        {
```

```
        return(1);
        }
#endif

    if(re_comp(s1) != 0){
      fprintf(stderr, "Error: re_comp fails\n");
      return(1);  /* not equal */
    }

    flag1 = re_exec(s2);

    /*
     * Do not allow user enter any regular expression string.
     */

#ifdef NO
    if(re_comp(s2) != 0){
      fprintf(stderr, "Error: re_comp fails\n");
      return(1);  /* not equal */
    }
    flag2 = re_exec(s1);
#endif

    return(!(flag1 || flag2));
  }
  else {
    return(strcmp(s1, s2));
  }
}
```

4.2.4 Type_verification

Description of Type_verification

The type_verification function will check to make sure that the two arguments it is passed have the same types. This is done using the parameter tables. This function will be used by the binding function, discussed below in Section 4.2.7, to check that the input parameters match the type in the parameter table.

Code of Type_verification

```
int type_verification(struct _elem_tab *arg1,
```

```
                  struct _elem_tab *arg2)
{
char * ptr1;
char * ptr2;

/*
 * If there is no type information, we will
 * assume the type information is correct.
 */

if(arg1 == NULL)
    return(0);

if ((strcmp(arg2->postfix_type,
             arg1->postfix_type) ==0))
    {
    return (0);
    }
if ((strcmp(arg2->c_type_desc,
            arg1->c_type_desc) ==0))
    {
    return(0);
    }
/*
 * Let relax some type checking rule.
 * Pointer and array are equivalent.
 */

ptr1 = arg2->c_type_desc;
ptr2 = arg1->c_type_desc;

while ( (*ptr1 != NULL) || (*ptr2 != NULL))
    {
    if ((*ptr1 == '*') && (*ptr2 == '*'))
        {
        ptr1 ++;
        ptr2 ++;
        continue;
        }
    else if (*ptr1 == '*')
        {
        if (*ptr2 == '[')
```

```
            {
            ptr2 = strchr(ptr2,']') + 1;
            }
        else
            break;
        ptr1 ++;
        continue;
        }
    else if (*ptr2 == '*')
        {
        if (*ptr1 == '[')
            {
            ptr1 = strchr(ptr1,']') + 1;
            }
        else
            break;
        ptr2 ++;
        continue;
        }
    else
        break;
    }
if (strcmp (ptr1,ptr2) == 0)
    {
    return(0);
    }
/*
 * Class and generic object are equivalent.
 */

ptr1 = arg2->postfix_type;
ptr2 = arg1->postfix_type;

if (strcmp(ptr1,"object") == 0)
    if (strncmp(ptr2,"class ",6) == 0)
        {
        return(0);
        }

if (strcmp(ptr2,"object") == 0)
    if (strncmp(ptr1,"class ",6) == 0)
        {
```

```
        return(0);
        }
    return(1);
    }
```

4.2.5 Internal_binding

Description of Internal_binding

The internal_binding structure has the responsibility of filling in all of the input parameters from the message structure constructed by the compiler. It will use the binding function within the DynamicBinding class to do so.

Code of Internal_binding

```
struct _frame *internal_binding(struct _frame *fp,
                       object obj, int *ret,
                       struct _elem_tab * usr_ret_type,
                       enum msg_types msg_type,
                       enum msg_types flag)
    {
    char method[MAX_METHOD_NAME];

    (void)strncpy(method,_frame->$.tmp_msg->_msg_name,
                  MAX_METHOD_NAME);

    fp = $self.binding(fp,obj,ret,msg_type,flag,NULL,
                       method,(char *)NULL);
    return(fp);
    }
```

4.2.6 External_binding

Description of External_binding

The external binding structure will be called if the input frame does not contain the message structure and all of the parameters are in linearized form. This will happen when we receive the frame from an external source. This structure will also call the DynamicBinding class binding function.

Code of External_binding

```
struct _frame *external_binding(struct _frame *fp, object obj,
```

```
                    char * input_buf,int *ret,
                    char * method_name, enum msg_types msg,
                    char * obj_param)
{
fp = $self.binding(fp,obj,ret,msg,SYNC_MSG,input_buf,
                    method_name,obj_param);
return(fp);
}
```

4.2.7 Binding

Description of Binding

The binding function will follow a series of basic steps, which are:

1. Look through the method table and find the method name.

2. Compare the type of each parameter to check for the proper format.

3. Then enqueue the frame so that it can be processed.

The function must first call the topological_sort function for the given method_name. This function will return to us a list that will be used to solve name conflicts. The list that is returned will be stored in the sorted_parents structure. See Section 4.2.8 for more on this function. Next, we will get the pre_sync_func and post_sync_func. These two functions are used to synchronize instance variables shared by different frames. When an instance variable is changed by one frame, we must make sure that all of the frames see this change. The pre_sync_func will synchronize the instance variables before the execution of a frame. The post_sync_func frame will synchronize the instance variables after the execution of a frame. Then we will get the str_method_table and the sym_method_table.

Now we must find the user specified method name. If we found the method name, then allocate a new frame, fp. If the input buffer passed to the binding function is not NULL, then we were passed parameters in linearized form. We must unlinearize these by calling the unlinearize function, see Section 2.9.9 for more. Now we must check to see if the function takes any arguments. This is done by checking the tmp_msg value. If it is not NULL, then parameters were passed. We must now fill in the parameter table and check the parameter types to make sure they match. Now we must take care of any optional parameters that are left over. Once the parameters for the frame are ready, we must call the pre_sync_func. Then we can put the frame on the ready queue to wait for execution, ENQUEUE_FRAME. Then we will YIELD, to ensure that the parent frame is executed before the child's frame. After we have finished executing the frame, we will do a post_sync_func call. Now that this method is done, repeat for the next method if more exist.

If we did not find the method name we must use the delgation concept. That is, check to see if there is a meldc_default method defined by the user. If so, we must call that method. We will assume that the meldc_default takes at least one argument, the name of the method. If the meldc_default function succeeds, then allocate a new frame. Copy the parameters of the frame into the parameter structure. Then do the same for the optional parameters. When we are done, return the frame.

If we could not find the method name and meldc_default failed, then binding also failed. When this happens, return a NULL frame structure.

Code of Binding

```
struct _frame *binding(struct _frame *fp, object obj,
                       int *ret, enum msg_types msg_type,
                       enum msg_types flag,
                       char * input_buffer,
                       char * method, char * obj_param)
{
  struct _method *mp;
  struct _para *pp = NULL;
  char token[MAX_METHOD_NAME];
  MetaClass parent_class;

  struct _method *str_method_table;
  struct _method *sym_method_table;

  struct OBJ_INSTANCE *sorted_parents = NULL;
  struct OBJ_INSTANCE *sub_objects = NULL;
  struct OBJ_INSTANCE *co, *o;

  *ret = NULL;
  if (obj == (object) NULL)
      {
      printf("Dynamic Binding: MeldC internal
              inconsistent error. \n");
      printf("NULL object detected. \n");
      exit(-1);
      }
```

```
  parent_class =
      (MetaClass)(((struct _object *)obj)->$.class_ptr);

  /*
   * Obtain the method name.
   */
  if (!strcmp(method,"$init") || !strcmp(method,"init"))
       {
       /*
        * special case for $init()
        * Don't merge $init()
        */
       sorted_parents = (struct OBJ_INSTANCE *)
                        m_malloc(&(_frame->$.add_memory),
                        sizeof(struct OBJ_INSTANCE));
       sorted_parents->obj = parent_class;
       sorted_parents->next = NULL;

       sub_objects = (struct OBJ_INSTANCE *)
                        m_malloc(&(_frame->$.add_memory),
                        sizeof(struct OBJ_INSTANCE));
       sub_objects->obj = obj;
       sub_objects->next = NULL;
       }
  else
       {
#ifdef OVERRIDE
       /*
        * Set the object equal to its virtual object.
        */

       if ((msg_type == VIRTUAL_SYM_MSG) ||
           (msg_type == VIRTUAL_STR_MSG))
           {
           if (msg_type == VIRTUAL_SYM_MSG)
               msg_type = SYM_MSG;
           else
               msg_type = STR_MSG;

           obj = (object)
            (((struct _object *)obj)->$.virtual_self);
```

```
          }
#endif
       /*
        * traverse up the class hierarchy
        */
       sorted_parents =
                   $self.topological_sort(method,
                                         parent_class,
                                         msg_type,
                                         &obj,
                                         sorted_parents,
                                         &sub_objects);
       }

  for (co = sorted_parents, o = sub_objects;
       co && o ;
       co = co->next, o = o->next)
    {
    MetaClass ancester;
    void (*pre_sync_func)(), (*post_sync_func)();
    int need_to_sync;
    char ancester_name[BUFSIZ];

#ifdef OVERRIDE
    /*
     * override semantics for inheritance find the
     * last class object that contains the method
     * to bind
     */
    if (co->next && o->next) continue;

#endif OVERRIDE

    ancester = (MetaClass) co->obj;
    sprintf(ancester_name, "$%s$%s",
            $self.class_instance_variable
                         (C_FEATURE_NAME,ancester),
            $self.class_instance_variable
                         (C_NAME,ancester));

    need_to_sync = FALSE;
```

```
    pre_sync_func =
             (void (*)())$self.
             class_instance_variable(C_PRE_SYNC,
             ancester);
    post_sync_func =
             (void (*)())$self.
             class_instance_variable(C_POST_SYNC,
             ancester);
#ifndef OVERRIDE
    /*
     * merge semantics for inheritance, thus need
     * to call functions to synchronize duplicating
     * instance variables
     */
if (pre_sync_func!=NULL && post_sync_func!=NULL)
    need_to_sync = TRUE;
#else
    /*
     * override semantics for inheritance
     * should NOT need to sync instance variables
     */
    need_to_sync = FALSE;

#endif OVERRIDE

    str_method_table = (struct _method *)
              $self.class_instance_variable(
              C_STR_METHOD_TABLE, ancester);
    sym_method_table = (struct _method *)
              $self.class_instance_variable(
              C_SYM_METHOD_TABLE, ancester);

     for (mp =
         (msg_type == (enum msg_types)STR_MSG ?
                        str_method_table :
                       sym_method_table);
                       mp->name != 0;
                       mp++)
        {
            if ($self.msg_compare(mp->name, method,
                                  msg_type) == 0)
               {
```

```
#ifndef OVERRIDE
               /*
                * merge semantics for inheritance
                * if ASYNC_MSG but merging parent's
                * methods still want to setup caller,
                * otherwise it never comes back to
                * queue other frames !!
                */
#ifndef STACK_BASE
               NEW_FRAME(fp, mp->frame_size, o->obj,
                        *((int *) (((char *)o->
                        obj)+mp->func_disp)),
                        (flag == ASYNC_MSG &&
                         !co->next ? 0 : _frame),mp,
                        (flag == ASYNC_MSG ? ASYNC :
                         SYNC));
#else
               NEW_DYNAMIC_FRAME(fp, mp->frame_size,
                        o->obj, *((int *)(((char *)o->
                        obj)+mp->func_disp)),
                        (flag == ASYNC_MSG &&
                        !co->next ? 0 : _frame),mp,
                        (flag == ASYNC_MSG ? ASYNC :
                        SYNC));
#endif

#else
               /*
                * override semantics for inheritance
                */
               NEW_FRAME(fp, mp->frame_size, o->obj,
                        *((int *)(((char *)o->
                        obj)+mp->func_disp)),
                        (flag == ASYNC_MSG ? 0 :
                        _frame),mp,
                        (flag == ASYNC_MSG ? ASYNC :
                        SYNC));
#endif OVERRIDE
               /*
                * If the input buffer is not NULL, then
                * the paramters is passed as the
                * linearize form.  We will unlinearize
```

```
 * the parameters.
 */

if (input_buffer != NULL)
    {
    unlinearize(fp,input_buffer,
                obj_param,NULL);
    }

/*
 * Is this a symbol message or a string
 * message ?
 */

fp->$.str_or_sym = msg_type;

/*
 * Make sure that, this function takes
 * some arguements.
 */

if ((input_buffer == NULL) &&
    (_frame->$.tmp_msg) &&
    (_frame->$.tmp_msg->_num_arg !=0))
    {
    int counter = 0;
    struct _elem_tab * elem_tab;

    for (pp =
         (struct _para *)mp->para_table;
         pp->name != 0 &&
         strcmp("$RET", pp->name) != 0;
         pp++)
        {

        /*
         * Make sure type of the input
         * paramtere match the type in
         * the parameter table.
         */
       if ((counter+1) > _frame->
                   $.tmp_msg->_num_arg)
```

```
    {
    fprintf(stderr,
            "MeldC rutnime error :
             insufficient paramters
             pass into the method [%s],
             class [%s].  Process
             continues anyway. Good
             luck!\n", method, $self.
             class_instance_variable
             (C_NAME,ancester));
    }

if ($self.type_verification(
    _frame->$.tmp_msg->
                   _para[counter].type,
    pp->decl_type_ptr))
    {
    fprintf(stderr,
          "MeldC runtime error:
           inconsistent type [%s] on
           parameter binding [%s]\non
           method [%s], class [%s].
           Process continues anyway.
           Good luck!\n", pp->
           decl_type_ptr->postfix_type,
           _frame->$.tmp_msg->
           _para[counter].type->
           postfix_type, method, $self.
           class_instance_variable
           (C_NAME,ancester));
    }

/*
 * Obtain the type pointer of the
 * variable.
 */
elem_tab = _frame->$.tmp_msg->
           _para[counter].type;

if (_frame->$.tmp_msg->
           _para[counter].dereference)
    {
```

```
            memcpy(((char *) fp + pp->offset),
               (char *)((int) (_frame->
               $.tmp_msg->_para[counter].offset)
               + (int)(_frame->$.tmp_msg)), 4);
            }
        else
            memcpy(((char *) fp + pp->offset),
               (char *)((int) (_frame->
               $.tmp_msg->_para[counter].offset)
               + (int)(_frame->$.tmp_msg)),
               elem_tab->size);

        counter ++;

        }

    /*
     * method has optional parameters
     */
    if (mp->opt && counter !=
         _frame->$.tmp_msg->_num_arg)
        {
        int size;
        int num_args;
        int size_copied;
        /*
         * pack optional parameters
         */
        num_args=_frame->$.tmp_msg->_num_arg -
                 counter;

        if (_frame->$.tmp_msg->
            _para[counter].dereference)
            {
            size_copied =
            ((char *)_frame->$.tmp_msg->_size)
            - (_frame->$.tmp_msg->
              _para[counter].offset);
            }
        else
            size_copied =
            (_frame->$.tmp_msg->_size)    -
```

```
    (_frame->$.tmp_msg->
    _para[counter].offset);

size = sizeof(struct _msg) +
       num_args *
       sizeof(struct _arg_desc) +
       size_copied;

if (size < BUFSIZ)
    size = BUFSIZ;
fp->$.msg =
  (struct _msg *)
  m_malloc(&(fp->$.add_memory),
  size);

fp->$.msg->_type =
          _frame->$.tmp_msg->_type;
fp->$.msg->_size = size;
fp->$.msg->_msg_type =
      _frame->$.tmp_msg->_msg_type;
fp->$.msg->_msg_name =
         STRDUP(_frame->$.tmp_msg->
         _msg_name);
fp->$.msg->_return_type =
   _frame->$.tmp_msg->_return_type;
fp->$.msg->_return_addr =
   _frame->$.tmp_msg->_return_addr;
fp->$.msg->_num_arg = num_args;

memcpy(&fp->$.msg->_para[0],
       &_frame->$.tmp_msg->
       _para[counter], num_args *
       sizeof(struct _arg_desc));

memcpy(&fp->$.msg->_para[num_args],
      (int)(_frame->$.tmp_msg) +
      (int)(_frame->$.tmp_msg->
      _para[counter].offset),
      size_copied);
/*
 * Readjust the offset.
 */
```

```
     MSG_ADJUST_OFFSET(fp->$.msg,
                       &fp->$.msg->
                       _para[num_args],
                       0);
   }
else
   fp->$.msg = NULL;
} /* end if num_args != 0 */

if (pp == NULL)
   {
   pp = (struct _para *)mp->para_table;
   if (pp == NULL)
       {
       if (co->next)
          {
          if (need_to_sync)
             pre_sync_func(o->obj,
                        ancester_name);
          ENQUEUE_FRAME(fp);
          YIELD; /* ensure that parent
                  * frame is executed
                  * before child's
                  * frame */
          if (need_to_sync)
             post_sync_func(o->obj,
                        ancester_name);
          }
       else
          {
          return(fp);
          }
       break;
       }
   }
/*
 * We are assume that $selector do not
 * take any arguement.
 */
if(pp->name != 0 &&
   strcmp("$selector", pp->name) == 0)
    {
```

```
    char * pointer;
    /*
     * Allocate the memory for $selector.
     * we need to free it later.
     */
    if (method != (char *)NULL)
        pointer =
            m_malloc(&(fp->$.add_memory),
                     strlen(method) + 1);

    strncpy(pointer,method,
            strlen(method)+1);
    /*
    bcopy(pointer,
          ((char *) fp + pp->offset),
          strlen(pointer)+1);
    */
    bcopy(&pointer,
          ((char *) fp + pp->offset),
          sizeof(char *));
    pp++;
    }

for (;pp->name != 0 &&
     strcmp("$RET", pp->name) != 0;
     pp++);

if (pp->name != 0 &&
    strcmp("$RET", pp->name) == 0)
   {
   if (flag != ASYNC_MSG)
     *ret =
         m_malloc(&(_frame->$.add_memory),
              pp->decl_type_ptr->size);
   else
     *ret =
     m_malloc(&(fp->$.add_memory),
              pp->decl_type_ptr->size);

   bcopy(ret,((char *)fp + pp->offset),
         sizeof(ret));
   }
```

```
                  if (co->next)
                     {
                     if (need_to_sync)
                        pre_sync_func(o->obj,
                                      ancester_name);
                     ENQUEUE_FRAME(fp);
                     YIELD; /* ensure that parent's
                             * frame is executed
                             * before child's */
                     if (need_to_sync)
                        post_sync_func(o->obj,
                                       ancester_name);
                     }
                  else
                     {
                     return(fp);
                     }
                  break;
               }/* end if msg_compare */

        } /* try next method */

    } /* end for */

  /*
   * We are unable to find the method that user specified.
   *
   * If there is a "default" method define by the user,
   * then the default method will be called. (The concept
   * of delegation)
   *
   * We will assume that the "default" function takes at
   * least one arguement (the method name).
   *
   * meldc_deault(char * method_name, optional);
   */

  sym_method_table = (struct _method *)
             $self.class_instance_variable(
             C_SYM_METHOD_TABLE,parent_class);
```

```
  for(mp = sym_method_table; mp->name != 0; mp++)
      {
      char * message;
      int  * size;
      struct _para *return_pp = NULL;

      if($self.msg_compare(mp->name, "meldc_default",
                           SYM_MSG) == 0)
          {
#ifndef STACK_BASE
          NEW_FRAME(fp, mp->frame_size, obj, *((int *)
                    (((char *)obj)+mp->func_disp)),
                    (flag ==ASYNC_MSG ? 0 : _frame),mp,
                    (flag == ASYNC_MSG ? ASYNC : SYNC));
#else
          NEW_DYNAMIC_FRAME(fp, mp->frame_size, obj,
                *((int *)(((char *)obj)+mp->func_disp)),
                (flag ==ASYNC_MSG ? 0 : _frame),mp,
                (flag == ASYNC_MSG ? ASYNC : SYNC));
#endif

          pp = (struct _para *)mp->para_table;

          /*
           * The default function need at least one
           * arguement.
           */

          if ((pp == NULL) || (pp->name == 0))
              {
              printf("MeldC Fatal Error: Incorrect
                     MeldC Default Method Fromat.\n");
              printf("                   : Incorrect
                     fromat meldc_default
                     (char * method_name, optional);
                     \n");

              return (struct _frame *)NULL;
              }
          /*
           * The first arguement should have the
           * type (char *).
```

```
 */
if (strcmp(pp->decl_type,"* char"))
    {
    /*
     * default function did not provide the
     * right type of parameters.
     */
    printf("MeldC Warning : Incorrect MeldC
            Default Method Fromat. \n");
    printf("                  :1st paramtere
            of meldc_default has
            a incompatible type \n");

    printf("                  : The Correct
            type = (char *), and
            input=(%s) \n",pp->decl_type);
    }

if((pp->name != 0) &&
   (strcmp("$RET", pp->name) != 0))
    {
    char * name;
    /*
     * The first parameter has to be the
     * methodname.
     */
    name  = m_malloc(&(fp->$.add_memory),
                      strlen(method)+1);

    strncpy(name,method,strlen(method)+1);

    *((int *)((char *)fp + pp->offset)) =
                                     name;
    pp ++;
    }

 if ((input_buffer == NULL) &&
     (_frame->$.tmp_msg->_num_arg !=0))
     {
     int counter = 0;
     struct _elem_tab * elem_tab;
```

```
for (; pp->name != 0 &&
     strcmp("$RET", pp->name) != 0;
     pp++)
    {
    /*
     * Make sure type of the input
     * paramtere match the type in
     * the parameter table.
     */
   if ($self.type_verification(
       _frame->$.tmp_msg->
       _para[counter].type,
       pp->decl_type_ptr))
       {
       fprintf(stderr, "MeldC runtime
               error: inconsistent type
               [%s] on parameter binding
               [%s]\non method
               meldc_default Process
               continues anyway.
               Good luck!\n",
               pp->decl_type_ptr->
               postfix_type,
               _frame->$.tmp_msg->
               _para[counter].type->
               postfix_type);
       }
   /*
    * Obtain the type pointer of the
    * variable.
    */
   elem_tab = _frame->$.tmp_msg->
              _para[counter].type;

   /*
    * Copy the value of the pointer into
    * the frame structure.
    */
   if (_frame->$.tmp_msg->_para[counter].
       dereference)
       {
       memcpy(((char *) fp + pp->offset),
```

```
                (char *)((int)(_frame->
                $.tmp_msg) + _frame->
                $.tmp_msg->
                _para[counter].offset),4);
        }
    else
        memcpy(((char *) fp + pp->offset),
                (char *)((int)(_frame->
                $.tmp_msg) + _frame->
                $.tmp_msg->
                _para[counter].offset),
                elem_tab->size);
    counter ++;
    }

/*
 * method has optional parameters
 */
if (mp->opt && counter !=
    _frame->$.tmp_msg->_num_arg)
   {
   int size;
   int num_args;
   int size_copied;
   /*
    * pack optional parameters
    */
   num_args = _frame->$.tmp_msg->_num_arg -
              counter;

   if (_frame->$.tmp_msg->_para[counter].
       dereference)
       {
       size_copied=(_frame->$.tmp_msg->
                  _size) - (_frame->
                  $.tmp_msg->
                  _para[counter].offset);
       }
   else
       size_copied=(_frame->$.tmp_msg->
                  _size) - _frame->
                  $.tmp_msg->
```

```
                        _para[counter].offset;

                size = sizeof(struct _msg) + num_args
                       * sizeof(struct _arg_desc) +
                       size_copied;
                if (size < BUFSIZ)
                    size = BUFSIZ;
                fp->$.msg = (struct _msg *)
                            m_malloc(&(fp->$.add_memory),
                                       size);
                fp->$.msg->_type =
                                _frame->$.tmp_msg->_type;
                fp->$.msg->_size = size;
                fp->$.msg->_msg_type =
                            _frame->$.tmp_msg->_msg_type;
                fp->$.msg->_msg_name =
                    STRDUP(_frame->$.tmp_msg->_msg_name);
                fp->$.msg->_return_type =
                         _frame->$.tmp_msg->_return_type;
                fp->$.msg->_return_addr =
                         _frame->$.tmp_msg->_return_addr;
                fp->$.msg->_num_arg = num_args;

                memcpy(&fp->$.msg->_para[0],
                    &_frame->$.tmp_msg->_para[counter],
                    num_args * sizeof(struct _arg_desc));

                memcpy(&fp->$.msg->_para[num_args],
                       (int)(_frame->$.tmp_msg)+
                       _frame->$.tmp_msg->
                       _para[counter].offset,
                       size_copied);

                MSG_ADJUST_OFFSET(fp->$.msg, &fp->
                                  $.msg->_para[num_args],
                                  0);
              }
            else
               fp->$.msg = NULL;
            } /* end if num_args != 0 */

        for (;pp->name != 0 &&
```

```
               strcmp("$RET", pp->name) != 0;
               pp++);

          if (pp->name != 0 &&
              strcmp("$RET", pp->name) == 0)
             {
             if (flag != ASYNC_MSG)
                 *ret=m_malloc(&(_frame->$.add_memory),
                              pp->decl_type_ptr->size);
             else
                 *ret=m_malloc(&(fp->$.add_memory),
                              pp->decl_type_ptr->size);
             bcopy(ret, ((char *) fp + pp->offset),
                   sizeof(ret));
             }
          return(fp);
          }
      }
  /*
   * Dynamic binding has failed.
   */

  fprintf(stderr, "MeldC runtime error: no %s method [%s]
          in class [%s]\n", msg_type ==
          (enum msg_types)STR_MSG ? "str" : "sym", method,
          $self.class_instance_variable
          (C_NAME,parent_class));
  return((struct _frame *) NULL);
}
```

4.2.8 Topological_sort

Description of Topological_sort

The topological_sort function is used by the binding function for inheritance to resolve name conflict. The function will only search on the given method name. If we are using OVERRIDE, then the function will find the name to be used to resolve the name conflict and return a pointer to that name's node in the linked list, the tail node. If we are using MERGE, then we will find the order to be used to resolve a name conflict and return a linked list. In this case, we will resolve any conflicts by starting at the head of the linked list and work our way to the end of the list. See Figure 4.9 for

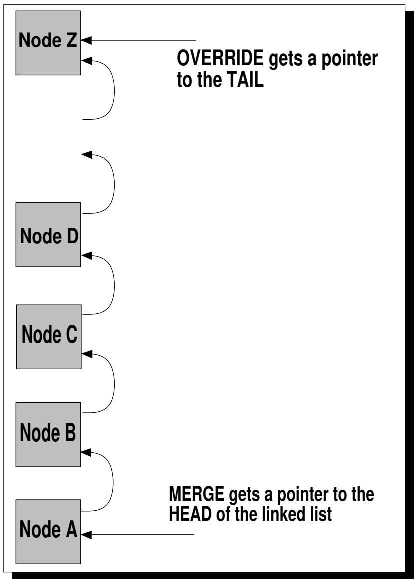

Figure 4.9: An example of which node resolves name conflict.

an example of this. The MERGE semantics would begin at node A. The OVERRIDE semantics would use node Z.

The method name is obtained first by the function. Next, a "critical section" is declared around the sort. We then try to obtain the method. Now that we have the method we can search the parent objects. To find the parent_objects, we use the class_instance_variable function with the parameter type of C_PARENT_OBJECTS. This information was stored by the compiler in the instance variables. With the parent_objects, we will call the topological_sort on each of them recursively. These calls on the parent_objects will climb up the hierarchy tree for us. When it is done going up the tree, it will return the list.

Note that it is assumed that there is no circular dependency. If there was, then the function would never terminate.

Code of Topological_sort

```
/*
 * returns a list of class objects sorted in reverse
 * topological order up the class hierarchy starting
 * from 'obj' which is a class object, and contains
```

```
 * the method to do dynamic binding on
 */
struct OBJ_INSTANCE *topological_sort(char * method,
                    MetaClass class_obj,
                    enum msg_types msg_type,
                    object *obj_p,
                    struct OBJ_INSTANCE *sorted_parents,
                    struct OBJ_INSTANCE **sub_objects)
{
  struct _method *mp;
  struct _method *str_method_table;
  struct _method *sym_method_table;
  struct OBJ_INSTANCE *tmp_object;
  struct OBJ_INSTANCE *parent_objects;
  object *part_of_obj_p;

  BEGIN_CRITICAL_SEC;
  str_method_table =
  (struct _method *)$self.class_instance_variable(
                          C_STR_METHOD_TABLE,class_obj);

  sym_method_table =
  (struct _method *)$self.class_instance_variable(
                          C_SYM_METHOD_TABLE,class_obj);

  /* find the method */
  for(mp = (msg_type == (enum msg_types)STR_MSG ?
                   str_method_table : sym_method_table);
      mp->name != 0;
      mp++) {

      if ($self.msg_compare(mp->name, method,
                            msg_type) == 0) {
         /*
          * add the class of class_obj to head of the list
          */
         tmp_object = (struct OBJ_INSTANCE *)
                      m_malloc(&(_frame->$.add_memory),
                      sizeof(struct OBJ_INSTANCE));
         tmp_object->obj = class_obj;
         tmp_object->next = NULL;
```

