RESTRUCTURING THE RUN TIME SUPPORT OF A DISTRIBUTED LANGUAGE

by

Hugh Bawtree

B.Sc., University of Victoria, 1984

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the School of

Computing Science

SIMON FRASER UNIVERSITY

October, 1991

APPROVAL

Name: Hugh Bawtree
Degree: Master of Science
Title of Thesis: Restructuring the Run Time Support of a Distributed Language

Dr. Peter Triantafillou
Assistant Professor
Chairman

Dr. M. Stella Atkins
Associate Professor
Senior Supervisor

Dr. Warren Burton
Professor

Dr. Robert Cameron
Associate Professor
Examiner

30 Oct 91
Date Approved

Title of Thesis/Project/Extended Essay

Restructuring the Run Time Support of a Distributed Language.

Author:

(signature)

Hugh Alexander Bawtree

(name)

October 24, 1991

(date)

ABSTRACT

Distributed programming languages are designed to make distributed programming simple through the use of powerful concurrent programming features and program checking which the compiler provides. Unfortunately, current distributed programming languages are not yet sufficiently fast, dependable and portable enough to make them more appealing to use than the alternatives. Distributed programs are commonly programmed in third generation languages with system calls embedded in the code. These programs are fast but notoriously difficult to program.

These problems can be alleviated by improving the Run Time Support of a distributed programming language. The Run Time Support implements the distributed constructs and other language constructs whose exact execution can only be determined at run-time.

We re-designed the Run Time Support for the distributed programming language called Synchronizing Resources (SR). We succeeded in making it simpler, faster, easier to maintain, more portable, and easier to test.

This thesis describes the software engineering techniques we used to improve the Run Time Support, the application of the techniques, and the improved design. Through our implementation, we justify our claims of simplicity, speed, maintainability, portability and testability.

Table of Contents

Table of Figures

CHAPTER 1

INTRODUCTION

1.1 Goals

In the implementation of a distributed language, it is common practise to hide the implementation of the distributed concepts in a Run Time Support (RTS) system. The procedures in the RTS are invoked from the code generated by the compiler. This design separates, and simplifies the design of both the compiler and the RTS.

The goal of this project is to improve the Run Time System (RTS) design of the distributed programming language SR (Synchronizing Resources) [ANDR86]. We applied software engineering techniques to make the system simpler, easier to port, and more secure. In the process of re-designing the system, we gained a greater understanding of the RTS design issues. In general, this understanding leads to a better understanding of the issues associated with the design of distributed programming languages.

1.2 Performance

The initial goal of our project was to reduce the communication overhead of SR, running on a network of SUN workstations, using the UNIX operating system. Another operating system called V-system [ChLa86] has much faster communication primitives.

We believed that by porting SR from UNIX to V-system, and replacing the UNIX communication sockets with V-system messages, we could greatly reduce the communication time of SR programs. Appendix A has the complete details on SR communication time.

1.3 System Design Problems

However, during the testing of the SR RTS on V-system (SR/V), we encountered many difficulties. We found some very difficult bugs (errors), some of which took weeks to analyze. We determined that these errors were caused by a faulty system design. The design faults were so severe, that we were forced to re-design the system.

This thesis describes our re-designed version of the SR RTS, and the techniques we used to avoid further system design errors. We believe these techniques are applicable to other large systems, in particular, other distributed systems.

1.4 Thesis Overview

The remainder of this thesis is divided into the following chapters:

2. Related Work: a review of other papers on the implementation of distributed programming languages.

3. Background: a review of Software Engineering, the SR language and the V-system operating system.

4. System Design: a description of the system design for our implementation of SR, and the techniques we used in the design.

5. Operating System Subsystem Design: a description of the design issues faced in the implementation of the Operating System subsystem.

6. Conclusions: a summary of system design techniques and distributed programming language design.

CHAPTER 2

RELATED WORK

2.1 Run Time Support

There have been very few papers written on the subject of distributed language RTSs. The ones that have been written tend to concentrate on the design details rather than the overall system design.

Both Almes [ALME85] and Lohr [LOHR88] implemented Remote Procedure Call (RPC) mechanisms to convert sequential Modula-2 programs into distributed programs. Almes' RPC mechanism is implemented on the V-system and Lohr's RPC mechanism is implemented on both MS-DOS and UNIX. However, there is no concurrency in these programs. If a program component of program P is currently executing on machine B, then the program component of P on machine A is suspended. These researchers use the RPC mechanism because it is easily adapted to existing sequential languages. However, we do not believe the major criteria for a successful distributed language is its similiarity to a sequential language. In fact, a distributed language should have mechanisms to support as much concurrency as possible, since a major advantage of distributing a program is to reduce the program's execution time.

Almes evaluated the V-system in terms of its support for his RPC mechanism. He found the performance to be quite fast for both the small, fixed size messages implemented by the Send, Receive and Reply primitives and the large, variable size messages. The performance is analyzed in detail in his paper. On the ease-of-programming side, he found the V-system's kernel mechanisms are simple to understand, compared to the interprocess mechanisms of many other systems. However, he found that due to the two methods of communication (Send/Receive and MoveTo/MoveFrom), the code must decide before sending a message which communication method to use. This adds complexity to the RPC stub generator code. The communication method used for each RPC is determined by checking the amount of data being sent, and matching the data size to the most appropriate communication method. Data less than 32 bytes long can be sent with a Send; larger data are sent using a combination of a Send and MoveTo.

Lohr's RPC mechanism was implemented for both the MS-DOS and the UNIX operating systems (OS). Unfortunately for our purposes, he does not analyze or evaluate either of the two OS. Instead, he develops his own simple distributed operating system which runs on top of MS-DOS and UNIX. In his distributed operating system, communication is performed with RPC calls and abort messages, and security is maintained with user-names and the help of the local OS. The RPC calls are implemented in the standard manner. The abort messages are sent to all remote components of program, when one component of a distributed program dies. For security checks, the distributed operating system assumes that a user has the same username on each machine. Then all security checks can be handled by the local OS.

Newton [NEWT87] implemented an RTS for Ada tasking which supports concurrency on the Mach operating system. Ada tasking is complicated to implement but the only process interaction mechanism supported is the *rendezvous*. SR is a more complete distributed language because of its flexible process interaction mechanism, of which *rendezvous* is but one example.

Newton does not explicitly analyze the performance of communication primitives in Mach. However, it appears from some of his timing tests, that Mach performs context switches between processes in 0.5ms on a four processor VAX 8200 which is almost twice as fast as the V-system context switch on a 10-MHz 68000 microprocessor. Since there is no common machine which both Mach and V-system are implemented on, it is difficult to compare their performance. Newton does not evaluate the ease-of-programming using Mach primitives.

Finley [FIN89] modified the SR/UNIX RTS for the Sequent multiprocessor, which runs a variant of UNIX. Curtis performed extensive performance tests to analyze performance problems. He also addresses many of the design issues associated with implementing a distributed language on a multiprocessor machine. Many of these design issues are associated with protecting critical sections. He does not comment on any system design issues. It appears he did not have to make any major changes to the SR/UNIX design.

Swinehart et al [SWI86] describe the system design of the Cedar Programming Environment which includes an operating system, programming environment and programming language. The system design of this large project has some similiarities with the SR/V design. It is interesting to note the similiarity between the Cedar machine layer and the SR/V machine subsystem (described in section 4.3.4), and between the Cedar Nucleus level and the SR/V Operating_System subsystem (described in section 4.3.2). The Cedar system, like the SR/V system, had problems with circular dependencies (described in section 4.2.1), which the authors call "loops". The Cedar approach to resolving the circular dependencies is to use sophisticated programming techniques: call-back procedures, registered procedures, procedural objects and object classes. All of these techniques are explained in the [SWI86] paper. The SR/V approach has been to eliminate the circular dependencies through re-design, using standard programming techniques. We believe the elimination of circular dependencies is preferable to using unusual programming techniques which are not supported in every programming language.

Our research is different than the above named research. We concentrate on the system design of an RTS. We attempt to eliminate the system design problems through the application of some basic software engineering principles, and we implemented the system in a standard third-generation language (C). Finally, we attempt to generalize the issues to all RTSs.

CHAPTER 3

BACKGROUND

3.1 Design Principles useful in System Restructuring

The Software Engineering (SE) field has been under investigation for a long time and the general principles are well understood. In this section we review the general principles that we found useful in the SR/RTS system restructuring, and introduce a set of techniques which use these general principles.

3.1.1 Modularization

The most important design technique we use is **modularization**. We used modularization to divide the RTS system into subsystems, and subsystems into modules. We also used modularization to extract modules whose functionality was originally duplicated in several other modules. In designing the modules we used the SE concepts of cohesion and coupling. More information about these concepts can be found in any SE textbook.

3.1.2 Abstraction

Abstraction is the separation of the interface from the implementation. The abstraction design technique is used to provide several layers of functionality [DIJ68]. For example, memory modules in an operating system can provide several layers of increasing functionality. At the lowest layer, a memory module could provide a memory block from any area of main memory. At the middle layer, another memory module provides a virtual memory block which, depending on the current access, is stored in main memory or on disk. At the top layer, a third memory module provides a virtual memory block in the current user's memory address space.

3.1.3 Dependency Diagrams

Another key design technique for clarifing the RTS design is the **dependency diagram**. These diagrams are used to show the dependencies between subsystems and modules. We define the depend relationship in the following manner: subsystem A depends on subsystem B if A uses a procedure, a data type, or anything which is implemented in subsystem B. The dependency diagram for the A and B subsystems is drawn below:

The depend relationship and the dependency diagrams are defined similiarly for modules.

We sometimes use the word use as a synonym for depend.

3.2 SR language

SR supports heavyweight virtual machines (VM) containing resources which contain lightweight processes. Each VM contains one address space unshared with any other VM. VMs may execute on the same or different physical machines. All communication, i.e. inter-VM, inter-resource and inter-process, is achieved through operation invocation. An operation is a generalization of a procedure.

The remainder of this section describes resources, and the mechanisms for implementing and invoking operations.

Resources, like modules in Modula-2, are the building blocks of SR programs.

Following software design principles, a resource is used to implement a software abstraction such as a bounded buffer, a file system, or a process manager. Resources may use other resources. For example, the file system resource could use the bounded buffer resource.

Each resource has a specification component, which declares the operations exported by the resource. The bounded buffer resource specification which exports the deposit and fetch operations looks like:

```
resource bounded_buffer
      op deposit (val item:int)              # val means value param.
      op fetch (res item:int)                # res means result param.
body bounded_buffer (size:int) separate     # size is size of buffer
```

The bounded buffer resource code that we use here is taken from [AnO187].

Within a resource, the operations may be implemented by either a **proc** or an **in** statement. The **proc** is similiar to a procedure. It can be invoked at any time, and there may be many copies of one **proc** being executed at the same time by different processes. The **in** statement is contained in a **proc**. In its simplest form it waits for one particular operation to be invoked. When it receives that invocation, it executes the body of the **in** statement, sends a reply, and continues with the execution of the **proc**. In the more complicated form, an **in** statement may wait for any of several operations to be invoked. Receiving any of the operation invocations will cause the corresponding body of code to be executed, send a reply, and continue with the execution of the **proc**. Our example of a bounded buffer implements the *deposit* and *fetch* operations with an **in** statement inside a process. The resource body for the bounded buffer follows:

```
body bounded_buffer
var buff[0:size-1]: int
var count:=0, front:=0, rear:=0

process worker
      do true ->                          # repeat in stmt forever
            in deposit(item) & count<size ->     # receive deposit invoc.
                  buf[rear] := item
                  rear := (rear+1) % size
                  count++
            [] fetch(item) & count>0 ->          # receive fetch invoc.
                  item := buf[front]
                  front := (front+1) % size
                  count--
            ni
      od
end worker

end bounded_buffer
```

Before an operation can be invoked, the resource which implements the operation must be created. The **create** statement creates an instance of a resource on a VM, and returns a unique object identifying the resource instance, called a **capability**, which identifies the resource instance. Possession of resource A's capability by resource B allows B to invoke A's operations. Every invocation of an operation must specify the resource instance by including the resource's capability in the invocation statement.

SR provides two invocation statements: **call** and **send**. A **call** statement causes the invoking process to be suspended until the operation is completed. The **send** statement causes the operation to start executing as a separate process.

This means the invoking process executes concurrently with the invoked operation. An example of a resource which invokes the bounded buffer resource follows:

```
resource user
      import bounded_buffer
body user()
      var bb: cap bounded_buffer # capability of b.b. resource

      initial
            var item: int      # integer variable

            # create a buffer with room for 20 items.
            bb := create bounded_buffer(20)

            send bb.deposit(5)       # create process to deposit 5
            send bb.deposit(3)       # create process to deposit 3
            call bb.fetch(item)      # suspend until item is fetched
            write (item)

            call bb.deposit(2)       # suspend until 2 is deposited
            call bb.fetch(item)      # suspend until item is fetched
            write (item)

            destroy bb  # destroy bb instance of resource
      end initial

end user
```

When an SR program starts, the default VM is located on the initiating machine. The main resource is created on the default VM and starts to execute. The main resource creates other resources which may contain new processes. The processes then communicate between themselves using the operation invocation and implementation statements described above.

Together, the SR statements **call, send, proc** and **in** implement the following process interactions:

Invocation	Implementation	Process Interaction
call	**proc**	remote/local procedure call
call	**in**	rendezvous
send	**proc**	dynamic process creation
send	**in**	message passing/semaphore

The performance of these communication primitives is described in Appendix A, and detailed in [ATKI88]. More information on the SR language is in [ANDR86] and [AnOl87].

3.3 V-system

The V-system supports teams which contain lightweight processes [CHER84]. Each team has its own address space, unshared by any other team. A process can create another process on the same team but it can not create a process on another team. It may create a new team with an initial process on either the same machine or another machine.

The V-system communication is implemented with messages sent between client and server processes. A client process X sends a message to a server process Y on either the same team or on another team, when it requires the service controlled by Y. The client process is suspended until the server process replies to the message indicating the service has been performed. The above model is implemented with three system calls: *Send, Receive* and *Reply*. The *Send* call sends a 32 byte message to the specified process and blocks the sending process until a Reply is received. The *Receive* call blocks the receiving process until a message is received. The *Reply* from the receiver sends a message back to the process which is Send-blocked. This model is much simpler than the UNIX socket model and no initial startup is required.

Once a server has received a Send message from a client, the server may initiate variable-size message transfers, using the *MoveFrom* and *MoveTo* system calls. The server may copy either to or from the client's address space. The portion of the address space available to the server is passed to the server in the initial fixed-size *Send* message. Since the client is suspended, there should be no problems with two processes accessing the same memory location. With this feature V-system ensures that large message transfers are still efficient. This would not be the case if only fixed-size messages were implemented. Note that care must be taken if there are other processes running on the client's team since they are not suspended.

Cheriton explains the reasons for the V-system inter-process communication (ipc) primitives in [CHER84]. He designed the ipc primitives to "efficiently support procedural interfaces". In this respect he has certainly succeeded since V-system is still one of the fastest distributed operating system in terms of message passing, and the Send primitive can easily be used to simulate a procedure call. Furthermore, he claims that implementing a non-blocking Send primitive is unnecessary for two reasons. First, he has experience with a distributed operating system that implements a non-blocking Send. He writes "practise showed that during execution, a process typically suspended execution to wait for a reply immediately after sending a message." Second, "such concurrency in communication is difficult to use and imposes an excessively high cost on the implementation", due to the message buffer management. He prefers to use additional lightweight processes to achieve concurrency. More detailed information on the V-system is in the manual [ChLa86].

Note that the V-system synchronous, blocking communication primitives contrast with the SR **send** invocation which is asynchronous and non-blocking. Our implementation of the **send** using V-system communication primitives is explained in section 5.2.4.

CHAPTER 4

DESIGN ISSUES

4.1 Overview

4.1.1 Introduction

In this chapter, we describe the general design problems that we encountered in the SR/V RTS design and describe our solutions. The complete description of the SR/V design is in Appendix B. We have divided the design issues into three categories: system design, module design and management issues. System design answers questions to do with the structure of the system. For example, how do the modules fit together? How is the system divided into modules? On the other hand, module design answers questions about individual modules. For example, how does the scheduler module decide which process to execute next? How does the semaphore module store the data about processes blocked on a semaphore? Of course, the system design can not be completely separated from the module design. Often a change in a module design will cause a change in the system design, and vice versa. Nevertheless, the division between system and module design provides us with two levels of abstraction, which makes the entire design easier to understand. Management issues arise because the project is large, complicated and requires much time and effort to complete. Management issues deal with questions such as: How large? How complicated? How much time, and how much effort?

4.1.2 SR RTS

The SR RTS is responsible for implementing the following SR concepts: VMs, resources, processes, operation types, and SR call and send invocations. The

SR call and send invocations must be executed either locally or remotely, depending on the context of the invocation.

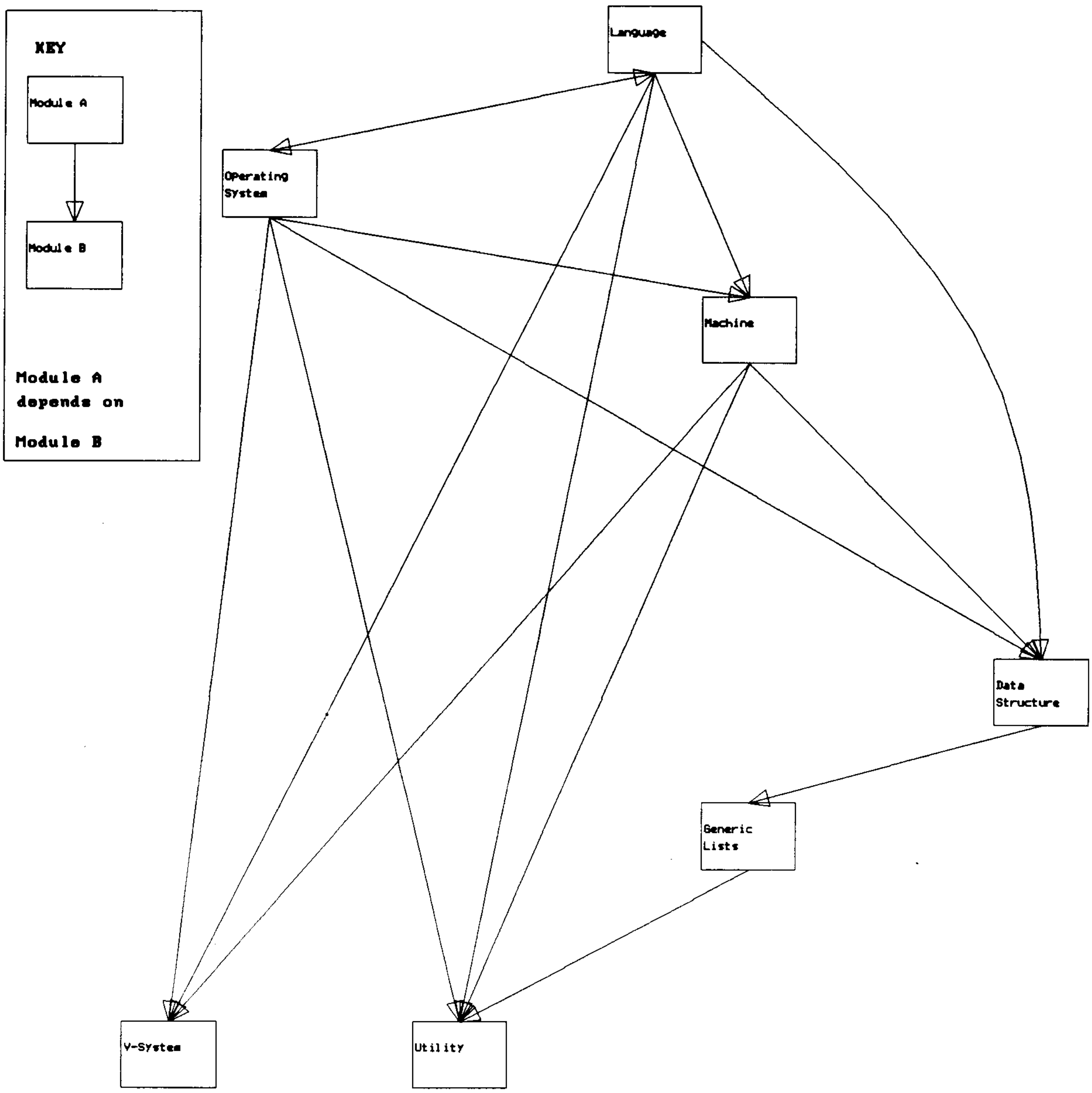
Figure 1 shows the SR RunTime System Dependency Diagram with dependencies between the RTS subsystems. We applied the software engineering techniques described in section 3.1 and divided the RTS into the six subsystems shown in Figure 1 (the V-system box is not a part of the RTS): Language (LG) subsystem, Operating System (OS) subsystem, Machine (MC) subsystem, Data Structure (DS) subsystem, Generic Lists (GL) subsystem and Utility (UT) subsystem.

We did not implement and test the entire RTS. We wrote a system design and documented it for the entire RTS. However, we only implemented the Operating System (OS) level and below. That is all of the RTS, except the Language (LG) subsystem. The OS level is the most significant part of our design and required the most design effort. The LG level would require much work to implement but the design issues are minor. We feel that we have verified our design and our design approach by implementing the OS level.

4.1.3 Design Goals and Tests

Throughout our design, we have striven to achieve the following goals: simplicity, security, and portability. The goal of simplicity means we choose to use standard designs instead of custom, elaborate designs, whenever we can. We use Hoare's explanation of security in language design [HOA81]. Hoare suggests that every result and error message must be understandable in terms of the source code. A secure language, again according to Hoare, means that it must be "logically impossible" for a program to cause the computer to run wild at compile-time or run-time. Portability means the language implementation can be easily changed to run on a different machine and/or a different operating system. The issue of portability will be dealt with in the next chapter on Operating Systems.

Figure 1
SR/V RTS Dependency Diagram



In this chapter, we explain our system design and show what we have done to make it simpler and more secure. We have used two tests in our attempt to weed out overly complex and insecure designs.

The first test involves writing. For each module and subsystem, there is a description. The purpose of the module is described in one sentence and the internal design is described in one paragraph. Any special provisions for the module's security are described in further paragraphs. If the module purpose can not be described in one sentence, then its cohesion is too low. If the design can not be described in one paragraph then it is too complex and it should be divided into two or more modules. Several times, we found that the process of writing brought out new and better designs. Although this test is not rigorous, in practise it always helps us find errors and improve the design.

The second test follows the first test. The module or subsystem design document is submitted to one or two other reviewers who review the documents for simplicity, security and errors. Any concerns the reviewers have are passed onto the designer who is responsible for improving the design. This system seems to work best if the reviewers are the designer's peers; eg. in this case, the designer's peers are fellow graduate students.

4.2 Application of the Design Techniques

4.2.1 A System Design Problem: Circular Dependencies

A major problem in RTS system design is circular dependencies. Swinehart et al describe circular dependency problems, which he calls "loops" in [SWI86]. The simplest example of a circular dependency is a mutual dependency which occurs when Subsystem A depends on Subsystem B and Subsystem B depends on A. There are also indirect circular dependencies with 'larger' circles. There may be four or five subsystems in the circular dependency, each subsystem depending on the next subsystem, and the last subsystem depending on the first subsystem (Eg. A -> B -> C -> D -> A). Circular dependencies may also occur between modules, in either the mutual or indirect form.

These circular dependencies are a problem for several reasons. First, they may indicate a mutually recursive procedure call. If this recursion is not completely understood, it could cause infinite recursion to occur every time the program is run, or, worse, just under special circumstances! Therefore, every circular dependency on the dependency diagram must be investigated to make sure that the design has safeguards against infinite recursion.

The second problem is deadlock due to resource contention. This type of deadlock occurs in the following scenario. Subsystem A has control of resource X, and it calls subsystem B. B needs to use X, and attempts to get control of it, but fails because A already has X. B then waits for the resource to be released. Unfortunately, it will wait forever, since A is not going to release the resource until B is finished. A common example of this scenario occurs in systems which attempt to report an 'out of memory' error but hang instead. The system hangs because the exception report mechanism attempts to allocate memory to hold the error message, but is unable to because the system is already out of memory!

The third problem with circular dependencies occurs during the testing of the final system. There are two general strategies that can be applied to this testing: top-down testing and bottom-up testing. In the first case, the top-most module on the dependency diagram is tested first, with all the lower level modules stubbed out. Then, one of the immediately lower modules is tested with the top-most module. The testing continues in this manner, adding lower-level modules until the entire system is included in the tests. In bottom-up testing, one of the bottom level modules is tested first, and the upper modules are added, one at a time, until the entire system is being tested. In both cases, the testing procedures depend on the assumption that bugs found during testing are most likely to be caused by the last module added to the test system. This assumption can enormously simplify and speed-up the testing process when a large system is being tested.

The problem with circular dependencies is that they do not have a top or a bottom! Therefore, we can not use the top-down, or bottom-up testing procedures. We have to develop special testing procedures for the system. These special procedures will complicate and slow down the testing process. When bugs are found, they will be more difficult to find because we can not assume that the original modules in the system have been completely tested.

In general, removing a circular dependency removes any chance of infinite recursion and simplifies the design. The simpler design avoids some tricky deadlock errors, and makes the testing simpler and quicker.

4.2.2 Circular Dependency Solutions

Due to all the problems with circular dependencies, much effort was devoted to removing them from the system design. In this section we describe some of the original circular dependencies, and the techniques used to remove them.

Figure 2 shows a simplified dependency diagram for an early version of the SR/V RTS system design before the circular dependencies were removed. Note the many circular dependencies. This is much more complex than the new SR RunTime System Dependency Diagram in Figure 1 and the Operating System (OS) Module Dependency Diagram in Figure 3. Taken together Figure 1 and Figure 3 represent most of the complexity of the latest SR/V design. The major improvement in the new design is the removal of most of the circular dependencies.

Figure 2

Early SR/V RTS Dependency Diagram

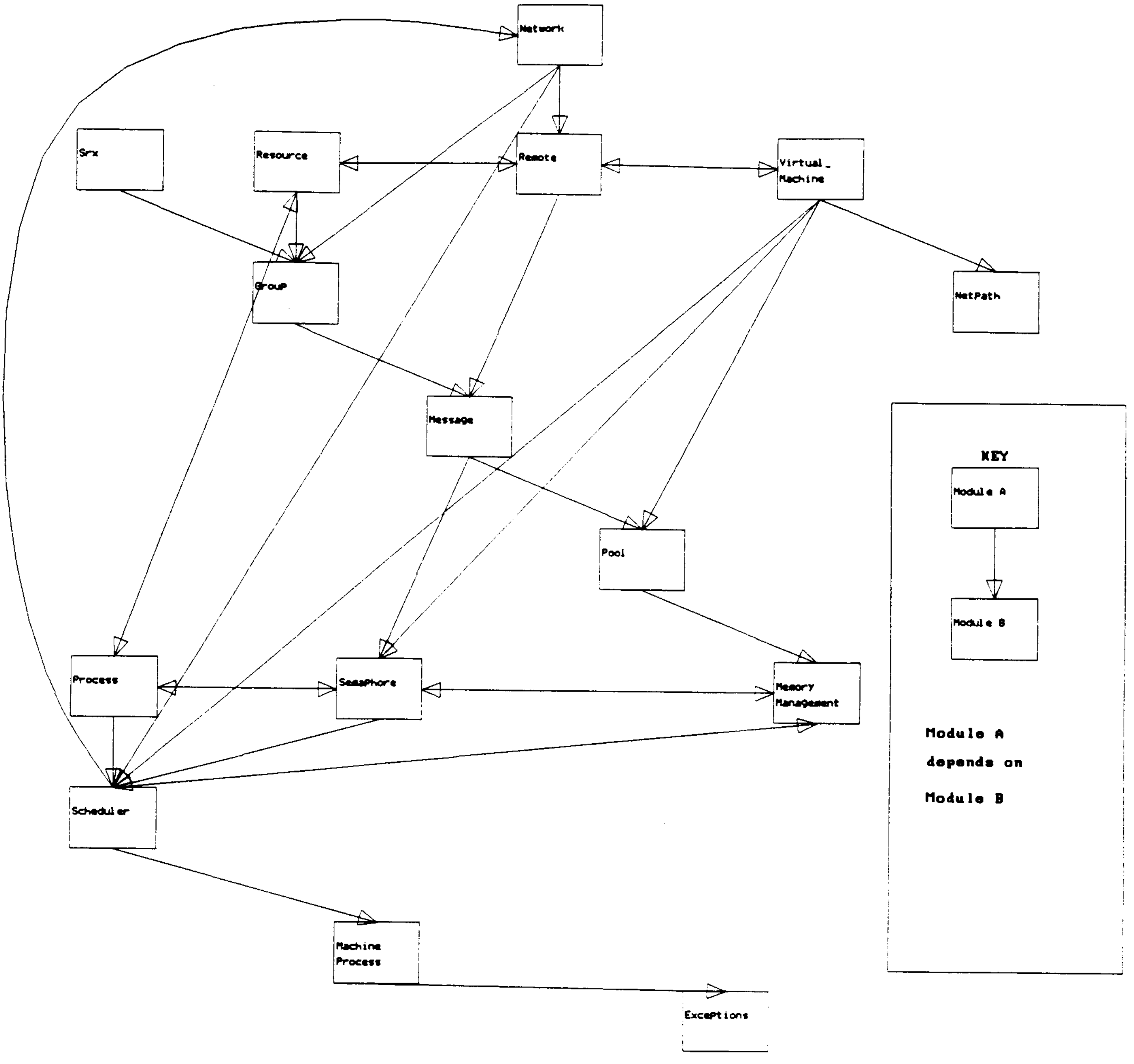
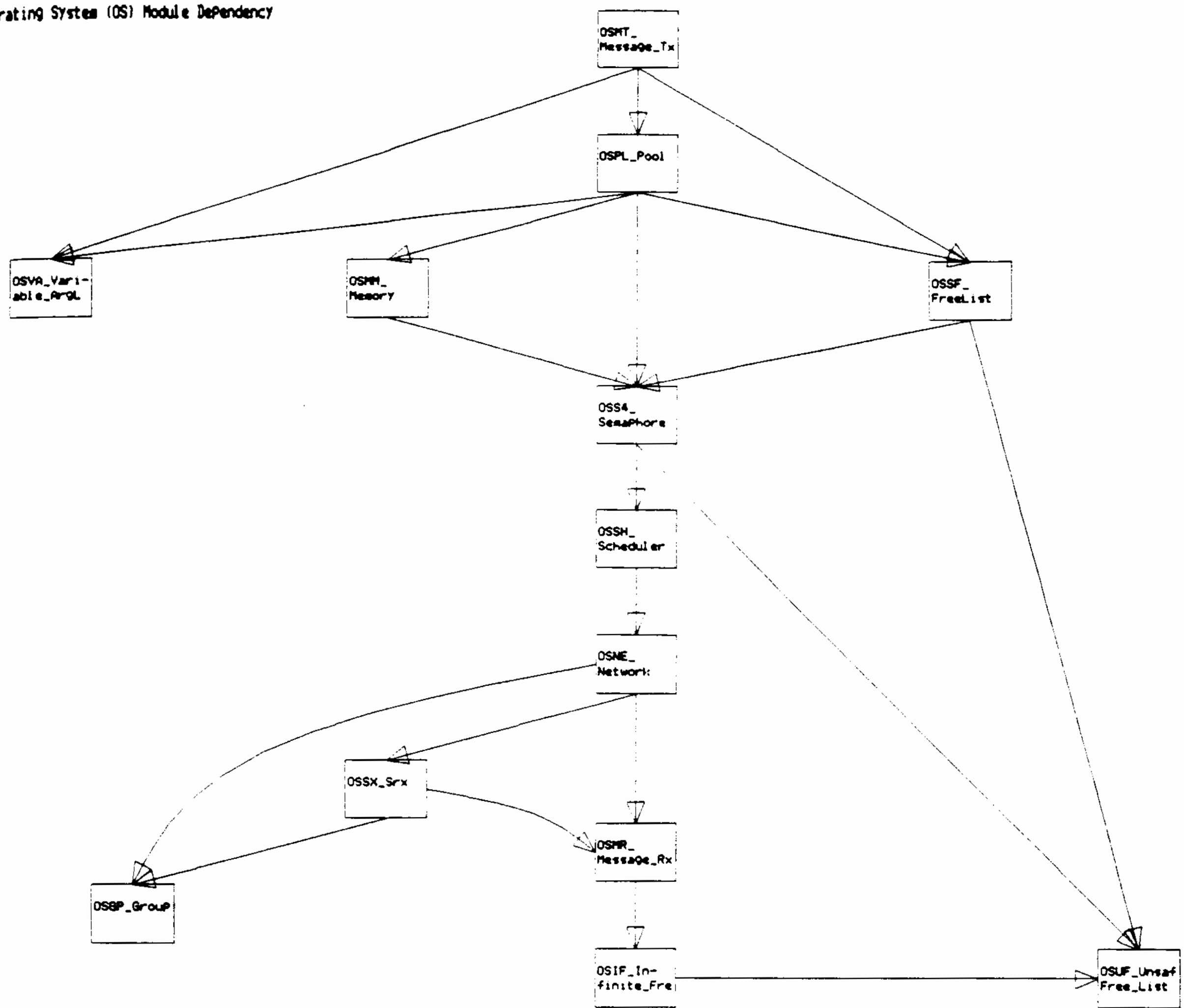


Figure 3

Operating System (OS) Module Dependency



4.2.2.1 Subsystem-Level Modularization

The biggest change to the design occurred when we realized that several of the circular dependencies were caused by the underlying RTS data structure. For example, each resource object has a list of processes associated with it, and each process object has a reference to its resource. In the original design, the resource module calls the process module to delete all process objects on a resource object when the resource object is deleted. Similarly, the process module calls the resource module to remove one process object from the resource object's list when the process is deleted. Thus we have a circular dependency!