```
        if (sorted_parents == NULL) {
           sorted_parents = tmp_object;
        }
        else {
           tmp_object->next = sorted_parents;
           sorted_parents = tmp_object;
        }
        /*
         * add the obj to the sub_objects list
         */
        tmp_object = (struct OBJ_INSTANCE *)
                     m_malloc(&(_frame->$.add_memory),
                     sizeof(struct OBJ_INSTANCE));
        tmp_object->obj = *obj_p;
        tmp_object->next = NULL;

        if (sub_objects == NULL) {
           *sub_objects = tmp_object;
        }
        else {
           tmp_object->next = *sub_objects;
           *sub_objects = tmp_object;
        }
        break;
    }
}
/*
 * search in its parent class objects. The parent
 * classes information are generated by the compiler,
 * they are store as one of the instance variables
 * in object's class.
 *
 * Find the parent objects.
 */

parent_objects =(struct OBJ_INSTANCE *)
                $self.class_instance_variable(
                C_PARENT_OBJECTS,class_obj);

for (; parent_objects;
     parent_objects = parent_objects->next) {
```

```
struct _para *ip = NULL;
char buf[BUFSIZ];
/*
 * Run through each parent object, and call
 * the topological_sort recursively for each
 * parent objects.
 */

/*
 * find from the instance_var_table the offset
 * of "__$<f_name>$<c_name>" which represent
 * the parent sub_object
 */
bzero(buf, BUFSIZ);
sprintf(buf, "__$%s$%s",
     $self.class_instance_variable(
           C_FEATURE_NAME,parent_objects->obj),
     $self.class_instance_variable(
           C_NAME,parent_objects->obj));
/*
 * The instance variables table, not only
 * contain the information about object's
 * instance. It also contain the information
 * about the parent sub_object.  The address
 * of parent sub_object is a part of the
 * "object". Instance variables table contain
 * such information, and it is described as
 * the offset into the "object"
 */

for(ip = (struct _para *)
         $self.class_instance_variable(
              C_INSTANCE_VAR_TABLE,class_obj);
    ip->name != 0 && strcmp(ip->name, buf);
    ip++);

if (!ip) {
   fprintf(stderr, "MeldC runtime error:
           topological sort failed\n");
   exit(-1);
}
```

```
        part_of_obj_p = (object*) ((char *) *obj_p +
                                          ip->offset);

        /*
         * recursively call on parents' class objects
         * up the hierarchy
         */
        sorted_parents = $self.topological_sort(method,
                         (MetaClass) parent_objects->obj,
                         msg_type,
                         part_of_obj_p,
                         sorted_parents,
                         sub_objects);
  }
  END_CRITICAL_SEC;

  return (sorted_parents);
}


end class DynamicBinding

end feature MeldC
```

Chapter 5

Shadow, Remote & Persistent Objects

This chapter will first discuss the concept of Shadow Objects. Afterwards, in Section 5.2 and Section 5.3, we will discuss two special types of Shadow Objects of the MELDC environment, Remote Objects and Persistent Objects. Both of these object types are developed ideas that have been implemented in MELDC. In each respective section, we will discuss what they are used for in MELDC and how they are generally implemented in the system. In the Runtime Persistent Object chapter, Chapter 7, we will discuss Persistent Objects in much more detail by discussing the code used to implement them. In the Runtime Remote Objects chapter, Chapter 8, we will discuss Remote Objects in more detail by also discussing this code used to implement Remote Objects.

5.1 Introduction to Shadow Objects

One of the important concepts in MELDC is supporting dynamic extension of object behavior based on the reflective architecture of MELDC. The extended behavior of an object is referred to as its *secondary behavior* to distinguish from the *primary behavior* defined in the class of the object. Extending object behavior in MELDC is characterized by two properties: (1) composability and (2) decomposability. Composability states that primary behavior, which implements the interface of objects, can be modified by composing with multiple secondary behaviors without changing the objects' interface. Decomposability describes the reverse property of composability. Primary behavior encompasses an object's functionality as defined in the object's class definition or provided through an inheritance mechanism. Secondary behavior encompasses dynamically added functionality which is in most cases orthogonal to primary behavior and to other secondary behaviors.

MELDC provides a mechanism called *shadowing* to implement secondary behaviors. The "shadow" idea implies dynamic, transient and orthogonal effects upon primary behavior. A secondary behavior of an object **o** is implemented by being attached with a second object, called the *shadow object* of **o** (referred to as $S_{\mathbf{o}}$). We say that object **o** is *shadowed* by $S_{\mathbf{o}}$ or $S_{\mathbf{o}}$ casts a shadow upon **o**. The primary behavior of $S_{\mathbf{o}}$ is the secondary behavior of **o**. The shadow object $S_{\mathbf{o}}$ alone is a regular object, whose primary behavior is defined by its own class . It has its own private data as well as its own threads of control. It can be shadowed by yet another object, which would then implement a secondary behavior of $S_{\mathbf{o}}$. It should be noted though, that an object cannot be attached directly or indirectly to itself as a shadow object.

The attachment used to establish the connection between objects and their shadow objects is a method (`attach`) which is defined in the metaclass. To attach $S_{\mathbf{o}}$ to **o**, a message is sent to the class of **o** with four parameters:

$$\mathrm{Class}_{\mathbf{o}}.\mathrm{attach}(\mathbf{o}, \mathrm{S}_{\mathbf{o}}, \mathrm{m}_{entry}, \mathrm{m}_{exit})$$

The third parameter, m_{entry}, indicates the entry method to the shadow object $S_{\mathbf{o}}$ and the forth, m_{exit}, indicates the exit method. Any message m sent to object **o** will be trapped and forwarded to the method m_{entry} of $S_{\mathbf{o}}$ before it is delivered to **o**. After **o** completes executing method m, m_{exit} of $S_{\mathbf{o}}$ is invoked.

An object may have a sequence of shadow objects, $S_1, S_2 \ldots S_n$, attached to it, where S_n is the last one attached and S_1 is the first one being attached. The entry method of S_n is the first one to be executed and that of S_1 is the last. After **o** completes executing its method, the exit method of S_1 is executed first and that of S_n is the last one to be executed. Any shadow object $S_{\mathbf{o}}$ may be detached from object **o** as long as there is no active message traveling through the set of shadow objects. If there are any active messages, the detachment operation is delayed until the message is completed. Shadow objects are the basic building blocks to dynamically change program behaviors. Figure 5.1 shows the control flow of an object shadowed by a set of objects.

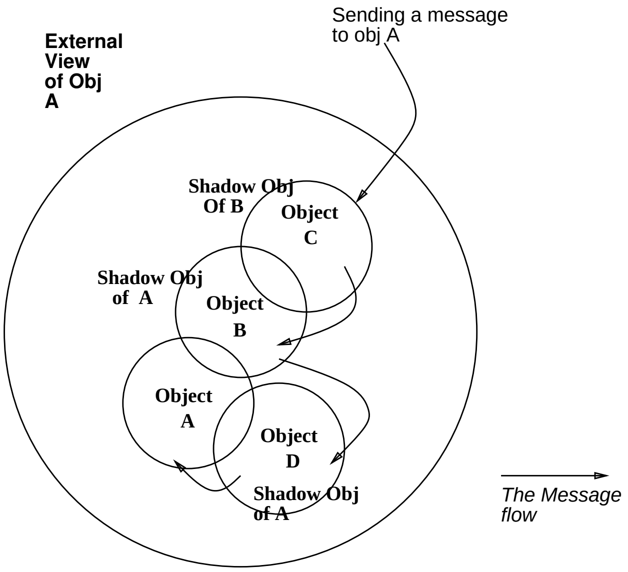

Figure 5.1: Simple Examples of Shadow Objects

5.2 Introduction to Remote Objects

Basically, Remote Objects are just like any other object in MELDC, with one exception. These objects have only one objective when they are executed and it is always the same objective. This objective deals with the external sources to be used by some MELDC object. When an object wants to interact with some other remote object, an object on an external machine, it will need to communicate with this object. To do so, Remote Objects have been created using the Shadow Object concept. These objects will be created at such a time as when an object tries to "call" the Remote Object. When the "call" is made, the system will create a Shadow Object and "attach" this shadow to the object making the "call". This shadow will now be used to communicate with the object it is attached to, with the Remote Object that communication has been established with.

5.2.1 Setting Up a Connection

Refer to Figure 5.2 first to see a picture of what is happening. Here we can see that object A is on it's own local machine. Object A though, wants access to object B on the remote side. Object a will first check it's own

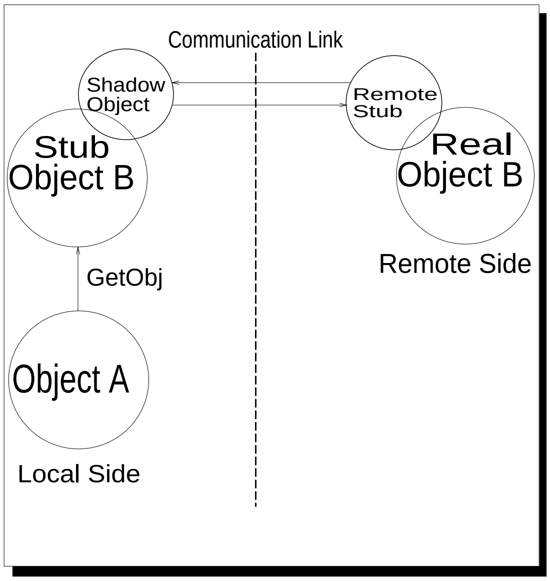

Figure 5.2:

import table and see if object B has already been accessed by object A. If object A has not yet imported object B, then object A will do a GetObj call. This call is a high level call that allows the user to gain access to a Remote Object at runtime and hide all of the details of the process from the user. The GetObj will trigger a whole series of events. It will create the Shadow Object and then attach the Shadow Object to object A. Then a remote stub, a Remote Object, will be created allowing the two sides communication. Now this GetObj call will be sent across the network to the remote stub or Remote Object.

The remote stub will see the message come in and recognize that it is a request for access to one of it's local objects. At this point, the remote stub will check an export table to see if object A has been granted permission to access this object B. The only way that the local side can gain access, is if the remote side has already done a PutObj. This PutObj is also a high level call that puts an object into the export table, thereby granting permission to some other source access to that object. So, when the remote stub checks to see if the local side has access to object B, the remote stub will check the export table to see if it has been exported yet. If the object has not been exported, the GetObj will return NULL. Otherwise, the address of the remote object B will be given as a return value. This will allow the local side to access the object B directly.

5.2.2 Creation & Destruction of Remote Objects

When a Shadow Object is created, a series of events must occur. We must first allow the MELDC environment to know that the shadow exists. This is done by first changing the object's method table. Here, we will point to the shadow_object_entry_interrupt point. This is where frames will be manipulated so that the Shadow Object will be able to execute the frame properly. This is especially true in the case of passing an object as a parameter, see Section 8.5.6 for more detail. We must also inform the object by putting the Shadow Object's address into the object structure. We also will define a shadow_obj_init_point. This is used to initialize certain values of the Shadow Object when first created. When a Shadow Object is created, it must then be associated with some Remote Object. It can be either associated with a Remote Object one-to-one. That is, there is two Shadow Objects, one on either side. There can also be a one-to-many relationship. Here, there is a Shadow Object that deals with many other Remote Objects. This is true for the $sender part of MELDC, see Section 8.7 for more details on the $sender implementation.

Now when a Shadow Object is to be terminated, we have another series of steps to follow. We must tell our Remote Object that we are terminating. In the one-to-one relationship, the Remote Object will also be destroyed. If the relationship is one-to-many, then the Shadow Object will not be destroyed if there are still more Remote Objects associated with it. Then the Remote Object must also call the shadow_obj_term_point. This will clean up the Remote Object.

5.3 Persistent Objects

5.3.1 Introduction to Persistent Objects

Often in large systems it is desirable to imbue objects with the quality of *persistence* for purposes of fault tolerance or system-related aspects such as transaction and database. This persistence can have many forms, but the common thread among them is that the data for the objects with the property of persistence should be safe from deletion upon the object's (or system's) demise. Acquiring persistence, should not in any way alter an object's original behavior. That is, persistence should be transparent to the objects. Persistence by itself makes very little sense. Persistence has to co-exist with objects. The way to acquire persistence is by the objects dynamically. The property of persistence makes it an ideal candidate for dynamic composition. Because of this, persistence can be implemented as a shadow object. Attaching a persistent shadow object to an object, transforms the object into a persistent object. Once the object has acquired the persistence behavior, it will stay persistent until the object is destroyed or detached from the persistent property. Notice that adding and altering the persistent behavior of objects does not affect objects themselves and the objects are completely unaware of this behavior. One of the biggest benefits of implementing persistence qualities with shadow objects lies in their dynamic nature. Traditionally, the policy of persistent behaviors has been incorporated as a part of the language system. A program has very little control over a persistence policy. Treating persistence as a secondary behavior will separate the concept of persistence from the system, as well as the objects. Systems can then dynamically configure the objects with a different policy. For example, not every object in a system requires the same degree of persistence (degree of persistence implying the rate of data flushing). It is wasteful to assign a high degree of persistence to every object, simply because of the requirements of a few objects in the system. A policy function can dynamically determine the persistence requirement of objects and designate a persistent policy to each object. This policy function can also periodically reevaluate the persistence requirement of each object and adjust their need accordingly by attaching a different persistent object.

5.3.2 The Saving & Restoring of Persistent Objects

The idea of persistence in MeldC, is to export the object's state to the external environment, in this case to disk. To implement persistence, we use Shadow Objects to do so. As with Remote Objects, we will also use PutObj and GetObj as a protocol to hide the details involved at runtime. A PutObj will flush an object to disk and save the state of the object. Once a PutObj has been done on an object, the object will be marked as persistent. From this point on, the object's state will be flushed to disk periodically. Note, that since the Shadow Object intercepts the message, the flush rate (the rate at which the object is flushed to disk) can be controlled by the Shadow Object. The GetObj will simply recover Some saved state of a Persistent Object from disk where it was saved.

Let us outline the basic steps followed for handling persistency. If a Shadow Object is attached to a Persistent Object, then the Shadow Object will do the following. It will first intercept any messages for it's destination object. The Shadow Object will then flush the Persistent Object to a disk to save the object's state, keeping the object persistent. Once the object has been flushed to disk, the Shadow Object will give the intercepted message to the destination object. The destination object will then execute the message. Once it is done executing, the destination object will then return some value to the Shadow Object. The Shadow Object will then forward the message to the object that originally sent the message.

It should be noted that the current implementation of Persistent Objects in MeldC does not handle the case of crash recovery. A programmer is free though to enhance this implementation and add this feature.

Chapter 6

Introduction to the Runtime

People have recognized the complexity of programming in large-scale distributed environments. Object-oriented programming is a key solution to this problem due to its resembling to the real-world distributed objects and data encapsulation. Many object-oriented languages/systems have been designed and built to reduce the complexity of programming distributed applications by providing high-level abstract views of the real world. In order to support these high-level abstractions, most of these languages hard code the complex details of distribution into their language internals and provide fixed choices of static mechanisms for distribution and communication. This approach usually makes extensions and the alteration to runtime systems difficult and, maybe, impossible.

The MELDC project is an attempt to provide a programming language that supports clear separation of the distribution and communication policies from the language internals. Runtime policies should be able to be designed or customized by MELDC programmers rather than only by the language designers. For example, systems programmers may desire to provide to application programmers the view of transparently accessing distributed resources with an efficient and reliable communication protocol. This can be simply implemented in MELDC without changing any MELDC internals.

MELDC provides a unique programming model, the model of *secondary behavior*. We believe that secondary behavior is a useful language construct to reduce the complexity for programming large-scale distributed object systems. Secondary behaviors encompass dynamically extended functionalities, which are, in most cases, independent of behaviors described by the primary definition of classes (referred to as primary behaviors). Secondary behaviors are composable and can be added or removed at runtime. Programming with secondary behaviors provides a mechanism of separating runtime policies from behaviors encapsulated by classes. MELDC programmers can design a system that intelligently alters its behavior while achieving reasonable efficiency.

The proper way to look at the MELDC runtime is as collection of objects. Together, these collective objects provide a simplistic view of the world, such as the concepts of transparent remote access and persistent objects. The MELDC runtime encapsulates these concepts and represents them as objects. In order to access the facilities in the MELDC runtime, the MELDC MetaClass provides two generic interfaces, GetObj and PutObj, both of which are discussed in Section 4.1.27 of the MetaClass chapter. The idea of GetObj is to obtain an object which is the representation of the external environment. On the other hand, the idea of PutObj is expose itself to the external environment, and allow others to access this object. Sometime PutObj will cause some side affects to the export object, but such side affects are completely transparent the original object. To high level programmers, the access to the external world can be requested through the interface of GetObj and PutObj. The resulting object from such a request can be treated as a local object. External information can be accessed as easily as the local information. In the mind of the high level MELDC programmer, there is a little or no distinguish between the local object and the object representing the external resources. The program can first implement the program locally and then substitute the local object creation by GetObj or PutObj.

The concept of remoteness does not exist in the MELDC language. In order to access a remote object, one has to rely on the mechanism provide by the MELDC runtime. That is, using GetObj to inform the MELDC runtime the desire to access the remote information. The MELDC runtime will identify the remote object and return the "local" representation of the remote object back to the programmer. The programmer can now access the remote information through this local object. The concept of remoteness is completely transparent to the programmer.

For a picture of how some of the runtime features interact with the MELDC features, refer to Figure 6.1.

It is very important to understand that the MELDC runtime is not part of the MELDC language. Without it, one could still write a program in MELDC, but the programmer would require much thorough understanding of the MELDC language. Also, most of the MELDC runtime mechanisms are implemented through the use of Shadow Objects. If you are interested

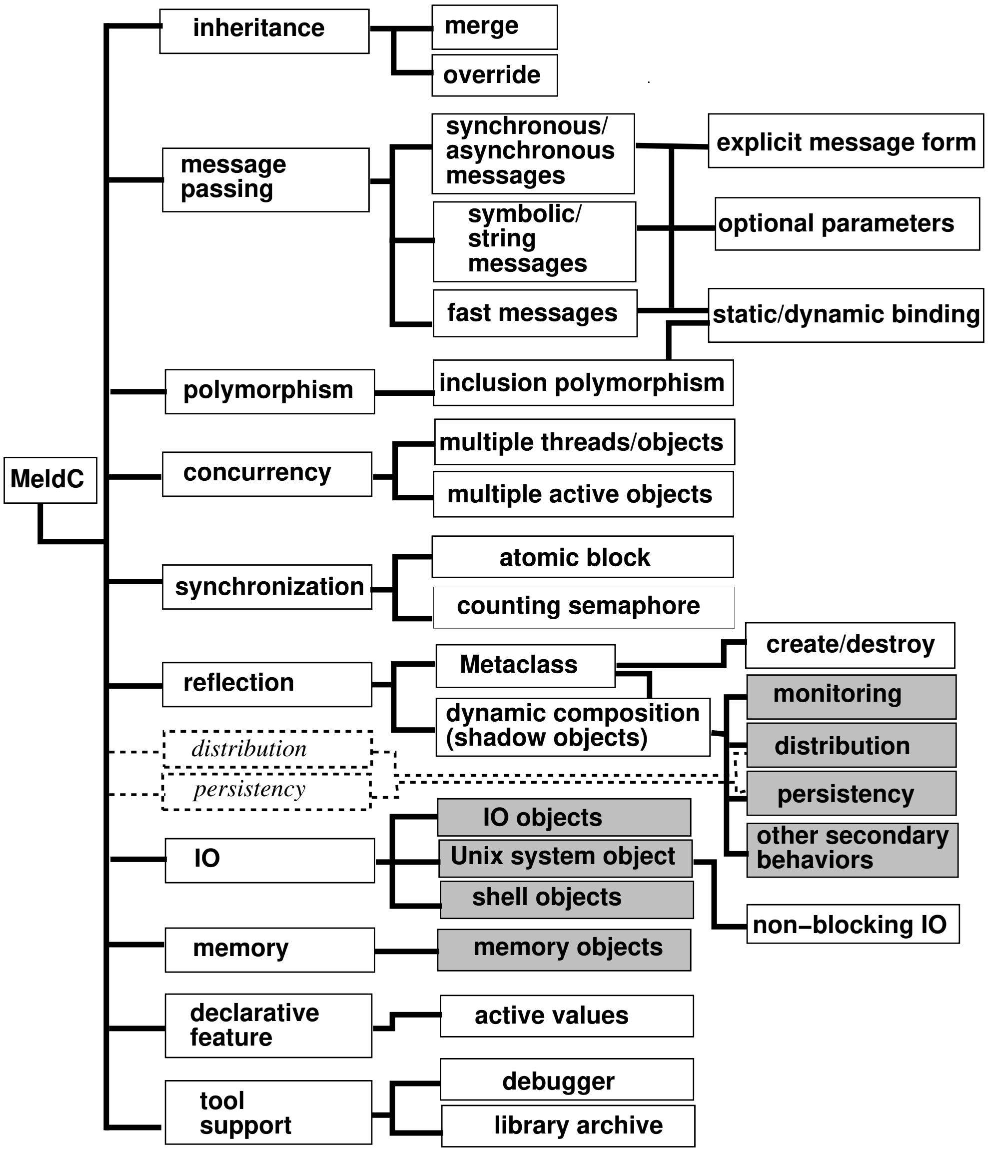

Figure 6.1: Important Runtime Features

in finding out about the runtime implementation using Shadow Objects, please refer to Chapter 7, for the Persistent Objects implementation and to Chapter 8, for the Remote Objects implementation.

Chapter 7

Runtime Component: Persistent Objects

7.1 Introduction to per_obj.m

We now will give a detailed description of the code used to implement the Persistent Object concept in MELDC. The code is contained in the per_obj.m file. This file contains all of the code needed for Persistent Objects. We will discuss each of the functions separatily, in the following sections, by dicussing the function and giving the respective code used to implement the function. The first four functions are trival, since they contain code that is used to manipulate the database. These functions go from Section 7.3 to Section 7.6. The following sections after these first four actually implement Persistency and are therefore the interesting ones. These functions go from Section 7.7 to Section 7.17.

7.2 Headers to per_obj.m

```
#include <stdio.h>
#include <fcntl.h>
#include <ndbm.h>
#include "meldc_user.h"
#include "protocol_obj.h"

extern int linearize_obj();
extern int unlinearize_obj();
```

```
extern int gethostname();
extern int errno();

feature PersistentObj

interface :

    imports Unix[sys_obj]
    imports MemoryAllocation[Memory]
    exports persistent_obj_protocol


implementation:

#define RESTORE 1
#define SAVE    2

PersistentObjProtocol persistent_obj_protocol =
                       PersistentObjProtocol.Create(0);



/*
 * Creating the persistent object database.
 */

class PersistentObjDataBase ::=
DBM *db;
char * name;

/*
 * This is a very simple database implementation, with
 * no concurrency control
 * (Since ndbm do not provide any concurrency control).
 */

methods:
```

7.3 Init

7.3.1 Description of Init

This function will be used to open or create the database. This must initially be done so that objects can be flushed to the database so that their states may be saved.

7.3.2 Code of Init

```
void init(char *  data_base_name)
    {
    /*
     * Attmept to open the database.
     */
    <
    if (!(db = dbm_open(data_base_name,O_RDWR|O_CREAT,0755)))
        {
        printf("Meldc Error PersistentObjDataBase init : Unable to ");
        printf("Open/Create database %d\n",errno);
        }
    name = data_base_name;
    >
    }
```

7.4 Write_to_database

7.4.1 Description of Write_to_database

This function will write the given information to the database for storage. This will occur only when the object is to be flushed to disk, PutObj.

7.4.2 Code of Write_to_database

```
void write_to_database(char * key, char * buffer)
    {
    <
    datum db_key;
    datum result;

    db_key.dptr = key;
    db_key.dsize = strlen(key);
```

```
    result.dptr = buffer;
    result.dsize = strlen(buffer);


    dbm_store(db,db_key,result,DBM_REPLACE);

    if (dbm_error(db))
        {
        printf("MeldC Error : PersistentObjDataBase
                write failed. %d\n",errno);
        }
    >
    }
```

7.5 Read_from_database

7.5.1 Description of Read_from_database

This function will find the given key value in the database and read the information from the database. The information will be the object's state that was last flushed to disk. The function will be called when a GetObj call is done.

7.5.2 Code of Read_from_database

```
char *  read_from_database(char * key)
    {
    datum db_key;
    datum result;

    <
    db_key.dptr = key;
    db_key.dsize = strlen(key);
    result = dbm_fetch(db,db_key);

    if ( (!dbm_error(db)) && (result.dsize))
        {
        result.dptr[result.dsize] = 0;
        return(result.dptr);
        }
```

```
    else
        return((char *) NULL);
    >
    }
```

7.6 Delete_from_database

7.6.1 Description of Delete_from_database

This function will find the given key value in the database and delete the entry from the database for us. This would only be done if we a sure that we will not need the object any longer. Once this is done, we will no longer be able to do a GetObj for that object.

7.6.2 Code of Delete_from_database

```
void delete_from_database(char * key)
    {
    datum db_key;
    datum result;
    int error;

    <
    db_key.dptr = key;
    db_key.dsize = strlen(key);
    dbm_delete(db,db_key);
    >

    }

end class PersistentObjDataBase
```

7.7 Obj_init

7.7.1 Description of Obj_init

This function will be used to create a persistent shadow object and then attach this persistent shadow object to the destination object. This will then cause the destination object to become persistent. It will first find the object name. Once we have the object name, we will be able to create the shadow object, including all of the inheritance hierarchy. Then we will be able to attach the shadow object to the destination. All of this will take place using the CreateAndAttachObj function, see Section 4.1.11. The parameters "persistent_obj_entry_point", "persistent_obj_exit_point", "persistent_obj_init_attach", "persistent_obj_init_detach" are similar to the shadow object parameters "shadow_object_entry_point", "shadow_object_init_point", "shadow_object_term_point". The "persistent_obj_entry_point" is used to handle an incoming message for a given shadow object that is persistent, see Section 7.15 for more. The "persistent_obj_exit_point" is used when the persistent shadow object is done executing, see Section 7.16 for more. The "persistent_obj_init_attach" is used when the persistent shadow object is first attached to the destination object, see Section 7.11 for more. And finally, the "persistent_obj_init_detach" is used when the persistent shadow object is to be detached from it's destination object, see Section 7.12 for more. The parameter RESTORE passed to the CreateAndAttachObj function means that we want to get the old flushed state from disk of the given destination object.

7.7.2 Code of Obj_init

```
class PersistentObjProtocol ::=
PersistentShadowObj per_shadow_obj;

methods :

void init(int rate)
    {
    <
    per_shadow_obj = PersistentShadowObj.Create(rate);
    >
    }
/*
 * Call by GetObj.
 */

object obj_init(MetaClass class_ptr, char * obj_name)
    {
    object obj;
    object temp_obj;

    if (class_ptr == (MetaClass)NULL)
        return((object) NULL);
```

```
/*
 * Create the object, and do not call the init
 * function of that object.
 */
if(temp_obj = class_ptr.find_obj_from_name(obj_name))
    {
    return(temp_obj);
    }


/*
 * creating the object.
 * including all its inheritance hierarchy.
 * We will create all parent objects recursively.
 */

/*
 * We are flush the data, when we enter and
 * exit the function.
 */

obj = class_ptr.CreateAndAttachObj(obj_name,
               per_shadow_obj,
               "persistent_obj_entry_point",
               "persistent_obj_exit_point",
               "persistent_obj_init_attach",
               "persistent_obj_init_detach",
               RESTORE);
return(obj);
}
```

7.8 Register_obj

7.8.1 Description of Register_obj

This function will take the given object and attach a persistent shadow object to it. This will result in making the given object persistent. The function uses the AttachObject function to attach the persistent shadow object to the destination object, see Section 4.1.12 for more. Note that we also pass the "persistent_obj_entry_point", "persistent_obj_exit_point", "persistent_obj_init_attach" and "persistent_obj_init _detach" methods as in the previous section, the obj_init function. The parameter SAVE passed to the AttachObject function means that we want to flush the object to disk.

7.8.2 Code of Register_obj

```
void register_obj(MetaClass class_ptr, object obj,
                  char * name)
    {
     class_ptr.set_obj_info(obj,
                            (enum obj_field)OBJ_NAME,
                                name);

    /*
     * Attach the the persistent shadow object to the
     * regular object.
     *
     * Attach method tak four methods, they are the
     * entry and exit point of the shadow object, and
     * the initialization routine for Attach and
     * Detach.
     */
    (void)class_ptr.AttachObject(obj,
        per_shadow_obj,
        "persistent_obj_entry_point",
        "persistent_obj_exit_point",
        "persistent_obj_init_attach",
        "persistent_obj_init_detach",
        SAVE);
    }
```

7.9 Change_rate

7.9.1 Description of Change_rate

This function simply changes the flush rate to be used. The flush rate is used to determine how often an object should be saved to disk. The higher the flush rate, the more often the object will be flushed to disk. If the flush rate is equal to zero, it then signifies that we are to only flush the object when the persistent object terminates.

7.9.2 Code of Change_rate

```
int change_rate(int rate)
    {
    per_shadow_obj.change_rate(rate);
    }

end class PersistentObjProtocol
```

7.10 Miscellanous Functions

7.10.1 Description of Miscellanous Functions

The init function will set the flush rate and then either open or create the database. The change_rate function will make sure that the flush rate is never greater than the MAX_FLUSH_RATE.

7.10.2 Code of Miscellanous Functions