Figure 4 shows a picture of the circular dependency between the process and resource modules with the delete procedures and the data structures hidden inside the modules.

Note that the circular dependency is caused by the circularity in the data structure. The software engineering principle of **information hiding** states that data structures should be hidden inside each module. In the original design, this principle is followed perfectly. The resource module hides the resource data structure and the process module hides the process data structure. Unfortunately, the resource data structure depends on the process data structure, and the process data structure depends on the resource data structure. Since each module hides one data structure, the circular dependency in the data structure causes a circularity in the module dependency. In particular, a delete operation on either a process or a resource requires an invocation of an operation from the other module.

Since we could not see an easy way to remove the circularity from the data structure, we decided to limit the effects of the circularity. Using the modularization technique, we extracted the data structure access and list manipulation procedures to another subsystem called the Data Structure (DS) subsystem. Now, all the circular dependencies caused by the data structure are isolated to the DS subsystem. Furthermore, these circular dependencies are all declaration dependencies. Eg. the DS process module depends on the DS resource module to have a resource object declaration, and the DS resource module depends on the DS process module to have a process object declaration. These circular data declaration dependencies are a small problem compared to the circular procedural dependencies.

Figure 5 shows a picture of the new process and resource modules with the delete procedures associated with each module.

As a side effect of this design decision, we noticed that the DS modules shared many of the same list operations. So, we created yet another subsystem called the Generic List (GL) subsystem to hold these list operations. This modularization reduces the amount of duplicate code and makes the remaining code easier to read.

4.2.2.2 Layers of Abstraction

Another kind of circular dependency, where one module encompasses different abstraction layers, can be removed by dividing a module into two layers of abstraction.

In our case, a circular dependency occurs between the Memory and Semaphore modules. The Memory module depends on the Semaphore module to provide semaphores which protect the critical sections in the memory list operations. The Semaphore module depends on the Memory module to provide memory blocks for the semaphore data structures. These data structures must be allocated at run time because the size of the data structure is determined by a run time parameter. Thus we have a circular dependency: Memory -> Semaphore -> Memory.

Figure 4

Circular Process-Resource Dependency

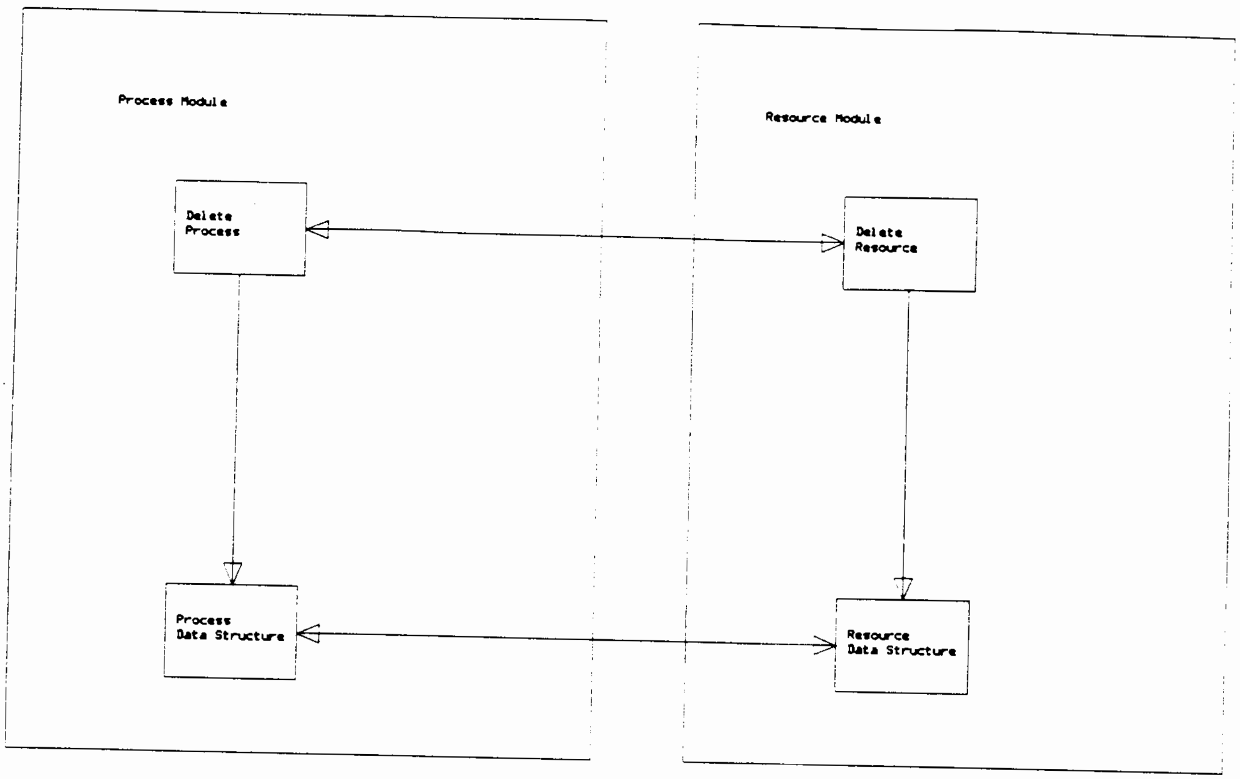
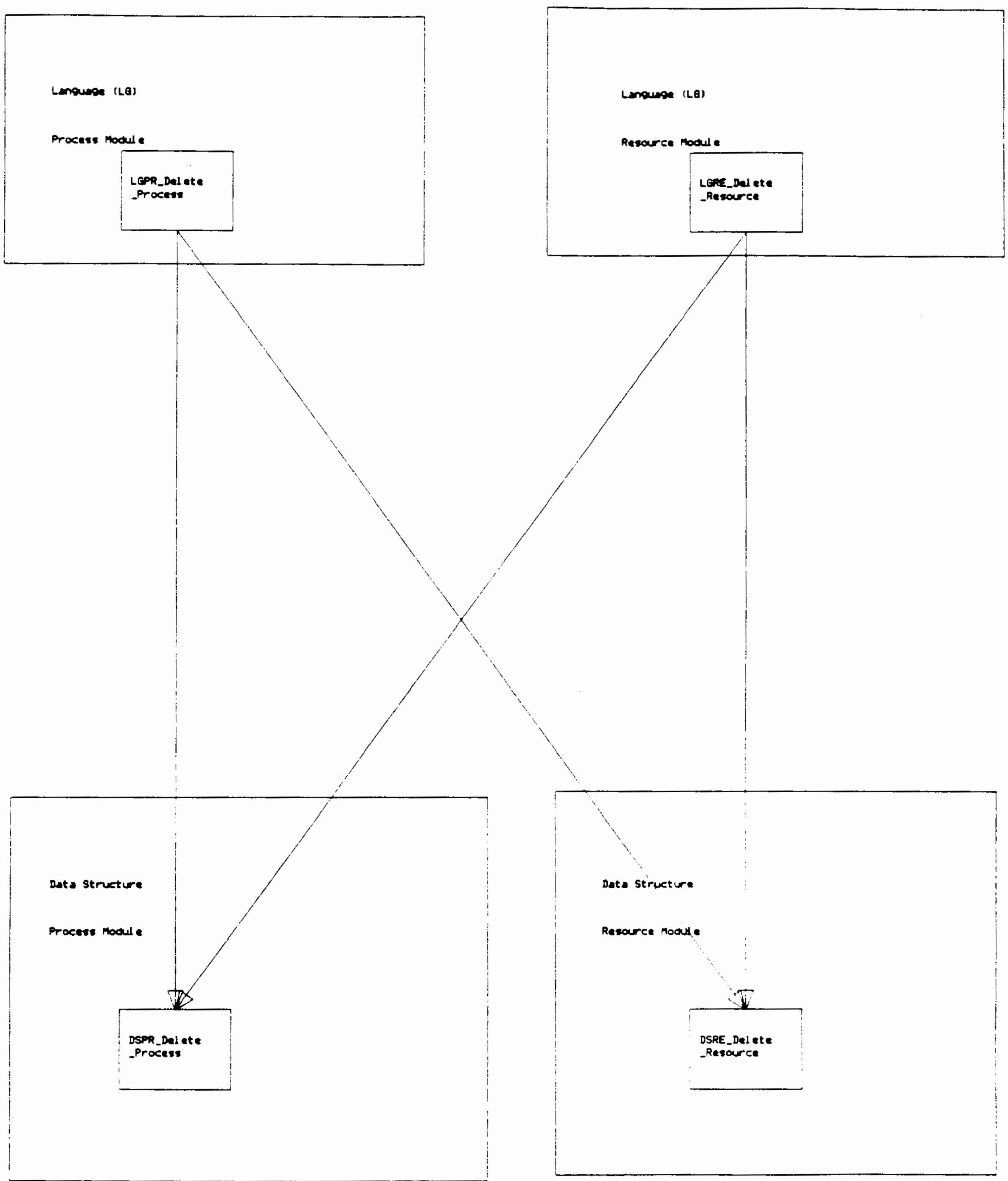


Figure 5

Linear Process-Resource Dependency Diagram



This circular dependency is broken by dividing the Memory module into two smaller modules. The simplest Memory module is called the machine (MC) level Memory module (MCMM_Memory). It uses the V-system memory management routines to allocate and free memory. It reports an error if there is any problem, but it does not keep track of the memory blocks allocated. The more complex Memory module is called the operating system (OS) level Memory module (OSMM_Memory). It uses the MCMM_Memory module to allocate and free memory, and it keeps track of all the memory allocated, with the help of the Semaphore module. The Semaphore module now depends on MCMM_Memory to allocate and free semaphore data structures. We now have a linear dependency: OSMM_Memory -> OSS4_Semaphore -> MCMM_Memory.

This new linear dependency design requires that OSS4_Semaphore keep track of the memory blocks it allocates. This turns out to be very simple because OSS4_Semaphore never really frees any memory blocks, it just re-uses them for other semaphores.

4.2.2.3 Module Splitting

Another kind of circular dependency, where one module performs two or more functions at the same level, can be removed by splitting a module in half. From the outside, the original module appeared to represent a well defined, highly cohesive module. However, after the division, the two new modules were found to have simpler internal designs and, most important, the circular dependency is gone.

The circular dependency involves four modules as follows: Network -> Message -> Semaphore -> Process (Scheduler) -> Network. The Network module is responsible for processing all requests from other VMs. It uses the Message module to read the incoming messages. The Message module uses the Semaphore module to protect the critical sections in the message list operations. The

Semaphore module uses the Process (Scheduler) module to block processes that have blocked on a semaphore and to awaken processes that are woken by a semaphore operation. Finally, the Process (Scheduler) module calls the Network module periodically to read the latest requests from other VMs. This long chain creates a circular dependency.

In fact, this dependency was not found until we started testing. The system hung in an infinite loop! The loop occurs as soon as the system runs out of message blocks. Then the Message module blocks on its message list semaphore, which causes a context switch. The Scheduler calls the Network module to read the incoming requests. Network calls the Message module and it blocks on the message list semaphore, and so on, and so on. . .

The solution to this circular dependency was to divide the Message module into two modules. The Message_Rx module is responsible for receiving (Rx) messages and the Message_Tx module is responsible for transmitting (Tx) messages. Although these two modules depend on each other to share a common message format and a communication protocal, they do not have any direct procedural dependencies. In fact, the internal design of either module can be changed completely without affecting the other module. Therefore, both of the new modules have high cohesion.

Since the two new modules have no procedural dependencies, they break the circular dependency. The new linear dependency is OSMT_Message_Tx -> OSS4_Semaphore -> OSSH_Scheduler -> OSNE_Network -> OSMR_Message_Rx

4.2.2.4 Unresolved Circular Dependencies

There are several circular dependencies that remain in the final RTS design. We keep these circular dependencies for two reasons. Either they can not be removed because of the inherent circular dependency between communications and processes in a distributed system; or, in the case of very small circles, the effort to remove the circular dependency is more work than the benefit gained.

For each circular dependency, we describe the dependency, why it is not removed, and what we did to avoid the problems associated with circular dependencies.

The biggest and most important remaining circular dependency occurs between the Operating System (OS), and Language (LG) subsystems, as shown in Figure 1. The OS subsystem depends on the LG subsystem to execute remote requests received from other VMs, by the OSNE_Network module. The LG subsystem contains both the procedures to implement the requests and the processes (LGPR_Process) which execute the procedures. In turn, the LG subsystem depends on the OS subsystem for memory management, process pools, free lists, variable argument lists, semaphores, process scheduling and message communication. This is a very complicated circular dependency.

We feel the OS - LG circular dependency is rooted in the core design of a distributed, message-passing system. Such a system is built around the intertwined concepts of process and message. Some process operations depend on messages to deliver the operation request, and message receivers depend on the process operations to execute the operations they receive. Furthermore, the VM operations and the Resource operations both depend on messages to deliver their requests and they are invoked by the message receivers. Therefore, the final design reflects a central problem with the underlying concept of intertwining the process and message concepts.

We had some trouble debugging this large circular dependency. In the end, we traced the procedure calls to make sure that there are no procedures which end up calling themselves. We were able to break this procedure circularity by making the message receiver create another process to execute the operation. The process creation was simplified to ensure it could not call the message receiver. Since the operations are invoked from another process, they can call whatever they wish, without creating a procedure circularity.

For the testing of the OS procedures, we have written an entire module of stubs to replace the LG procedures called from the message receiver. This extra code is necessary to isolate the OS subsystem during testing.

There are many circular dependencies in the DS subsystem. Figure 6 shows a picture of the DS dependencies. These are all data type dependencies. They do not cause any control problems. Each module only depends on the other modules to supply it with a type declaration name. There is no possibility for infinite recursion, because there is no executable code in these circular dependencies. For testing, we must ensure that all these data declarations compile without error. Then, we must test the modules and subsystems which use the DS subsystem. Note there is no direct testing of the DS subsystem.

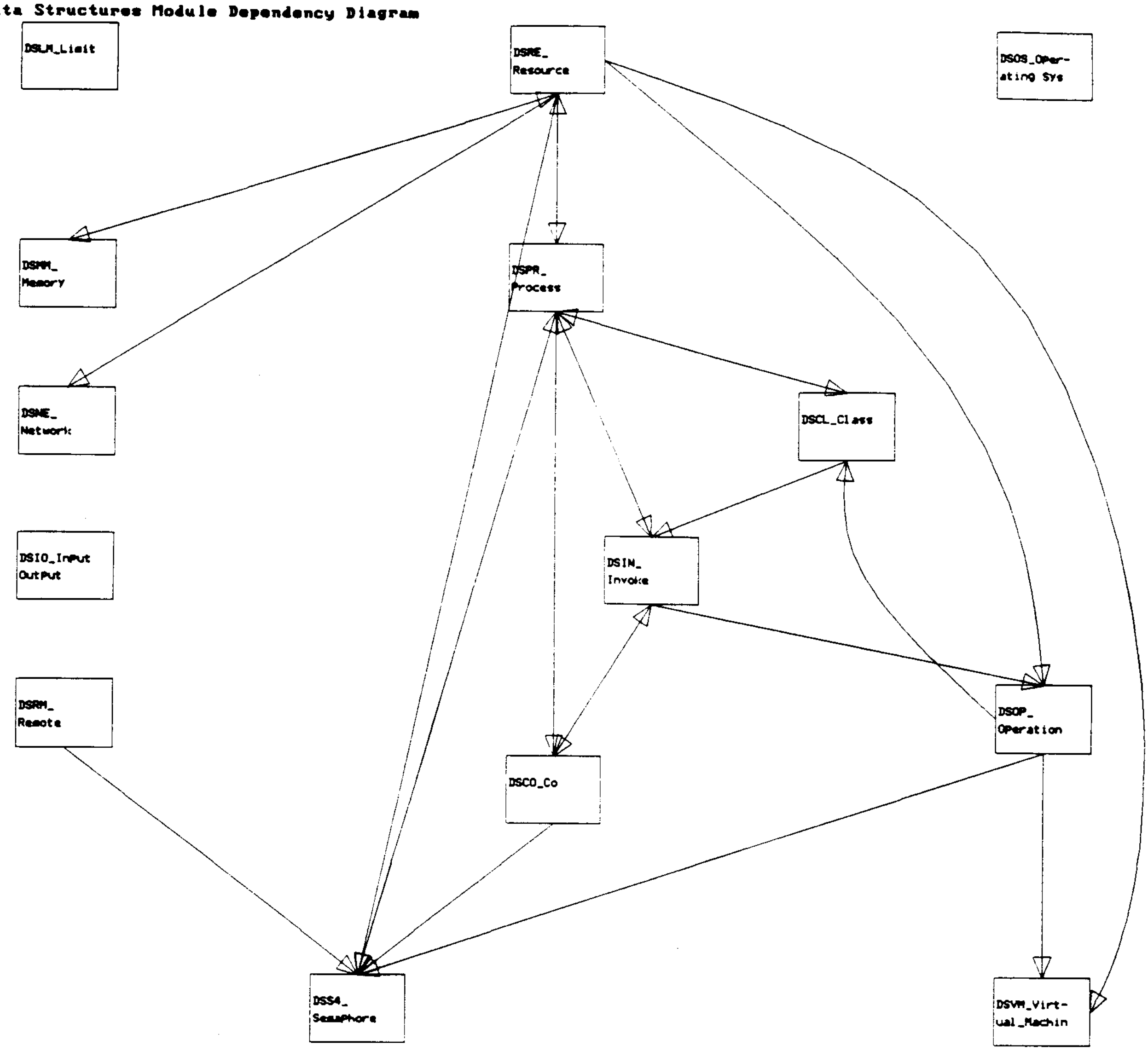
Finally there are two small circular dependencies in the LG subsystem. Figure 7 shows the LG dependencies. The LGIV_Invoke module depends on the LGCO_Concurrent module to manage concurrent invocations. The LGCO_Concurrent module depends on the LGIV_Invoke module to provide the procedure to create and initialize an invocation descriptor. There is no possibility for infinite recursion, and the descriptor creation procedure is easily stubbed out during testing.

The LGVM_Virtual_Machine module depends on the LGRT_Remote_Tx to deliver requests to remote machines. The LGRT_Remote_Tx module depends on the LGVM_Virtual_Machine to retrieve and store information about the remote VM's communication addresses. Again, there is no possibility for infinite recursion and the two LGVM procedures are easily stubbed out during testing.

In summary, there are only a few circular dependencies left in the RTS design. In the worst case, the circular dependency is caused by the interdependency between messages and processes in this design, which is common to many distributed systems. In the other cases, the circular dependencies are small, easily explained, trouble-free and require very little work during testing.

Figure 6

Data Structures Module Dependency Diagram



4.3 Subsystem Design Issues

4.3.1 Language (LG) Subsystem Design

The Language (LG) Subsystem provides the functionality for SR Language-specific concepts, which are too complex to implement with in-line code. For example, the LG subsystem implements Virtual Machines, Resources, and Operations. Almost every module in LG implements an SR concept or statement directly.

The dependencies between LG modules are fairly simple. Most modules only depend on one or two other LG modules. The two exceptions are LGMN_Main which calls almost every other module to initialize the RTS, and LGIV_Invoke which calls several other modules to implement the several different types of invocation.

The LG dependencies on other Subsystem modules are more complex. The LG modules only depend on two or three OS_Operating_System modules, but they often depend on six or seven DS_Data_Structure modules. The reason for the large number of DS modules is that the LG modules often must traverse the RTS data structure to find the information they need. In the course of traversing the data structure, they use the DS descriptors and data access procedures. Most LG modules also use several of the MC_Machine modules. Taken collectively, the LG modules use almost every other module in the RTS. This is not surprising since LG supplies most of the interface to the Generated Code (GC), and the rest of the RTS is written to support that interface.

There are two circular dependencies in the LG Dependency Diagram, shown in Figure 7. Neither of them are cause for concern.

The circular dependency between LGVM_Virtual_Machine and LGRT_Remote_Tx occurs because the LGVM sr_create and sr_destroy procedures need to do sr_remote calls, and the LGRT sr_remote procedure needs to call sr_vm_connect in LGVM if the requested VM's communication address is unknown. Since the sr_vm_connect procedure does not depend on any other LG modules, there is no possibility of recursion or deadlock. We will need a stub for sr_vm_connect during the testing of LGRT_Remote_Tx.

The circular dependency between LGIV_Invoke and LGCO_Concurrent occurs because the LGIV sr_invoke procedure depends on LGCO to implement concurrent invocations, and LGCO must sometimes make a copy of an invocation descriptor, which it does by calling sr_dup_inv in LGIV. The sr_dup_inv procedure has no dependencies other than the obvious need to use the invocation descriptor. SR_dup_inv is a simple copy procedure. There is no possibility of recursion or deadlock. We will need a stub for sr_dup_inv during the testing of LGCO_Concurrent.

The internal design of some of the LG modules is quite complex. In particular the LGIV_Invoke and the LGIN_Input_Op modules must distinguish between many different types of invocations and implement each type as efficiently as possible. These design issues are described in greater detail in [ANDR86].

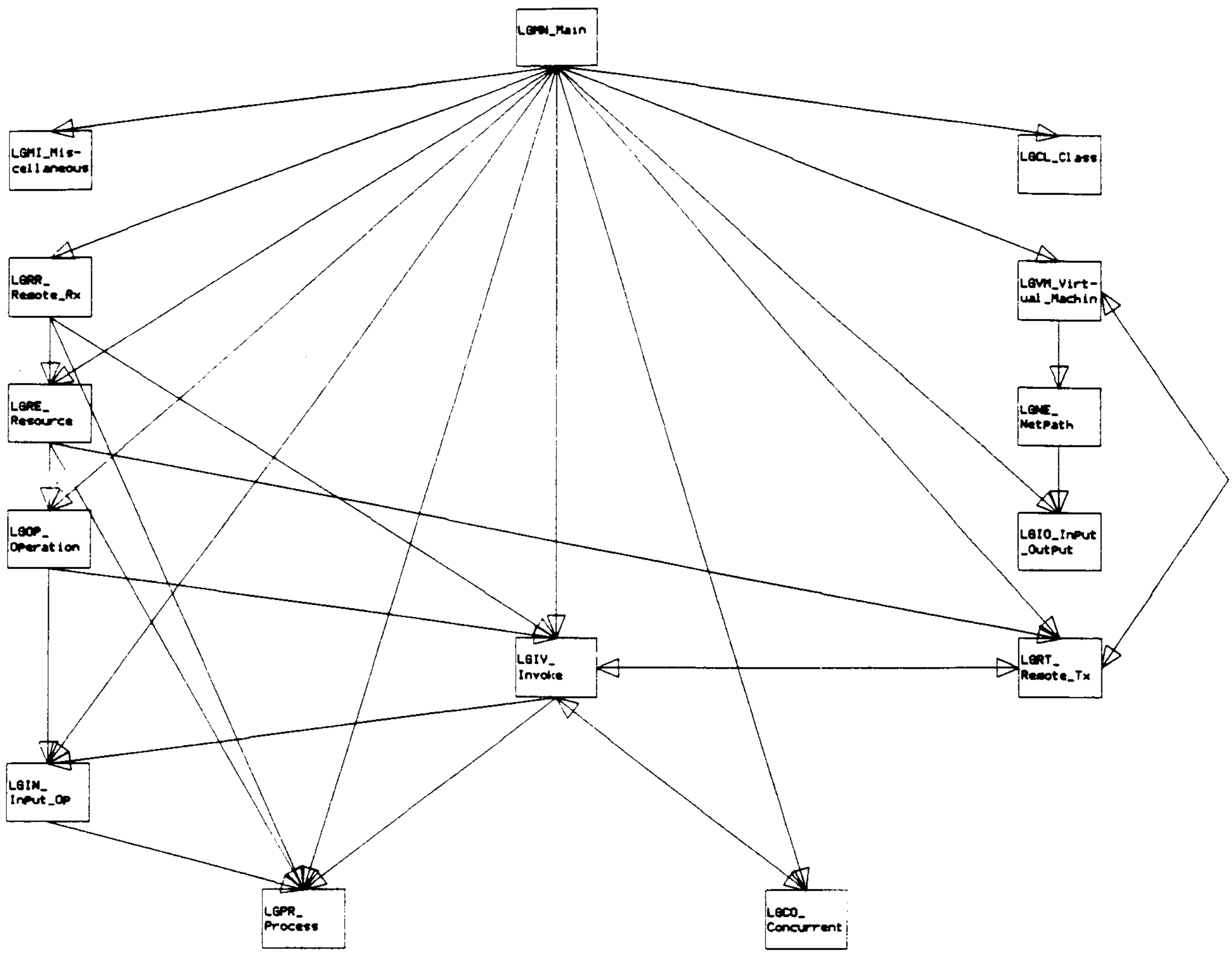
4.3.2 Operating System (OS) Subsystem Design

The OS Subsystem shown in Figure 3 provides the functionality that is normally associated with an Operating System. For example, it supplies Message passing, Memory Management, a Network interface, and SR Process Scheduling.

The OS Subsystem is quite complex. There are over a dozen modules and many of these modules depend on ten or more other modules. To complicate the design further, this subsystem seems to have a tendency to develop circular dependencies. Fortunately, we have managed to break most of the circular dependencies. However, there is one circular dependency left.

Figure 7

Language (LG) Module Dependency Diagram



The circular dependency that is left is 'caused' by the OSNE_Network module's dependency on several LG_Language modules. This particular dependency seems to be unavoidable. Section 4.2.2.4 explains this dependency in greater detail.

The other modules are fairly simple when regarded in isolation. There are several different types of Free Lists to manage the lists of descriptors. There are the OS-type modules like the Message modules, the OSSH_Scheduler module, the OSNE_Network module, and OSS4_Semaphore module. There are also several modules which are peculiar to SR or the V-system implementation. The OSSX_Srx module is peculiar to SR. It ensures that each VM number is unique. The OSPL_Pool module is peculiar to the V-system implementation. It supplies a pool of V-system processes to perform V-system blocking operations. Although the connections between these modules are complex, each module is straightforward.

4.3.3 Machine (MC) Subsystem Design

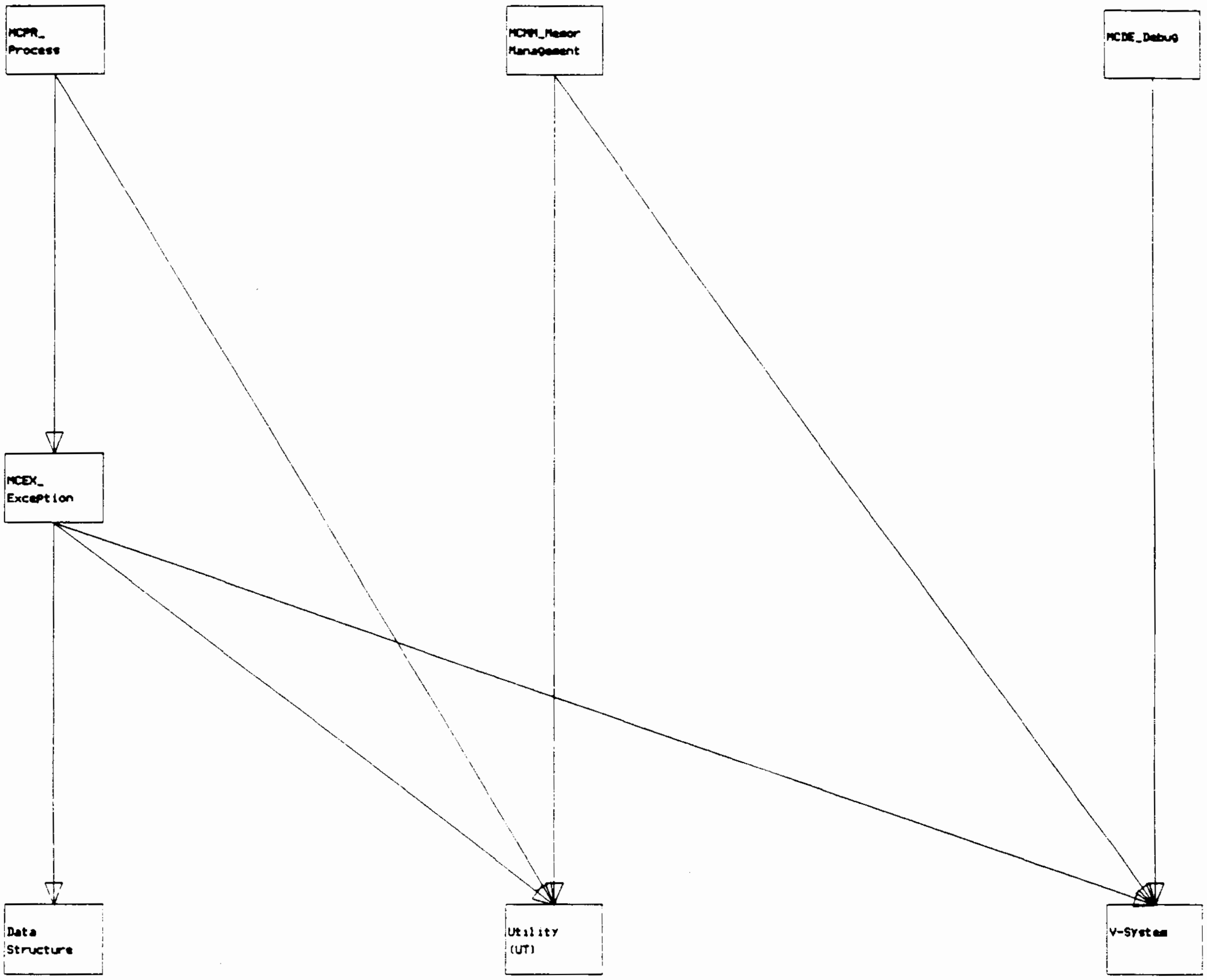
The Machine Subsystem is the lowest level of the RTS. Every other subsystem in the RTS depends on it, either directly or indirectly. Figure 8 shows the dependencies.

This subsystem is a mixed collection of modules. There are two main reasons for including modules in this subsystem. Some modules are included because they are used by almost every other module in the RTS. Eg. the MCDE_Debug module. Others are included because they hide machine-specific details. Eg. the MCPR_Process module. In general, modules are put in this subsystem because they belong at the bottom of the dependency diagram.

Most of the Machine subsystem design is straightforward. Each of the modules supplies a few procedures to manipulate their simple module.

Figure 8

Machine (MC) Module Dependency Diagram



4.3.4 Data Structures (DS) Subsystem Design

In the RTS design, there is one RTS for each Virtual Machine (VM). Each RTS implements a very complicated data structure to keep track of all the SR entities on its VM, and the relations between those entities. It is the purpose of the Data Structure Subsystem to implement the entity descriptors (data types) and supply primitive procedures to allow higher-level modules to access the data in the descriptors.

In Object-Oriented Programming Systems (OOPS) terminology, each DS module is a 'server' class. Since the DS modules only supply data types and data access procedures, we call the DS modules **data servers**. For each data server, there is one higher-level module in the OS_Operating_System or LG_Language subsystems which has the same module name, but a different prefix. We call the corresponding higher-level module, the **function server**, because it implements the corresponding functions. For example, the server class DSS4_Semaphore module implements the semaphore data type and one data access procedure: dss4_sem_count. OSS4_Semaphore is the corresponding function server which implements the standard semaphore functions: create, kill, P, and V.

The DS_Data_Structures subsystem is designed to let all modules access the RTS data structure through the interface specified by the module description. However, the function server for a DS module may manipulate any fields in the DS module, even those that are 'hidden'. **Hidden** fields are not specified as part of the interface. An example of a hidden field is the blocked field in the semaphore descriptor which is a list of the processes blocked on the semaphore. The OSS4_Semaphore function server needs to access the blocked field to implement the P and V operations. The need of the function server such as OSS4_Semaphore to access the hidden fields of a data server, reflects the tight relationship between the data server and function server pairs. Unfortunately, there is no way to document this relationship in the C code other than to use the same root name on the code files. In an OOPS programming language, we could reflect this relationship by having the function server inherit the data server, and redefine the interface.

Much of the complexity of the Data Structure subsystem originates from two requirements. The SR entities must be created dynamically and the many inter-entity relationships must be stored in the data structure in order to perform the operations efficiently. For example, in the case of the resource and process entities, we have a bidirectional relationship. Each resource may contain any number of processes, and each process must have an owner resource. Both relationships must be stored if we are to perform both process and resource operations efficiently.

To satisfy the dynamic requirements, the RTS implements descriptor records which exist in main memory. To satisfy the need to keep track of relationships between entities, each descriptor record contains pointers to other entities which are related to it. For example, the resource instance descriptor has a pointer to a list of processes in the resource and the process descriptor has a pointer to the 'owner' resource of the process.