```
class PersistentShadowObj merges MemoryAllocation[Memory] ::=
int flush_rats = 0;
int virtual_time = MAX_FLUSH_RATE;
PersistentObjDataBase data_base;

methods :

void init(int rate)
    {
    char name[MAX_HOSTNAME];
    char * filename;

    <

    /*
     * If the flush rate == MAX_FLUSH_RATE, we will
     * flush the peristetnt object to the disk,
     * every time it received a message.
     * Otherwise, if will wait until it receive
     * (MAX_FLUSH_RATE-rate) number of messages
     * before the object get flush to the disk.
     */
    flush_rate = rate;
```

```
    if (flush_rate > MAX_FLUSH_RATE)
        flush_rate = MAX_FLUSH_RATE;
    /*
     * open or create the database.
     */

    gethostname(name,MAX_HOSTNAME);
    filename = $self.malloc(MAX_HOSTNAME);
    sprintf(filename,"MeldC.Per.%s",name);
    data_base =
           PersistentObjDataBase.Create(filename);
    >
    }

void change_rate(int rate)
    {
    flush_rate = rate;
    if (flush_rate > MAX_FLUSH_RATE)
        flush_rate = MAX_FLUSH_RATE;
    }
```

7.11 Persistent_obj_init_attach

7.11.1 Description of Persistent_obj_init_attach

This method is used when a persistent shadow object is first attached to an object. Once the persistent shadow has been attached, this method will be executed. The function will simply check the status passed in. If the status is SAVE, we will flush the object to disk using the flush_data function, see Section 7.17. If the status passed in is RESTORE, then we will get the old state of the destination object, which has already been flushed to disk, using the restore_presistent_obj function, see Section 7.14 for more.

7.11.2 Code of Persistent_obj_init_attach

```
int persistent_obj_init_attach(object obj, int status)
    {
    switch (status)
        {
        case SAVE :
            {
```

```
            $self.flush_data(obj,NULL);
            return(TRUE);
            }
        case RESTORE :
            {
            return($self.restore_persistent_obj(obj,NULL));
            }
        }
    }
```

7.12 Persistent_obj_init_detach

7.12.1 Description of Persistent_obj_init_detach

This method is called whenever we want to detach the persistent shadow object from it's parent object. The method will simply call the destroy_persistent_obj, see Section 7.13 for more.

7.12.2 Code of Persistent_obj_init_detach

```
int persistent_obj_init_detach(object obj)
    {
    $self.destroy_persistent_obj(obj,NULL);
    return;
    }
```

7.13 Destroy_persistent_obj

7.13.1 Description of Destroy_persistent_obj

This functions main goal is to delete the entry from the database. When this is called for an object, it can only mean that the object's state no longer needs to be saved on disk. In otherwords, this object is done being persistent. To do this, the function first finds the class of the object. Then the function will read the object's information from the database, using the read_from_database function. Once we have read this information from the database, we will delete the entry from the database using the delete_from_database function. Then we can unlinearize the object information using the unlinearize_obj function, see Section 2.9.10 for more. We must then check to see if there are any objects that are inherited from this object. If there is, we must destroy their persistence as well. After we have destroyed the persistence of all objects that are inherited from the parent object, we are then done.

7.13.2 Code of Destroy_persistent_obj

```
/*
 * Delete the object from the database.
 * It is usually called by Detach.
 */

int destroy_persistent_obj(object obj, char * parent)
    {
    char * buf;
    struct _para * para;
    struct object_as_param * obj_param = NULL;
    object temp_obj;
    object class_ptr;
    char * obj_name;
    char new_obj_name[_NAME_SIZE];

    /*
     * find the class of the object.
     */
    class_ptr
        = (object)MetaClass.obtain_obj_info(obj,
                   (enum obj_field)OBJ_CLASS_PTR);

    if (parent != NULL)
        {
        obj_name
            = (char *)MetaClass.obtain_obj_info(obj,
                          (enum obj_field)OBJ_NAME);
        sprintf(new_obj_name,"%s$%s",obj_name,parent);
        obj_name = new_obj_name;
        }
    else
        {
        obj_name
            = (char *)MetaClass.obtain_obj_info(obj,
                          (enum obj_field)OBJ_NAME);
        }
```

```
buf = data_base.read_from_database(obj_name);

/*
 * Once we extracted the persistent information,
 * we will delete the entry from the database.
 */

if (buf)
    data_base.delete_from_database(obj_name);

if (buf == NULL)
    return FALSE;

para = (struct _para *)
       (class_ptr.class_instance_variable(
                        C_INSTANCE_VAR_TABLE));

/*
 * identify the instance variable that are objects.
 */

unlinearize_obj(obj,para,buf,&obj_param,obj);

while(obj_param != NULL)
    {
    object temp_obj;
    /*
     * There are objects pass as instance variables,
     * look through the database & try to fill the value.
     */

    if (strcmp(obj_param->object_name,
               "__$__NULL_OBJ") == 0)
        {
        /*
         * The NULL object.
         */
        obj_param = obj_param->next;
        continue;
        }
    else
        {
```

```
struct _para * para ;
char * pointer;
object class_obj = (object) NULL;

/*
 *  we need to find the class pointer.
 */

para =
     ((struct _para *)(obj_param->pointer));


if ( (para != NULL) &&
     (strncmp(para->decl_type_ptr->
               postfix_type,"class",5) ==0))
    {
    pointer = strchr(para->decl_type_ptr->
                          postfix_type,' ');

    pointer ++;

    class_obj =
MetaClass.find_class_obj_from_name(pointer);
    }

if (class_obj == (object) NULL)
    {
    /*
     * Set the object to NULL.
     */
    obj_param = obj_param->next;
    continue;
    }
else if (strncmp(obj_param->object_name,
                            "__$__",5) == 0)
    {
    /*
     * Part of the inheritance hierarchy.
     */

    if ($self.destroy_persistent_obj(
       (object)(*(char **) (((int)obj +
```

```
                    (int)((struct _para *)
                     (obj_param->pointer))-> offset))),
                    obj_name)
                    == FALSE)
                   {
                   return(FALSE);
                   }
               }
           else
               {
               obj_param = obj_param->next;
               continue;
               }
           }
       obj_param = obj_param->next;
       }
   return(TRUE);
   }
```

7.14 Restore_persistent_obj

7.14.1 Description of Restore_persistent_obj

This function's main job is to load the persistent object from the database. This is done when a GetObj is executed. To do this the function first finds the class of the object. Then the function will call the read_from_database function to get the object's entry. Now that we have the object's information, we can delete the entry from the database with a delete_from_database function. We must then unlinearize the object's information using the unlinearize_obj function, see Section 2.9.10 for more. If other objects are inherited from this object, we must restore all of those object's as well. This is done using a recursive call to this function, restore_persistent_obj. After we have restored the object and the object's hierarchy, we are then done.

7.14.2 Code of Restore_persistent_obj

```
int restore_persistent_obj(object obj,char * parent)
    {
    char * buf;
    struct _para * para;
```

```
struct object_as_param * obj_param = NULL;
object temp_obj;
object class_ptr;
char * obj_name;
char new_obj_name[_NAME_SIZE];

/*
 * find the class of the object.
 */
class_ptr
    = (object)MetaClass.obtain_obj_info(obj,
                 (enum obj_field)OBJ_CLASS_PTR);

if (parent != NULL)
    {
    obj_name
        = (char *)MetaClass.obtain_obj_info(obj,
                      (enum obj_field)OBJ_NAME);
    sprintf(new_obj_name,"%s$%s",obj_name,parent);
    obj_name = new_obj_name;
    }
else
    {
    obj_name
        = (char *)MetaClass.obtain_obj_info(obj,
                        (enum obj_field)OBJ_NAME);
    parent = obj_name;
    }

buf = data_base.read_from_database(obj_name);

/*
 * Once we extracted the persistent information,
 * we will delete the entry from the database.
 */

if (buf)
    data_base.delete_from_database(obj_name);

if (buf == NULL)
    return FALSE;
```

```
para = (struct _para *)
             (class_ptr.class_instance_variable(
                           C_INSTANCE_VAR_TABLE));

/*
 * identify the instance variable that are objects.
 */

unlinearize_obj(obj,para,buf,&obj_param,obj);

while(obj_param != NULL)
    {
    object temp_obj;
    /*
     * There are objects pass as instance variables,
     * look through the database & try to fill the
     * value.
     */

    if (strcmp(obj_param->object_name,
               "__$__NULL_OBJ") == 0)
        {
        /*
         * The NULL object.
         */
        *(char *)(((int) obj +
                   (int)((struct _para *)
                 (obj_param->pointer))-> offset)) = 0;
        }
    else
        {
        struct _para * para ;
        char * pointer;
        object class_obj = (object) NULL;

        /*
         *  we need to find the class pointer.
         */

        para=((struct _para *)(obj_param->pointer));
```

```
    if ( (para != NULL) &&
         (strncmp(para->decl_type_ptr->
                   postfix_type,"class",5) ==0))
        {
        pointer = strchr(para->decl_type_ptr->
                                     postfix_type,' ');

        pointer ++;

        class_obj =
    MetaClass.find_class_obj_from_name(pointer);
        }

    if (class_obj == (object) NULL)
        {
        /*
         * Set the object to NULL.
         */
        *(char *)(((int) obj +
             (int)((struct _para *)
           (obj_param->pointer))-> offset)) = 0;
        }
    else if (strncmp(obj_param->object_name,
                    "__$__",5) == 0)
        {
        /*
         * Part of the inheritance hierarchy.
         */

        if ($self.restore_persistent_obj(
           (object)(*(char **) (((int)obj +
             (int)((struct _para *)
             (obj_param->pointer))-> offset))),
             parent)
             == FALSE)
            {
            return(FALSE);
            }
        }
    else if(temp_obj =
(object)class_ptr.find_obj_from_name(obj_name))
        {
```

```
              *(char **) (((int) obj +
                  (int)((struct _para *)
                     (obj_param->pointer))-> offset))
                      = (char *)temp_obj;
              }
          else
              {
              /*
               * One of the instance variable itself
               * is a persistent object.
               */
              *(char **) (((int) obj +
                 (int)((struct _para *)(obj_param->
                                pointer))-> offset))
           = class_obj.GetObj(obj_param->object_name,
                           persistent_obj_protocol);
              }
          }
      obj_param = obj_param->next;
      }
  return(TRUE);
  }
```

7.15 Persistent_obj_entry_point

7.15.1 Description of Persistent_obj_entry_point

This function will use the shadow object concept of intercepting messages to decide when and when not to flush a persistent object to disk. That is, when this function is called, we will check the virtual_time variable to see if it is time to save the object to disk. If the flush_rate variable is 0, then we will call the flush_data function, which will flush the object to disk and therefore save the object's state. Otherwise, we will compare the flush_rate to the variable_time variable. If they are equal, then we will also call the flush_data function and also flush the object to disk. If the two variables are not equal, then we will decrement the variable_time variable by one.

7.15.2 Code of Persistent_obj_entry_point

```
/*
 * After the shadow object intercept the message, it will forward
```

```
 * the message to this routine.
 */

struct _frame * persistent_obj_entry_point(struct _frame * fp)
    {
    if ((flush_rate == 0) || (strncmp("$term",
                                       fp->$.mp->name,6) == 0))
        {
        object obj;

        obj = (object) (fp->$.obj);

        $self.flush_data(obj,NULL);

        return(fp);
        }
    else
        {
        if (virtual_time == flush_rate)
            {
            object obj;

            obj = (object) (fp->$.obj);

            $self.flush_data(obj,NULL);

            virtual_time = MAX_FLUSH_RATE;

            return(fp);
            }
        else
            {
            virtual_time --;
            return(fp);
            }
        }
    }
```

7.16 Persistent_obj_exit_point

7.16.1 Description of Persistent_obj_exit_point

This function will check to see if it is time to flush the object to disk. This check will be done when the persistent shadow object is done executing. When it is done, it will automatically call this method to be executed. If the flush rate equals zero or the virtual time equals the flush rate, then we will flush the object to disk using the flush_data function, see Section 7.17 for more. If neither of these cases exist, then we will just decrement the virtual time variable by one.

7.16.2 Code of Persistent_obj_exit_point

```
void persistent_obj_exit_point(object obj, optional)
    {
    if(flush_rate == 0)
        {
        $self.flush_data(obj,NULL);
        return;
        }
    else if (virtual_time == flush_rate)
        {
        $self.flush_data(obj,NULL);

        virtual_time = MAX_FLUSH_RATE;
        return;
        }
    else
        {
        virtual_time --;
        return;
        }
    }
```

7.17 Flush_data

7.17.1 Description of Flush_data

This functions main goal is to flush the object to the database for us. In otherwords, this function will save the object's state on disk so that it can be restored at a later time. The data stored on disk is saved in a certain format.

The format used is for the object name to be used as the key for finding the object's information at some later time if the object is not inherited from any other object. If the object is inherited from another object, we will use "obj_name$parent_name" as the key. We will not linearize another object, but put the name of the other object in the linearized buffer. Any object names that begin with "__$__" is an inherited object. For example:

JAMES O{ __$__1} - James represents the objects name & the parent object of JAMES is __$__1.

jessie O{ __$__NULL_OBJ} - Here jessie has no parent since the parent of jessie is represented as a NULL_OBJ.

The function will first get the object's class and then the object's name. Then we must linearize the object's information by calling the linearize_obj function, see Section 2.9.3 for more. Once we have linearized the data, we can then flush this information to disk using the object's name as the key. If there are objects that are inherited from this one, we must also flush them to disk by calling the flush_data function recursively. Once the object's hierarchy has been saved, we are then done saving the object's state.

7.17.2 Code of Flush_data

```
void flush_data(object obj, char * parent)
    {
    int size;
    struct _para * para;
    char *bufp;
    struct object_as_param * obj_param = NULL;
    MetaClass class_ptr;
    char * obj_name;
    char new_obj_name[_NAME_SIZE];


    class_ptr = (MetaClass)(MetaClass.obtain_obj_info(obj,
                              (enum obj_field)OBJ_CLASS_PTR));
    obj_name = (char *)(MetaClass.obtain_obj_info(obj,
                                   (enum obj_field)OBJ_NAME));
    /*
     * Linearize and flush to the disk.
```

```
     * Flush to the disk.
     */

    para = (struct _para *)
           (class_ptr.class_instance_variable(
                                   C_INSTANCE_VAR_TABLE));

    size = linearize_obj(obj,para,&bufp,&obj_param);

    if (parent == NULL)
        {
        parent = obj_name;
        data_base.write_to_database(obj_name,bufp);
        }
    else
        {
        sprintf(new_obj_name,"%s$%s",obj_name,parent);
        data_base.write_to_database(new_obj_name,bufp);
        }


    while(obj_param != NULL)
        {
        if (strncmp(obj_param->object_name,"__$__",5)==0)
            {
            if (strcmp(obj_param->object_name,
                       "__$__NULL_OBJ") != 0)
                $self.flush_data((object)
                                (obj_param->obj),parent);
            }
        obj_param = obj_param->next;
        }
    }

end class PersistentShadowObj

end feature PersistentObj
```

Chapter 8

Runtime Component: Remote Objects

This last chapter of the manual is an outline of how Remote Objects has been implementation for the Runtime of MeldC. Let us begin by giving a detailed discription of the necessary steps needed to access an object from an external source. The local side will first do a GetObj call, which is a high level protocol used at runtime to get an object. The local side will contact the remote side and inform the remote side that this object is wanted. The remote side must have by this point, have made a call to PutObj. PutObj is another high level protocol used to allow access to the desired remote object at runtime. If the remote source has done a PutObj for the desired object, then the remote side will inform the local side that access has been granted to the object. Now here we are being very general and vague. There is much more to it since the shadow object concept is used here. When an object copy is sent to the local side, a shadow object stub is created also to allow proper communication between the objects on both the remote and local side. This shadow will forward the messages it receives to the real object on the remote side.

Let us now outline how the GetObj call works in detail so that you can get a general sense of how all of the sections below fit together. A GetObj, as stated above, will get the remote object's network address. It does this by using the obj_init function, see Section 8.3.4. We will in this process, verify that the object exists, see Section 8.8. Then we will get the object's id and establish a connection.

Now let's look at PutObj in detail. This function, also as stated before, will allow a remote source access to an object. To do so, we must call the register_obj function, see Section 8.9.4. This will put the name into the export table.

Now that we know where the object is and we have gained access to an object we can import this object. That is, set up a way of using the object. This means we must create a shadow object to handle the communication between our object and the remote object. This is done by using the CreateAndAttachObj or the Create function and the Attach function separately. See Section 4.1.9 for the Create function and see Section 4.1.11 for the CreateAndAttachObj function. We must also use the initialization functions of the generic_remote_obj.m section to set up the shadow objects, see Section 8.5, This will allow us to catch and process any incoming messages from the remote source. It will also allow us to send messages. We can also export an object to a remote source to inform our local side that we have granted access to one of our objects.

The next thing we must discuss in general, is an overview of sending and receiving messages. To send a message to some remote object, we must have established a connection to the remote source with a shadow object. We must have also been granted access to the remote object. If we have been granted the access, then we know where the object is and we can send a message to the remote source. There are a series of steps to follow. These include encoding the message, put message into a message format, see Section 8.1. We must also linearize the message, see Section 2.9.1, and save our outgoing message. We will then send the message with the send_message_request, see Section 8.6.22. This function will use the functions of the remote_udp section to do so, see Section 8.2. This message will then be received by the remote source will be processed. After processing this message, we will receive a response message. We will use the functions of the remote_udp section to also do so. When this occurs, we must perform what is known as a Kernel trap. This will capture the message and perform a series of steps that allows the shadow objects, that are attached to the object, to process this message. This is done in the shadow_object_entry_point function, see Section 8.5.9. It should be noted, that there are numerous low level functions that are performed by functions contained in the generic_remote_interface.m section, see Section 8.6. As we come across these functions while discussing other functions we will make references to them.

To see how all of the main pieces of this chapter fit, or in other words, a general hierarchy, see Figure 8.1. The lower files of the figure are only accessable through the higher ones. This hides the actual implementation from the user so that the user does not have to worry about the specific steps involved. The programmer does have the option though of replacing the lower level protocol.

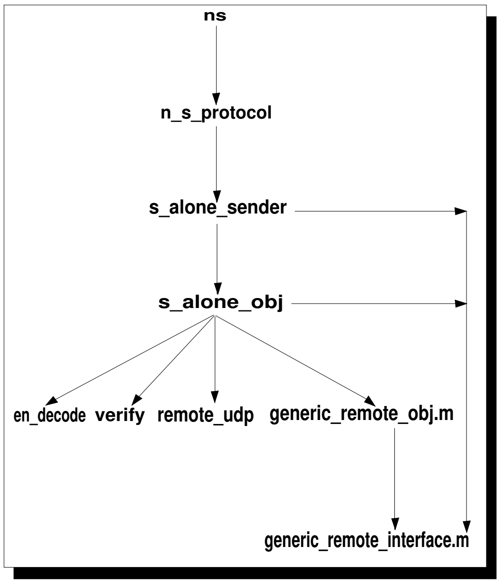

Figure 8.1: Runtime file hierarchy for remote objects

8.1 En_decode.m

8.1.1 Description of En_decode.m

The file en_decode.m will preform the simple, but highly needed task of encoding a message and decoding a message into a certain format. The function will first determine the size of the message buffer that will be needed. Then the function will allocate the memory for the message buffer. It will then procede to pack the message buffer into a predefined form. The form is packed in the following fashion:

- The first six bytes indicate the length of the buffer
- The variables are then put in the following order:
  - destination of object name
  - source object name
  - type of message in input buffer
  - return code from the remote host
  - the input buffer that contains the user data needed to be sent to the remote host

- The first six bytes of each variable field represents the length of the data
- The data will be located right after the length

8.1.2 Headers of En_decode.m

```
feature EnDecode

interface:

    imports Unix[sys_obj]

    imports MemoryAllocation[Memory]

    exports Encode_Decode_Buffer

implementation:

class Encode_Decode_Buffer merges MemoryAllocation[Memory]::=

methods :
```

8.1.3 Encode_message

Description of Encode_message

The function encode_message will do the actual work. It first finds the size of the buffer. The size will equal the size of all the strings plus the bytes that indicate the length of each variable and the overall length. It will then use malloc to allocate the memory to put the message into. It then packs it into the above format.

Code of Encode_message

```
int encode_message(char ** message, char * input_buf,
                   int length, char * odest, char * oname,
                   int type_of_message,
                   enum response_message return_status,
                   int remote_obj_id)
    {
    /*
     * MAXINT is only 9 digit, choose a buffer with 20
```

```
     * characters, should be enough.
     */

    char type_of_mess_str[20];
    char return_status_str[20];
    char remote_obj_id_str[20];
    char length_str[7][MAX_DIGIT+1];
    int size_of_message_buffer;
    /*
     * Convert the number to string.
     */

#ifdef TRACE
    printf("Enter encode_message \n");
#endif

    $self.convert_num_str(type_of_mess_str,
  type_of_message,0);

    $self.convert_num_str(return_status_str,
 (int)return_status,0);

    $self.convert_num_str(remote_obj_id_str,
  (int)remote_obj_id,0);

    /*
     * First compute the size of the buffer.
     * size_of_message_buffer = size of all the string
     * + bytes that indicates
     * the length of individual
     *                         and overall buffer size.
     */
    size_of_message_buffer = strlen(type_of_mess_str) +
                             strlen(return_status_str) +
                             strlen(remote_obj_id_str) +
                             strlen(odest) +
                             strlen(oname) +
                             length + 7 * MAX_DIGIT ;

    *message= $self.malloc(size_of_message_buffer + 10);

    $self.convert_num_str(length_str[0],
```

```
                          size_of_message_buffer,
                          MAX_DIGIT);
    $self.convert_num_str(length_str[1],strlen(odest),
                          MAX_DIGIT);
    $self.convert_num_str(length_str[2],strlen(oname),
                          MAX_DIGIT);
    $self.convert_num_str(length_str[3],
                          strlen(type_of_mess_str),
                          MAX_DIGIT);
    $self.convert_num_str(length_str[4],
                          strlen(return_status_str),
                          MAX_DIGIT);
    $self.convert_num_str(length_str[5],
                          strlen(remote_obj_id_str),
                          MAX_DIGIT);
    $self.convert_num_str(length_str[6],length,
                          MAX_DIGIT);

    sprintf(*message,"%s%s%s%s%s%s%s%s%s%s%s%s",
            length_str[0], length_str[1],odest,
            length_str[2],oname, length_str[3],
    type_of_mess_str, length_str[4],
            return_status_str, length_str[5],
            remote_obj_id_str, length_str[6]);

    if (input_buf)
        memcpy(((*message)+size_of_message_buffer -
                         length),input_buf,length);

    return(size_of_message_buffer);
    }
```

8.1.4 Decode_message

Description of Decode_message

The decode_message function is even easier. Knowning the field set up, it is a simple task of just pulling the message apart and placing the pieces into the respective variables.

Code of Decode_message

```
void decode_message(char * buf,
```

```
                    struct client_sendto_req_type
                    *data_packet)
    {
    /*
     * Skip the first MAX_DIGIT bytes.
     */
    char length_str[MAX_DIGIT+1];
    int length;
    char digit[20];

#ifdef TRACE
    printf("Enter decode_message \n");
#endif

    length_str[MAX_DIGIT] = NULL;

    buf += MAX_DIGIT;

    strncpy(length_str,buf,MAX_DIGIT);

    length = atoi(length_str);

    buf += MAX_DIGIT;

    (void)strncpy(data_packet->odest,buf,length);

    *((char *)((int)(data_packet->odest) + length)) = NULL;

    buf += length;

    strncpy(length_str,buf,MAX_DIGIT);

    length = atoi(length_str);

    buf += MAX_DIGIT;

    strncpy(data_packet->oname,buf,length);

    *((char *)((int)data_packet->oname + length)) = NULL;

    buf += length;
```

```
strncpy(length_str,buf,MAX_DIGIT);

length = atoi(length_str);

buf += MAX_DIGIT;

strncpy(digit,buf,length);

*((char *)((int)digit + length)) = NULL;

data_packet->type = atoi(digit);

buf += length;

strncpy(length_str,buf,MAX_DIGIT);

length = atoi(length_str);

buf += MAX_DIGIT;

strncpy(digit,buf,length);

*((char *)((int)digit + length)) = NULL;

data_packet->response =
                 (enum response_message)atoi(digit);

buf += length;

strncpy(length_str,buf,MAX_DIGIT);

length = atoi(length_str);

buf += MAX_DIGIT;

strncpy(digit,buf,length);

*((char *)((int)digit + length)) = NULL;

data_packet->remote_obj_id = (int)atoi(digit);

buf += length;
```

```
    strncpy(length_str,buf,MAX_DIGIT);

    length = atoi(length_str);

    data_packet->msg_size = length;

    data_packet->msg = $self.malloc(length);

    buf += MAX_DIGIT;

    strncpy(data_packet->msg,buf,length);
    }
```

8.1.5 Convert_num_str & free_message

Code of convert_num_str & free_message

```
void convert_num_str(char * buf,int num, int max_width)
    {
    if (max_width == 0)
        {
        sprintf(buf,"%d",num);
        }
    else
        {
        int counter;
        char temp_buf[20];
        int temp_length;

        sprintf(temp_buf,"%d",num);
        temp_length = strlen(temp_buf);

        for (counter = 0; counter < max_width; counter ++)
            *(buf + counter) = '0';

        temp_length = max_width - temp_length;

        strcpy((buf+temp_length),temp_buf);
        *(buf+max_width+1) = NULL;
        }
    }
```

```
void free_message(struct client_sendto_req_type data_packet)
    {
    $self.free(data_packet.msg);
    return;
    }

end class Encode_Decode_Buffer

end feature EnDecode
```

8.2 Remote_udp.m

8.2.1 Description of Remote_udp.m

The remote_udp file has the task of setting up a connection on the network with udp from the local to the remote end. The basic concept for this is to create the socket and bind it to a specific port. The address of the socket is assumed to be a internet address format. This part is done with the function init, to initialize the connection.

It should be noted that this implementation is a very simple one. The verification process is very limited in it's usage. Therefore, it is up to the programmer to substitute a better protocol for the current one if so desired.

8.2.2 Headers of Remote_udp.m & the init function

```
feature NetworkUdp

interface:

    imports Unix[sys_obj]

    imports HashTable[Hash]

    imports MemoryAllocation[Memory]

    exports NetworkProtocol

implementation:
class NetworkProtocol merges MemoryAllocation[Memory] ::=
int sock;
```

```
int port_num;
int disable = 0;
   /* Indicates whether the remote object is available */
char local_hostname[MAX_HOSTNAME];
   /* The local hostname. */
struct sockaddr_in server_addr;

methods:

/*
 * Initialize the protocol class.
 * The socket address format will assume to be the
 * internet address format.
 */

int init()
    {
    struct sockaddr_in sin;
    int sin_len;

    /*
     * Creating the socket, and bind it to a specific
     * port.
     */

    if (sys_obj.gethostname(local_hostname,MAX_HOSTNAME))
        {
        printf("Unable to obtain local hostname. Errno =
               %d\n",sys_obj.errno());
        printf("Disable the use of remote object. \n");

        disable = 1;
        return(0);
        }

#ifdef STREAM
    sock = sys_obj.socket(AF_INET,SOCK_STREAM,0);
#else STREAM
    sock = sys_obj.socket(AF_INET,SOCK_DGRAM,0);
#endif STREAM

    if (sock < 0)
```

```
        {
        printf("Unable to create socket for remote object.
Errno = %d\n", sys_obj.errno());
        printf("Disable the use of remote object. \n");

        disable = 1;
        return(0);
        }

    sin.sin_family = AF_INET;         /* Internet family */
    sin.sin_addr.s_addr = INADDR_ANY; /* Any local address. */
    sin.sin_port = (u_short)htons(0); /* random choose a port */
                                      /* between 1024-5000    */
    if (sys_obj.bind (sock,(struct sockaddr *)&sin,
                      sizeof(sin)))
        {
        printf("Unable to bind the socket for remote
                object. Errno = %d\n", sys_obj.errno());
        printf("Disable the use of remote object. \n");

        disable = 1;
        return(0);
        }
    /*
     * Find out the port number.
     */

    sin_len = sizeof(sin);
    if (sys_obj.getsockname(sock,
                            (struct sockaddr *)&sin,
                            &sin_len))
        {
        printf("Unable to get sockname for remote
                object. Errno = %d\n", sys_obj.errno());
        printf("Disable the use of remote object. \n");

        disable = 1;
        return(0);
        }

    /*
     * Save the port number.
```

```
     */
    port_num = (int)ntohs(sin.sin_port);

    return(0);
    }
```

8.2.3 Reassign_port_num

Description of Reassign_port_num

The user can reassign their own port number using the reassign_port_num. The system picks the port number by default and makes this port number unique and is usually not necessary to change it. Sometimes, when a port number of a server has to be well known, the automatic port number generation is no longer useful. In this case, this function can be used to change the port number to that well known number so that everyone can use it. This change starts with closing the old socket first. The socket will than be bound to the specific port. Get the port number and then save it.

Code of Reassign_port_num

```
/*
 * User want to change the default port number, and
 * reassign it own port number.
 */

int reassign_port_num(int new_port_num)
    {
    struct sockaddr_in sin;
    int sin_len;

    /*
     * First let's close our old socket.
     * When we issue close call to the system object, all
     * the receive that is waiting for this particular fd
     * will be waken up.
     */

    (void)sys_obj.close(sock);

    /*
     * Rereating the socket & bind it to a specific port.
```

```
     */

    if (sys_obj.gethostname(local_hostname,MAX_HOSTNAME))
        {
        printf("Unable to obtain local hostname. Errno =
%d\n", sys_obj.errno());
        printf("Disable the use of remote object. \n");

        disable = 1;
        return(0);
        }

    sock = sys_obj.socket(AF_INET,SOCK_DGRAM,0);

    if (sock < 0)
        {
        printf("Unable to create socket for remote object.
                Errno = %d\n", sys_obj.errno());
        printf("Disable the use of remote object. \n");

        disable = 1;
        return(-1);
        }

    sin.sin_family = AF_INET;        /* Internet family */
    sin.sin_addr.s_addr = INADDR_ANY;/* Any local addr */
    sin.sin_port = (u_short)htons(new_port_num);

    if (sys_obj.bind (sock,(struct sockaddr *)&sin,
                      1sizeof(sin)))
        {
        printf("Unable to bind the socket for remote
                object. Errno = %d\n", sys_obj.errno());
        printf("Disable the use of remote object. \n");

        disable = 1;
        return(-1);
        }
    /*
     * Get the port number.
     */
```

```
    sin_len = sizeof(sin);
    if (sys_obj.getsockname(sock,
    (struct sockaddr *)&sin,
    &sin_len))
        {
        printf("Unable to get sockname for remote object.
                Errno = %d\n", sys_obj.errno());
        printf("Disable the use of remote object. \n");

        disable = 1;
        return(-1);
        }
    /*
     * Save the port number.
     */

    int old_port_num;

    old_port_num = port_num;

    port_num = (int)ntohs(sin.sin_port);

    return(old_port_num);
    }
```

8.2.4 Send_to_remote

Description of Send_to_remote

The actual sending of the message will be sent with the function send_to_remote. It will be sent using a sendto call passing the socket number, the buffer, the size of the buffer and the socket address. Of course error handling is done to make sure that it is sent without error.

Code of Send_to_remote

```
int send_to_remote(char * buffer,int size,struct sockaddr *sin)
    {
    int s ;
    int total_bytes = 0;

#ifdef TRACE
```

```
    printf("Enter send_to_remote \n");
#endif

    while (size > 0)
        {
        s = sys_obj.sendto(sock,buffer,size,0,
     (struct sockaddr *)sin,
                           sizeof(struct sockaddr));
        if (s < 0)
            {
            /*
             * Unable to send to the remote
             *
             * PS ***********
             * We alos need to add the SIGPIPE into the event
     * handling mechanism in meldc, so that when
     * SIGPIPE happen we do not generate the core.
             */
            printf("Send Failed. Errno = [%d]\n",
                   sys_obj.errno());

            /*
             * If we are unable to send to the name server,
     * the program will not be able to continue, so
             * we should exit.
             */

            return(-1);
            }
#ifdef TRACE
         printf("send %u === %d %s \n",
               ((struct sockaddr_in *)sin)->sin_addr,
               size,buffer);
#endif
         buffer += s;
         size -= s;
         }

    return(0);
    }
```

8.2.5 Receive_from_remote

Description of Receive_from_remote

The remote side will then receive the message from the send function on the local side with the receive_from_remote function. It does so with the recvfrom call. The message will then be processed by the remote side. The remote side can send a response back using the same connection.

Code of Receive_from_remote

```
int receive_from_remote(char * buffer, int size,
                        struct sockaddr *server_addr,
                        int *fromlen, int flags)
    {
    int s ;
    int total_bytes = 0;

    if (flags & MSG_PEEK)
        {
        s = sys_obj.recvfrom(sock,buffer,size,flags,
                             (struct sockaddr *)server_addr,
                             fromlen);
        if (s < 0)
            {
            /*
             * return the error back to the parent.
             */

            if (sys_obj.errno() == READ_ABORT)
                {
                sys_obj.set_parent_errno(READ_ABORT);
                return(s);
                }
            printf("MeldC Error: receive_from_remote()\n");
            printf("Receive Failed. Errno = [%d]\n",
    sys_obj.errno());
            }
        return(s);
        }
    while (size > 0)
        {
        s = sys_obj.recvfrom(sock,buffer,size,flags,
```

```
                            (struct sockaddr *)server_addr,
                            fromlen);
        if (s < 0)
            {
            if (sys_obj.errno() == READ_ABORT)
                {
                sys_obj.set_parent_errno(READ_ABORT);
                return(s);
                }
            /*
             * Unable to receive from the name server.
             *
             * PS ***********
             * We alos need to add the SIGPIPE into the
     * event handling mechanism in meldc, so that
     * when SIGPIPE happen we do not generate the
             * core.
             */
             printf("Unable to receive from the remote
     object. Errno = %d\n",
     sys_obj.errno());

            /*
             * If we are unable to send to the name server,
     * the program will not be able to continue, so
             * we should exit.
             */

            return(-1);
            }

#ifdef TRACE
        printf("receive === %d %s \n",size,buffer);
#endif
        size -= s;
        buffer += size;
        total_bytes += s;
        }
    return(total_bytes);
    }
```

8.2.6 Miscellaneous functions

Code of Miscellaneous functions

```
int local_port()
    {
    return(port_num);
    }

char * local_hostname()
    {
    return(local_hostname);
    }

/*
 * is the remote object disable ?
 */
int is_disable()
    {
    return(disable);
    }

end class NetworkProtocol

end feature NetworkUdp
```

8.3 S_alone_obj

8.3.1 Description of S_alone_obj

This next file, s_alone_obj is also a simple interface to the shadow objects. The first function is the create_shadow_obj_protocol. This function will create the shadow object protocol,shadow_obj_protocol, using the metaClass Create function, see Section 4.1.9 for more on this function.

NOTE – Both this section and the s_alone_obj are generic. That is, any system dependent information such as the protocol between shadow objects are hidden away from both files.

8.3.2 Headers of S_alone_obj

```
extern char * construct_object_id();
extern char * construct_thread_id();
```

```
feature RemoteObj

interface:

    imports MemoryAllocation[Memory]

    imports GenericObj[GenericRemoteClass]

    imports ShadowObjInterface[ShadowObjProtocol]

    exports RemoteProtocol

    exports remote_protocol

implementation:

    RemoteProtocol remote_protocol = RemoteProtocol.Create();

class RemoteProtocol merges MemoryAllocation[Memory] ::=
object shadow_obj_protocol;


methods:
int init()
    {
    <
    $vself.create_shadow_obj_protocol();
    >
    }

void create_shadow_obj_protocol()
    {
    shadow_obj_protocol =
               ShadowObjProtocol.Create(GenericRemoteClass);
    }
```

8.3.3 Register_obj

Description of Register_obj

The next function of importance is the register_obj. It's main job is to register the object. That is, to allow the remote process the right to access the requested object by putting the object's name into the export table.

First it will recall the object information with the obtain_obj_info function, located in Section 4.1.24. Next we change the object name to the name that the user assigned, using the set_obj_info function, see Section 4.1.25. Now we can check to see if the object already exists by calling the export function, export_object. This function, located in the generic_remote_interface section, will look the object up in the export table to see if it does exist. See Section 8.6.15 for more information on this function.

Code of Register_obj

```
/* Register the object.                                  */
/* Give the permission to the rmeote process to access */
/* this object.        */
/*                                                       */

int register_obj(MetaClass class_ptr, object obj,
 char * name)
    {
    HObjEntry *o_entry;

    if ((char *)class_ptr !=
        (MetaClass.obtain_obj_info(obj,
 (enum obj_field)OBJ_CLASS_PTR)))
        {
        printf("Fatal error : inconsistent class name of
exporting object");
        return(0);
        }
    /*
     * Change the obj name, to name that user assigned.
     */
    class_ptr.set_obj_info(obj,(enum obj_field)OBJ_NAME,
                           name);

    shadow_obj_protocol.export_object(obj,name);
```

```
return(0);
}
```

8.3.4 Obj_init

Description of Obj_init

The next function to examine is the obj_init function. This function will attach the shadow object to the newly created object and specify the exit and entry point to the shadow objects. This shadow object will then intercept all messages being sent to the object that the shadow is attached to and then will forward the messages to the remote location.

First, we will create the shadow object for the sender, using the metaClass Create function, and specify the protocol to use, the shadow_obj_protocol. We will initialize the dummy remote stub so that it looks like a regular object, except that the method address will point to a predefined function. The predefined function will be the remote object itself. We will create a regular object and attach a shadow object to it by using the CreateAndAttachObj function, discussed in Section 4.1.11. Here we will also pass the shadow_object_entry_point and the shadow_object_exit_point. This will tell the shadow where to forward the message to. We also send the shadow_object_init_point. This is for the initialization of the shadow object. And finally, we pass the shadow_object_term_point. This is used for the cleaning up of the shadow object connection during the "detachment" phase of the shadow objects life. For more on these three functions, see the generic_remote_obj section below, Sectiongenericremoteobj. We will then add this object as an entry into the import hash table by calling the import_object function in the generic_remote_interface section below, Section 8.6.16. After this, the initialization routine of the shadow object will be called.

Code of Obj_init

```
/* Construct the remote shadow object. All the message */
/* send to this object will be reconstructed and       */
/* forwarded to the remote side.        */

object obj_init(MetaClass class_ptr, char * obj_name,
                struct sockaddr *addr)
    {
    struct sockaddr sin;
    GenericRemoteClass shadow_object;
```

```
HObjEntry *o_entry;
object o;
enum response_message message_type;

sin = *addr;

/*
 * Lets find out whether this remote object is
 * available on the remote host.
 *
 * If the object is not available on the remote
 * host, but the remote object do know where to
 * find this object, then the remote object will
 * return back the hostname and the port number.
 */

/*
if (!shadow_obj_protocol.is_object_exist(
 obj_name,&sin))
    {
    return((object)NULL);
    }
*/

/*
 * Creating the shadow object for the sender.
 * The port id for each remote object store
 * inside the object structure.
 */

shadow_object = GenericRemoteClass.Create(
                            shadow_obj_protocol,
                            &sin, obj_name);
/*
 * Initialize the dummy remote object stub.
 * It will look like an regular object, except
 * the method address of all the method will
 * point at a predefine function.
 */

/*
 * Creating a regular object, and attach shadow
```

```
     * object to this regular object.
     */
    o = class_ptr.CreateAndAttachObj(obj_name,
                       shadow_object,
                       "shadow_object_entry_point",
                       "",
                       "shadow_object_init_point",
                       "shadow_object_term_point");
    /*
     * Set the object name.
     */
    /*
     * Add the entry into the import table.
     */

    shadow_obj_protocol.import_object(o,obj_name);

    /*
     * Waiting for the initialization of the
     * shadow object.
     */

    return(o);
    }
```

8.3.5 Miscellaneous functions

Code of Miscellaneous functions

```
int reassign_port_num(int port_num)
    {
    return(shadow_obj_protocol.reassign_port_num(
port_num));
    }

struct sockaddr * dest_addr(struct sockaddr * sin,
                     char * hostname, int port_num)
    {
    return(shadow_obj_protocol.dest_addr(sin,
                               hostname,port_num));
    }
```

```
struct host_entry * construct_host_entry(struct
      host_entry * h_entry)
    {
    return(shadow_obj_protocol.
                    construct_host_entry(h_entry));
    }

end class  RemoteProtocol

end feature RemoteObj
```

8.4 S_alone_sender

8.4.1 Description of S_alone_sender

This next section, s_alone_sender, is a simple interface to the shadow objects. It is inherited from s_alone_obj, but differs from s_alone_obj by the shadow object protocol used. For this reason, s_alone_sender has only one function. That is the create_shadow_obj_protocol function. It is used to create the shadow object protocol and associate it with a shadow object. This is done with a call to the Create function in the metaClass.

It should be noted that the major difference between the s_alone_obj and a_alone_sender is that, s_alone_obj is a one to one implementation. That is, s_alone_obj will have a shadow object for each and every remote shadow object. The s_alone_sender implementation will have a one to many implementation. That is, a single shadow object may be associated with numerous remote shadow objects.

Code of S_alone_sender

```
feature RemoteObjSender

interface:

    imports MemoryAllocation[Memory]

    imports GenericObj[GenericRemoteClass]

    imports RemoteObj[RemoteProtocol]

    imports ShadowObjInterface[ShadowObjSenderProtocol]
```

```
    exports RemoteProtocolSender

    exports remote_protocol_sender

implementation:

    RemoteProtocolSender remote_protocol_sender =
 RemoteProtocolSender.Create();

/*
 * This protocol support the idea of $sender.
 */

class RemoteProtocolSender merges RemoteObj[RemoteProtocol] ::=

methods :

void create_shadow_obj_protocol()
    {
    shadow_obj_protocol =
    ShadowObjSenderProtocol.Create(GenericRemoteClass);
    }

end class  RemoteProtocolSender

end feature RemoteObjSender
```

8.5 Generic_remote_obj

8.5.1 Description of Generic_remote_obj

The next file, generic_remote_obj.m, contains all of the functions that implement the remote shadow object. Let us look at these functions closely, since the shadow object implementation depends on how these functions work.

Headers of Generic_remote_obj & init function

```
void reset_obj_and_frame();
extern int linearize_return_value();
extern int linearize();
extern int unlinearize_return_value();
```

```
extern int linearize_return_value_size();
extern int unlinearize_method_name();
extern int linearize_msg_size();
extern int linearize_msg();

extern int *
     $GenericObj$GenericRemoteClass$remote_object_handler();
extern char * construct_object_id();
extern struct _threadid deconstruct_thread_id();

feature GenericObj

interface:

    imports Unix[sys_obj]

    imports HashTable[Hash]

    imports MemoryAllocation[Memory]

    exports GenericRemoteClass

implementation:

    extern struct _frame * restore_current_obj();
    extern char * construct_thread_id();

class GenericRemoteClass merges MemoryAllocation[Memory] ::=

struct sockaddr sin; /* remote object internet address. */
int    remote_obj_id;
       /* Identify each unique GenericRemoteClass object. */
char   dest_obj_name[MAX_HOSTNAME];
object shadow_protocol;
int    object_status = 0; /* 0 imply ok, -1 error. */
int    export_caller = 0; /* If we implementing $sender,
     we need to export
                             the caller object. */

methods :

void init(object shadow_obj_protocol,
```

```
struct sockaddr * input_sin,
        char * obj_name)
  {
  <
  remote_obj_id = -1;
  /*
   * We have to know what shadow object protocol we
   * are going to use to connect to remote.
   */
  shadow_protocol = shadow_obj_protocol;

  sin = *input_sin;
  if (obj_name != NULL)
      {
      /*
       * If name equal to NULL, then user do not wish
       * to associate this shadow object with the
       * remote object.
       */
      strcpy(dest_obj_name,obj_name);
      }
  else
      dest_obj_name[0] = (char) NULL;
  >
  }
```

8.5.2 Shadow_object_init_point

Description of Shadow_object_init_point

The function will be called whenever an object is attached to a shadow object. It will first check to see if the object exists using the is_object_exist function in Section 8.6.18, in the generic_remote_interface chapter. This will ask the remote side if the object exists on their side. If the function returns false, then the object does not exist on their side. When this happens, the attachment is seen as failing, so a NULL object is returned. If the object does exist on the remote side, then we will issue the connect_remote call. This will set up the link between the shadow object and the remote machines shadow object stud.

It should be noted that the function name is not fixed, but is predefined by the user.

Code of Shadow_object_init_point

```
/*
 * When an object is attached to a shadow object, this
 * routine will be called.
 */

int shadow_object_init_point(object  obj)
    {
    if (!shadow_protocol.is_object_exist(dest_obj_name,
 &sin))
        {
        printf("MeldC Error : Remote object %s does not
exist. \n", dest_obj_name);
        return(FALSE);
        }
    shadow_protocol.connect_remote((object)obj,&sin);
    return(TRUE);
    }
```

8.5.3 Shadow_object_term_point

Description of Shadow_object_term_point

This function will be called when an object is detached from a shadow object. It will be used to clean up the shadow object. That is, remove the shadow object from the system. We do this by first calling the obj_destroy function, in Section 8.6.21, to destroy the connection. Now we may call the Destroy function, in Section 4.1.17 of the MetaClass chapter, to remove the shadow object.

It should be noted that the function name is not fixed, but is predefined by the user.

Code of Shadow_object_term_point

```
/*
 * When an object is deattached from a shadow object, this
 * routine will be called.
 */

void shadow_object_term_point(object obj)
    {
    shadow_protocol.obj_destroy((object)obj,$self);
```

```
    /*
     * Since we are destroyed ourselves, we will not be
     * able to reference this object after this point.
     */
    GenericRemoteClass.Destroy($self);
    return;
    }
```

8.5.4 Miscellaneous functions

Code of Miscellaneous functions

```
struct sockaddr *return_sockaddr()
    {
    return (&sin);
    }

void set_sockaddr(struct sockaddr *my_sin)
    {
    sin = *my_sin;
    }

char * return_remote_obj_name()
    {
    return(dest_obj_name);
    }

void set_remote_obj_id(int id)
    {
    remote_obj_id = id;
    }

int return_remote_obj_id()
    {
    return(remote_obj_id);
    }

int export_caller_name()
    {
    export_caller = 1;
    }
```

8.5.5 Receive_object_as_parameter

Description of Receive_object_as_parameter

The next function of interest is the receive_object_as_parameter. This function will allow the object to be passed to the remote location. It will basically do this by importing the object from the remote location using the network address that was passed by the pass_object_as_parameter. The network address is saved in front of the message received and will be stripped off of the message to be used to get the object from the remote source. It should be noted that the name, receive_object_as_parameter, is misleading. The function does not actually get the object by the object itself being passed, but the network address of where the object is located. See Section 8.5.7 for an example of receiving an object as a parameter.

The function will first find the remote address and store it. It will then construct the remote address by calling dest_addr, in Section 8.6.7. We then must do a malloc for the size of the object_as_param structure. Then the function will check to see if the object is in the import or the export tables. If the object is in the import table, it means that we have previously discovered where the object is and already have access to that object. If the object is in the export table, it means that we are the owner of the object or simply put, it is on the local side. The function can do this by using the is_object_imported and the is_object_exported functions located in the generic_remote_interface file, discussed below in Section 8.6.17 and Section 8.6.14, respectively. If the object has already been imported or is on the local side, we will then just assign the allocated structure the objects address from the entry in the table. If the object is not a local or imported object, then construct the absolute classname and import the object. This will be done using the function obj_init, in Section 8.3.4. It basically will construct the remote shadow object, which will capture all messages and forward these to the remote object, and assign the shadow object to the object structure allocated.

It should be noted that this function will always be called when a new message arrives from some remote source. This is due to the fact that this is the only function that can decide if an object was passed or not and then decide what to do with it. In other words, no other function can understand the linearized message string pass in.

Code of Receive_object_as_parameter

```
char * receive_object_as_parameter(char * message, struct
        object_as_param ** obj_param)
    {
```

```
    struct object_as_param *ptr = NULL;
    struct sockaddr addr;
    HObjEntry * o_entry;
    int remote_port_num;
    char remote_hostname[MAX_HOSTNAME];
    char class_name[MAX_HOSTNAME];
    char feature_name[MAX_HOSTNAME];
    char absolute_obj_name[MAX_HOSTNAME];

#ifdef TRACE
    printf("receive_object_as_parameter %s obj_param %x\n",
    message,*obj_param);
#endif

    if (*(message++) == '@')
        {
        /*
         * object are pass as parameter.
         */

        if (sscanf(message,"{ %s %d }",remote_hostname,
           &remote_port_num) != 2)
            {
            printf("MeldC Fatal error : receive_object
                     _as_parameter()\n");
            printf("                   : Invalid message
                    received. \n");
            sys_obj.exit(-1);
            }
        /*
         * Construct the remote address.
         */

        shadow_protocol.dest_addr(&addr,remote_hostname,
  remote_port_num);

        message = strchr(message,'#');

        while (*(message++) != '@')
            {
            if( *(message-1) != '#')
                {
```

```
                printf("MeldC Fatal error 1 : receive_
                        object_as_parameter()\n");
                printf("                    : Invalid
                        message received. \n");
                return(0);
                }

            ptr =(struct object_as_param *)
                        $self.malloc(sizeof(struct
                                    object_as_param));

            ptr->next=(struct object_as_param *) NULL;
            if (sscanf(message,"{ %s %s %s %s }",
                       feature_name,class_name,
                       ptr->object_name,
                       absolute_obj_name) != 4)
                {
                printf("MeldC Fatal error 2 : receive_
object_as_parameter()\n");
                printf("                    : Invalid
message received. \n");
                return(0);
                }

            message  = strchr(message,'}');

            message ++;

            o_entry = (HObjEntry *)
                shadow_protocol.is_object_imported(
                                    ptr->object_name);

#ifdef TRACE
            if (o_entry)
                printf("from the import table %x \n",
                        o_entry->obj_ptr);
#endif
            if (o_entry == NULL)
                {
#ifdef TRACE
                printf("Not from the imported table.\n");
#endif
```

```
                o_entry = (HObjEntry *) shadow_protocol.
                    is_object_exported(ptr->object_name);
                }

            sprintf(ptr->class_name,"$%s$%s",
                    feature_name,class_name);

            if ((o_entry == NULL) ||
                (o_entry->obj_ptr) == NULL)
                {
                MetaClass class_obj;

#ifdef TRACE
                printf("Not from the exported table.\n");
#endif
                /*
                 * The obj is not a local or imported obj
                 */

                /*
                 * construct the absolute classname.
                 */

                class_obj = MetaClass.
               find_class_obj_from_name(ptr->class_name);

                /*
                 * We need to import the object.
                 */

                if (class_obj == (MetaClass)NULL)
                    {
                    printf("MeldC Fatal error : receive_
    object_as_parameter()\n");
                    printf("                  : Unknown
    classname received. \n");
                    printf("                  : Perhaps
      you did not import the class.\n");
                    return(0);
                    }

                if (strcmp (ptr->object_name,
```

```
    absolute_obj_name) == 0)
                    {
                    /*
                     * The absolute object name is the
     * same as the object name.
                     */
                    ptr->obj =(struct _object *)
                    shadow_protocol.obj_init(class_obj,
              ptr->object_name,
      &addr);
                    }
                else
                    {
                    ptr->obj =(struct _object *)
      shadow_protocol.obj_init(
                                    class_obj,
                                    absolute_obj_name,
     &addr);
                    }
                }
            else
                {
                ptr->obj = o_entry->obj_ptr;
                }
#ifdef TRACE
            printf("The Object return %x \n",ptr->obj);
#endif
            if (*obj_param != NULL)
                ptr -> next = (*obj_param);

            *obj_param = ptr;
            }
        }
    else
        message--;

    return(message);
    }
```

8.5.6 Pass_object_as_parameter

Description of Pass_object_as_parameter

Next is the pass_object_as_parameter function, the other side of the above receive_object_as_parameter. It will basically pass an object to a remote source. This can be done by simply passing the object's network address to the remote source using the message being sent. This function will put the network address of the object in front of the linearized message. The remote side will see this piece of the message and use it to access the object. See Section 8.5.7 for an example of sending an object as a parameter. Let's look at the steps to do this.

The function will first find it's own network address using the return_my_local_address in the generic_remote_interface file. Then the function will put the local hostname and port number into the message using the return_my_local_hostname and return_my_local_port functions. Then, it will check to see if the object is exported. It does this with the is_object_exported function discussed in Section 8.6.14. If the object is exported, it means that we must update the proper address of the object at the remote location. This is done by sending the object's name and the network address of where the object exists to the remote source in a message.

If the object was not exported, we check to see if the oject was imported. That is, see if the object exists on another machine. In this case, the function will first find the network address of where the object exists. Then the function will send the name and network address to the remote source in a message.

If the object is local and it has not yet been exported to the remote machine yet, then we must first choose an absolute name for the object, using the construct_object_id function, in Section 2.14.7. This is done to identify it from other objects, by using the object id. Once it has an identity, the function can set the object's name using the set_obj_info function, Section 4.1.25, and put the name into the absolute_obj_name field. Then the function will declare the object to be exported by calling the export_object function. Then the function will set the class_name and feature_name. The function will then put all three names into the message and return it.

Code of Pass_object_as_parameter

```
char * pass_object_as_parameter(char ** message, int *size,
                                struct object_as_param * ptr)
    {
    HObjEntry * o_entry;
    struct sockaddr * my_address;
```

```
    char * class_name;
    char * feature_name;

#ifdef TRACE
    printf ("pass_object_as_parameter \n");
#endif
    my_address = (struct sockaddr *)
 shadow_protocol.return_my_local_address();


    char  data[MAX_HOSTNAME*10];/* We assume the user is
                                 * not going to pass more
                                 * than 10 objects to the
                                 * remote machine.
                                 */
    char * new_message = data;

    /*
     * if ptr is not equal to NULL, we will reconstruct the
     * pointer.
     */

    if (ptr != NULL)
        {
        *(new_message++) = '@';
        sprintf(new_message,"{ %s %d }",
                shadow_protocol.return_my_local_hostname(),
                shadow_protocol.return_my_local_port());

        new_message  = strchr(new_message,'}');
        new_message ++;
        *(new_message++) = '#';
        }

    while (ptr != NULL)
        {
        char obj_id[MAX_HOSTNAME];
        char absolute_obj_name[MAX_HOSTNAME];

        o_entry = (HObjEntry *) shadow_protocol.
      is_object_exported(ptr->object_name);
```

```
        if (o_entry == NULL)
            {
            /*
             * We will not export any shadow object.
             */

            o_entry = (HObjEntry *) shadow_protocol.
      is_object_imported(ptr->object_name);

            if (o_entry == NULL)
                {
                /*
                 * This is a local object, and we did not
 * export this obj to the remote machine yet.
                 *
                 * Since the object did not export by the
 * user, the system need to choose an
 * absolute name for the object.  This
 * absolute name will be construct by using
 * object id.
                 */

                (void)construct_object_id(obj_id,
                ptr->obj->$.objid);

                sprintf(absolute_obj_name,"%s%s",
obj_id,ptr->object_name);

                if ((ptr->obj != NULL) && (o_entry == NULL))
                    {
                    /*
                     * Export the obj to the remote machine.
                     */

                    /*
                     * Equivalent to PutObj.
                     */
                    ((MetaClass)(ptr->obj->$.class_ptr)).
set_obj_info((object)ptr->obj,
(enum obj_field)OBJ_NAME,
                                absolute_obj_name);
                    shadow_protocol.
```

```
                              export_object((object)ptr->obj,
                              absolute_obj_name);
                      }
                  }
              else
                  {
                  strncpy(absolute_obj_name,ptr->object_name,
MAX_HOSTNAME);
                  }
              }
          else
              strncpy(absolute_obj_name,ptr->object_name,
    MAX_HOSTNAME);

          class_name = ((MetaClass)(ptr->obj->$.class_ptr)).
  class_instance_variable(C_NAME);

          feature_name = ((MetaClass)(ptr->obj->$.class_ptr)).
      class_instance_variable(C_FEATURE_NAME);
          /*
           * Construct the pre-condition linear list, which
           * require remote object to import these objects
           * before executing the method.
           */

          sprintf(new_message,"{ %s %s %s %s }",feature_name,
                                     class_name,
                                     ptr->object_name,
                                     absolute_obj_name);

          new_message = strchr(new_message,'}') + 1;
                  {
                  /*
                   * This is a local object, and we did not
 * export this obj to the remote machine yet
                   *
                   * Since the object did not export by the
 * user, the system need to choose an
 * absolute name for the object.  This
 * absolute name will be construct by using
 * object id.
                   */
```

```
                (void)construct_object_id(obj_id,
  ptr->obj->$.objid);

                sprintf(absolute_obj_name,"%s%s",obj_id,
ptr->object_name);

                if ((ptr->obj != NULL) && (o_entry == NULL))
                    {
                    /*
                     * Export the obj to the remote machine.
                     */

                    /*
                     * Equivalent to PutObj.
                     */
                    ((MetaClass)(ptr->obj->$.class_ptr)).
set_obj_info(
                                (object)ptr->obj,
                                (enum obj_field)OBJ_NAME,
                                absolute_obj_name);
                    shadow_protocol.export_object(
(object)ptr->obj,
                                absolute_obj_name);
                    }
                }
            else
                {
                strncpy(absolute_obj_name,ptr->object_name,
MAX_HOSTNAME);
                }
            }
        else
            strncpy(absolute_obj_name,ptr->object_name,
    MAX_HOSTNAME);

        class_name = ((MetaClass)(ptr->obj->$.class_ptr)).
     class_instance_variable(C_NAME);

        feature_name = ((MetaClass)(ptr->obj->$.class_ptr)).
    class_instance_variable(C_FEATURE_NAME);
        /*
```

```
         * Construct the pre-condition linear list, which
         * require remote object to import these objects
 * before executing the method.
         */

        sprintf(new_message,"{ %s %s %s %s }",feature_name,
                                   class_name,
                                   ptr->object_name,
                                   absolute_obj_name);

        new_message = strchr(new_message,'}') + 1;

        if (ptr->next == NULL)
            *((new_message)++) = '@';
        else
            *((new_message)++) = '#';
        *(new_message) = (char ) 0;
        ptr = ptr -> next;
        }

    /*
     * Compute the size of the new message.
     */

    if (new_message != data)
        {

        /*
         * We will reconstruct the message that we send it
 * to remote.
         */

        int new_size;

        new_size = *size + (int)new_message - (int)data;

        new_message = $self.malloc(new_size+1);

        memcpy(new_message,data,new_size-(*size));
        memcpy((new_message + (new_size-(*size))),
message,*size);
        *(new_message + (new_size)) = (char ) 0;
```

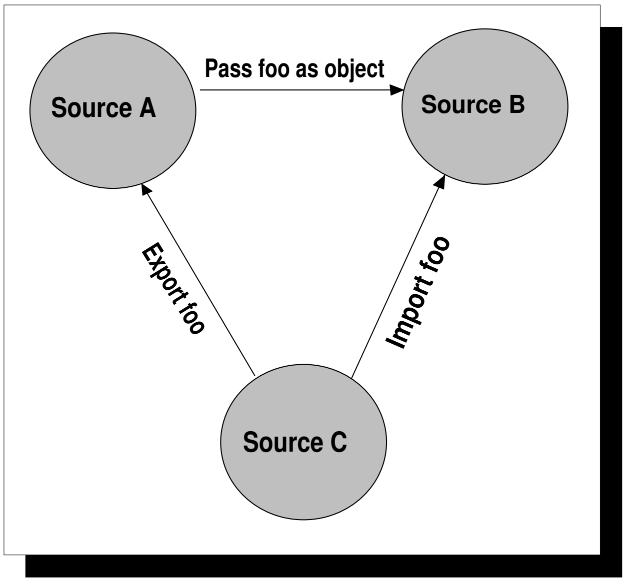

Figure 8.2: An example of passing and receiving an object as a parameter.