The DS subsystem is essentially a very primitive DBMS. It is responsible for storing all the data and data relations necessary for the operation of a VM.

4.3.5 Generic Lists (GL) Subsystem Design

Many of the SR entities are implemented using data structures called descriptors, eg. the resource and process descriptors. These descriptors are often stored in linked lists of various types, because of the SR requirement that the entities be created and destroyed dynamically. Since these list types have very little to do with the type of descriptor they contain, it is appropriate that the lists are implemented separately from the SR entities. For example, the resource descriptor is implemented by the LGRE_Resource module, but it uses a linked list which is implemented by the GLLL_Linked_List module. The Generic Lists Subsystem has been created to implement modules for all the list types required by the RTS.

This subsystem has very few dependencies because it is usually only working with pointer fields. It initializes pointer fields, and assigns one field to another. GL_Generic_Lists does depend on MC_Machine for some generic data type definitions.

All of the instances of Generic Lists (GL) modules are implemented using standard list manipulation algorithms. Therefore, this section merely describes some implementation techniques common to all the modules which affect the design and use of these modules.

Each instance of the Generic Lists (GL) module defines its own data type. However, this is little more than a syntactic convention. In fact, the procedures in these modules can work with any C record structure. This works because C has very loose type checking and all the GL procedures are implemented as #define statements.

The #define statements are processed by the C preprocessor. In essence, the GL procedures, implemented by #define statements, are 'invoked' before the code is compiled. Therefore, they can accept parameters containing C types, and C field names. These parameters allow the GL procedures to be more general than if they were implemented with the standard C functions.

Since all the modules in the GL subsystem are working on lists, they tend to supply very similiar procedures. To make this similiarity explicit, we have used the following standard procedure names:

create_list	- Create a list and initialize it.
is_empty_list	- Determine if a list is empty. Return TRUE for an empty list, and FALSE otherwise.
pop	- Remove the node from the front of the list and

	return a pointer to it.
chop	- Remove a node from the end of the list and return a pointer to it.
delete	- Remove the given node from the list. The node may be anywhere in the list.
push	- Add a node to the front of the list.
append	- Add a node to the end of the list.
append_list	- Add a new list to the end of the old list.
insert	- Add a node after the given node in the list. The given node may be anywhere in the list.

Not all of the above procedures are implemented for all of the GL modules.

The GL subsystem could be simplified if it was implemented in a language which supports generic modules, such as Ada, Miranda, Modula-3, or CLU. Then there would be no need for #define statements, or the passing of type names and field names. We would create a list of type X by creating an instance of a generic list module. The procedures for the list would be defined to work on the elements of type X. Therefore, they would not require the type names and field names as parameters.

4.4 Module Design Issues

We now explain the module design issues, module by module. The module designs use the principles of information hiding, abstraction and modularity. This allows us to concentrate on the module interfaces and some of the more interesting implementation details, without having to explain the internal design of every module. The following descriptions are ordered from top to bottom of the dependency diagrams:

4.4.1 LGMN_Main

This module initializes all the modules in the RTS. If this is the first RTS then it creates the main resource. Otherwise, it just waits for requests from remote VMs.

This module starts the RTS on each VM. The first RTS is invoked from the operating system command-line. This initial invocation is the **program startup** which may include program parameters. These parameters are ignored by the RTS and passed to the SR program. Every subsequent invocation is a **VM startup** which is the result of a VM create statement. In this case, all the parameters are used for the RTS initialization.

4.4.2 LGVM_Virtual_Machine

This module implements the Virtual Machine (VM) module. This module supplies the operations to create and destroy virtual machines. Each virtual machine has its own memory space, communication address, and RTS. Once a virtual machine is created, then resource instances may be started on it.

The SR concept of VMs is described further in [ANDR86].

4.4.3 Other LG Modules

The remaining LG module interfaces are unchanged from their UNIX implementation. Some minor, uninteresting changes were made to conform to changes in the operation of the OS procedures.

4.4.4 Process Modules
(LGPR_Process, OSSH_Scheduler, MCPR_Process, DSPR_Process)

The process modules are layered one on top of each other. Each level depends on the lower levels, and adds its own level of functionality.

The LGPR_Process module implements SR processes. SR processes are very lightweight with no time-slicing between processes. This means that an SR process will monopolize the cpu until it blocks itself. More information about processes and the standard operations can be found in any operating systems text.

The OSSH_Scheduler module controls the processor. It assigns the processor to the ready process which has been waiting the longest.

The MCPR_Process module implements the process module at the machine level. This includes creating a process context, changing contexts, and context error checking. These operations can only be done at the machine level because they manipulate machine registers and the process stack.

The DSPR_Process module implements the data structures and data access functions for the process data types. These data types support the implementation of SR processes.

4.4.5 OSMT_Message_Tx & OSMR_Message_Rx Modules

The OSMT_Message_Tx module implements the message transmit operations with the V-system Send operation.

The OSMR_Message_Rx module implements the message receive operations with the appropriate V-system operations: Receive, and Reply.

4.4.6 OSPL_Pool

The OSPL_Pool module implements a process pool module. This module is implemented to accommodate the V-system blocking operations. In the V-system, if you want to execute a blocking operation without blocking the current process, then you must put the code for the blocking operation in another process, called a helper process, and send a message to the helper process.

The message contains the blocking operation code and any parameters required for the operation.

In the V-system implementation of SR, we follow this V-system model of one main process, and many helper processes. However, the main process is also receiving messages from other VMs as well as the helper processes. Plus, there are different types of helper processes. There are helper processes to perform IO operations, processes for Message operations, and processes for VM operations.

This module simplifies the implementation by containing all the code to create a V-system process pool, report process pool errors, and synchronize with the other in-coming messages.

This module supplies the operations to communicate with process pools, and the operations used to implement the process pools.

4.4.7 OSNE_Network

The OSNE_Network module implements a network interface. This module is responsible for receiving all messages from the network and calling the appropriate module to perform the requested operations.

This module is a 'design problem'. It is called from the OSSH_Scheduler module, which is in the middle of the OS Dependency Diagram, but it calls several of the LG_Language modules, which depend on the OS subsystem. Unfortunately, there does not seem to be any way to avoid this circular dependency.

This circular dependency is unavoidable because OSNE must be called from OSSH_Scheduler and it must call the LG modules. Before we go any further, we will explain why the OSSH_Scheduler must call OSNE and why OSNE must call the LG modules.

The OSSH_Scheduler module is responsible for scheduling tasks. Since the OSNE module must periodically check for messages on the network, OSSH_Scheduler is responsible for scheduling OSNE periodically. Therefore, OSSH_Scheduler must call OSNE_Network.

The OSNE module must call the LG_Language modules because OSNE is responsible for ensuring the operations requested by the in-coming messages are executed. Unfortunately, all these operations are implemented in the LG_Language subsystem. Therefore, OSNE must call the LG_Language modules.

Fortunately, the circular dependency is not as serious as it appears. OSNE spawns SR processes to perform most of the message operations. Therefore, very little of the LG_Language code is actually executed when OSNE calls the LG_Language modules. Furthermore, the code that is executed never calls OSNE either directly or indirectly. Therefore, we do not have to worry about infinite recursion.

However, this dependency does make testing more difficult. OSNE can not be completely tested until the LG_Language subsystem is working, but it must be working in order to test the OS_Operating_System subsystem. We suggest that a special test program with stubbed procedures be set up to test the OSNE_Network module by itself. Then it can be used with confidence in the OS_Operating_System tests.

4.4.8 OSSX_Srx

The OSSX_Srx module is responsible for supplying a unique VM number for each new VM.

Currently this module is implemented as a separate V-system process. This implementation affects the interface. This module is initialized by starting the process rather than by calling a procedure, and operations are 'called' by sending messages to the process. Therefore, some of the 'procedures' listed in the Invocation Interface have the word 'Message' appended to indicate they are really messages, not procedures.

4.4.9 OSGP_Group

The OSGP_Group module implements the messages to process groups. There is a very close dependency on the VM data structures because the VM modules are the only modules that use process groups.

4.5 Management Issues

Management issues appear in large projects such as this SR RTS implementation. When systems become this large it is difficult to measure the size of the system, and thus to estimate how much time and effort is required. If we can't measure the size of the system how do we know if it will take one year or five years to complete? Without any estimate of the time required, how can we tell if we will ever finish? Estimating the size of a large system is necessary to ensure successful completion.

The module interface documents supply us with the raw data necessary to estimate the size of the system. We now have several methods to develop accurate estimates of the system size, complexity, and time and effort needed to complete it. Each of these methods will be more accurate than estimates made without the benefit of the module interface documents, because these methods are based on a better understanding of the system.

An informal method of estimating is to study each module interface, and, based on our experience, estimate the lines of code necessary to implement the module. This becomes our estimate of module complexity. Based on the complexity, we can estimate the time to code and test this module. The system size is equal to the sum of the lines of code for each module. The total time to implement is equal to the sum of the time needed for each module.

A more formal method of estimating involves measuring the complexity of the module interface and the module dependencies. The module interface complexity is measured by counting the number of procedures, procedure parameters, data types, and data items in the interface. The module dependency complexity is measured by counting the same items in the module's dependency list. Based on these numbers, a measure of complexity is calculated. This complexity measurement can then be used to estimate the system size and implementation time.

For example, Henry and Selig [HeSe90] present a metric of design complexity called information flow. They tested the information flow metric on the documented design and implementation of projects created by University students in a Software Engineering class. Their results indicate that the information flow metric is a good predictor of a project's size and complexity.

With either the informal or the formal method, experience will lead to better estimates. For example, the first module implemented usually takes much longer to implement than estimated. However, now that we have measurements of the module's complexity, we have a chance to improve our estimates for the remaining modules. To determine the problem with the first module's estimate we must re-evaluate the module's complexity. Is it more complex than originally designed? Have more procedures, parameters, types and data items been added to the interface or the dependency list? If so, then the original estimate is not wrong; the original design is wrong. In this case, we should re-examine the remaining module designs to see if they too will have to be changed. If the module's complexity has not changed from the original design, then the estimate is wrong. In this case, we should change the estimates for all the remaining modules. In either case, we will quickly gain better measurements of system complexity and thus better estimates of time and effort required.

In general, the module interface documents help us to understand the system better, make better system design decisions, better module design decisions, and manage the project better. However, it does require that we think carefully about our design before we code it, and it requires that we document the design before we code it!

CHAPTER 5

OPERATING SYSTEM SUBSYSTEM DESIGN

5.1 Introduction

This chapter discusses the Operating System (OS) subsystem design shown in Figure 3. The discussion is devoted mainly to our implementation of the OS subsystem on V-system. However, we attempt to generalize the issues and solutions to all applications or operating systems. We first discuss the major design decisions in the design of the OS subsystem. Then, we discuss the remaining problems and suggested solutions.

5.2 Design Decisions

This section describes the mapping of SR entities onto the Operating System concepts. The most important mappings are the VM, process, input/output and communication mappings. Section 3.2 explains these SR entities in greater detail.

5.2.1 SR VM (Virtual Machine) Mapping

The V-system implementation of VMs is similiar to the UNIX implementation. In the UNIX implementation, each VM maps to one UNIX process. On the V-system (abbreviated as V), each VM maps to a V team. This means communication between all VMs is identical even if the VMs are on the same physical machine, because V hides the physical location of the teams.

Communication would be faster for VMs on the same machine, if we had all VMs on the same machine in the same V team, but we feel this would greatly complicate communication for a relatively small increase in the overall speed of the SR program. If the programmer wants to increase the speed of his program, he can reduce the number of VMs, so there is only one VM per physical machine.

5.2.2 SR Process Mapping

Within the VM team, there is one V process which controls all the SR processes and the context switching between them. This is the same design as the UNIX implementation. We choose to map all SR processes within a VM to one V process, because it is faster than the V process management. In particular we are concerned with the time to perform a context switch. The RTS implementation of context switching is faster than the V context switching. Within both a V-system team and an SR VM, a context switch occurs when a process voluntarily blocks itself, usually on a communication request. The V-system communication primitives are synchronous Send(), Receive(), and Reply(), as described in 3.3. The SR communication statements are call, send, proc, and in, as described in 3.2.

To compare the SR context switch with the V-system context switch, we estimated the performance of each SR communication pair in the two implementations. Appendix A has a complete description of the performance estimates and the method of estimating. Our main finding is that communication using SR process managment is faster between two resources in the same VM, but slower between two resources in different VMs on different machines. The following table shows, for each SR communication pair, the estimated performance using SR process management, and the estimated performance using V-system process management, for both local and remote communication:

SR communic.	Estimated Performance (context sw + overhead)			
	LOCAL		REMOTE (different machines)	
	SR process	V process	SR process	V process
send-proc	1.54 ms	2.40 ms	9.43 ms	7.79 ms
send-in	1.56	2.55	9.30	7.66
call-proc	1.11	1.11	9.00	7.36
call-in	1.85	2.55	9.30	7.66

There is a tradeoff between SR process management and V-system process management as SR has faster local communication and V-system has faster remote communication. We decided on fast local communication, because we believe there is much more local communication in most programs than remote communication. In fact, based on the above estimates, a program using SR process management will be faster than one using V-system process management unless there are 15 times more remote communications than local communications.

5.2.3 Input/Output

An implication of the decision to put all the SR processes into one V process is that we can never invoke a blocking V-system call from this V process. If we did, then all the SR processes on the VM would also be blocked. Instead, we create a pool of V helper processes which perform all the blocking V-system calls. Any SR code which requests a blocking operation is translated into a request to the appropriate V helper process. The V helper process executes the blocking operation, and blocks waiting for the operation to finish. When the blocking operation is finished, the helper process informs the main V process. Meanwhile, the main V process is free to continue executing other SR processes.

The input-output (IO) operations are affected by this decision to use V helper processes. All IO operations are blocking operations and therefore must use the pool of V helper processes. In addition, the IO data structures in the V-system are quite different from the UNIX IO data structures, because of their use of the V-system's message-based kernel. Therefore, all the SR IO functions in the V-system implementation are different from the UNIX implementation.

5.2.4 SR Communication Mapping

Communication between resources on the same VM remains the same as in the UNIX implementation of SR, since resources are implemented the same as in the UNIX implementation. All communication is accomplished through operation invocation and operation implementation. Invocations of operations implemented within the VM are optimized to avoid the use of the slow inter-team communication facilities. Invocations between VMs must use V-system primitives because they are the only means of communication between teams in the V-system.

There are two major problems with communication between resources on different VMs. First, there is the need to avoid blocking V calls from the main V process, as described in 5.2.3. To solve this problem, each VM maintains a pool of V processes called invoke processes which execute the *Send* primitives, and one V process called the receive process which executes the *Receive* primitive to receive messages from other VMs. Second, V-system communication is done through synchronous, blocking *Send* messages, but the SR **send** invocation is asynchronous and non-blocking. To implement the **send** invocation, we use an invoke process to send the message to the remove VM. When the receiver process receives a **send** invocation, it creates an SR process to perform the operation, and immediately replies to the invocation so that the invoker is only blocked as long as it takes to ensure the invocation is started.

5.3 Remaining Problems

5.3.1 The Need for Distributed Data Structures

In SR, executing a locate statement associates a machine number with the system-defined machine name (p 24 of Andr87). This machine number can then be used anywhere else in the code to specify the location of a new VM.

The SR language stores the following information about each physical machine in use by the SR program: machine identifier, name, and communication address. This information is needed to create a VM on the machine. Since VMs may be created from any existing VM, this information must be available to all VMs. This creates the need for a data structure shared between VMs on separate machines. In other words, we need a shared data structure. This can be implemented in two ways: a centralized server process which manages the data structure, or a distributed data structure which is updated using a distributed transaction manager.

The problem with the current implementation is that it uses a central server process to store the physical machine information. This central process is vulnerable to processor failure. If the processor it is running on crashes, or becomes separated from the network, the process and all the machine information is lost. The rest of the SR program will fail if it requires access to that information. Since this design is part of the SR RTS, the SR programmer can do very little to protect himself from processor failure. This is a great weakness in a distributed programming language. One of the great advantages of distributed programs is their ability to survive processor crashes, but, in this case, the implementation used to achieve the distribution advantages is itself vulnerable to processor crashes!

A distributed transaction manager design [CER84] would be better than the current design. In a distributed design, we could store the machine information in every VM. Then, whenever a locate statement is executed, we could broadcast the information to every VM. Since every locate statement adds information to the data structure, the distributed transaction management can be very simple.

However, the SR language does require that an unique machine number be assigned to each machine. Currently these numbers are assigned in sequential order. In a distributed design, the machine numbers could be mapped to the machine's network address which is guaranteed to be unique, but unlikely to be sequential.

The distributed transaction manager design may be more work to implement but we believe it is required if SR programs are to exploit all the advantages of a network.