```
/* Set the last byte to NULL*/

    *size = new_size;
    return(new_message);
    }
}
```

8.5.7 Example of Receiving and Passing an Object as a Parameter

Here we will give an example of both the receive_object_as_parameter and the pass_object_as_parameter functions. First, see Figure 8.2 for a picture of what will happen. Here, source C will export object foo to source A. This means that source A knows where object foo is. At some point, Source A will pass object foo to Source B. Source B will receive_object_as_parameter from Source A and then import the object from Source C.

8.5.8 Remote_obj_dispatch

Description of Remote_obj_dispatch

Next is the remote_obj_dispatch function. This function will dispatch the incoming message to the local object. This message though contains an object. The function must also put the message into a form that the object may process it. It should be noted that the variable sender_info contains the sender's unique threadid so that a return message can be sent back.

The first thing to do is change the threadid of the current frame's thread id to the thread id of the remote thread. This will give the local object the illusion that the remote object is identical to the local thread. Next we will find the class pointer using the obtain_obj_info function, Section 4.1.24. Then the function will do a receive_object_as_parameter call. This will get for us the source message. Then we will need to find the method pointer from the source message. To do so we need to unlinearize the method name first, since it was linearized to send it across the network. This is done using the unlinearize_method_name function, see Section 2.9.12. At this point we must dynamically bind the method, so we will call the binding function, Section 4.2.7, to do so. Then the frame will be ready to be enqueued to run. After it is processed, we must then send back the return value. The means we must reverse what we just did, linearize the return value and find the size of the value first using the linearize_return_value_size and linearize_return_value. The linearize size function will tell us how much memory to malloc to hold the return value. Then we must call the pass_object_as_parameter function if the return value is an object. If the return value is not an object, then return the value. Finally, we free up the memory allocated using the malloc call.

Code of Remote_obj_dispatch

```
char * remote_obj_dispatch(object obj,char * obj_name,
   char * threadid_string,
                           char * src_msg,
                           enum msg_types msg,
                           int *return_length)
    {
    int * ret;
    object class_ptr;
    struct _method * mp ;
    struct _threadid threadid;
    char method[MAX_METHOD_NAME] ;
    char * ret_mess;
```

```
    struct object_as_param * obj_param = NULL;

    /*
     * First we need to obtain the sender information.
     */

#ifdef TRACE
    printf("Enter remote_obj_dispatch \n");
#endif

    /*
     * Changing the threadid of the current thread.
     */

    if (deconstruct_thread_id(threadid_string,&threadid))
        {

        /*
         * this is a synchronous call.
         */

        /*
         * Changing the current frame's thread id to the
 * thread id of the remote thread.
         */

        MetaClass.reset_current_frame_thread_id(threadid);
        }

    strcpy(dest_obj_name,obj_name);

#ifdef DEBUG
    printf("%s %s \n",obj_name,threadid_string);
#endif

    /*
     * Find out the class poiointer.
     */

    class_ptr = (object)(MetaClass.obtain_obj_info(
                     obj, (enum obj_field)OBJ_CLASS_PTR));
    /*
```

```
     * Find out the method pointer from the input data.
     */

    /*
     * Fisrt we should strip out any header information...
     */

    if ((src_msg = $self.receive_object_as_parameter(
             src_msg,&obj_param)) == (char *)NULL)
      {
      printf("MeldC Error:Unable to parse the message.\n");
      return(0);
      }

    if (unlinearize_method_name(src_msg,method,
MAX_METHOD_NAME,NULL) == 0)
        {
        printf("Unable to unlinearize the method name.\n");
        return (0);
        }

    ret_mess = NULL;

    /*
     * dynamic bind the method, and issue the call.
     */

    /*
     * Dynamic bind the method, and than enqueue the frame.
     * We also need to change the thread id .
     */

    ((MetaClass)(class_ptr)).binding ($self,obj,
          src_msg,
          &ret_mess,
          method,
          &mp,
          (enum msg_types)msg,(char *)obj_param);

    while (obj_param != NULL)
        {
        struct object_as_param * ptr;
```

```
        ptr = obj_param;
        obj_param = obj_param->next;
        $self.free((char *)ptr);
        }

    obj_param = NULL;

    char * l_return;
    int    l_return_size;

    l_return_size=linearize_return_value_size(mp,ret_mess);

    l_return = $self.malloc(l_return_size);

    linearize_return_value(mp,l_return,ret_mess,&obj_param);

    if (obj_param)
        {
        /*
         * We are returning an object...
         */
#ifdef DEBUG
        printf("Returning object as parameter. \n");
#endif
        if (obj_param)
            ret_mess = $self.pass_object_as_parameter(
           l_return,
           &l_return_size,
                                         obj_param);
        /*
         * Free up the old return value.
         */

        $self.free(l_return);
        }
    else
        ret_mess = l_return;

#ifdef DEBUG
    printf("ret_mess = %x l_return = %x\n",ret_mess,
    l_return);
#endif
```

```
#ifdef DEBUG
    printf("ret_mess = %x l_return = %x\n",ret_mess,
    l_return);
#endif

    /*
     * WE NEED TO FREE ret_mess some how ???????
     */
#ifdef DEBUG
    printf("The method is called. %s\n",ret_mess);
#endif
    * return_length =  l_return_size;

    return(ret_mess);
    }
```

8.5.9 Shadow_object_entry_point

Description of Shadow_object_entry_point

The next function is the shadow_object_entry_point. The function is bascially used to handle an incoming message coming from a remote source. That is, this function is used to handle the Kernel trap. Here is the detail of how this is done.

First, the frame is linearized so that it can be sent across to the remote location. This is done using the linearize function, Section 2.9.1. If the object_param structure is returned from the linearize function not equal to NULL, then we are trying to pass an object as a parameter to the remote source. In this case we must call the pass_object_as_parameter function. This will enable the remote side to access the object by passing the address of the object. We must also change our hash table so that it shows that the object has been exported to the remote source. Then, depending on if the call was synchronous or asynchronous, we will first call the obtain_obj_info function, Section 4.1.24. This will give us the object name. Then we will construct the threads id with the construct_thread_id function. Finally, we will call the export_object function, see Section 8.6.15. In the case of the asynchronous call, we will actually be exporting the sender. This will put the object into the export hash table for us to show that the object has been exported. Next, we must build a remote request message. So, we will call malloc to allocate memory for a message. Then we make a call to enqueue_remote_request, in Section 8.6.8. This will save the message structure for us into the message hash table, so that we can determine who to wake up later when a response message arrives back. We will then do a send_message_request to actually send our message to the remote location, see Section 8.6.22. Notice that our message will have the CSSENDTOREQUEST command. At this point, we will do a delayuntil caller_name. This will suspend this process until a new message is received back. Then the process will be woken up to continue at this point again.

The first thing to do after being woken up, is to check the hash table with a query_remote_request. This will check the hash table and return the entry placed there by us. If the entry is NULL, then we will set the return value to zero by making a unlinearize_return_value function call with the buffer field equal to "i0" and the obj_param field equal to NULL. If the entry is not NULL, then we will take the response message from the entry and then we can then do a dequeue_remote_request, see Section 8.6.9, to delete our entry from the hash table. If the frame's caller is NULL, then we do not have to return a value. In this case, we will free the message memory allocated and return NULL. If the response message was not delivered, we will also free up the allocated message with a free function call and also return NULL. If the message was received and the caller is not NULL, then return to the caller. This will allow us to return the value back to the caller. To do so, we first must check to see if the remote source sent an object as the return value. This is done using the receive_object_as_parameter, Section 8.5.5. If the return value is an object, then we will identify the object and obtain access to that object by importing it. If the return value is not an object, then we will unlinearize the the return value, once again with the unlinearize_return_value. We will then obtain the original object information, free the original message memory and return to the caller.

It should be noted, that if the entry function returns a NULL pointer, this implies that the original function should not be resumed. If the entry function returns a frame, then the Kernel will resume the execution of the frame right after the execution of the shadow object.

Code of Shadow_object_entry_point

```
struct _frame * shadow_object_entry_point(struct _frame *fp)
    {
    int return_value;
    char caller_name[Max_Object_Name_Size];
    RESP_MESS * resp_message;
    int *return_message;
    HEntry *entry;
```

```
/*
 * We need the linearzation algorithm here.
 */

char thread_buf[100];
char * message;
int size;
struct object_as_param * ptr = NULL;

/*
 * Obtain my local address.
 */

/*
 * We need to find out how many objects that we are
 * passing as parameter.
 */

/*
 * ptr == contain the name of the objects that we will
 *        pass as parameter.
 */

size = linearize(fp,&message,&ptr);

if (ptr)
    message = $self.pass_object_as_parameter(message,
        &size,ptr);

if (fp->$.caller != (struct _frame *)NULL)
    {
    char * obj_name;
    /*
     * Synchronous call.
     */

    obj_name = MetaClass.obtain_obj_info(
                (object)(fp->$.caller->$.obj),
                (enum obj_field)OBJ_NAME);

    sprintf(caller_name,"%s#%s", obj_name,
                construct_thread_id(thread_buf,
```

```
$thread,SYNC));

        /*
         * Export the caller object automatically.
         */

        shadow_protocol.export_object((object)(
fp->$.caller->$.obj),obj_name);
        }
    else
        {
        char * obj_name;

        obj_name = MetaClass.obtain_obj_info(
                            (object)(fp->$.sender),
                            (enum obj_field)OBJ_NAME);

        sprintf(caller_name,"%s#%s",obj_name,
                construct_thread_id(thread_buf,$thread,
    ASYNC));

        shadow_protocol.export_object((object)(
       fp->$.sender),obj_name);
        }

    resp_message = (RESP_MESS *)$self.malloc(
    sizeof(RESP_MESS));

    shadow_protocol.enqueue_remote_request(resp_message,
      caller_name,&sin);

    if ((return_value = shadow_protocol.
send_message_request(
                          dest_obj_name,
                          caller_name,
                          remote_obj_id,
                          CSSENDTOREQUEST |(int)
 (_frame->$.str_or_sym),
                          INIT_REQ,
                          (char *)message,
                          size,
                          &sin)
```

```
                        )== NULL)
        {
        /*
         * Unable to send to message to the name server.
         *
         * We need to define a set of exception handler.
         */
        shadow_protocol.dequeue_remote_request(
   caller_name);

        (void)$self.free((char *)resp_message);

        printf("ERROR in remote object handler. \n");
        }

    /*
     * Waiting for the return code.
     */

    if (ptr)
        {
        /*
         * If the new_message is not equal to NULL,
         * we will free up its memory.
         */
        $self.free(message);
        }

    delayuntil caller_name;

    entry = (HEntry *)shadow_protocol.
      query_remote_request(caller_name);

    if (entry == NULL)
        {
        /*
         * The entry do not exist, and we should just
 * return.
         */
        (void)unlinearize_return_value(fp,"i0 ",NULL);

        return(NULL);
```

```
        }

#ifdef DEBUG
    printf("return_from_dequeue %x \n",entry->ptr);
#endif

    resp_message = (RESP_MESS*)entry->ptr;

    shadow_protocol.dequeue_remote_request(caller_name);

    if (fp->$.caller == (struct _frame *)NULL)
        {
        /*
         * Asynchronous call, no need to parse the return
 * value.
         */
        if (resp_message->resp_mess)
           shadow_protocol.free(resp_message->resp_mess);

        $self.free((char *)resp_message);

        return(NULL);
        }

    if (resp_message->resp_type ==
(int)NOT_DELIVERED_MESSAGE)
        {
#ifdef DEBUG
        printf("No response from remote, TIMEOUT \n");
#endif

        (void)$self.free((char *)resp_message);
        /*
         * We should generate a event, that indicates
 * we are unable to deliver the message to the
         * remote object.
         */
        (void)unlinearize_return_value(fp,"i0 ",NULL);
        return(NULL);
        }

    /*
```

```
     * We have not yet connect the local shadow object
     * with any particular remote object.
     */

    if (remote_obj_id == -1)
        {
#ifdef TRACE
        printf("Setting remote object id . \n");
#endif
        remote_obj_id = resp_message->remote_obj_id;
        }

    /*
     * We migh want to run through the linearzation
     * algorithm.
     */

    return_message = (int *)(resp_message->resp_mess);

    (void)$self.free((char *)resp_message);

    if (return_message == 0)
      {
      printf("POSSIBLE ERROR in the remote object.\n");
      (void)unlinearize_return_value(fp,"i0 ",NULL);
      return(NULL);
      }
    /*
     * We need send this into the linerzation algorithm
     * and return the the actual value back to the
     * calling program.
     */

    /*
     * Restore the fp ptr back to the orginal frame
     * pointer, so that we will able to return the
     * value back to the caller.
     */

    enum return_type ret_type;
    struct _para * para;
    struct object_as_param * obj_param = NULL;
```

```
    char * original_message;

    original_message = (char *)return_message;

    if ((return_message = $self.
    receive_object_as_parameter(return_message,
                                    &obj_param))
            == (char *)NULL)
        {
        printf("MeldC Error : Unable to parse the
message. \n");

        return(NULL);
        }

#ifdef DEBUG
    if (obj_param)
        {
        printf("receive object as return value %s \n",
return_message);
        }
#endif

    /*
     * If the return message is a pointer, the pointer
     * should be allocate as part of the shadow obj's
     * memory pool.
     */

    (void)unlinearize_return_value(fp,return_message,
   obj_param,
   fp->$.caller);

    while (obj_param != NULL)
        {
        struct object_as_param * ptr;
        ptr = obj_param;
        obj_param = obj_param->next;
        $self.free((char *)ptr);
        }

    shadow_protocol.free(original_message);
```

```
    return(NULL);
    }
```

8.5.10 Meldc_default

Description of Meldc_default

The final function in the Generic_remote_obj section is the meldc_default function. This function is used when someone sends a message to it's caller. In this case, the caller is located at the remote machine. The function will set the shadow object so that it represents the caller. This can be seen as a function that implements the $sender command. There is a problem though right off the start, the remote sender information is not accessable to this object. This does not allow us to determine the return type from $sender. Therefore it is assumed to be char *.

We first will obtain the object information and the thread id from the obtain_obj_info and construct_thread_id functions respectively. Then we will find the message size needed with the linearize_msg_size function. We will then linearize the message with the linearize_msg function. We will then make the object accessable using the pass_object_as_parameter function, see Section 8.5.6. Then we must put the request that we are to send into the message hash table using the enqueue_remote_request function. Then we can send the message using the send_message_request function. We then free up the message memory and do a delayuntil caller_name to wait for a response back.

The response must then be parsed. So we will do a query_remote_request call to get the entry value saved in the hash table. Then do a dequeue_remote_request to take the entry out of the message hash table. We can then free up the response message memory. Next we must check to see if the response is an object by calling the receive_object_as_parameter function. If it is an object, will then import the object with the message information recevied in the response message. If the return value is not an object, then we will unlinearize the return value with the unlinearize_return_value function. And finally, we can then free up the return the message.

Code of Meldc_default

```
char * meldc_default(char * method_name, optional)
    {
    char caller_name[Max_Object_Name_Size];
    char thread_buf[100];
```

```
    RESP_MESS * resp_message;
    char * tmp_ptr;
    int  sender_shadow_object_id = 0;
    int size;
    char * message;
    struct object_as_param * obj_param = NULL;

    sprintf(caller_name,"%s#%s",
           MetaClass.obtain_obj_info($sender,
               (enum obj_field)OBJ_NAME),
           construct_thread_id(thread_buf,$thread,SYNC));
    /*
     * The remote sender infomration is not visable to
     * this object.  There is no way for us to identify
     * the return type from $sender.  We will simply
     * assume it is char *.  We WILL NOT perform any
     * type checking on the return value.
     */

    /*
     * Linearize the message.
     */

    size = linearize_msg_size(method_name,_frame);

    message = (char *)$self.malloc(size);

    (void)linearize_msg(method_name,_frame,message,
&obj_param);

    if (obj_param)
        message = $self.pass_object_as_parameter(
  message,&size,obj_param);

    resp_message = (RESP_MESS *)$self.malloc(
sizeof(RESP_MESS));

    shadow_protocol.enqueue_remote_request(
    resp_message,caller_name,&sin);

    if ((shadow_protocol.send_message_request(
     dest_obj_name,
```

```
                              caller_name,
                              remote_obj_id,
                              CSSENDTOREQUEST,
                              INIT_REQ,
                              (char *)message,
                              size,
                              &sin)
                              )== NULL)
        {
        /*
         * Unable to send to message to the remote
         */
        shadow_protocol.dequeue_remote_request(
caller_name);
        printf("MeldC Fatal ERROR : $sender : Send
Failed, Unable to send \n");
        printf("                           : message
to $sender. \n");
        return(0);
        }
    /*
     * Waiting for the return code.
     */

    $self.free(message);

    delayuntil caller_name;

    obj_param = NULL;

    /*
     * Parse the result.
     */

    int *return_message;
    HEntry *entry;
    char *return_buf;

    entry = (HEntry *)shadow_protocol.
                   query_remote_request(caller_name);

    if (entry == NULL)
```

```
        {
        /*
         * The entry do not exist, and we should just
         * return.
         */
        return(0);
        }

#ifdef DEBUG
    printf("return_from_dequeue %x \n",entry->ptr);
#endif

    resp_message = (RESP_MESS*)entry->ptr;

    shadow_protocol.dequeue_remote_request(
      caller_name);

    if (resp_message->resp_type ==
(int)NOT_DELIVERED_MESSAGE)
        {

        printf("MeldC Error : meldc_default() : No
response from remote. \n");

        (void)$self.free((char *)resp_message);

        /*
         * We should generate a event, that indicates
 * we are unable to deliver the message to the
 * remote object.
         */

        (void)unlinearize_return_value(_frame,"i0 ",
       NULL);
        return 0;
        }
    /*
     * We migh want to run through the linearzation
     * algorithm.
     */

    return_message = (int *)(resp_message->resp_mess);
```

```
    (void)$self.free((char *)resp_message);

    if (return_message == 0)
        {
        printf("MeldC Error : $sender : Invalid return
message. \n");
        return(0);
        }

    if ((return_message =
       $self.receive_object_as_parameter(return_message,
                                          &obj_param))
            == (char *)NULL)
        {
        printf("MeldC Error : Unable to parse the
message. \n");
        return(0);
        }

   (void)unlinearize_return_value(_frame,return_message,
  obj_param);

    while (obj_param != NULL)
        {
        $self.free((char *)obj_param);
        obj_param = obj_param->next;
        }

    if (return_message)
        shadow_protocol.free((char *) return_message);
    }

end class  GenericRemoteClass

end feature GenericObj
```

8.6 Generic_remote_interface, ShadowObjProtocol Class

8.6.1 Description of Generic_remote_interface, ShadowObjProtocol Class

The file generic_remote_interface contains two different classes. See Figure 8.3 to see the pictorial representation of the two classes. The first is the ShadowObjProtocol class. This class will implement shadow objects as one to one. This means that there is one shadow object on the local side for every remote shadow object. In this case, the shadow object will have the identity of the remote shadow object. This class will be discussed in this section. Below we will describe the three functions used to implement the saving, finding and deleting of these remote generic objects.

The other class will be used to implement the $sender command, the ShadowObjSenderProtocol class. This class for the $sender will implement shadow objects for the case of one to many. That is, for a local shadow object, there could be many remote shadow objects associated with it. In this case, the sender is responsible for keeping track of the remote shadow objects identity and the shadow uses this to find the correct remote shadow object when needed. This class will be discussed in the next section, Section 8.7.

This file also contains the lowest level of code for the sending and receiving of messages and objects. Most of the functions will manipulate the network itself. Here is a look at the functions that do so.

Headers of Generic_remote_interface, ShadowObjProtocol Class & init function

```
extern char * construct_object_id();
extern char * construct_thread_id();

feature ShadowObjInterface

interface:

    imports Unix[sys_obj]

    imports HashTable[Hash]

    imports MemoryAllocation[Memory]
```

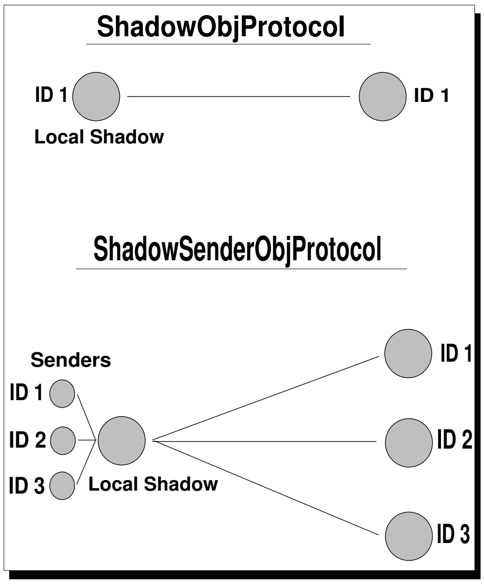

Figure 8.3: Pictorial representation of the two classes.

```
    imports NetworkUdp[NetworkProtocol]

    imports Verify[RemoteVerification]

    imports EnDecode[Encode_Decode_Buffer]

    exports ShadowObjProtocol

    exports ShadowObjSenderProtocol

implementation:

/*
 * The portocol that is required to communicate with the
 * name server.
 */
class ShadowObjProtocol merges MemoryAllocation[Memory] ::=

NetworkProtocol net_pro =
                       NetworkProtocol.Create();
```

```
RemoteVerification remote_verify =
                       RemoteVerification.Create();

Encode_Decode_Buffer en_de_buf =
                       Encode_Decode_Buffer.Create();

Hash export_table  = Hash.Create();
Hash import_table  = Hash.Create();
Hash message_table = Hash.Create();
Hash generic_obj_table = Hash.Create();
int shadow_obj_id = 0; /* shadow _object id */
struct sockaddr my_address;
int initialize = 0;
object shadow_object_class;
                      /* The shadow object's class. */


methods:
int init(object obj)
    {
    <
    /*
     * Attach an shadow object implementation onto the
     * shadow object interface.
     *
     * The shadow object interface can be combine with
     * multiple implemnetation of the shadow object.
     */

    shadow_object_class = obj;
    (void)remote_verify.set_protocol_obj($vself);

    if (!net_pro.is_disable())
        {
        send receive_message() to $vself;
        }
    /*
     * Initialize the message table.
     */
    message_table.meldcNewHashTable(STRINGHASH,
    DEFAULTSIZE, 20);
```

```
    (void)$self.dest_addr(&my_address,
  net_pro.local_hostname(),
                        net_pro.local_port());
    >
    }
```

8.6.2 Reassign_port_num

Description of Reassign_port_num

The reassign_port_num function, will take a port number as an argument and change the old port number to that given port number. This is useful for making a port number a well known port number instead of a system assigned port.

Code of Reassign_port_num

```
int reassign_port_num(int port_num)
    {
    int old_port_num;

#ifdef TRACE
    printf("reassign port number \n");
#endif

    old_port_num = net_pro.reassign_port_num(port_num);
    (void)$self.dest_addr(&my_address,
  net_pro.local_hostname(),
                        net_pro.local_port());
    send receive_message() to $vself;
    return(old_port_num);
    }
```

8.6.3 Return_my_local_address

Description of Return_my_local_address

There is also the return_my_local_address function that will return your address.

Code of Return_my_local_address

```
struct sockaddr * return_my_local_address()
```

```
    {
    return(&my_address);
    }
```

8.6.4 Return_my_local_port

Description of Return_my_local_port

There is also the return_my_local_port function that will return the port number that is local to you.

Code of Return_my_local_port

```
int return_my_local_port()
    {
    return(net_pro.local_port());
    }
```

8.6.5 Return_my_local_hostname

Description of Return_my_local_hostname

You can also find out your local hostname by using the return_my_local_hostname function.

Code of Return_my_local_hostname

```
char * return_my_local_hostname()
    {
    return(net_pro.local_hostname());
    }
```

8.6.6 Construct_host_entry

Description of Construct_host_entry

This function, construct_host_entry, will fill in the host_entry structure. This structure contains the port number and the hostname.

Code of Construct_host_entry

```
struct host_entry * construct_host_entry(struct host_entry
                      * h_entry)
    {
```

```
    strcpy(h_entry->host_name,net_pro.local_hostname());
    h_entry->port_num = net_pro.local_port();
    return(h_entry);
    }
```

8.6.7 Dest_addr

Description of Dest_addr

This function will construct the destination address given the hastname and port number as passed parameters.

Code of Dest_addr

```
struct sockaddr * dest_addr(struct sockaddr * sin,
    char * hostname, int port_num)
    {
    struct hostent * hp;
    struct sockaddr_in * sin_in=(struct sockaddr_in *)sin;

    /*
     * Get network host entry.
     *
     * IMPORTANT : C LIBRAY, the info is contain in a
     *             static area.
     */

    hp = gethostbyname(hostname);

    if (hp == NULL)
        {
        printf("gethostbyname failed Hostname = %s\n",
               hostname);
        return(NULL);
        }
    /*
     * Set up the server address.
     */

    bcopy (*(hp->h_addr_list),&(sin_in->sin_addr),
           hp->h_length);
```

```
#ifdef TRACE
    printf("hostname %s \n",hostname);
#endif

    sin_in->sin_family = AF_INET;
    /*
     * Right the now the port number is hard code it, later
     * we should change it, so that it will perfrom a lookuo
     * from a table.
     */
    sin_in->sin_port = (u_short)htons(port_num);

    bzero(sin_in->sin_zero,sizeof(sin_in->sin_zero));

    return(sin);
    }
```

8.6.8 Enqueue_remote_request

Description of Enqueue_remote_request

The next major function that is used quite often is the enqueue_remote_request. When we send a message to the remote side, we will save the message structure into the message hash table under the message id and then suspend ourselves. When we get a response message back, we can check the message table for the message id and see which thread to wake up and who responded to us. This function will take care of saving the request into the message hash table for us.

The function will initialize the message to be saved with the message name, the remote_obj_id, the response type, and the destination address. After it has done that, the function will then insert the request into the message hash table with the message_table.insertHash command.

Code of Enqueue_remote_request

```
void enqueue_remote_request(RESP_MESS * resp_message,
                            char * resp_name,
    struct sockaddr *sin)
    {
    HEntry * entry;
    <
#ifdef DEBUG
```

```
    printf("enqueue message %s %x\n", resp_name,
    resp_message);
#endif

    /*
     * initialize the resp_message packet.
     */
    strcpy(resp_message->resp_name,resp_name);
    /*
     * Save the dest address.
     */
    resp_message->remote_obj_id = -1;
    resp_message->resp_type = 0;
    resp_message->resp_mess = (char *)NULL;
    resp_message->dest_addr = *sin;

    entry = message_table.insertHash(resp_name);

    if (entry == NULL)
        {
        printf("Duplicate entry in the response table.
        %s\n",resp_name);
        entry = message_table.meldcHash(resp_name);
        if (entry == NULL)
            {
            printf("Meldc internal inconsistency \n");
            }
        }
    entry->ptr = (int *)resp_message;
    >
    }
```

8.6.9 Dequeue_remote_request

Description of Dequeue_remote_request

Next is the opposite of the enqueue, the dequeue_remote_request function. This function simply takes the entry off of the message hash table. This will occur after a request has been made and the process was put to sleep. When the process is woken up, it can then remove the entry.

It will do this by using the response name that was saved in the table during the enqueue function. It will then use this response name, resp_name and the message_table.deleteHash call.

Code of Dequeue_remote_request

```
HEntry * dequeue_remote_request(char * resp_name)
    {
    HEntry * entry;

#ifdef DEBUG
    printf("dequeue message %s \n", resp_name);
#endif
    <
    message_table.deleteHash(resp_name);
    >
    }
```

8.6.10 Query_remote_request

Description of Query_remote_request

The query_remote_request function has the purpose of looking up the entry in the message hash table and returning it. This was also done after a delayuntil for the sending of a request to a remote location.

To do so, the function will use the response name saved into the table by the enqueue function. The resp_name will be used by the message_table.meldcHash call. The value found is returned.

Code of Query_remote_request

```
HEntry * query_remote_request(char * resp_name)
    {
    HEntry * entry;
    <
#ifdef DEBUG
    printf("query remote  message %s \n", resp_name);
#endif

    entry =  message_table.meldcHash(resp_name);

#ifdef DEBUG
    if (entry)
        printf("query message addr %x \n",entry->ptr);
#endif
```

```
    >
    return(entry);
    }
```

8.6.11 Save_generic_obj

Description of Save_generic_obj

The next function is the save_generic_obj function. This function will save the identity of some generic remote shadow object so that the local shadow object can find it later. The function will simply take that object and insert it into the generic_obj_table using the generic_obj_table.insertHash call. The objects are saved into the hash table using the shadow_obj_id.

Code of Save_generic_obj

```
int save_generic_obj(object obj)
    {
    HEntry * entry;
    char key[20];
    <
    sprintf(key,"%d",shadow_obj_id);

#ifdef DEBUG
    printf("Key = %s \n",key);
#endif

    entry = generic_obj_table.insertHash(key);

    if (entry != NULL)
        {
        entry->ptr = (int *)obj;
        shadow_obj_id ++;

#ifdef DEBUG
        printf("In save_generic_object shadow_obj_id =
%d \n",shadow_obj_id);
#endif
        return(shadow_obj_id-1);
        }
    return(-1);
    >
    }
```

8.6.12 Find_generic_obj

Description of Find_generic_obj

Next we have the find_generic_obj function. The function will find the generic object given the object id from the remote side. The hash table will be searched by using the generic_obj_table.meldcHash call. The entry in the table will be returned if the entry was not NULL, meaning the generic object exists. This entry will contain the identity of the generic remote shadow object. Otherwise, the value of NULL will be returned stating that the generic object does not exist.

Code of Find_generic_obj

```
object find_generic_obj(int generic_obj_id)
    {
    HEntry * entry;
    char key[20];
    <
#ifdef TRACE
    printf("Enter find_Generic_obj. \n");
#endif
    sprintf(key,"%d",generic_obj_id);

    entry = generic_obj_table.meldcHash(key);

    if (entry)
        return((object) entry->ptr);
    else
        return((object) NULL);
    >
    }
```

8.6.13 Delete_generic_obj

Description of Delete_generic_obj

The delete_generic_obj function will simply find the identity of the generic object in the generic object table using the generic_obj_id passed in. The function will delete the identity of the generic shadow object and the function calling this one will use the identity to destroy the remote object, thereby removing the connection between the local and remote shadow object stubs.

The function will first check that the entry exists using the generic_obj_table.meldcHash call. If the call returns NULL, then the entry does not exist. If the entry exists, then the entry will be saved into an object o. The generic_obj_table.deleteHash will be used to remove the entry into the table. Then we will return the entry saved in o.

Code of Delete_generic_obj

```
object delete_generic_obj(int generic_obj_id)
    {
    HEntry * entry;
    char key[20];
    object o;
    <
    sprintf(key,"%d",generic_obj_id);

    entry = generic_obj_table.meldcHash(key);

    if (entry == (HEntry *)NULL)
        {
        printf("MeldC Error : Invalid shadow object id.");
        return;
        }
    if (entry->ptr)
        {
        o = (object)entry->ptr;
        }
    generic_obj_table.deleteHash(key);
    >
    return(o);
    }
```

8.6.14 Is_object_exported

Description of Is_object_exported

The next function, is_object_exported, will simply look up the given object name in the export table and return the entry there. It will do this using the export_table.meldcHash call. If the entry is NULL, then the object was not exported. Else, the object has been exported and the entry will be returned.

Code of Is_object_exported

```
HObjEntry * is_object_exported(char * obj_name)
    {
    HObjEntry * o_entry;

    o_entry=(HObjEntry *)export_table.meldcHash(obj_name);

    return(o_entry);
    }
```

8.6.15 Export_object

Description of Export_object

This function, export_object, is used to put an object into the export table after some remote location has been granted access to the object. It will be passed the object, obj, and the object name, obj_name.

The function will insert the obj_name into the hash table using the export_table.insertHash call. The call will return the value into the o_entry variable. This variable will be used for error checking. When the table is checked from here on out for this object name, it will verify that the object can be accessed from the remote source. This is true only until the entry is removed.

Code of Export_object

```
void export_object(object obj, char * obj_name)
    {
    HObjEntry * o_entry;

    o_entry=(HObjEntry *)export_table.insertHash(obj_name);

    if (o_entry == NULL)
        {
        o_entry = (HObjEntry *)export_table.
       meldcHash((char *)(
                            MetaClass.obtain_obj_info(
                            obj,
                            (enum obj_field)OBJ_NAME)));
        if  (o_entry == NULL)
            {
            printf ("Meldc fatal error, hash table internal
```

```
     inconsistency \n");
            return;
            }
        }
    o_entry->obj_ptr    = (struct _object *)obj;

    return;
    }
```

8.6.16 Import_object

Description of Import_object

This function, import_object, is also used to put an object into a table, the import table. This is done after an onject has been imported from a remote source. It will also be passed the object, obj, and the object name, obj_name.

It will do a import_table.insertHash call to put the object name into the table. This call will also return a value into the o_entry variable for error checking as well.

Code of Import_object

```
void import_object(object obj, char * obj_name)
    {
    HObjEntry * o_entry;

    o_entry=(HObjEntry *)import_table.insertHash(obj_name);

    if (o_entry != NULL)
       o_entry->obj_ptr  = (struct _object *)obj;
    else
        {
        /*
         * If user ask to import the same object twice.
         */
        o_entry = (HObjEntry *)
  import_table.meldcHash(obj_name);

        if (o_entry != NULL)
            o_entry->obj_ptr  = (struct _object *)obj;
        }
```

```
    return;
    }
```

8.6.17 Is_object_imported

Description of Is_object_imported

The next function, is_object_imported, which as discussed earlier, is similiar to the export version above. It will simply look up the given object name, obj_name, in the import table.

It will do so using the import_table.meldcHash call and return the value into the entry variable. If the entry is NULL, then the object was not imported. Else, the object was already imported and the entry is returned.

Code of Is_object_imported

```
HObjEntry * is_object_imported(char * obj_name)
    {
    HObjEntry * o_entry;

    o_entry=(HObjEntry *)import_table.meldcHash(obj_name);

    return(o_entry);
    }
```

8.6.18 Is_object_exist

Description of Is_object_exist

The is_object_exist function will ask the remote side if the object exists on their side. To do so, we must be passed the object name assigned to the object we are looking for and we also need the address of the object on the remote side.

We will first take the host information passed in and add it to the host table to be used to verify the health of the remote object. This is done with the add_active_remote_obj function, discussed in Section 8.8.3. This function will basically add the remote object into a list to be used to verify that the object is still alive on the remote side. The is_object_exist function will then get the object name and thread id with the obtain_obj_info and construct_thread_id functions respectively. Next, we will do a malloc to get the memory needed for the message to be placed in. We will now execute a enqueue_remote_request call, discussed in Section 8.6.8, to store the request message into the message hash table. Then the message will be sent using the send_message_request function, discussed in Section 8.6.22. Notice, that we make this call using the CSSENDTOEXIST status. This will be used by the remote side to realize that we want to verify the existence of the object in question. We will do a delayuntil to suspend this process until the remote side is done.

The remote side will then wake this process up when the message is received from it. At this time we will call the query_remote_request function, discussed in Section 8.6.10. This will get the entry for us from the message table. Then we will do a dequeue_remote_request call to remove our entry from the message hash table. At this time, the response message will be checked to see what the response type, resp_type, is equal to. If the value is either OBJECT_EXIST or OBJECT_EXIST_IN_REMOTE, then the check succeed and we will return a value of TRUE. If the type is OBJECT_EXIST_IN_REMOTE, this means that the address we have of the object is incorrect and should be changed to the address received in the message. You see, if the remote object is not where we thought it was, then the remote side we contacted will send the proper address back to us. If the type is equal to NOT_DELIVERED_MESSAGE, then we got no response back from the remote side and we timed out. Here we return the value of FALSE. We do this, since we assume the worst and that is the remote host went down. And finally, the last type is that it does not exist, so we obviously return FALSE.

Code of Is_object_exist

```
int is_object_exist(char * obj_name, struct sockaddr * sin)
    {
    char message[MSG_LENGTH];
    char buf[100];
    char caller_name[Max_Object_Name_Size];
    HEntry *entry;
    RESP_MESS * resp_message;

    if (net_pro.is_disable())
        {
        printf("The remote object service via Protcol obj
is not available.\n");
        /*
         * we can not post a receive.
         */
        return;
        }
```

```
    /*
     * We will add this host, and its port number into our
     * hostable, we will use this information, to verify
     * the health of our remote object.
     */

    remote_verify.add_active_remote_obj(sin);

    /*
     * NOTE::::
     */
    strcpy(message,"?");

    /*
     * Store the request into a table.
     */

    sprintf(caller_name,"%s#%s",
            (char *)MetaClass.obtain_obj_info($self,
    (enum obj_field)OBJ_NAME),
            construct_thread_id(buf,$thread,SYNC));

    resp_message = (RESP_MESS *)
$self.malloc(sizeof(RESP_MESS));

    $self.enqueue_remote_request(
  resp_message,caller_name,sin);

    $self.send_message_request(obj_name,caller_name,-1,
       CSSENDTOEXIST,
                            INIT_REQ,
                            (char *)message,
       strlen(message),sin);
    delayuntil caller_name;

    entry = (HEntry *)$self.query_remote_request(
   caller_name);

    resp_message = (RESP_MESS *)(entry->ptr);

    $self.dequeue_remote_request(caller_name);
```

```
    /*
     * OBJECT_EXIST imply the real object is physically
     * located at the remote host.
     *
     * OBJECT_EXIST_IN_REMOTE indicates that the object
     * is not physically located at the host your are
     * accessing.  This call will return the actual
     * location (hostname and address) of the object.
     */

    if ((resp_message->resp_type==(int)OBJECT_EXIST) ||
        (resp_message->resp_type ==
 (int)OBJECT_EXIST_IN_REMOTE))
        {
        if (resp_message->resp_type ==
    (int)OBJECT_EXIST_IN_REMOTE)
            {
            char hostname[MAX_HOSTNAME];
            int port_number;

            /*
             * find the port number and the hostname
     * of the remote object.
             */
            sscanf(resp_message,"{ %s %d }",hostname,
  port_number);

            /*
             * Reconstruct the address.
             */
            (void)$self.dest_addr(sin,hostname,
 port_number);
            }
        if (resp_message->resp_mess) $self.
      free((char *)resp_message->resp_mess);
        (void)$self.free((char *)resp_message);

        return(TRUE);
        }
    else if (resp_message->resp_type ==
             (int)NOT_DELIVERED_MESSAGE)
        {
```

```
        printf("No response from remote, TIMEOUT
                 \n");
        if (resp_message->resp_mess)
          $self.free((char *)resp_message->resp_mess);
        (void)$self.free((char *)resp_message);
        return(FALSE);
         }
    else
         {
        if (resp_message->resp_mess)
          $self.free((char *)resp_message->resp_mess);
        (void)$self.free((char *)resp_message);
        return(FALSE);
        }
    }
```

8.6.19 Obtain_remote_obj_id

Description of Obtain_remote_obj_id

The next function, obtain_remote_obj_id, will get for us the object id from a remote source. To do so, we need the address of where the remote object resides. This is passed to us from the calling function.

First we will get the object name and the thread id. Then we will store our request into the message table with a enqueue_remote_request call. Then we will send the message with the send_message_request call, with a message type of CSCONNECT. We will then do a delayuntil and wait for the remote side to finish and wake us up with a response message. We will then do a query_remote_request call to get the entry. Then we will take our entry out of the message hash table with a dequeue_remote_request. The function will then return the message we received from the remote side.

Code of Obtain_remote_obj_id

```
RESP_MESS *  obtain_remote_obj_id(struct sockaddr * sin,
             RESP_MESS * resp_message)
    {
    char message[MSG_LENGTH];
    char buf[100];
    char caller_name[Max_Object_Name_Size];
    HEntry *entry;
```

```
   if (net_pro.is_disable())
       {
       printf("The remote object service via Protcol obj
       is not available.\n");
       /*
        * we can not post a receive.
        */
       return;
       }
   strcpy(message,"?");

   /*
    * Store the request into a table.
    */
   sprintf(caller_name,"%s#%s",
           (char *)MetaClass.obtain_obj_info($self,
   (enum obj_field)OBJ_NAME),
           construct_thread_id(buf,$thread,SYNC));

   $self.enqueue_remote_request(resp_message,
caller_name,sin);

   $self.send_message_request("NONE",caller_name,
      -1,CSCONNECT,
                             INIT_REQ,
                             (char *)message,
      strlen(message),sin);
   delayuntil caller_name;

   entry = (HEntry *)$self.query_remote_request(
      caller_name);

   resp_message = (RESP_MESS *)(entry->ptr);

   $self.dequeue_remote_request(caller_name);

   return(resp_message);
   }
```

8.6.20 Connect_remote

Description of Connect_remote

We then have the connect_remote function, which was mentioned earlier in the obj_init function. It will basically set the shadow object, that shadow attached to the passed in object, remote object field to the remote side.

It will do so by first allocating memory for a message. Then we will call the obtain_remote_obj_id function. This will give us the remote object id for the remote object. If we received an id, then we will get the shadow object for the passed in object. Then we will call the set_remote_obj_id to set the shadow objects remote object field to that returned by the obtain_remote_obj_id function. Then we will set the message memory free and return with a value of TRUE.

Code of Connect_remote

```
int connect_remote(object obj, struct sockaddr * sin)
    {
    RESP_MESS * resp_message;
    object shadow_object;

    resp_message = (RESP_MESS *)$self.malloc(
    sizeof(RESP_MESS));

    resp_message = $self.obtain_remote_obj_id(sin,
 resp_message);

    if (resp_message)
        {
        shadow_object
                = (object)MetaClass.obtain_obj_info(obj,
(enum obj_field)OBJ_SHADOW_OBJ);

        shadow_object.set_remote_obj_id(resp_message->
 remote_obj_id);
        (void)$self.free((char *)resp_message);
        return(TRUE);
        }
    else
        {
        printf("Fatal Error : in connect_remote . \n");
        return(FALSE);
```

```
        }
    }
```

8.6.21 Obj_destroy

Description of Obj_destroy

Next we have the obj_destroy function. It will send a message to the remote side to destroy the remote shadow object stub, basically tear down the connection.

First we will call the obtain_obj_info, to get the object's name. Then we will check the import table by executing the import_table.meldcHash call. This will give us the entry in the table and put it into the o_entry variable. Then we will delete the entry in the import table with the import_table.deleteHash call. We will get the remote object id with the return_remote_obj_id function. Then a message will be sent to the remote side to ask it to destroy the remote shadow object stub. This is done with the send_message_request call with a message type of CSDESTROY. This function could easily have been called UnGetObj, since it is the ungetting of the object received by the GetObj call.

Code of Obj_destroy

```
void obj_destroy(object obj,object shadow_obj)
    {
    HObjEntry *o_entry;
    int remote_obj_id ;

    char  obj_name[256];
    /*
     * Remove object entry from import table.
     */
    strncpy(obj_name,
            (char *)(MetaClass.obtain_obj_info(obj,
    (enum obj_field)OBJ_NAME)),
            256);

    o_entry=(HObjEntry *)import_table.meldcHash(obj_name);
    /*
     * Just in case there is a duplicate entry.
     */
    while ( (o_entry !=  (HObjEntry *) NULL) &&
```

```
            (o_entry->obj_ptr != (struct _object *)obj))
        {
        import_table.deleteHash(obj_name);
        o_entry =  (HObjEntry *)import_table.meldcHash(
obj_name);
        }
    import_table.deleteHash(obj_name);

    remote_obj_id = shadow_obj.return_remote_obj_id();

    /*
     * Send a request to remote, and ask remote to destroy
     * the remote shadow object stub.
     *
     * There is no need to wait for the response.
     */
    $self.send_message_request(obj_name,"NONE",
  remote_obj_id,CSDESTROY,
                         INIT_REQ,
                         (char *)"?",2,
                         (struct sockaddr *)
     shadow_obj.return_sockaddr());
    }
```

8.6.22 Send_message_request

Description of Send_message_request

The send_message_request function has been talked about greatly so far. It has been mentioned so much for the reason that, everytime it is desired to communicate with the remote side in someway, it is necessary to go through this function to get that message sent there. The variables used here are the:

- odest – which is the destination object name
- oname – is the source object name. It is used to send a response back to the sender.
- return_status – the status of the message we sent out.
- type_of_message – the type of message we sent.

To send a message, we must encode the data using the encode_message function in the en_decode file, Section 8.1.3. This function will allocate the message buffer for us. It will also linearize the message for us so that we can send the message out on the network to the remote location. The encode_message will finally send us the size of the buffer allocated for the message. Then the message is ready to be sent to the remote and is done so by using the send_to_remote function, discussed in Section 8.2.4. And finally, we should free up the memory buffer.

Code of Send_message_request

```
int   send_message_request(char *  odest,
                  char *  oname,
                  int     remote_obj_id,
                  int     type_of_message,
                  enum    response_message return_staus,
                  char *  input_buf,
                  int     size,
                  struct sockaddr *sin)
    {
    static int packet_id = 0;
    char * message;
    int size_message;
    int length;

#ifdef TRACE
    printf("Enter send_message_request \n");
#endif

    if (net_pro.is_disable())
        {
        printf("The remote object service via name server
is not available.\n");
        /*
         * we can not post a receive.
         */
        return;
        }
    /*
     * Encoind the data.
     * The message buffer will be allocated by the routine
     * encode_message.  After we send out the message to
```

```
     * the remote side, we should free up the message.
     */

    size_message = en_de_buf.encode_message(&message,
    input_buf,size,
                                            odest,oname,
                                            type_of_message,
                                            return_staus,
    remote_obj_id);
    /*
     * Transfer the meory ownership from en_de_buf to $self.
     */

#ifdef DEBUG
    printf("The message we send out = %s\n",message);
#endif

    if (net_pro.send_to_remote((char *)message,
       size_message,sin))
        {
        /*
         * Unable to register the object with name server.
         * I think this is a fatal error, the meldc program
         * should exit.
         */
        printf("Error Can not send the message to the name
                server. \n");
        return(0);
        }
    en_de_buf.free(message);

    return(1);
    }
```

8.6.23 Receive_message

Description of Receive_message

The receive_message function is just the opposite of the send function above. It will receive a message sent out from a remote location. The function though will always be waiting for a new message to be interrcepted or will be handling a message received already. The messages will be passed from the name_server.

The function will not return until all of the bytes that you requested have arrived at the remote location. It will peek at the data that is present when it first discovers a new message. What it will look at is the first bytes of the message, the size of the message to be received. This is done with a receive_from_remote call, located also in the remote_udp file, with a type of MSG_PEEK. Then it will allocate a buffer of the message size. Then it will receive the entire message into the buffer with another receive_from_remote call. This is true only if after the MSG_PEEK, the message_size does not equal 0. If it does equal 0, then it was unable to decode the incoming message. The message will be dropped and it will be read from the buffer instead. In the case of the datagram method, the buffer will not be large enough, so some data will be discarded. After doing so, we will make a asynchronous call to the function process_received_message to allow another message to be received immediately. When it is processed, we will free up the message.

Code of Receive_message

```
int receive_message()
    {
    int bytes_receive;
    int size_byte;

    if (net_pro.is_disable())
        {
        printf("The remote object service via name server is
                not available.\n");
        /*
         * we can not post a receive.
         */
        return;
        }
    /*
     * We will always posting a receive which waits the data
     * from the name server.
     */

    struct sockaddr sin; /* source address. */
    char buf[MAX_DIGIT+1];
    char * message;
    int  message_size;
```

```
    int      sin_len;

    /*
     * receive_from_remote will not return, until all the
     * bytes that you requested are arrived.
     */

    sin_len = sizeof(struct sockaddr);

    /*
     * Peek the data that are present in the socket.
     * The first MAX_DIGIT bytes should contain the size of
     * the entire buffer, which we about to received.
     */

    buf[MAX_DIGIT] = (char *)NULL;

    if ((size_byte= net_pro.receive_from_remote((char *)buf,
        MAX_DIGIT,&sin,&sin_len,MSG_PEEK)) == -1)
        {
        /*
         * Unable to receive the message from name server.
         */
        if (_frame->$.errno == READ_ABORT)
            return(0);
        sys_obj.exit(-1);
        }
    message_size = atoi(buf);

    if (message_size != 0)
        {
        /*
         * We need to allocate the message buffer.
         */
        message = 0;
        message = (char *)$self.malloc(message_size);

        if ((size_byte = net_pro.receive_from_remote(
                                            (char *)message,
              message_size,
&sin,&sin_len,0))
== -1)
```

```
            {
            /*
             * Unable to receive the message from the
     * name server.
             */
            if (_frame->$.errno == READ_ABORT)
                return(0);
            sys_obj.exit(-1);
            }

#ifdef DEBUG
        printf("We received the message %s\n",message);
#endif
        }
    else
        {
        /*
         * Unable to decode the ncoming message.
         * We will drop the message. Read the data from the
         * buffer and discard the buffer. In the case of
         * datagram, if the buffer we posted is not large
         * enough, the rest of the data will be discarded.
         */

        printf("Unable to decode the incoming buffer. \n");

        if ((size_byte = net_pro.receive_from_remote(
      (char *)buf,
                   MAX_DIGIT,
      &sin,&sin_len,
      0)) == -1)
            {
            /*
             * Unable to receive the message from the name
     * server.
             */
            if (_frame->$.errno == READ_ABORT)
                return(0);
            sys_obj.exit(-1);
            }
        }
    /*
```

```
     * Process the incoming buffer & free up the "message",
     * when it is done.
     */

    send process_received_message(message,sin,
  sin_len) to $vself;

    /*
     * Post another receive ...
     */

    send receive_message() to $vself;
    }
```

8.6.24 Process_received_message

Description of Process_received_message

The next function is the process_received_message function. It's main job is to identify an incoming message that was just received and issue the proper command. This is all done by using the message type received with the message. The type is in the first three bits of the message and is used by a switch statement to decide the command to be issued. We first must decode the message, since it was linearized, encoded, before being sent from the remote side. The decoding is done using the function decode_message, discussed in Section 8.1.4. Then we can do a check of the type with the switch statement just mentioned. Below are the types to be received and the commands to issue for each.

CS_HELLO_ARE_YOU_THERE

The first type is the CS_HELLO_ARE_YOU_THERE. This type is used by the remote side to check if this side of the connection is still alive. The command to issue in this case is to simply return a message to the remote machine that CS_YES_I_AM_A_LIVE. This will be the message type of the return message and is sent using the send_message_request function, discussed in Section 8.6.22.

CS_YES_I_AM_A_LIVE

The next message type is the CS_YES_I_AM_A_LIVE type. This is returned when you send a message to a remote machine to check if it is still up and running, as discussed directly above. The command to issue in this case is to simply make a call to the receive_verification function, discussed in Section 8.8.7. That function will then process the message we received.

CSSENDTOEXIST

Next we have the CSSENDTOEXIST message type. This is used to verify that the object in question does in fact exist. We check this by using the is_object_exported function, discussed in Section 8.6.14. The export table will be searched and the entry is returned. If the entry is not NULL, then it does exist in the table, so has been exported. In this case, a return status of OBJECT_EXIST will be returned.

If the entry is NULL, then it was not exported. So we check if the object was imported from another machine. This is done using the is_object_imported function, discussed in Section 8.6.17. If the entry does not return NULL, then it means that the object does exist but on a remote source. In this case, we must return the status of OBJECT_EXIST_IN_REMOTE.

If both the import and export tables have entries of NULL, then the object does not exist. In this case, we will return a status of OBJECT_NOT_EXIST.

All of these return statuses will be put into a message that will be returned using the send_message_request function.

CSCONNECT

The next message type is the CSCONNECT type. This one is used to set up a connection between the local and remote side. This is done when a user imports a remote object. We must then create a shadow object which will act as a bridge between the local and remote object to allow communication between them. We must therefore call the Create function, discussed in Section 4.1.9, to create the shadow object needed. We then save the object into the database with the save_generic_obj function. The return_status value is set to OBJECT_EXIST and a return message is sent back.

CSDESTROY

The reverse of the CSCONNECT is the CSDESTROY message type. Here we want to destroy the shadow object and severe the connection between the remote and local objects. A simple call to the delete_generic_obj, discussed in Section 8.6.13, will do this for us.

CSSENDTORETURN

Next we have the CSSENDTORETURN message type. This is used whenever we have processed a message and we want to return the return_status to the caller. The message was received from a remote side and now we must try to forward the result to the proper object. We must first make a query to the message table to get the entry under the destination. This is done by making a call to the query_remote_request, with the destination as the parameter passed. We then update the imformation of the request with the new return_status. Then we identify the remote shadow object and send back the response to it with a 'respond data_packet.odest' call.

CSSENDTOREQUEST

The final message type is the CSSENDTOREQUEST type. This means that we received a request from the remote side. This function will be done with dynamically binding the method. We first get an entry in the hash table for exporting a new generic remote object. We will make a call to the find_generic_obj function. This will give us a generic objecton the remote side. We then call the remote_obj_dispatch function to allocate the return message for us. Then we can send back a return message with the send_message_request function. And finally we free up the memory of the return message.

Code of Process_received_message

```
void process_received_message(char *buf,
                              struct sockaddr sin,
                              int sin_len)
    {
    HObjEntry *o_entry = NULL;
    enum response_message return_status;
    int    return_value;
    HEntry *entry;
    RESP_MESS * resp_message;
    struct client_sendto_req_type data_packet;

    /*
     * Decode the message, and store them into the
     * variable data_packet.  When we finish the
     * data_packet, we need to free the memory location.
     */
```

```
#ifdef TRACE
    printf("Enter decoding routine. \n");
#endif

    en_de_buf.decode_message(buf,&data_packet);

    $self.free(buf);

#ifdef TRACE
    printf("Enter process_receive_message routine. \n");
#endif
    /*
     * we need to find out the type of this message.
     * The first three bits of the data_packet.type,
     * represent the msg type.  It could be a str
     * message ot a sym message.
     */

    switch (data_packet.type & 0xfffffff8)
        {
        case CS_HELLO_ARE_YOU_THERE :
            {
            /*
             * Send back a response indicates that I am
     * live and well.
             */
          if ((return_value=$self.send_message_request(
                          "NONE",
                          "NONE",
                          -1,
                          CS_YES_I_AM_A_LIVE,
                          INIT_REQ,
                          (char *) data_packet.msg,
                          (char *) data_packet.msg_size,
                          &sin))
                          == NULL)
                {
                /*
                 * Unable to send to message to the
                 * remote object.
                 *
                 * We need to define a set of exception
```

```
                 * handler.
                 */
                printf("Unable to send to message to the
remote object. \n");
                }
            break;
            }

        case  CS_YES_I_AM_A_LIVE :
            {
            /*
             * We receive the process verification
     * message.
             */
            remote_verify.receive_verification(&sin,
      data_packet.msg);
            break;
            }

        case  CSSENDTOEXIST :
            {
            /*
             * check and make sure that the object has
     * been exported.
             */
            char msg[MSG_LENGTH];
            char caller_name[Max_Object_Name_Size];
            int msg_length = 1;

            o_entry=(HObjEntry *)$self.
                 is_object_exported(data_packet.odest);

                {
                object shadow_obj;
                struct sockaddr *addr;

                /*
                 * We did not export the object.
 *
                 * Did we import this object from other
 * machine ?
                 */
```