5.3.2 Time-Slicing

The lack of time-slicing in the SR RTS implementation is a serious problem. Without time-slicing, we can not take full advantage of the concurrency offered by a network of processors. The RTS design can be optimized to improve the amount of concurrency but without time-slicing there will always be problems of one SR process hogging the processor on machine A, while other processors go idle waiting for information from other processes on machine A. Without time-slicing, the SR program designer trying to design concurrency into her program will have to understand the SR RTS before she can achieve her goals.

In the current RTS implementation, the slow down occurs when one VM makes a series of remote requests to other VMs, which are blocked waiting for requests:

The main V process sends request messages to remote teams.

- these requests are translated into V Replys to helper processes.

The main V process on the local team continues to execute until it blocks itself.

Now, and only now, are the V helper processes able to execute and send their messages to other V teams.

The requests are executed on the other V teams.

When the reply messages are sent back to the helper processes, the messages will not be processed until the main V process blocks itself again.

The reply messages are processed by the helper process, and sent back to the main V process.

The main V process will not receive the message until it blocks itself, yet again.

The overall effect is that there are many places where a remote request message may be delayed. Each of these delays reduces the concurrency of the program, and thus the speed of program execution.

Without time-slicing or interrupts, it is impossible to ensure prompt service of the messages, and impossible to guarantee the concurrency which gives us the speed advantage of distributed programs. Unfortunately, time-slicing is a complex concept to implement and imposes a high performance penalty.

We do not see any great solution to this dilemma. We do believe that time-slicing or some such processor sharing scheme is needed in order to take full advantage of concurrency.

5.4 Environment Issues

5.4.1 V-system communication primitives

The V-system communication primitives have a very important influence on our SR RTS design. (section 3.3 has a description of these primitives.) They are the motivation for implementing SR on the V-system. They are much faster and simpler than the socket mechanism used in the UNIX operating system. (Appendix A gives the precise performance figures.)

However, there are a few disadvantages to using the V communication primitives. They are not quite as simple as they first appear, and there are circumstances where we do not get the speed increase that we expect.

The V communication primitives can be complicated to use since there are several variations on the basic *Send, Receive, Reply* primitives. The SR RTS must be able to send a message of any size. This requirement is not efficiently supported on the V-system. If the program is to get the full advantage of V message speed then the V programmer must write his own functions to determine, based on the size of the message, which communication primitive is most appropriate. We believe this function should be implemented in the V-system library since it is useful in many applications.

Since the *Send* and *Receive* primitives are synchronous, the asynchronous communications are difficult to implement and they are a slow. Any program which does asynchronous communication must either create a helper process for every async message, or, as we do in our implementation, keep a pool of helper processes to perform all the message communication with other V teams. This adds a level of complexity to all inter-team communication. It also adds a small amount to the communication time since there is an extra intra-team pair of messages between the main V process and the helper V process. There may be a further decrease in the real communication time. In our implementation, a message can not be sent until there is a helper process available. If the system is busy and all the helper processes are busy then a message will be blocked until a helper process becomes available.

As usual in a message-based operating system, the V-system process concept is closely tied to the communication concept. In V-system, these two concepts work together to make the message communication very fast.

5.4.2 V-system and SR Missing Information

In porting the RTS from UNIX to V-system, we ran into many small problems. We believe these problems to be symptomatic of the current UNIX environment. There are many systems with UNIX-like interfaces which, although similiar, are not quite the same. System developers never find all the these small differences until they reach the implementation stage.

Here we give an indication of the annoying 'small' differences between the SUN UNIX system and the V-system. The V-system requires a blank space between every operand and operator on the command line, which is not required in the SUN UNIX system. Eg. the command

```
copy file1 file2 >file3
```

is invalid on the V-system because there is no space between the '>' operator and the file3 operand.

The V-system also requires a blank line at the end of .h include file. If the blank line is not there, the C compiler will issue typical uncomprehensible C error messages.

Although these problems may sound trivial and inconsequential, each occurrence of one of these problems may take hours, and sometimes days to fix. The only solution is to continue the drive toward standards, and document the idiosyncrasies of the new systems we build. At least that will allow future implementors to find the problems faster. With the current systems which skimp on the documentation of details, the only way to solve problems is by trial and error, or appeal to a guru.

5.5 Summary - Minimal Operating System Requirements

In porting the SR RTS from UNIX to V-system, we have seen that the operating system (OS) has a major effect on the design of a distributed language RTS. However, despite wide differences between OSs, the OS pecularities can still be hidden in a few key modules of the RTS: Process, Communication, Memory, and Input/Output.

Despite our ability to hide OS pecularities, there is still a list of OS requirements that the RTS depends on in order to implement the SR language. These requirements and some desirable OS features are summarized in the following list:

For Processes:

- create process
- delete process
- fast context switching
- Desirable: time slicing. This feature would improve the level of concurrency in an SR program.

Communication

- send and receive variable size messages
- Desirable: asynchronous messages. This feature would make the implementation easier.

Memory Management

- standard, variable-size memory allocation and deallocation
- Desirable: multiple processes in one address space. This feature makes implementation easier if the OS uses synchronous communication, as

V-system does.

Input/Output

- input and output operations to standard input, standard output, standard error, and files.

Machine Addresses

- Machine names. This feature is necessary for the implementation of the SR **locate** statement.

- Desirable: unique machine address numbers. This feature would ease the implementation of a distributed, shared data structure.

CHAPTER 6

CONCLUSION

This thesis describes and documents the application of software engineering techniques to the design of a distributed programming language. We found the concepts of modularization and abstraction to be very useful in the design of the system, and the dependency diagrams are a wonderful aid to documenting and understanding the system design. With the aid of the software engineering techniques, we were able to identify our most serious system design problem - circular dependencies - and reduce both the extent and the danger of this problem.

The modularization and abstraction of Run Time System concepts helped make the design issues explicit. It led to the recognition of a simple Data Base Management System (Data Structure Subsystem) and Operating System (Operating System Subsystem), among other systems, embedded in the Run Time System. Having recognized these subsystems, we have simplified our work. We can now understand the system better by examining it at the different layers of complexity: system level, subsystem level, and module level. At each level there is less complexity than if all the details were presented in one document (Eg. the code).

The isolation of the Operating System Subsystem also led to an improvement in portability. We identified a list of operating system features which are required for the porting of SR, and we have isolated these features to the Operating System Subsystem. The list of required operating system features follows:

Process features

- creation, deletion, and fast context switching.

Communication

- send and receive message operations.

Memory Management

- variable-size memory allocation and deallocation.

Input/Output

- C-type input and output operations to standard input, standard output, standard error and files.

Machine Addresses

- Unique machine names.

In summary, the use of software engineering techniques in the design of the SR Run Time System has simplified the system and improved the portability.

BIBLIOGRAPHY

[ALME85] G. Almes, "The Impact of Language and System on Remote Procedure Call Design", TR 85-26, Dept. of Computer Science, Rice University, Houston, Texas, October, 1985.

[ANDR86] G. Andrews, et al, "An Overview of the SR Language and Implementation", TR 86-6c, Dept of Computer Science, U. of Arizona, Tucson, October 1987.

[AnOl87] G. Andrews, R Olsson, "Revised Report on the SR Programming Language", TR 87-27, Dept of Computer Science, U. of Arizona, November 1987.

[ATKI88] S. Atkins, R. Olsson, "Performance of multitasking and synchronisation mechanisms", Software Practise and Experience, Vol 18(9), Sept 1988: 880-895.

[CER84] S. Ceri, G. Pelagatti, Distributed Databases: Principles & Systems, McGraw-Hill, New York, NY.

[CHER84] D. Cheriton, "The V Kernel: A Software Base for Distributed Systems", IEEE Software, April 1984: 19-42.

[ChLa86] D. Cheriton, K. Lantz, V System 6.0 Reference Manual, Depts. of Computer Science and Electrical Engineering, Stanford U., California, June 1986.

[DEM82] T. DeMarco, Controlling Software Projects, Yourdon Press (New York), 1982.

[DIJ68] E.W. Dijkstra, "The Structure of the THE Multiprogramming System," Comm ACM, 11(5), May 1968: 341-346.

[FIN89] C. Finley, "A Multiprocessor SR Implementation", CSE-89-6, Div of Computer Science, U. of California, Davis, March, 1989.

[HeSe90] S. Henry, C. Selig, "Predicting Source-Code Complexity at the Design Stage", IEEE Software, March 1990: 36-44.

[HOA81] C.A.R. Hoare, "The Emperor's Old Clothes", Comm ACM, 24(2): 75-83.

[LOHR88] Klaus-Peter Lohr, Joachim Muller, Lutz Nentwig, "DAPHNE - Support for Distributed Applications Programming in Heterogeneous Computer Networks", Proc. Int. Conf. on Distributed Computing Systems (June 1988): 63-71.

[NEWT87] T. Newton, "An Implementation of Ada Tasking", CMU-CS-87-169, Carnegie Mellon U., October, 1987.

[SWI86] D. Swinehart, P. Zellweger, R. Beach, "A Structural View of the Cedar Programming Environment", ACM TOPLAS, 8(4): 419-490.

APPENDIX A:

SR COMMUNICATION PERFORMANCE

To estimate the performance of SR communication pairs on the V-system, we measured the performance of communication pairs on the current UNIX implementation of SR, and we measured the performance of V-system processes. Since the UNIX implementation performs its own process management, and it uses only the malloc system call, we believe it leads to an accurate estimation of the performance of SR process management on the V-system.

The performance figures are used to estimate the performance of SR communication using V-system process management, versus the performance using SR process management, for both the local and remote cases.

The first section describes the SR performance tests, and the UNIX implementation results for local communication. The second section describes the V-system performance tests, and the results for both local and remote communication. The third section compares the local communication performance of an SR implementation using V processes, with the SR UNIX implementation using SR process management. The fourth section estimates the SR remote communication performance using V-system process management and using SR process management.

All the performance tests are performed on SUN-2 workstations on an Ethernet 10Mbit Local Area Network.

1)SR Process Performance Tests

The tests in this section are based on the tests in the paper "Performance of Multi-tasking and Synchronisation Mechanisms" by M. Stella Atkins and Ronald A. Olsson, which appeared in Software Practice and Experience, 1988.

In the following discussion, the name of the program denotes the time to execute the program. For instance, the sisema program executes 1,000,000 iterations of the two semaphore operations, P and V, in 71.8 seconds. Therefore, we can say sisema = 71.8 sec. On the other hand, the word 'sema' is used to indicate the performance of one semaphore operation. So, we say sema = 0.0718 ms.

The following list describes the performance terms:

sisema: time of 1,000,000 pair of P and V operations.

sema: time of 1 pair of P and V ops without context switch.

semaCS: time of context switch associated with sema.

mesg: time of 1 send-in operation without context switch.

mesgCS: time of context switch associated with mesg.

b3: time of 100,000 sema, 500 mesg, 500 mesgCS, 500 semaCS, and b3overhd.

b3overhd: overhead associated with starting and terminating b3.

b4: time of 500 sema, 100,000 mesg, 500 mesgCS, 500 semaCS, and b4overhd.

b4overhd: overhead associated with starting and terminating b4.

semswitch: time of 200,000 sema and their context switches.

msgswitch: time of 200,000 mesg and their context switches.

cirndz: time of 100,000 call-in ops, with context switch.

rndz: time of 1 call-in op, with context switch.

rndzCS: time of 1 context switch associated with rndz.

a5: time of 100,000 send-proc and sema ops.

creat: time of 1 send-proc op.

cpprcd1: time of 1,000,000 call-proc ops within a resource.

prcd1: time of 1 call-proc op within a resource.

cpprcd2: time of 100,000 call-proc ops between resources.

prcd2: time of 1 call-proc op between resources.

overhd1M: overhead associated with 1,000,000 iterations.

overhd100k: overhead associated with 100,000 iterations.

The performance results and calculated statistics follow:

sisema = 71.8 sec

=> sema = 0.072 ms

b3 = 100,000 sema + 500 mesg + 500 mesgCS + 500 semaCS + b3overhd b4 = 500 sema + 100,000 mesg + 500 mesgCS + 500 semaCS + b4overhd b3 = 10.2 sec

b4 = 144.5 sec

b3overhd = 3.1 sec

b4overhd = 3.8 sec

=> (b4-b4overhd) - (b3-b3overhd) = -99,500 sema + 99,500 mesg

=> mesg = 1.41 ms

msgswitch = 200,000 mesg + 200,000 mesgCS

msgswitch = 312.1 sec

=> mesgCS = 0.15 ms

semswitch = 200,000 sema + 200,000 semaCS

semswitch = 57.7 sec

=> semaCS = 0.22 ms

overhd1M = 1 sec

overhd100k = 1 sec

cirndz = 186.0 sec - overhd100k = 185.0 sec

=> rndz = 1.85 ms

rndz = rndzCS + mesg + mesgCS

=> rndzCS = 0.29 ms

a5 = 100,000 creat + 100,000 sema + 100,000 semaCS + overhd100k a5 = 184.4 sec

=> creat = 1.54 ms

cpprcd1 = 1,000,000 prcd1 + overhd1M

cpprcd1 = 44.6 sec

=> prcd1 = 0.043 ms

cpprcd2 = 100,000 prcd2 + overhd100k

cpprcd2 = 112.4 sec

=> prcd2 = 1.11 ms

2) V-system Process Performance Tests

This section describes three tests: local and remote Send-Receive-Reply communication, and V process creation. The Send-Receive-Reply tests are performed using the V-system utility timeipc. The V process creation test is performed by our own program.

Send-Receive-Reply		V process
Local	Remote	creation
1.14 ms	5.11 ms	24 ms *

* This figure is the fastest, consistently reproducible performance measurement. It is difficult to get an accurate measurement because of the limited number of processes the test creates.

3) Local Communication Performance

This section estimates the local communication performance using SR process management and using V-system process management. For SR process management, we estimate the local communication will be the same as the UNIX implementation, because the UNIX implementation of local communication uses only one system call, malloc.

For V-system process management, all communication between processes must be done using V-system calls. Therefore, for the SR send-proc operation, which creates a new process, we must use the V-system calls, Create and Ready. For the send-in and call-in operations which communicate between existing processes, we must use the V-system calls, Send, Receive, and Reply, to implement the context switch. The overhead, which determines the V-process a send is sending to, is added to the cost of the V-system calls. For the call-proc operation, we can use the same optimizations as the UNIX implementation since the proc is not in a separate V process. However, the send-in semaphore optimization between processes can not be done using V processes. Instead, they are implemented the same as the send-in message.

SR communic.	V implement.	Performance (context sw + overhead)	
		SR Process	V Process
send-proc	Create-Ready	1.54 ms	24 ms
send-in (message)	Send-Receive-Reply	1.56	2.55 (1.14 + 1.41)
send-in (semaphore)	Send-Receive-Reply	0.29	2.55 (1.14 + 1.41)
call-proc	C procedure call	1.11	1.11 (same as UNIX)
call-in	Send-Receive-Reply	1.85	2.55 (1.14 + 1.41)

4) Remote Communication

This section estimates the SR remote communication performance using SR process management and using V-system process management. For each method, we first calculate the basic cost of the remote communication, which depends on the V-system calls. Then, we calculate the cost of each SR operation by adding the overhead associated with the operation to the basic cost of remote communication.

Using SR process management, a remote request message is implemented by the following steps:

1) notify a V invoke process of the request, and block invoking SR process (local Send-Receive-Reply).

2) If there are no more SR processes on the ready queue, then the main V process sleeps (Delay).

3) V invoke process executes the V-system Send call, which blocks the invoke process (remote Send).

4) request is received at the remote VM by the VM's receive process (remote Receive).

5) receive process creates an SR process to execute the request, and blocks itself on a Receive (SR process creation).

6) remote SR process executes the request, Replys to the request and kills itself (remote Reply).

7) invoke process is unblocked by Reply. It unblocks the invoking SR process and blocks itself (local Receive for next request).

8) SR process continues execution.

In addition to the above process handling, there is some SR overhead required to determine which SR process should receive the request. This overhead depends on the SR communication operation.

The total time required for this remote request is:

local Send-Receive-Reply	(1.14 ms) +
Delay	(0.05 ms) +
remote Send	(5.11 ms) +
remote Receive	(included in remote Send time) +
SR process creation	(1.54 ms) +
remote Reply	(included in remote Send time) +
SR overhead	
total	7.84 ms + SR overhead

Using V-system process management, a remote request message is implemented by the following steps:

	remote Send-Receive	(5.11 ms) +
	Reply to V pool process	(1.14 ms) +
	remote Reply	(included in Send-Receive time) +
	SR overhead	
	total	6.25 ms + SR overhead

Combining the SR performance figures (not including context switch times) and remote request times, we obtain the following estimated performance times for the SR communication pairs:

	SR	SR process		V process	
	Overhead	Mesg Time	Total	Mesg Time	Total
send-proc	1.54 ms	7.89 ms	9.43 ms	6.25 ms	7.79 ms
send-in	1.41	7.89	9.30	6.25	7.66
call-proc	1.11	7.89	9.00	6.25	7.36
call-in	1.41	7.89	9.30	6.25	7.66

APPENDIX B:

SR on V-System SYSTEM DESIGN

SR RUN TIME SYSTEM (RTS) DESIGN

Function of the RTS

The SR compiler/linker compiles SR resources into object code and links the object code together with the SR Run-Time System (RTS) to form SR executable programs.

The Run-Time System (RTS) contains all the data structures and operations to support the dynamic creation of SR entities, the deletion of entities, and the various operations on these entities. The SR entities include Virtual Machines (VMs), resources, processes, operations, semaphores, and messages. Since these entities are created, deleted and operated on during run-time, SR must supply an RTS.

RTS Documentation

We call this document the Design Document. It describes the re-design of the SR RTS into an Object-Oriented (OO) design suitable for porting to the V-system. The major differences between the previous UNIX design and the OO, V-system design are due to the Object-Oriented nature of the design. The changes due to the V-system are contained within modules.

The design document is broken into the following sections: RTS introduction (this section), Dependency Diagrams, and Abstract Data Types (modules). The modules are grouped into five subsystems: Machine, Generic Lists, Data Structures, Operating System, and Language. Each module describes the data structure and the operations for an SR entity, or an RTS data type which is used to implement an SR entity. Also, each module identifies the C or Assembler code files which implement the module. When the modules are implemented, they form the entire RTS.

The module breakdown was chosen for several reasons:

- It allows the designer and reviewer to understand small portions of the RTS design without having to understand the entire RTS system.
- It allows design changes to be accomplished relatively easily, because it tells the designer where the code that implements each SR entity is located. For example, in the current RTS design, semaphores are implemented by RTS code. If we decided to change that design to use semaphores that are implemented by the operating system, then, by inspecting the design document, we would find that semaphores are implemented by the OSS4_Semaphore module in the Operating System subsystem. Furthermore, that module is implemented in the OSS4_Semaphore.c, and the OSS4_Semaphore.h files. Now, all we have to do is rewrite those files to use the operating system semaphores. The design document has allowed us to quickly locate the semaphore code without having to understand the entire RTS.
- It encourages the designer to place all code which is related

to one module into that module's implementation files. Any other code that uses that module must then call the operations of that module. This results in less coding and testing because each portion of code is only written once. For example, single linked lists are used in many different places in the RTS code. However, the code to implement these lists is written once, in the Generic Lists subsystem, GLLL_Linked_List module. Every other module which uses a linked list, calls the appropriate procedures in the GLLL_Linked_List module, thereby reducing the total code in the RTS. Furthermore, if during testing, we find a mistake in the linked list implementation, we only have to fix it once, and we only have to test it once to make sure the mistake is fixed. We do not have to re-test the linked list for resources, the linked list for processes, the linked list for memory blocks, etc. In other words, the subsystem/module breakdown puts the design effort into creating good modules, which results in good design. The simpler design simplifies the implementation and testing.

The RTS documentation is divided into subsystems and modules in order to help reviewers understand the design. The simpler design that results reduces the amount of coding and thus the amount of testing. We will now describe the RTS design in detail.

Dependency Diagrams and Circular Dependencies

A key tool in understanding the RTS design is the dependency diagram. This diagram is used to show the dependencies between subsystems. We define depend by saying that subsystem A depends on subsystem B if A uses a procedure, a data type, or anything which is implemented in subsystem B. The dependency diagram for the A and B subsystems is drawn below:

A major problem in the RTS design is circular dependencies. The simplest example of a circular dependency occurs when Subsystem A depends on Subsystem B and Subsystem B depends on A. There are also circular dependencies with 'larger' circles. That is there may be four or five subsystems in the circular dependency, each subsystem depending on the next subsystem, and the last subsystem depending on the first subsystem. (Eg. A -> B -> C -> D -> A)

These circular dependencies are a problem for several reasons. First, they may indicate a mutually recursive procedure call. If this recursion is not completely understood, it could possibly cause infinite recursion to occur every time the program is run, or, worse, just under special circumstances! Therefore, every circular dependency on the dependency diagram must be investigated to make sure that the design has safeguards against infinite recursion.

The second problem is deadlock due to resource contention. This type of deadlock occurs in the following scenario. Subsystem A has control of resource X, and it calls subsystem B. B attempts to get control of resource X, but fails because A already has X. B then waits for the resource to be released. Unfortunately, it will wait forever, since A is not going to release the resource until B is finished. The most common example of this scenario occurs in systems which attempt to report an 'out of memory' error but hang instead. The system hangs because the exception report mechanism attempts to allocate memory to hold the error message, but is unable to because the system is already out of memory!

The third problem with circular dependencies occurs during the testing of the final system. There are two general strategies that can be applied to this testing: top-down testing and bottom-up testing. In the first case, the top-most module on the dependency diagram is tested first, with all the lower level modules stubbed out. Then, one of the lower modules is tested with the top-most module. The testing continues in this manner, adding lower-level modules until the entire system is included in the tests. In bottom-up testing, one of the bottom level modules is tested first, and the upper modules are added, one at a time, until the entire system is being tested. In both cases, the testing procedures depend on the assumption that bugs found during testing are most likely to be caused by the last module added to the test system. This assumption can enormously simplify and speed-up the testing process when a large system is being tested.

The problem with circular dependencies is that they do not have a top or a bottom! Therefore, we can not use the top-down, or bottom-up testing procedures. We have to develop special testing procedures for the system. These special procedures will complicate and slow down the testing process. When bugs are found, they will be more difficult to find because we can not assume that the original modules in the system have been completely tested.

In general, removing a circular dependency removes any chance of infinite recursion and simplifies the design. The simpler design avoids some tricky deadlock errors, and makes the testing simpler and quicker.

Naming Standards

We have followed a few simple rules in abbreviating the subsystem and module names. The abbreviation for each subsystem is two letters long. When naming the subsystem, it is common to prefix the name of the subsystem with its abbreviation. For example, the Operating System subsystem's abbreviation is OS. It is often referred to as OS_Operating_System.

The abbreviation for each module name is also two letters long but it is combined with its subsystem abbreviation to make a four letter abbreviation. For example, the abbreviation for the Semaphore module in the Operating System subsystem is S4. When combined with the subsystem name, the abbreviation is OSS4, and the common name for the module is OSS4_Semaphore module.

The module four letter abbreviation is often prefixed to the module operations. For example, the operation to create an empty linked list, using the GLLL_Linked_List module, is called glll_create_empty_list. (This standard has not been completely implemented simply because, it is so much work to go and change all the code which uses operations with non-standard names.)

Module Descriptions

We use a standard format for all the module descriptions. The following example shows the module description for the mythical Stack module in the mythical UT_Useful_Things subsystem. We have included comments in every section of the description to explain the purpose and meaning of the format and terms used in that section. The module abbreviation is UTST, and the description was last modified on February 9, 1991.

UTST STACK MODULE

February 9, 1991

PURPOSE:

This section describes the purpose of the module. In this case, the stack module implements the data types and operations to create, delete and perform operations on a stack.

DATA INTERFACE:

This section describes any variables that may be used by other modules to change the operation of this module.

Name	**Description**
None.	- In the Stack module, there are no variables which other modules may access or modify.

DATA TYPE INTERFACE:

This section describes data type definitions that may be used by other modules to declare their own variables.

Name	**Description**
stack	Pointer to a stack variable.

INVOCATION INTERFACE:

This section describes operations which may be invoked by other modules to modify the stack variables they have declared. Under the **Procedure** heading is the name of the operation. On the same line, or the very next line, is a **Description** of the Procedure.

Following the description of each procedure there is a list of the procedure parameters under the **Parameters** heading. On the same line as the parameter name is a parameter **Description.** The parameter description includes, in order, the data type, the flow of data (INput, OUTput, or INput-OUTput), and a short English description.

Procedure **Description**
 Parameters **Description** (Type,IN/OUT, etc.)

utst_init_stack
 Initialize the stack data structure.
 NewStack stack, IN-OUT, The stack to be initialized.

utst_kill_stack
 Free up all resources used by this stack.
 OldStack stack, IN-OUT, This stack will become unusable.

utst_push — Add an item to the top of the stack.

- Item — stk_item, IN, The item.
- AStack — stack, IN-OUT, The stack receiving Item.

utst_pop — Remove the item at the top of the stack.

- AStack — stack, IN-OUT, The stack.
- Item — stk_item, IN, The item removed from AStack.

IMPLEMENTATION FILES:

This section names the files which contain the code implementing this module. Sometimes there is more than one file implementing the module. However, one file is never used to implement more than one module!

UTST_Stack.h — - The .h files contain data, data type and procedure declarations for this module. They must be included in any other module which uses this module.

IMPORTED ELEMENTS:

This section lists all the elements used by this module that are implemented in other modules. It also describes the element Type (Procedure, Data Type, or Data), and the module which implements the element.

Name	**Type**	**module**
glll_create_empty_list	Procedure	GLLL_Linked_List
glll_push	Procedure	GLLL_Linked_List
glll_pop	Procedure	GLLL_Linked_List

NOTES:

None. — - This section will sometimes contain special comments explaining design decisions or suggestions for implementing the module.

RTS Design

The dependency diagram in Figure 1 shows the dependency relationship between the RTS subsystems. Note that there is one circular dependency in this diagram. The LG -> OS -> LG circular dependency is described in greater detail in the OSNE_Network module description.

This appendix is divided into one section for each subsystem of the RTS. In each section there is a dependency diagram for the subsystem and a module description for each module in the subsystem. The module dependency diagram is identical to the subsystem diagram in meaning except that an arrow from the A module to the B module means that the A module depends on the B module. I.e. the only difference is that the subsystem dependency diagram contains subsystems and the module dependency diagrams contain modules.

HOW TO START

The design document is intended to be used as a reference, rather than an introduction to the RTS, so there is no good place to start reading. This document works best when there is a particular design question that must be answered. In that case, the reader uses the document like a dictionary, with no intention of understanding everything, but simply intending to get information about one topic, or, in this case, one design issue.

However, if this is your first introduction to the RTS, then it is best to start by perusing the Dependency Diagrams. After that, you can start by reading the LG_Language subsystem, since that contains the highest level modules.

RTS DATA STRUCTURE SUBSYSTEM (DS) DESCRIPTION

Function of the Data Structure Subsystem

In the RTS design, there is one RTS for each Virtual Machine (VM). Each RTS implements a very complicated data structure to keep track of all the SR entities on its VM, and the relations between those entities. It is the purpose of the Data Structure Subsystem to implement the entity descriptors (data types) and supply primitive procedures to allow higher-level modules to access the data in the descriptors.

In Object-Oriented Programming Systems (OOPS) terminology, each DS module is a 'server' class. Since the DS modules only supply data types and data access procedures, we call the DS modules **data servers**.

For each data server, there is one higher-level module in the OS_Operating_System or LG_Language subsystems which has the same module name, but a different prefix. We call the corresponding higher-level module, the **function server**, because it implements the corresponding functions. For example, the server class DSS4_Semaphore module implements the semaphore data type and one data access procedure: dss4_sem_count. OSS4_Semaphore is the corresponding function server which implements the standard semaphore functions: create, kill, P, and V.

The DS_Data_Structures subsystem is designed to let all modules access the RTS data structure through the interface specified by the module description. However, the function server for a DS module may manipulate any fields in the DS module, even those that are 'hidden'. **Hidden** fields are not specified as part of the interface. An example of a hidden field is the blocked field in the semaphore descriptor which is a list of the processes blocked on the semaphore. The OSS4_Semaphore function server needs to access the blocked field to implement the P and V operations. The need of the function server such as OSS4-_Semaphore to access the hidden fields of a data server, reflects the tight relationship between the data server and function server pairs. Unfortunately, there is no way to document this relationship in the C code other than to use the same root name on the code files. In an OOPS programming language, we could reflect this relationship by having the function server inherit the data server, and redefine the interface.

Data Structure Subsystem Design

Much of the complexity of the Data Structure is created by two requirements. The SR entities must be created dynamically and there are many relationships between the entities which must be stored in order to perform the operations efficiently. For example, in the case of the resource and process entities, we have at least two relationships between these entities. Each resource may contain any number of processes, and each process must have an 'owner' resource.

To satisfy the dynamic requirements, the RTS implements descriptor records which exist in main memory. To satisfy the need to keep track of relationships between entities, each descriptor record contains pointers to other entities which are related to it. For example, the resource instance descriptor has a pointer to a list of processes in the resource and the process descriptor has a pointer to the 'owner' resource of the process.

<u>Note</u>

The DS subsystem is really a very primitive DBMS. There may be alternative designs using DBMS technology which are more efficient, support data distribution, and supply other DBMS benefits.

DSCL CLASS MODULE

PURPOSE:

Implement the data structures and data access functions for the class data type. This 'class' refers to the SR implementation of equivalence classes for input operations. It has nothing to do with the 'class' of Object-Oriented programming. For more information about the SR class implementation refer to "An Overview of the SR language and Implementation", by Gregory Andrews, et al.

DATA INTERFACE:

Name	Description
None.	

DATA TYPE INTERFACE:

Name	Description
class	Pointer to an operation class descriptor.

INVOCATION INTERFACE:

Procedure — **Description**
  Parameters — **Description** (Type,IN/OUT, etc.)

dscl_class_pending
The number of pending invocations for this class.

Parameters	Description
clap	class, IN, This class data structure.
[return]	short, OUT, Number of pending invocations for clap.

dscl_class_num_ops
The number of operations in this class.

Parameters	Description
clap	class, IN, This class data structure.
[return]	short, OUT, Number of operations in clap.

dscl_class_count
The number of available class descriptors.

Parameters	Description
[return]	int, OUT, Number of available descriptors.

IMPLEMENTATION FILES:

DSCL_Class_i.h
DSCL_Class_h.h

IMPORTED ELEMENTS:

Name	Type	Module
inv_queue	Data Type	DSIN_Invocation
proc_queue	Data Type	DSPR_Process
proc	Data Type	DSPR_Process
sem	Data Type	DSS4_Semaphore
dss4_sem_count	Procedure	DSS4_Semaphore
Bool	Data Type	UT_Util

NOTES:

None.

DSCO CO MODULE

PURPOSE:

Implement the data structures and data access functions for the co data types. These data types support the implementation of the SR co statement.

DATA INTERFACE:

Name	**Description**
INIT_SEQ_CO	co initial sequence number.

DATA TYPE INTERFACE:

Name	**Description**
cob	pointer to a CO statement descriptor.
struct cii_st	Co Invocation Information data STructure.

INVOCATION INTERFACE:

Procedure **Description**
 Parameters **Description** (Type,IN/OUT, etc.)

dsco_cob_pending
 The number of pending invocations on this co statement.
 coStmt — cob, IN, This co statement descriptor.
 [return] — short, OUT, Number of pending invocations for coStmt.

dsco_cob_completions
 The list of completed invocations for this co statement.
 coStmt — cob, IN, This co statement descriptor.
 [return] — invb, OUT, List of completed invocations for coStmt.

dsco_cii_cob The co statement descriptor for this arm of the co statement.
 cii_arm — cii_st, IN, This arm of the co statement.
 [return] — cob, OUT, The co statement descriptor for cii_arm.

dsco_cii_completions
 The list of completed invocations for this arm of the co statement.
 cii_arm — cii_st, IN, This arm of the co statement.
 [return] — invb, OUT, The list of completed invocations for cii_arm.

dscl_co_count [return]	The number of available co descriptors.	int, OUT, Number of available descriptors.

IMPLEMENTATION FILES:

DSCO_Concurrent_i.h
DSCO_Concurrent_h.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
invb	Data Type	DSIN_Invoke
sem	Data Type	DSS4_Semaphore
dss4_sem_count	Procedure	DSS4_Semaphore
tindex	Data Type	GLAR_Array
seq	Data Type	UT_Util

NOTES:

None.

DSIN INVOKE MODULE

PURPOSE:

Implement the data structures and data access functions for the invoke data types.

DATA INTERFACE:

Name	Description
INVOCATION_HEADER_SIZE	Byte size of the invb data structure header (i.e. the part of the data structure that comes before the variable-length argument list).
OP_CAP_OFFSET	The byte offset of the operation capability inside the invb data structure.
OP_CAP_SIZE	The byte size of the operation capability inside the invb data structure.

DATA TYPE INTERFACE:

Name	Description
in_type	enumerated type specifying the valid INvocation TYPEs.
invb	pointer to an INVocation data structure.
inv_queue	an INVocation QUEUE data structure.

INVOCATION INTERFACE:

Procedure — **Description**

  Parameters — **Description** (Type, IN/OUT, etc.)

dsin_invb_pach

The packet header for this invocation.