```
                o_entry= (HObjEntry *)$self.
                  is_object_imported(data_packet.odest);

                if (o_entry == NULL)
                    {
                    return_status = OBJECT_NOT_EXIST;
                    }
                else
                    {
                    struct hostent * host;

                    /*
                     * The object exist at the other
     * machine.  Find the hostname and
     * it portnumber and send it back.
                     */
                    shadow_obj = (object) MetaClass.
                        obtain_obj_info(
                        (object)(o_entry->obj_ptr),
                        (enum obj_field)OBJ_SHADOW_OBJ);
                    addr = (struct sockaddr *)shadow_obj.
         return_sockaddr();

                    /*
                     * Reconstruct the host entry.
                     */

                    host = gethostbyaddr(
                             ((struct sockaddr_in *)
                             addr)->sin_addr,
                             sizeof(struct in_addr),
                             AF_INET);
                    sprintf(msg,"{ %s %d }",
                            host->h_name,
                            ((struct sockaddr_in *)
                            addr)->sin_port);

                    return_status=OBJECT_EXIST_IN_REMOTE;
                    msg_length = strlen(msg);
                    }
                }
            else
```

```
                {
                return_status = OBJECT_EXIST;
                }

            char buf[100];

            sprintf(caller_name,"%s#%s",
                MetaClass.obtain_obj_info($self,OBJ_NAME),
                construct_thread_id(buf,$thread,SYNC));

            if ((return_value=$self.send_message_request(
                                data_packet.oname,
                                caller_name,
                                data_packet.remote_obj_id,
                                CSSENDTORETURN,
                                return_status,
                                (char *) msg,
                                msg_length,&sin))
                                == NULL)
                {
                /*
                 * Unable to send to message to the name
* server.
                 *
                 * We need to define a set of exception
* handler.
                 */
                printf("ERROR in remote obj handler.\n");
                }

            break;
            }

        case CSCONNECT :
            {
            /*
             * check and make sure that the object has
    * been exported.
             */
            char msg[MSG_LENGTH];
                char caller_name[Max_Object_Name_Size];
            char buf[100];
```

```
            object generic_remote;

            /*
             * Important :
             * Every time a remote user import a remote
     * object, and shadow object will be created,
     * the shadow object will act like a bridge,
     * which is responsible for all the
     * communication between local and the remote
             * object.
             */

            generic_remote = (object)shadow_object_class.
                         Create($self,&sin,(char *)NULL);
            /*
             * add the new generic obj into the database.
             */
            data_packet.remote_obj_id = $self.
                save_generic_obj((object)generic_remote);

#ifdef DEBUG
            printf("data_packet.remote_obj_id %d \n",
                   data_packet.remote_obj_id);
#endif
            generic_remote.set_remote_obj_id(
                              data_packet.remote_obj_id);

            sprintf(caller_name,"%s#%s",
                MetaClass.obtain_obj_info($self,OBJ_NAME),
                construct_thread_id(buf,$thread,SYNC));

            return_status = OBJECT_EXIST;

            if ((return_value=$self.send_message_request(
                               data_packet.oname,
                               caller_name,
                               data_packet.remote_obj_id,
                               CSSENDTORETURN,
                               return_status,
                               (char *) msg,
                               1,&sin))
                               == NULL)
```

```
                {
                /*
                 * Unable to send to message to the name
 * server.
                 *
                 * We need to define a set of exception
                 * handler.
                 */
                printf("ERROR in remote obj handler.\n");
                }
            break;
            }

        case  CSDESTROY :
            {
            object o;
            o = $self.delete_generic_obj(
      data_packet.remote_obj_id);
            shadow_object_class.Destroy(o);
            remote_verify.destroy_active_remote_obj(&sin);
            break;
            }

        case CSSENDTORETURN :
            {

            /*
             * We received the result from the remote
             * machine.  We will attempt to forward this
             * result to the proper object.
             */

            /*
             * We have receive the return value.
             */

            entry = $self.query_remote_request(
     data_packet.odest);
            /*
             * update the information of the request.
             */
            if (entry == NULL)
```

```
                {
                printf("Unable to deliver the return
message, Unknown object. \n");
                return;
                }
            resp_message = (RESP_MESS *)(entry->ptr);

            /*
             * resp_type ---> the result status from
     *       remote object.
             *
             * The valid value can be
             * OBJECT_EXIST
             * OBJECT_NOT_EXIST
             * DELIVERED_MESSAGE
             * NOT_DELIVERED_MESSAGE
             */

            resp_message->resp_type = (int)(
data_packet.response);

#ifdef DEBUG
            printf("data_packet.msg_size = %d \n",
data_packet.msg_size);
#endif
            resp_message->resp_mess = (char *)$self.
                    malloc(data_packet.msg_size);
            /*
             * Indentify which remote shadow object
             * that send back the response.
             */
            resp_message->remote_obj_id =
    data_packet.remote_obj_id;
            memcpy(resp_message->
            resp_mess,data_packet.msg,
                   data_packet.msg_size);

            resp_message->dest_addr = sin;
            resp_message->dest_len  = sin_len;

            respond data_packet.odest;
```

```
            break ;
            }

      /*
       * We received a request from remote.
       */
      case  CSSENDTOREQUEST :
            {
            char * return_msg = NULL;
            int  return_length = 0;
            char caller_name[Max_Object_Name_Size];
            object generic_remote;
            char threadid_string[Max_Object_Name_Size];
            char obj_name[Max_Object_Name_Size];

            if (sscanf(data_packet.oname,"%[^#]#%s",
                    obj_name, threadid_string) != 2)
                {
                printf("MeldC Error: Unable to decode
the sender information.\n");
                printf("MeldC Error : sender_info =
%s \n",data_packet.oname);
                }

            o_entry=(HObjEntry *)export_table.
meldcHash(data_packet.odest);

            if (o_entry != NULL)
               {
               /*
                * Dynamic binding the method.
                */

               /*
                * creating the generic_remote object. Use
        * by $sender
                */

               if (data_packet.remote_obj_id == -1)
                   {
                   printf("MeldC Fatal Error : Unknow
   shadow object. \n");
```

```
                   return(0);
                       }
               else
                   {
                   generic_remote = (object) $self.
     find_generic_obj(
     data_packet.remote_obj_id);
                   }

               if (generic_remote == (object)NULL)
                    {
                    printf("Unable to execute the remote
    request. \n");
                    }
               else
                    {
                    /*
                     * the funct remote_obj_dispatch will
                     * allocate the return message.
                     */

                    return_msg = generic_remote.
remote_obj_dispatch(
                        (object)(o_entry->obj_ptr),
                        obj_name,
                        threadid_string,
                        (char *)(data_packet.msg),
                        (enum msg_types)
        (data_packet.type & 0x7),
                        &return_length);
                    }
               return_status = DELIVERED_MESSAGE;
               }
           else
               {
               return_status = NOT_DELIVERED_MESSAGE;
               }
           /*
            * Send back the return value.
            */

           char buf [100];
```

```
           sprintf(caller_name,"%s#%s",
                   MetaClass.obtain_obj_info($self,
     OBJ_NAME),
                   construct_thread_id(buf,$thread,
       SYNC));
#ifdef TRACE
           printf("return_msg before send  = %x \n",
   return_msg);
#endif
           if ((return_value =
                      $self.send_message_request(
                             data_packet.oname,
                             caller_name,
                             data_packet.remote_obj_id,
                             CSSENDTORETURN,
                             return_status,
                             (char *) return_msg,
                             return_length,&sin))
                             == NULL)
                {
                /*
                 * Unable to send to message to the
 * remote object.
                 *
                 * We need to define a set of exception
 * handler.
                 */
                printf("ERROR in remote object handler.
\n");
                }
            /*
             * If return_msg is not equal to NULL, this
     * implied that generic_remote.remote_obj_
     * dispatch has been called, and we should
             * free up the memory of return_msg.  Since
     * the object who own the return_msg is equal
             * to generic_remote, we should ask
     * generic_remote to free up the memory.
             */

#ifdef DEBUG
```

```
            printf("return_msg after send  = %x \n",
    return_msg);
#endif
            if (return_msg)
                generic_remote.free(return_msg);
            break;
            }
        default :
            {
            printf("ERROR Invalid message has been
    received. \n");
            }
        }
    /*
     * free up the memory location that is allocated
     * to the "data_packet".
     */

    /*
     * Free up the buffer.
     */

    en_de_buf.free_message(data_packet);
#ifdef TRACE
    printf("exit process_receive_message routine.\n");
#endif
    }
```

8.6.25 No_response_from_process

Description of No_response_from_process

The last function in this file is the no_response_from_process function. What happened here, is that we sent some message to remote source and never got a response back from the remote. We must clean up all of the processes that are waiting on this remote process. So we will go through the list of all of the messages in the message table. Any that match the remote object, we will extract the caller information and respond to that waiting process with a NOT_DELIVERED_MESSAGE to wake it up. This will be done with all of the message in the table. You should note that we are assumming that the remote process no longer exists.

Code of No_response_from_process

```
void no_response_from_process(struct sockaddr * sin)
    {
    RESP_MESS * resp_message;
    HEntry *entry = NULL;
    int key = 0;

#ifdef DEBUG
    printf ("Enter the no_response_from_process. \n");
#endif

    while ((entry = message_table.meldcNextEntry(
                         entry,&key)) != NULL)
        {
        resp_message = (RESP_MESS *)(entry->ptr);

        if (memcmp(&(resp_message->dest_addr),
                   sin, sizeof(struct sockaddr)) == 0)
            {
            /*
             * resp_type ---> the result status from
     *        remote object.
             *
             *     The valid value can be
             * OBJECT_EXIST
             * OBJECT_NOT_EXIST
             * DELIVERED_MESSAGE
             * NOT_DELIVERED_MESSAGE
             */

#ifdef DEBUG
            printf("NO RESPONSE FROM THE REMOTE PROCESS.
    \n");
#endif
            resp_message->resp_type =
   (int)(NOT_DELIVERED_MESSAGE);

            resp_message->resp_mess = NULL;

            respond entry->u.name;
            }
```

```
        }
    }


end class  ShadowObjProtocol
```

8.7 Generic_remote_interface, ShadowObjSenderProtocol Class

8.7.1 Description of Generic_remote_interface, ShadowObjSenderProtocol Class

This class contains the $sender specific implementation. It is also contained in the generic_remote_implementation file, but differs in the class. Without $sender, the association of the shadow object is one to one. That is, the shadow object on the local side as a counterpart on the remote side. All of the messages will be forwarded from the local shadow object to the remote shadow object. Because of the one to one correspondence, the shadow objects are created at the beginning.

On the other hand, implementing $sender is quite different. Each remote shadow object represents a unique sender on the remote machine. The relationship between local and remote counterparts are usually one to many. Unlike the other implementation, this one has the problem of garbage collection. The system can never be sure that the remote counterparts are no longer needed. This is because the object can respond to it's sender anytime it feels like. In this implementation, there are usually alot of the sender objects hanging around in the system.

Headers of Generic_remote_interface, ShadowObjSenderProtocol Class

```
class ShadowObjSenderProtocol merges ShadowObjProtocol ::=

Hash caller_shadow_obj  = Hash.Create();

methods:
```

8.7.2 Connect_remote

Description of Connect_remote

This connect_remote function is unlike the other we saw in Section 8.6.20. Here we implement this function for the $sender. We take the object, obj, and the address, sin, as parameters.

It will use the obtain_obj_info function to get the object information needed. That needed information is the shadow object of the object passed to us. Here we will just call the function, export_caller_name. This will set the caller name equal to one. This is done because the $sender implementation requires the interface to export the caller automatically. This will allow a remote object to reference a local caller.

Code of Connect_remote

```
int connect_remote(object obj, struct sockaddr * sin)
    {
    object shadow_object;

    shadow_object = (object)MetaClass.obtain_obj_info(obj,
                          (enum obj_field)OBJ_SHADOW_OBJ);
    if (shadow_object == (object)NULL)
        {
        printf("MeldC Error : connect_remote() Invalid
shadow object. ");
        }
    else
        {
        /*
         * The $sender implementation require the interface
         * to export the caller automatically.  So a remote
         * object is able to reference a local caller.
         */
        shadow_object.export_caller_name();
        }
    }
```

8.7.3 Save_generic_obj

Description of Save_generic_obj

The save_generic_obj function is also for the $sender implementation. Here we will receive the object, obj, and the key for the hash table, as parameters.

We will first insert the key into the generic hash table using the generic_obj_table.insertHash call. The call will return the entry to us. If the entry was not NULL, then we will increment the shadow_obj_id by one. Then we will return the shadow objects id minus one.

Code of Save_generic_obj

```
int save_generic_obj(object obj,char * key)
    {
    HEntry * entry;
    <
#ifdef DEBUG
    printf("Key = %s \n",key);
#endif

    entry = generic_obj_table.insertHash(key);

    if (entry != NULL)
        {
        entry->ptr = (int *)obj;

        shadow_obj_id ++;

#ifdef DEBUG
        printf("In save_generic_object shadow_obj_id =
%d \n",shadow_obj_id);
#endif
        return(shadow_obj_id-1);
        }
    return(-1);
    >
    }
```

8.7.4 Find_generic_obj

Description of Find_generic_obj

This $sender function, find_generic_obj will look in the hash table with the generic_obj_table.meldcHash call using the passed key value. We will return the entry pointer back to the calling function.

Code of Find_generic_obj

```
object find_generic_obj(char * key)
    {
    HEntry * entry;
    <
#ifdef TRACE
    printf("Enter find_Generic_obj. \n");
#endif
    entry = generic_obj_table.meldcHash(key);

    if (entry)
        return((object) entry->ptr);
    else
        return((object) NULL);
    >
    }
```

8.7.5 Delete_generic_obj

Description of Delete_generic_obj

This $sender function will take the key value passed in and search for it in the hash table. It will first make sure that the entry exists with the generic_obj_table.meldcHash call. This will return the entry back and save it into the entry variable. We will then save the entry pointer into the o variable. We can then delete the hash entry with the generic_obj_table.deleteHash call. When done, we will return the saved entry pointer in the o variable.

Code of Delete_generic_obj

```
object delete_generic_obj(char * key)
    {
    HEntry * entry;
    object o = (object)NULL;
```

```
<
entry = generic_obj_table.meldcHash(key);

if (entry == (HEntry *)NULL)
    {
    printf("MeldC Error : Invalid shadow object id.");
    return;
    }
if (entry->ptr)
    {
    o = (object)entry->ptr;
    }
generic_obj_table.deleteHash(key);
>
return(o);
}
```

8.7.6 Process_received_message

Description of Process_received_message

This is now the implementation of the process_received_message function for the $sender. Like the parent function, we will receive a message and then process the message based on the message type. It will first take the message and decode it from linearized form. It is in this form since we just received it from the network. We can decode it using the decode_message function, discussed in Section 8.1.4. After decoding the message, we will then use a switch statement on the message type. Here are all of the possible types.

CS_HELLO_ARE_YOU_THERE

The CS_HELLO_ARE_YOU_THERE request is for the verification, also discussed above. So in this case a response must be returned stating that it is alive and well, CS_YES_I_AM_A_LIVE.

CS_YES_I_AM_A_LIVE

Here we received a verification message stating that the remote location is still alive and functioning. Must inform the verification routine that we received this message so that it can update it's verifcation list.

CSSENDTOEXIST

Check and make sure that the object has been exported. If it was not exported, then we must check if the object was imported from another machine. If it was not, then it does not exist an error will be returned to the sender. Otherwise, we must find the host name and it's portnumber and send it back to the remote location. See the parent version in Section 8.6.24 for more details.

CSDESTROY

The object is to be destroyed, so call the function delete_generic_obj, in Section 8.7.5. This function will basically remove shadow objects attached to this object and remove all connections associated with the shadow objects. If the object is a remote object, a CSDESTROY must also be sent to the shadow object on the other machine to clean it up also. It will then proceed to remove the object from the generic hash table.

CSSENDTORETURN

The result was received from the remote machine so now forward the result to the object. See parent function in Section 8.6.24 for more details.

CSSENDTOREQUEST

This states that the remote side made a request and we must process it. To do so we must reset it's socket address. Then we will create the new generic object with the Create($self) command and then save_generic_obj function, also in this file. We then set the remote object name, the export caller name, the remote object id and the socket address. The remote_obj_dispatch will then dispatch the incoming message to the local object. This object will then process it and send back the return value. The process will return it here where the final value will then be sent back to the remote object.

Code of Process_received_message

```
void process_received_message(char *buf,
                              struct sockaddr sin,
                              int sin_len)
    {
    HObjEntry *o_entry = NULL;
    enum response_message return_status;
    int    return_value;
```

```
    HEntry *entry;
    RESP_MESS * resp_message;
    struct client_sendto_req_type data_packet;

    /*
     * Decode the message, and store them into the variable
     * data_packet.  When we finish the data_packet, we
     * need to free the memory location.
     */

#ifdef TRACE
    printf("Enter decoding routine. \n");
#endif

    en_de_buf.decode_message(buf,&data_packet);

    $self.free(buf);

#ifdef TRACE
    printf("Enter process_receive_message routine. \n");
#endif

    /*
     * we need to find out the type of this message. The
     * 1st 3 bits of the data_packet.type, represent the
     * msg type. It could be a str msg or a sym message.
     */

    /*
    switch (( (data_packet.type >> 3) << 3))
    */
    switch (data_packet.type & 0xfffffff8)
        {
        case CS_HELLO_ARE_YOU_THERE :
            {
            /*
             * Send back a response indicates that I am
             * live and well.
             */
            if ((return_value =
                        $self.send_message_request(
                              "NONE",
```

```
                            "NONE",
                            -1,
                            CS_YES_I_AM_A_LIVE,
                            INIT_REQ,
                            (char *) data_packet.msg,
                            data_packet.msg_size,
                            &sin))
                            == NULL)
                {

                /*
                 * Unable to send to message to the
 * remote object.
                 *
                 * We need to define a set of exception
 * handler.
                 */
                printf(" Unable to send to message to
 the remote object. \n");
                }
            break;
            }

        case  CS_YES_I_AM_A_LIVE :
            {
            /*
             * We receive the process verification msg
             */
            remote_verify.receive_verification(
&sin,data_packet.msg);
            break;
            }

        case  CSSENDTOEXIST :
            {
            /*
             * check and make sure that the object has
     * been exported.
             */
            char msg[MSG_LENGTH];
            char caller_name[Max_Object_Name_Size];
            int msg_length = 1;
```

```
          o_entry=(HObjEntry *)
                   $self.is_object_exported(
                   data_packet.odest);

          if (o_entry == NULL)
              {
              object shadow_obj;
              struct sockaddr *addr;

              /*
               * We did not export the object.
               *
               * Did we import this object from other
 * machine ?
               */
              o_entry= (HObjEntry *)
                       $self.is_object_imported(
                       data_packet.odest);

              if (o_entry == NULL)
                  {
                  return_status = OBJECT_NOT_EXIST;
                  }
              else
                  {
                  struct hostent * host;

                  /*
                   * The object exist at the other
     * machine.  Find the hostname and
     * it portnumber and send it back.
                   */

                  shadow_obj = (object)
                     MetaClass.obtain_obj_info(
                      (object)(o_entry->obj_ptr),
                      (enum obj_field)OBJ_SHADOW_OBJ);
                  addr = (struct sockaddr *)
   shadow_obj.return_sockaddr();

                  /*
```

```
                 * Reconstruct the host entry.
                 */

                host = gethostbyaddr(
                       ((struct sockaddr_in *)
                       addr)->sin_addr,
                       sizeof(struct in_addr),
                       AF_INET);
                sprintf(msg,"{ %s %d }",
                        host->h_name,
                        ((struct sockaddr_in *)
addr)->sin_port);

                return_status =
    OBJECT_EXIST_IN_REMOTE;
                msg_length = strlen(msg);
                }
            }
        else
            {
            return_status = OBJECT_EXIST;
            }
        char buf[100];

        sprintf(caller_name,"%s#%s", MetaClass.
obtain_obj_info($self,OBJ_NAME),
                construct_thread_id(buf,
$thread,SYNC));

        if ((return_value =
  $self.send_message_request(
                       data_packet.oname,
                       caller_name,
                       data_packet.remote_obj_id,
                       CSSENDTORETURN,
                       return_status,
                       (char *) msg,
                       msg_length,&sin))
                       == NULL)
            {
            /*
             * Unable to send to message to the
```

```
* name server.
                *
                * We need to define a set of exception
* handler.
                */
               printf("ERROR in remote obj handler\n");
               }
           break;
           }

       case  CSDESTROY :
           {
           object o;
           o = $self.delete_generic_obj(
    data_packet.odest);
           if ((struct _object *)o)
               shadow_object_class.Destroy(o);
           remote_verify.destroy_active_remote_obj(
 &sin);
           break;
           }

       case CSSENDTORETURN :
           {

           /*
            * We received the result from the remote
    * machine.  We will attempt to forward
    * this result to the proper object.
            */

           /*
            * We have receive the return value.
            */

           entry = $self.query_remote_request(
   data_packet.odest);
           /*
            * update the information of the request.
            */

           if (entry == NULL)
```

```
                {
                printf("Unable to deliver the return
message, Unknown object. \n");
                return;
                }
            resp_message = (RESP_MESS *)(entry->ptr);

            /*
             * resp_type ---> the result status from
     *        remote object.
             *
             * The valid value can be
             * OBJECT_EXIST
             * OBJECT_NOT_EXIST
             * DELIVERED_MESSAGE
             * NOT_DELIVERED_MESSAGE
             */
            resp_message->resp_type =
 (int)(data_packet.response);

#ifdef DEBUG
            printf("data_packet.msg_size = %d \n",
   data_packet.msg_size);
#endif

            resp_message->resp_mess = (char *)$self.
malloc(data_packet.msg_size);
            /*
             * Indentify which remote shadow object
     * that send back the response.
             */
            resp_message->remote_obj_id =
   data_packet.remote_obj_id;
            memcpy(resp_message->
   resp_mess,data_packet.msg,
                           data_packet.msg_size);
            resp_message->dest_addr = sin;
            resp_message->dest_len  = sin_len;

            respond data_packet.odest;

            break ;
```

```
            }

     /*
      * We received a request from remote.
      */
     case  CSSENDTOREQUEST :
            {
            char * return_msg = NULL;
            int  return_length = 0;
            char caller_name[Max_Object_Name_Size];
            object generic_remote;
            char threadid_string[Max_Object_Name_Size];
            char obj_name[Max_Object_Name_Size];
            HObjEntry * shadow_entry;

            if (sscanf(data_packet.oname,"%[^#]#%s",
                     obj_name, threadid_string) != 2)
                {
                printf("MeldC Error : Unable to decode
the sender information.\n");
                printf("MeldC Error : data_packet.oname
= %s \n", data_packet.oname);
                }

            o_entry=(HObjEntry *)export_table.
    meldcHash(data_packet.odest);

            if (o_entry != NULL)
                {
                struct sockaddr * sender_sin;

                /*
                 * For this implementation ($sender)
                 * We are using the caller's name as
 * the handle for the generic object.
                 */
                generic_remote = (object) $self.
  find_generic_obj(obj_name);

                if (generic_remote != (object)NULL)
                    {
                    /*
```

```
                   * reset its socket address.
                   */
                  generic_remote.set_sockaddr(&sin);
                  }

              if (generic_remote == (object)NULL)
                  {
                  /*
                   * we need to create the new
    * generic_object.
                   */
                generic_remote =
    (object)shadow_object_class.Create
                   ($self,&sin,(char *)obj_name);

                $self.save_generic_obj(generic_remote,
obj_name);

                generic_remote.export_caller_name();

                generic_remote.set_remote_obj_id(0);
                }

             if (generic_remote == (object)NULL)
                  {
                  printf("MeldC Error : process_
  received_message () \n");
                  printf("            : Unable to
  execute the remote request.
                         \n");
                  }
             else
                  {
                  /*
                   * the function remote_obj_dispatch
                   * will allocate the return message.
                   */
                  return_msg = generic_remote.
       remote_obj_dispatch(
                           (object)(o_entry->obj_ptr),
                           obj_name,
                           threadid_string,
```

```
                             (char *)(data_packet.msg),
                             (enum msg_types)
       (data_packet.type & 0x7),
                             &return_length);
                    }

               return_status = DELIVERED_MESSAGE;
#ifdef TRACE
               printf("Deliver message \n");
#endif
               }
           else
               {
               return_status = NOT_DELIVERED_MESSAGE;
#ifdef TRACE
               printf("Not Deliver message \n");
#endif
               }

           /*
            * Send back the return value.
            */
           char buf [100];

           sprintf(caller_name,"%s#%s",
                   MetaClass.obtain_obj_info($self,
     OBJ_NAME),
                   construct_thread_id(buf,$thread,
       SYNC));

#ifdef TRACE
           printf("return_msg before send  = %x \n",
   return_msg);
#endif
           if ((return_value =
                    $self.send_message_request(
                          data_packet.oname,
                          caller_name,
                          data_packet.remote_obj_id,
                          CSSENDTORETURN,
                          return_status,
                          (char *) return_msg,
```

```
                        return_length,&sin))
                        == NULL)
                {
                /*
                 * Unable to send to message to the
         * remote object.
                 *
                 * We need to define a set of exception
         * handler.
                 */
                printf("ERROR in remote obj handler\n");
                }
            /*
             * If return_msg is not equal to NULL, this
     * implied that generic_remote.remote_obj_
             * dispatch has been called, and we should
     * free up the memory of return_msg.  Since
     * the object who own the return_msg is = to
             * generic_remote, we should ask generic_
     * remote to free up the memory.
             */

#ifdef DEBUG
            printf("return_msg after send  = %x \n",
    return_msg);
#endif
            if (return_msg)
                generic_remote.free(return_msg);
            break;
            }
        default :
            {
            printf("ERROR Invalid message has been
    received. \n");
            }
        }
    /*
     * free up the memory location that is allocated
     * to the "data_packet".
     */

    /*
```

```
     * Free up the buffer.
     */

    en_de_buf.free_message(data_packet);
#ifdef TRACE
    printf("exit process_receive_message routine\n");
#endif
    }
```

8.7.7 No_response_from_process

Description of No_response_from_process

The final function, no_response_from_process, must deal with a nonresponding host. Not only does the remote object information need to be cleaned up here, but also the shadow object associated with it. The function will look up the shadow objects that are connected to the remote host and remove them.

This function is also different from it's parent class . Once we do not receive any response from the remote process, we will consider the remote process has terminated. Not only will we wake up those threads that are waiting on the response from the remote process, we will also clean up the remote shadow objects as well.

Code of No_response_from_process

```
void no_response_from_process(struct sockaddr * sin)
    {
    int key = 0;
    HObjEntry *o_entry = NULL;
    object obj;

    $self.ShadowObjProtocol::no_response_from_process(sin);

   /*
    * We also need to clean up the sahdow objects that are
    * pointed to non-responding host.
    */

   /*
    * If remote is not responding, we will also destroy the
    * shadow object that is associated with the remote.
```

```
    */

   /*
    * Very IMPORTANT ::
    *
    * This pieces of code is extremly inefficent,  Later
    * when we have time we need to rewrite this piece of
    * code.
    */

   while ((o_entry = (HObjEntry *)generic_obj_table.
                     meldcNextEntry(
                     (HEntry *)o_entry, &key)) != NULL)
       {
       struct sockaddr * sin_dest;

       sin_dest = (struct sockaddr *)
    ((object)(o_entry->obj_ptr)).return_sockaddr();

       if (memcmp(sin,sin_dest,sizeof(struct sockaddr))==0)
           {
           object o = (object)NULL;

           o = (object)o_entry->obj_ptr;

           $self.delete_generic_obj(((object)
                   (o_entry->obj_ptr)).
   return_remote_obj_name());

           shadow_object_class.Destroy(o);
           o_entry = NULL;
           }
       }
    }


end class  ShadowObjSenderProtocol

end feature ShadowObjInterface
```

8.8 Verify.m

8.8.1 Description of Verify.m

The verify.m file will take care of verifying that all of the remote objects are still alive. It does so by keeping track of all of the remote objects received. Then it will periodically go down the list and make sure that all of the remote objects are still there and working. This is done by sending a message to each. Those that respond are considered still fine. Those that do not respond though may have threads waiting for those remote objects to respond to them. Since this may be the case, we will wake up the threads that are waiting for the nonresponding processes and notify them of the problem. It should be noted that this verification scheme is a very simple mechanism. In many case, this mechanism is not sufficent. Part of the problem here is the network protocol. We do not take this into account and are not very concerned with the network. The local area network is usually very reliable, but may be the cause of a no response some of the time. Any programmer wanting a robust network protocol can implement their own by replacing the network_udp and verify objects.

Headers of Verify.m & init & set_protocol_obj functions

```
feature Verify

interface:

    imports Unix[sys_obj]

    imports HashTable[Hash]

    imports MemoryAllocation[Memory]

    exports RemoteVerification

implementation:

class RemoteVerification merges MemoryAllocation[Memory]::=
struct ObjStatus * head;
int    total_obj = 0;
object protocol_obj;
int   time_stamp = 0;
```

```
methods:

void init()
    {
    head = (struct ObjStatus *) NULL;
    }

/*
 * We will use the protocol object to verify whether
 * the other remote obj is still alive.
 */

void set_protocol_obj(object obj)
    {
    protocol_obj = obj;
    send sleep_and_alarm(SLEEP_TIME) to $self;
    }
```

8.8.2 Find_active_remote_obj

Description of Find_active_remote_obj

This function will be used to find new remote objects that are not on our list of active remote objects. We will be given the address of some remote object. We will check the linked list of active remote objects one at a time. If we find a match, then we will return the obj_entry structure. If we do not find a match, we will return NULL.

Code of Find_active_remote_obj

```
struct ObjStatus *  find_active_remote_obj(
struct sockaddr *sin)
    {
    struct ObjStatus *obj_entry;

    obj_entry = head;

    /*
     * If the entry are identical, do not add the
     * entry into the list.
     */
```

```
    while (obj_entry != NULL)
        {
        if (memcmp(&(obj_entry->sin),sin,
   sizeof(struct sockaddr)) == 0)
            return (obj_entry);
        obj_entry = obj_entry -> next;
        }
    return((struct ObjStatus *)NULL);
    }
```

8.8.3 Add_active_remote_obj

Description of Add_active_remote_obj

This function will add a new remote object on to the linked list of already active remote objects. We will be given the address of the remote object.

We will first call the find_active_remote_obj function, discussed in Section 8.8.2. If the function returns, then we know that the remote object is already on the list and we return. If it returns NULL, then we must add the remote object. Will first will fill in the o_entry struct values first. Then we will add the remote object structure, o_entry, on to the linked list of active objects.

Code of Add_active_remote_obj

```
void add_active_remote_obj(struct sockaddr *sin)
    {
    struct ObjStatus *obj_entry;

    obj_entry = $self.find_active_remote_obj(sin);

    if (obj_entry != NULL)
        return;

    obj_entry = (struct ObjStatus *)
$self.malloc(sizeof(struct ObjStatus));

    obj_entry->sin = *sin;

    obj_entry->status = ENABLE;
    obj_entry ->next  = NULL;
```

```
    if (head == (struct ObjStatus *)NULL)
        {
        head = obj_entry;
        total_obj ++;
        }
    else
        {
        obj_entry->next = head;
        head = obj_entry;
        total_obj ++;
        }
    }
```

8.8.4 Destroy_active_remote_obj

Description of Destroy_active_remote_obj

This next function will remove the remote object from the active objects list. That is, we will delete the o_entry structure from the linked list that we created in the add_active_remote_obj function, discussed in Section 8.8.3.

We will once agian get the address of the remote object and search through the remote object list one at a time until we find the remote object that matches. Once we find the o_entry that matches, we will free it. Then we will adjust the linked list so that the pointers around the o_entry deleted will be set properily.

Code of Destroy_active_remote_obj

```
void destroy_active_remote_obj(struct sockaddr *sin)
    {
    struct ObjStatus *obj_entry;
    struct ObjStatus *pre_obj_entry = NULL;

    obj_entry = head;
    while (obj_entry != NULL)
        {
        if (memcmp(&(obj_entry->sin),sin,
   sizeof(struct sockaddr)) == 0)
            {
            if (obj_entry == head)
                {
                head = head->next;
```

```
                $self.free((char *)obj_entry);
                return;
                }
            else
                {
                pre_obj_entry->next=obj_entry->next;
                $self.free((char *)obj_entry);
                return;
                }
            }
        pre_obj_entry = obj_entry;
        obj_entry = obj_entry -> next;
        }
    }
```

8.8.5 Sleep_and_alarm

Description of Sleep_and_alarm

The sleep_and_alarm function will send a verify message to all of the remote objects on it's active list every X seconds to determine which are still alive. If a remote object does not respond back in Y seconds after the verify message went out, the remote object will be considered dead. We must at this point check to see if there are any processes waiting for a response message back from this remote object. We will check all of the messages and see which processes are waiting. All of those waiting, will be waken up with a response message informing the processes that the remote object is no longer active. The sleep_and_alarm function will then call the destroy_active_remote_obj function to remove such remote objects. Here the remote object will be taken off of the list and we will then free the o_entry memory up.

Code of Sleep_and_alarm

```
void sleep_and_alarm(int time)
    {
    struct ObjStatus *obj_entry;

    while (1==1)
        {
        sys_obj.sleep(time-4);
        obj_entry = head;
```

```
    time_stamp ++;
    while (obj_entry != NULL)
        {
        /*
        obj_entry->status = VERIFY;
        */
        send send_verification(obj_entry) to $self;
        obj_entry = obj_entry->next;
        }
    /*
     * waiting for all the response to come back.
     */

    sys_obj.sleep(time);

    /*
     * Verify whether all the remote processes are
     * still alive.
     */

    obj_entry = head;
    while (obj_entry != NULL)
        {
        if (obj_entry->status == VERIFY)
            {
            /*
             * we have not yet received the
             * verification.
             */
            protocol_obj.no_response_from_process(
 &(obj_entry->sin));
            $self.destroy_active_remote_obj(
 &(obj_entry->sin));
            }
        obj_entry = obj_entry->next;
        }
    }
}
```

8.8.6 Send_verification

Description of Send_verification

This function has the job of sending out the verification messages to the obj_entry it is passed. It will take the obj_entry structure and change the status to VERIFY. Then it will send a message with the send_message_request function with a message status of CS_HELLO_ARE_YOU_THERE.

Code of Send_verification

```
void send_verification(struct ObjStatus * obj_entry)
    {
    char message[MSG_LENGTH];
    int return_value;

    /*
     * Every message that we send out we should time stamp
     * the message, just in case any of the response arrived
     * out of sync, we can identify it and drop the message.
     */

    sprintf(message,"%d",time_stamp);
    <
    obj_entry->status = VERIFY;

    if ((return_value = protocol_obj.send_message_request(
                                "NONE",
                                "NONE",
                                -1,
                                CS_HELLO_ARE_YOU_THERE,
                                INIT_REQ,
                                (char *)message,
                                strlen(message),
                                &(obj_entry->sin)))
                                == NULL)
        {
        /*
         * Unable to send to message to the remote object.
         *
         * We need to define a set of exception handler.
         */
        printf("Unable to send to message to the remote
```

```
object. \n");
        }
    >
    }
```

8.8.7 Receive_verification

Description of Receive_verification

This function will receive all of the messages that has to do with the verification of remote objects. The function will search the linked list of active remote objects with the find_active_remote_obj function. This function will return back to us the entry structure of the matched remote object. We will take the structure and set the status to ENABLE. This states that the remote object is still alive, so leave it on our active list.

Code of Receive_verification

```
void receive_verification(struct sockaddr *sin,
                          char * message)
    {
    int receive_timestamp ;
    struct ObjStatus *obj_entry;

#ifdef DEBUG

    printf("receive_verification \n");
#endif

    receive_timestamp = atoi(message);

    if (receive_timestamp != time_stamp)
        {
        /*
         * We have received an old message, this imply that
         * the remote process has received our verification
         * message, but unfortunatly the response message
         * got lost in the network.  Drop this message.
         */
        return;
        }
    <
```

```
    obj_entry = $self.find_active_remote_obj(sin);
    if (obj_entry != NULL)
         obj_entry->status = ENABLE;
    >
    }


end class RemoteVerification


end feature Verify
```

8.9 N_s_protocol.m

8.9.1 Description of N_s_protocol.m

This file is used to provide a high level protocol to communicate with the remote object by going through the name server. That is, the n_s_protocol sits on top of s_alone_obj.m. When s_alone_obj.m does a GetObj, the GetObj will come here. We will then ask the name server for the remote address, instead of sending the message directly to the remote object. The name server will then return the address back here. The n_s_protocol will then send the address and GetObj call back to s_alone_obj, where the message will then be forwarded to the remote side. The n_s_protocol will also be used to do a PutObj. When s_alone_obj does a PutObj, the call will come here and have us ask the name server to register the object, so that the object can be accessed by the remote side.

Headers of N_s_protocol.m

```
#define MAX_HOSTNAME 100

feature NS_Protocol_Obj

interface:

    imports RemoteObj[remote_protocol]
    imports NS[NameServer]
    imports Unix[sys_obj]
    exports ns_protocol

implementation:
```

```
  NameServerProtocol ns_protocol=
    NameServerProtocol.Create();

class NameServerProtocol ::=

NameServer name_obj;
struct sockaddr sin;

methods:
```

8.9.2 Init

Description of Init

The init function will try to set up a connection with the name server so that we can communicate with the server. We know the port number since it is hard codied into the system. (Note: This will be changed at some future date) We will get the hostname with the getenv("MELDC_NS") command. Then we will call the dest_addr function defined in the generic_remote_interface file, see Section 8.6.7 for more details. Then we will make a GetObj call. This will get the name server object for us at runtime. Now we will be able to get the messages from the server.

Code of Init

```
void init()
    {
    struct sockaddr sin;
    char * hostname;

    name_obj = (object) NULL;
    /*
     * Try to get the name server object.
     * The port number of the name server should be well
     * known by every one.  Right now the port number is
     * hard coded, we should change it later.
     */

    /*
     * Import the name server from the remote machine.
     * We will also hard code the hostname.
```

```
     */

    hostname = getenv("MELDC_NS");

    if (hostname == NULL)
        {
        remote_protocol.dest_addr(&sin,"tompkins",
  NAMESERVER_PORT);
        }
    else
        remote_protocol.dest_addr(&sin,hostname,
  NAMESERVER_PORT);

    name_obj = NameServer.GetObj("NameServer",
  remote_protocol,&sin);

    if (name_obj == (object)NULL)
        {
        /*
         * Unable to obtain the name server object,
 * try another one.
         */

        if (name_obj == (object)NULL)
            {
            printf("MeldC: Unable to connect to the name
    server.\n");
            sys_obj.exit(-1);
            }
        }
    respond "NameServerIsImported";
    }
```

8.9.3 Obj_init

Description of Obj_init

This function will be called by GetObj, from s_alone_obj, to get the object wanted from the network. We will first make sure that the name server has been imported by doing a delayuntil. This is necessary, since the name server must be used to perform this function. We will then lookup the name. This will give us the h_entry structure back. We then will call the dest_addr function to get the the address of the wanted object from the name server. We then will initialize the object's fields. Finally, we will return the object.

Code of Obj_init

```
object obj_init(MetaClass class_ptr, char * name)
    {
    object obj ;
    struct host_entry * h_entry;

    /*
     * sockaddr lookup should be done in the routine
     * is_object_exist.
     */

    if (name_obj == (NameServer)NULL)
        delayuntil "NameServerIsImported";

    h_entry = name_obj.lookup(name);

    if (h_entry == NULL)
        return((object)NULL);

    if (remote_protocol.dest_addr(&sin,h_entry->host_name,
  h_entry->port_num)
            == NULL)
        {
        return((object)NULL);
        }
    obj = remote_protocol.obj_init(class_ptr,name,&sin);

    return(obj);
    }
```

8.9.4 Register_obj

Description of Register_obj

This function is used by the PutObj call to give permission to the remote process to access this object. We will first obtain the object information. Then we will set the object name. We will then use the construct_host_entry function. We will then do a delayuntil, to make sure that the name server has been imported. Then we will register the object with the name server by calling the name_obj.register_obj function.

Code of Register_obj

```
void register_obj(MetaClass class_ptr,object obj,
  char * name)
    {
    struct sockaddr sin;
    struct host_entry h_entry;

    if (class_ptr != (MetaClass)
  (MetaClass.obtain_obj_info(obj,
                (enum obj_field)OBJ_CLASS_PTR)))
            {
            printf("Fatal error : inconsistent class
    name of exporting object");
            return(0);
            }
    class_ptr.set_obj_info(obj,
   (enum obj_field)OBJ_NAME,
   name);

    remote_protocol.construct_host_entry(&h_entry);

    remote_protocol.register_obj(class_ptr, obj,
         name);

    /*
     * Register the object with the name server.
     */

    if (name_obj == (NameServer)NULL)
        delayuntil "NameServerIsImported";

    name_obj.register_obj(name,h_entry);

    return;
    }
```

8.9.5 Miscellaneous functions

Description of Miscellaneous functions

```
int connect_remote(object obj)
    {
    remote_protocol.connect_remote(obj,&sin);
    return;
    }

void obj_destroy(object o, object shadow_object)
    {
    remote_protocol.obj_destroy(o,shadow_object);
    }

int reassign_port_num(int port_num)
    {
    return(remote_protocol.reassign_port_num(port_num));
    }

end class NameServerProtocol


end feature NS_Protocol_Obj
```

8.10 Malloc.m

8.10.1 Description of Malloc.m

The malloc file is obviousily used for the purpose of getting memory allocated when called. There are two functions here. The first is the malloc call. This malloc will be used to allocate the memory. The function will get the size of memory needed and will then do a m_malloc call, discussed in Section 2.8.2. We will then by returned a pointer to the memory allocated.

The other function is of course, the free function. Here we will be given a pointer to memory that has been allocated and now must be freed back to the system. This function will call the m_free function, discussed in Section 2.8.4. It should be noted at this point, what the hierarchy of freeing memory is. An object can free up it's own memory. An object can also free up the memory of it's own threads.

Code of Malloc.m

```
extern M_MEM m_create();
extern int m_free();
extern int kill();

feature MemoryAllocation

interface:

    imports Unix[sys_obj]

    exports Memory

implementation:

    Memory memory_obj = Memory.Create();

/*
 * The memory class should be inherited by a class who
 * needs additional memory from the heap.
 *
 * Important :
 * The allocated memory can only be freed by the owner
 * of the memory.
 */

class Memory ::=

methods:

char * malloc (int size)
    {
    long * ptr;

    /*
     * obtain the object id;
     */
    <
    ptr = (long *)m_malloc(&(_obj->$.add_memory),size);

    return((char *)ptr);
```

```
    >
    }

char * free (char * p)
    {
    long * ptr;

    if (p == NULL)
return;
    <
    m_free(_obj->$.add_memory,(char *)(p));
    >
    }

end class Memory

end feature MemoryAllocation
```

8.11 Ns.m & Init

8.11.1 Description of Ns.m & Init

The file contains the name server for the system. The name server is responsible for resolving the name of an object. To talk to the name server, the n_s_protocol will be used. The first function in this file is the init function. It will first initialize a table called the obj_addr_table. This table wil contain all of the address of objects on the local machine. This will allow us to look up the address of an object. Then the function will do a PutObj. This will allow the remote_protocol object access to the name server.

Headers of Ns.m & code of Init

```
#define DEBUG

#define MAX_BUF 256

feature NS

interface :

    imports RemoteObj[remote_protocol]
    imports HashTable[Hash]
```

```
    imports Unix[sys_obj]
    exports NameServer

implementation:

    NameServer  ns_obj = NameServer.Create();

class NameServer ::=

Hash    obj_addr_table = Hash.Create();
int     fd;
char    buf[MAX_BUF];

methods:

void init ()
        {
        fd = sys_obj.open("/tmp/meldc_ns.log",
  O_RDWR|O_CREAT,0666);

        if (fd == -1)
            {
            printf("Can not open the name server log file.
    %d \n", sys_obj.errno());
            fd = 1;
            }
        obj_addr_table.meldcNewHashTable(STRINGHASH,
 DEFAULTSIZE, 20);

        /*
         * We will hard code the nameserver address.
         */
        if (remote_protocol.reassign_port_num(
   NAMESERVER_PORT) == -1)
            {
            sprintf(buf,"port number assignment failed
     \n");
            sys_obj.write(fd,buf,strlen(buf));
            }

        NameServer.PutObj ($self, "NameServer",
   remote_protocol);
```

```
        }
```

8.11.2 Register_obj

Description of Register_obj

This function will register the object into the name server table, obj_addr_-table. In other words, this function wil inform the name server where the object exists in the system on the local side.

To do this, it will first check to see if the object name is already in the table by executing the obj_addr_table.meldcHash call. If the entry already exists, then the call will return non-NULL. In this case, we will first get the memory address of the entry. Then we will delete the old entry from the table by executing the obj_addr_table.deleteHash call. If the entry did not exist already, then we will just malloc the needed memory for the entry to be put into the table. If it did or did not exist in the table already, we will assign the entry to the address of the memory allocated. Then we will insert the entry into the hash table with the insertHash call for the obj_addr_table.

Code of Register_obj

```
int register_obj(char * name, struct host_entry h_entry)
    {
    HEntry * entry;
    struct host_entry * addr;

    /*
     * Delete the old entry if exist.
     */
    <
#ifdef DEBUG
    sprintf(buf,"Register the object %s with name server.
    \n",name);
    sys_obj.write(fd,buf,strlen(buf));
#endif
    entry = obj_addr_table.meldcHash ( name );

    if (entry != NULL)
        {
        sprintf(buf,"An entry already exist, remove the
old entry.\n");
```

```
        sys_obj.write(fd,buf,strlen(buf));
        addr = (struct host_entry *)entry->ptr;
        obj_addr_table.deleteHash(name);
        }
    else
        {
        addr = (struct host_entry *) MALLOC
       (sizeof(struct host_entry));
        }
    /*
     * Even though the $sender, correspond to the remote
     * object, but, it is actually points to the shadow
     * objects.  We will be able to access one of the
     * method in the shadow object.
     */

    *addr = h_entry;

    entry = obj_addr_table.insertHash (name);

    entry->ptr = (int *) addr;
    >
    return(0);
    }
```

8.11.3 Lookup

Description of Lookup

This next function in this file will check the hash table to see if the name of the object is in the table and return the respective entry. We will make the obj_addr_table.meldcHash call. This will return the respective entry for the given object name. If the entry exists, it is not equal to NULL, then we will return the entry to the caller. If the entry is NULL, then we will return NULL.

Code of Lookup

```
struct host_entry * lookup (char * name)
    {
    HEntry * entry;
    struct host_entry no_entry_available;
```

```
    /*
     * Do we need to call meldcNewHashTable
     */
    <
#ifdef DEBUG
    sprintf(buf,"Query the object %s with name server.
            \n",name);
    sys_obj.write(fd,buf,strlen(buf));
#endif

    entry = obj_addr_table.meldcHash (name);

    if ( entry != NULL )
      {
      return(((struct host_entry *)(entry->ptr)));
      }
    else
      {
#ifdef DEBUG
      sprintf(buf,"No entry available. %s \n",name);
      sys_obj.write(fd,buf,strlen(buf));
#endif
      no_entry_available.host_name[0] = 0;
      no_entry_available.port_num = 0;
      /*
      return(no_entry_available);
      */
      return((struct host_entry *)NULL);
      }
    >
    }
```

8.11.4 Deregister_obj

Description of Deregister_obj

This function will take an entry in the hash table of the name server and remove it. We will be given the name of the object entry to do so. It is a simple process of just making the obj_addr_table.meldcHash call. This will give us the entry of the given name. We will then call the free on the memory of the entry. Then we will do a obj_addr_table.deleteHash call.

Code of Deregister_obj

```
int deregister_obj (char * name)
    {
    HEntry * entry;
    <
#ifdef DEBUG
    sprintf(buf,"Deregister the object %s with name server.
            \n",name);
    sys_obj.write(fd,buf,strlen(buf));
#endif

    entry = obj_addr_table.meldcHash (name);
    FREE(entry->ptr);
    obj_addr_table.deleteHash(name);
    >
    return(0);
    }

end class NameServer

end feature NS
```

8.12 Hash.m

8.12.1 Description of Hash.m

This last file of the Runtime chapter is the well known, but mysterous hash function file, hash.m. Here is where all of the hash functions exist on the runtime side of the MeldC environment.

Headers of Hash.m & init function

```
feature HashTable

interface:

exports Hash

implementation:

#define NSIZE 256
```

```
class Hash merges MemoryAllocation[Memory]::=
HTable * hash_table;

methods :

void init()
{
<
$self.meldcNewHashTable(STRINGHASH, DEFAULTSIZE, 20);
>
}
```

8.12.2 meldcNewHashtable

Description of meldcNewHashtable

This function is ued to initialize a new hash table. It basically will malloc the needed memory for the table and then set some of the variables of the table with the given parameters passed in.

Code of meldcNewHashtable

```
void
meldcNewHashTable(int type, int increment,
  int table_size)
{
    HTable *new_table;

    new_table=(HTable *)$self.malloc(sizeof(HTable));
    new_table->type = type;
    new_table->increment = increment;
    new_table->number_of_entry = 0;
    new_table->table_size = table_size;

    if (table_size == 0)
        new_table->table = NULL;
    else if (table_size > 0)
        {
        new_table->table = (HEntry **)$self.
               malloc(sizeof(HEntry *) * table_size);
        bzero(new_table->table,
```

```
            sizeof(HEntry *) * table_size);
      }
    else printf("Error in table_size [%d] !\n",
              table_size);

    hash_table = new_table;
}
```

8.12.3 MeldcHash

Description of MeldcHash

This function as demostrated many times will find the given name in the table and return the entry found. This function must take the name given and perform the hash function on the name to find the proper key for the table slot.

Code of MeldcHash

```
HEntry *
meldcHash(char * h_name)
{
    HEntry *result;
    int key ;
    <
    key = $self.getHashKey(hash_table->type,
   (int *)h_name,
                          hash_table->table_size);
    if(key == -1 || hash_table->table[key] == NULL)
        return (HEntry *) NULL;

    result = (hash_table->table)[key];
    if (hash_table->type == STRINGHASH)
        while(strcmp(result->u.name, h_name) != 0)
        {
            if (result->next == NULL)
               return (HEntry *) NULL;
            else result = result->next;
        }
    else
        while(memcmp(result->u.objid, h_name,
     sizeof(ObjID)) != 0)
```

```
        {
            if (result->next == NULL)
               return (HEntry *) NULL;
            else result = result->next;
        }
    return result;
    >
}
```

8.12.4 InsertHash & DeleteHash

Description of InsertHash & DeleteHash

These two hash functions have also been used quite often. The insertHash function will simply find the slot for the given name in the table and put the entry into the table. The deleteHash table will just take the name and find the entry in the table and will simply remove the entry.

Code of InsertHash & DeleteHash

```
HEntry *
insertHash(char * h_name)
{
    int key;
    HEntry *entry;
    <
    if((HASHRATE * hash_table->table_size) <=
hash_table->number_of_entry)
        $self.reHash(POSITIVE);
    if($self.meldcHash(h_name) != NULL) return NULL;

    entry = $self.myInsert(h_name);
    hash_table->number_of_entry++;
    return entry;
    >
}

int
deleteHash(char *h_name)
{
    HEntry *previous, *entry;
    <
```

```
    int key = $self.getHashKey(hash_table->type,
       (int *)h_name,
                             hash_table->table_size);

    if(key == -1) return FALSE;
    if(hash_table->table[key] == NULL) return FALSE;

    entry = hash_table->table[key];
    if (hash_table->type == STRINGHASH)
    {
        if(strcmp(entry->u.name, h_name) == 0)
        {
            hash_table->table[key] = entry->next;
            return $self.myDelete(entry);
        }
        while(strcmp(entry->u.name, h_name) != 0)
        {
            if (entry->next == NULL) return FALSE;
            else
            {
                previous = entry;
                entry = entry->next;
            }
        }
    }
    else
    {
        if(memcmp(entry->u.objid, h_name,
  sizeof(ObjID)) == 0)
        {
            hash_table->table[key] = entry->next;
            return $self.myDelete(entry);
        }
        while(memcmp(entry->u.objid, h_name,
     sizeof(ObjID)) != 0)
        {
            if (entry->next == NULL) return FALSE;
            else
            {
                previous = entry;
                entry = entry->next;
            }
```

```
        }
    }
    previous->next = entry->next;
    return $self.myDelete(entry);
    >
}
```

8.12.5 Miscellanous Hash Functions

Description of the Miscellanous Hash Functions

Here are the rest of the functions that exist to help the other previousily mentioned functions:

- meldcNextEntry
- getHashKey
- getStringHashKey
- getBinaryHashKey
- getStringIndex
- myInsert
- myDelete
- reHash

Code of the Miscellanous Hash Functions

```
/*
 * Obtain the next entry from the hash table.
 */

HEntry *
meldcNextEntry(HEntry * entry , int * key)
    {
    HEntry *result;
    <
    if (entry == NULL)
        {
        *key = 0;
        }
```

```
    else if (entry->next != NULL)
        return(entry->next);

    for (;*key < hash_table->table_size; (*key) ++)
        {
        result = (hash_table->table)[*key];
        if (result != NULL)
            return(result);
        }
    return(0);
    >
    }

int
getHashKey(int type, int *h_something, int h_size)
{
    if (h_size > MAXSIZE || h_size <= 0)
    {
        fprintf(stderr, "Invalid Hash Table Size [ %d ]
\n", h_size);
        return -1;
    }
    if(type == STRINGHASH)
        return $self.getStringHashKey((char *) h_something,
      h_size);
    else
        return $self.getBinaryHashKey(h_something, h_size);
}

int
getStringHashKey(char *h_str, int h_size)
{
    int look, cut, h_code;

    h_code = $self.getStringIndex(*h_str);
    if(h_code == -1) return -1;

    if (h_size <= C_ALPHA_SIZE)
    {
        cut = C_ALPHA_SIZE / h_size;
        return $self.normalized(h_code / cut, h_size);
    }
```

```
    look = h_size / C_ALPHA_SIZE + 1;
    return
        $self.normalized
        (h_code + $self.getStringHashKey(
                          (char *)(h_str + 1), look) *
                          C_ALPHA_SIZE, h_size);
}

int
getBinaryHashKey(int *h_number, int h_size)
{
    int look, cut, h_code;

    h_code = $self.getBinaryIndex(*h_number);

    if (h_size <= BINARY_SIZE)
    {
        cut = BINARY_SIZE / h_size;
        return $self.normalized(h_code / cut, h_size);
    }
    look = h_size / BINARY_SIZE + 1;
    return
        $self.normalized
        (h_code + $self.getBinaryHashKey(
                         (int *)(h_number + 1), look) *
                         BINARY_SIZE, h_size);
}
int
getStringIndex(char my_c)
{
    if(my_c == '$' || my_c == '\0') return 0;
    if(my_c <= '9' && my_c >= '0') return my_c - '0' + 1;
    if(my_c <= 'Z' && my_c >= 'A') return my_c - 'A' + 11;
    if(my_c == '_') return 37;
    if(my_c <= 'z' && my_c >= 'a') return my_c - 'a' + 38;
    fprintf(stderr, "Invalid String [ %c ]\n", my_c);
    return -1;
}

int
getBinaryIndex(int my_i)
{
```

```
    return (my_i % BINARY_SIZE);
}

HEntry *
myInsert(char *h_name)
{
    int key;
    HEntry *entry;
    <
    key = $self.getHashKey(hash_table->type,
           (int *)h_name,
                           hash_table->table_size);
    entry = (HEntry *) $self.malloc(sizeof(HEntry));
    entry->next = hash_table->table[key];
    if (hash_table->type == STRINGHASH)
        {
        entry->u.name =
               (char *)$self.malloc(strlen(h_name));
        strcpy(entry->u.name,h_name);
        }
    else
    {
        ObjID *oidp;
        oidp =(ObjID *) $self.malloc(sizeof(ObjID));
        oidp = (ObjID *) memcpy(oidp, h_name,
                      sizeof(ObjID) / sizeof(char));
        entry->u.objid = oidp;
    }
    hash_table->table[key] = entry;
    return entry;
    >
}

int
myDelete(HEntry *entry)
{
    <
    $self.free((char *)(entry->u.name));
    $self.free((char *)entry);
    hash_table->number_of_entry--;
    if((HASHRATE * (hash_table->table_size -
            hash_table->increment))
```

```
       > hash_table->number_of_entry)
        $self.reHash(NEGATIVE);
    return TRUE;
    >
}

int
normalized(int h_code, int h_size)
{
    while(h_code > (h_size - 1)) h_code -= h_size;
    return h_code;
}
void
reHash(int my_sign)
{
    HEntry         **new_hash_table, **old_hash_table,
   *entry;
    int            new_table_size, old_table_size, i;

    new_table_size = hash_table->table_size +
     hash_table->increment * my_sign;
    new_hash_table = (HEntry **)
                     $self.malloc(sizeof(HEntry *) *
     new_table_size);
    for (i = 0; i < new_table_size; i++)
        new_hash_table[i] = NULL;

    old_table_size = hash_table->table_size;
    old_hash_table = hash_table->table;
    hash_table->table_size = new_table_size;
    hash_table->table = new_hash_table;

    for (i = 0; i < old_table_size; i++)
    {
    HEntry *  temp_entry;

        entry = old_hash_table[i];
        while (entry != NULL)
        {
            if((temp_entry =
                $self.myInsert(entry->u.name)) ==
                (HEntry *) NULL)
```

```
                printf("Error in ReHash!!!\n");
            else
                temp_entry->ptr = entry->ptr;
            entry = entry->next;
        }
        $self.free((char *)(old_hash_table[i]));
    }
    $self.free(old_hash_table);
}

end class Hash

end feature
```