Parameter	Description
invoke	invb, IN, This invocation data struct.
[return]	pach, OUT, The packet header for invoke.

dsin_invb_opcab

The operation capability for this invocation

Parameter	Description
invoke	invb, IN, This invocation data struct.
[return]	opcap, OUT, The operation capability for invoke.

dsin_invb_type

The invocation type of this invocation.

Parameter	Description
invoke	invb, IN, This invocation data struct.
[return]	in_type, OUT, The invocation type for invoke.

dsin_invb_proc

The process id for the invoker process.

invoke — invb, IN, This invocation data struct.

[return] — proc, OUT, The process id for invoke.

dsin_invb_co

The co data for this invocation.

invoke — invb, IN, This invocation data struct.

[return] — struct cii_st, OUT, The co statement data for invoke.

dsin_invb_arg_size

The byte size of the argument list in this invocation.

invoke — invb, IN, This invocation data struct.

[return] — pach, OUT, The argument list byte size for invoke.

dsin_create_inv_queue

Create an empty invocation queue.

invokeQ — inv_queue, IN-OUT, A new invoke queue.

dsin_is_empty_inv_queue

Test if invocation queue is empty.

invokeQ — inv_queue, IN, An existing invoke queue.

[return] — Bool, OUT, TRUE if invokeQ is empty, FALSE otherwise.

dsin_append

Add an invocation to the end of the queue.

invoke — invb, IN, An invocation.

invokeQ — inv_queue, IN-OUT, The queue.

dsin_append_list

Add a new list to the tail of an existing queue.

NewList — invb, IN, A new list.

ExistQ — inv_queue, IN-OUT, The existing queue.

dsin_delete

Delete an invocation from the middle of the queue.

invoke — invb, IN, The invocation to be deleted.

invokeQ — inv_queue, IN-OUT, The queue containing invoke.

IMPLEMENTATION FILES:

DSIN_Invoke_i.h
DSIN_Invoke_h.h

IMPORTED ELEMENTS:

Name	Type	Module
struct cii_st	Data Type	DSCO_Concurrent
struct pach_st	Data Type	DSNE_Network
pach	Data Type	DSNE_Network
opcap	Data Type	DSOP_Operation
proc_queue	Data Type	DSPR_Process
proc	Data Type	DSPR_Process
sem	Data Type	DSS4_Semaphore
gldd_create_empty_list	Procedure	GLDD_Double_Double_List
gldd_is_empty_list	Procedure	GLDD_Double_Double_List
gldd_append	Procedure	GLDD_Double_Double_List
gldd_append_list	Procedure	GLDD_Double_Double_List
gldd_delete	Procedure	GLDD_Double_Double_List
Bool	Data Type	UT_Utility
seq	Data Type	UT_Utility
ut_offsetof	Procedure	UT_Utility
ut_fieldsize	Procedure	UT_Utility

NOTES:

None.

DSIO INPUT OUTPUT MODULE

PURPOSE:

Implement the data structures and data access functions for the input and output data types.

DATA INTERFACE:

Name	Description

None.

DATA TYPE INTERFACE:

Name	Description
io_type	Input/Output TYPE. Values are: INPUT, OUTPUT.
access_mode	file ACCESS MODE. Values are: READ, WRITE, READ_WRITE.
file_offset	FILE OFFSET type. Values are: ABSOLUTE, RELATIVE, EXTEND.
FILE	FILE descriptor. Values include: STDIN, STDOUT, STDERR, NULL_FILE, NOOP_FILE.

INVOCATION INTERFACE:

Procedure **Description**

Parameters **Description** (Type,IN/OUT, etc.)

None.

IMPLEMENTATION FILES:

DSIO_IO_i.h

IMPORTED ELEMENTS:

Name	Type	Module
proc_queue	Data Type	DSPR_Process

NOTES:

None.

DSLM LIMITS MODULE

PURPOSE:

Implement the data structures to support the RTS runtime limits.

DATA INTERFACE:

Name **Description**

sr_max_rmt_reqs

Maximum number of remote requests that can be issued at any one time.

DATA TYPE INTERFACE:

Name **Description**

None.

INVOCATION INTERFACE:

Procedure **Description**

Parameters **Description** (Type, IN/OUT, etc.)

None.

IMPLEMENTATION FILES:

DSLM_Limit_i.h

IMPORTED ELEMENTS:

Name	Type	Module
None		

NOTES:

None.

DSMM MEMORY MODULE

PURPOSE:

Implement the data structures and data access functions for the memory block data type.

DATA INTERFACE:

Name	**Description**
RTS_OWN	If dsmm_memh_res returns this value then the RTS owns the memory block.
PROG_OWN	If dsmm_memh_res returns this value then the program owns the memory block. I have not seen this constant used anywhere in the RTS code, so I think it may be unused now (HB, Feb/91).

DATA TYPE INTERFACE:

Name	**Description**
memh	pointer to a MEMory block Header. This header exists for every memory block allocated for the SR program.
memhdr	pointer to a MEMory HeaDeR. This header only exists for certain cases where the Generated Code (GC) wishes to keep track of the memory it is allocating.

INVOCATION INTERFACE:

Procedure	**Description**	
Parameters		**Description** (Type,IN/OUT, etc.)
dsmm_memh_res	The resource for this memory block.	
memblock		memh, IN, A memory block header.
[return]		rint, OUT, The resource which owns memblock.

Memory List Operations

The following procedures perform the standard list operations for memory lists. Refer to the GL_Generic_Lists subsystem introduction for an explanation of the standard list operations.

dsmm_create_empty_mem_list
dsmm_push_mem
dsmm_delete_mem

IMPLEMENTATION FILES:

DSMM_Memory_i.h
DSMM_Memory_h.h

IMPORTED ELEMENTS:

Name	Type	Module
rint	Data Type	DSRE_Resource
gldl_create_empty_list	Procedure	GLDL_Double_Link
gldl_push	Procedure	GLDL_Double_Link
gldl_delete	Procedure	GLDL_Double_Link

NOTES:

None.

DSNE NETWORK MODULE

PURPOSE:

Implement the data structures and data access functions for the network interface data types.

DATA INTERFACE:

Name	**Description**
None.	

DATA TYPE INTERFACE:

Name	**Description**
ms_type	MeSsage TYPE. Values are: BLOCKFUNC_FINI, REQ_FINDVM, REQ_CREATE, REQ_INVOKE, REQ_DESTROY, REQ_DESTVM, MSG_EXIT, NO_OP.
pach_st	PACKet header structure. Contains information necessary for every packet sent over the network.
num_st	message structure to hold one NUMber. Several of the message types only send one number in their message.
srxreply	message structure for a REPLY message from the SRX.
findvm_reply_st	message structure for a message in REPLY to a req_FINDVM message.

INVOCATION INTERFACE:

Procedure **Description**
Parameters **Description** (Type, IN/OUT, etc.)

None.

IMPLEMENTATION FILES:

DSNE_Net_i.h
DSNE_Net_h.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
Pid	Data Type	DSPR_Process

NOTES:

None.

DSOP OPERATION MODULE

PURPOSE:

Implement the data structures and data access functions for the operation data types.

DATA INTERFACE:

Name	**Description**
INIT_SEQ_OP	operation initial sequence number.

DATA TYPE INTERFACE:

Name	**Description**
op_type	enumerated type which specifies the valid OPeration TYPEs.
opcap	OPeration CAPability descriptor.
oper	pointer to an OPERation descriptor.

INVOCATION INTERFACE:

Procedure **Description**
    Parameters **Description** (Type,IN/OUT, etc.)

dsop_opcap_vm The vm for this operation capability.
    OperationCap opcap, IN, This operation capability's data structure.
    [return] vmid, OUT, VM identifier for OperationCap.

dsop_oper_res The resource that this operation belongs to.
    operation oper, IN, This operation's data struct.
    [return] rint, OUT, Resource instance for operation.

dsop_oper_pending_inputs
    The number of pending inputs for this operation.
    operation oper, IN, This operation's data struct.
    [return] short, OUT, Number of pending inputs for operation.

dsop_oper_type
    The operation type of this operation.
    operation oper, IN, This operation's data struct.
    [return] op_type, OUT, Operation type of operation.

dsop_oper_code
    The code address for this operation, if this

is a proc type operation.

operation — oper, IN, This operation's data struct.

[return] — paddr, OUT, Code address for operation.

dsop_oper_class

The input operation class for this operation, if this is an input type operation.

operation — oper, IN, This operation's data struct.

[return] — class, OUT, Input operation class for operation.

dsop_oper_sema4

The semaphore for this operation, if this is a semaphore type operation.

operation — oper, IN, This operation's data struct.

[return] — sema, OUT, Semaphore for operation.

dscl_oper_count

The number of available operation descriptors.

[return] — int, OUT, Number of available descriptors.

IMPLEMENTATION FILES:

DSOP_Operation_i.h
DSOP_Operation_h.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
class	Data Type	DSCL_Class
rint	Data Type	DSRE_Resource
sem	Data Type	DSS4_Semaphore
dss4_sem_count	Procedure	DSS4_Semaphore
vmid	Data Type	DSVM_Virtual_Machine
paddr	Data Type	MCPR_Process
seqn	Data Type	UT_Utility

NOTES:

None.

DSOS OPERATING SYSTEM MODULE

PURPOSE:

Implement the data structures and data access functions that are peculiar to the V-system operating system.

DATA INTERFACE:

Name	**Description**
Message Constants	
MAX_SEGMENT_SIZE	
MIN_MESG_SIZE	
V-system Process Priorities	
VULTURE_PRIO	
BLOCK_OSPROCESS_PRIO	
MAIN_PROCESS_PRIO	
MESG_STKSIZE	

DATA TYPE INTERFACE:

Name	**Description**
system_errors	System errors - classified by system call. Values are: CREATE_ERROR, READY_ERROR, RECEIVESPEC_ERROR, REPLY_ERROR, SEND_ERROR, REPLYSEG_ERROR, OPEN_ERROR, CLOSE_ERROR, SEEK_ERROR

INVOCATION INTERFACE:

Procedure **Description**

Parameters **Description** (Type, IN/OUT, etc.)

None.

IMPLEMENTATION FILES:

DSOS_Operating_System_i.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
Message	Data Type	V-system

NOTES:

None.

DSPR PROCESS MODULE

PURPOSE:

Implement the data structures and data access functions for the process data types. These data types support the implementation of SR processes.

DATA INTERFACE:

Name	**Description**
None.	

DATA TYPE INTERFACE:

Name	**Description**
proc	A process descriptor.
proc_queue	A queue of processes. This is often used to sequence a list of blocked processes.
pr_type	valid sr PRocess TYPEs. Values are: INITIAL, FINAL, PROC.
pr_status	PRocess status codes. Values are: srACTIVE — Process is running. srREADY — Process is ready to run. srBLOCKED — Process is blocked, waiting for some operation. srINFANT — Process is created but not started. srFREE — Process descriptor is not in use.
Pid	Process IDentifier.

INVOCATION INTERFACE:

Procedure — **Description**

Parameters — **Description** (Type,IN/OUT, etc.)

dspr_proc_stack

The stack address for this process descriptor.

procDesc	proc, IN, This process descriptor.
[return]	daddr, OUT, The stack address for procDesc.

dspr_proc_status

The status of this process descriptor.

procDesc	proc, IN, This process descriptor.
[return]	int, OUT, The status for procDesc.

dspr_proc_in_type

The invocation type of this process descriptor.

procDesc	proc, IN, This process descriptor.

[return] in_type, OUT, The invocation type of procDesc.

dspr_proc_res The resource that owns this process descriptor.

procDesc proc, IN, This process descriptor.
[return] rint, OUT, The resource that owns procDesc.

dspr_proc_blocked
The blocked list that this process descriptor is on.

procDesc proc, IN, This process descriptor.
[return] proc_queue *, OUT, The blocked list for procDesc.

dspr_proc_invoke
The invocation descriptor for this process descriptor.

procDesc proc, IN, This process descriptor.
[return] invb, OUT, The invocation descriptor for procDesc.

dspr_proc_co_list
The list of co statements for this process descriptor.

procDesc proc, IN, This process descriptor.
[return] cob, OUT, The list of co statements for procDesc.

dspr_proc_class
The operation class for the current input statement in this process descriptor.

procDesc proc, IN, This process descriptor.
[return] class, OUT, The operation class for the current input statement in procDesc.

dspr_is_proc_else_leg
Is this process in an in-statement with an else leg?

procDesc proc, IN, This process descriptor.
[return] Bool, OUT, TRUE if this process is in an in-statement with an else leg. FALSE, otherwise.

Process List

The following procedures perform the standard list operations for process lists. Refer to the GL_Generic_Lists subsystem introduction for an explanation of the standard list operations.

dspr_create_empty_proc_list
dspr_is_empty_proc_list
dspr_append_proc
dspr_pop_proc
dspr_delete_proc

Process Queue

The following procedures perform the standard list operations for process queues. Refer to the GL_Generic_Lists subsystem introduction for an explanation of the standard list operations.

dspr_create_empty_procQ
dspr_is_empty_procQ
dspr_append_procQ
dspr_pop_procQ
dspr_delete_procQ

IMPLEMENTATION FILES:

DSPR_Process_i.h
DSPR_Process_h.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
class	Data Type	DSCL_Class
cob	Data Type	DSCO_Concurrent
in_type	Data Type	DSIN_Invoke
invb	Data Type	DSIN_Invoke
rint	Data Type	DSRE_Resource
sem	Data Type	DSS4_Semaphore
glll_create_empty_proc_list	Procedure	GLLL_Linked_List
glll_is_empty_proc_list	Procedure	GLLL_Linked_List
glll_append	Procedure	GLLL_Linked_List
glll_pop	Procedure	GLLL_Linked_List
glll_delete	Procedure	GLLL_Linked_List
glde_create_empty_proc_list	Procedure	GLDE_Double_Ended
glde_is_empty_proc_list	Procedure	GLDE_Double_Ended
glde_append	Procedure	GLDE_Double_Ended
glde_pop	Procedure	GLDE_Double_Ended
glde_delete	Procedure	GLDE_Double_Ended

Bool	Data Type	UT_Utility
daddr	Data Type	UT_Utility

NOTES:

None.

DSRE RESOURCE MODULE

PURPOSE:

Implement the data structures and data access functions for the resource data types.

DATA INTERFACE:

Name	Description
INIT_SEQ_RES	RESource INITial SEQuence number.
CRB_HEADER_SIZE	byte SIZE of the CReate Block HEADER.
RES_CAP_SIZE	byte SIZE of the RESource CAPability structure.

Resource Status Values:

INIT_REPLY	INITial process has REPLIed.
FINAL_REPLY	FINAL process has REPLIed.
FREE_SLOT	this resource descriptor SLOT is FREE.

DATA TYPE INTERFACE:

Name	Description
rescap	RESource CAPability data structure.
rint	pointer to a Resource INsTance descriptor data structure.
crb	pointer to a Create Request Block. It contains information necessary to perform the create operation.

INVOCATION INTERFACE:

Procedure **Description**
  Parameters **Description** (Type, IN/OUT, etc.)

dsre_rescap_vm
The VM of the resource specified by this rescap.

Parameters	Description
ResourceCap	rescap, IN, Resource capability.
[return]	vmid, OUT, The VM specified by ResourceCap.

dsre_rint_procs
The process list for this resource.

Parameters	Description
res	rint, IN, Resource instance.
[return]	proc, OUT, The list of processes for res.

dsre_rint_memory
The memory list for this resource.

Parameters	Description
res	rint, IN, Resource instance.
[return]	memh, OUT, The list of memory blocks for res.

dsre_rint_rescap
The resource capability for this resource.
res rint, IN, Resource instance.
[return] rescap, OUT, The rescap for res.

dsre_rint_rc_size
The resource capability size for this resource.
res rint, IN, Resource instance.
[return] short, OUT, The byte size of rescap for res.

dsre_rint_ops The operations list for this resource.
res rint, IN, Resource instance.
[return] oper, OUT, The list of operations for res.

dsre_rint_num_ops
The number of operations for this resource.
res rint, IN, Resource instance.
[return] short, OUT, The number of operations for res.

dsre_rint_status
The status flag for this resource's initial/final/reply proc.
res rint, IN, Resource instance.
[return] int, OUT, Initial/final/reply status flag for res.

dscl_rint_count
The number of available rint descriptors.
[return] int, OUT, Number of available descriptors.

dsre_crb_pach The packet header for this Create Request Block.
CreateReq crb, IN, Create request block.
[return] pach, OUT, Packet header for CreateReq.

dsre_crb_rescap
The resource capability for this Create Request Block.
CreateReq crb, IN, Create request block.
[return] rescap, OUT, Resource capability for CreateReq.

dsre_crb_size The byte size of this Create Request Block.
CreateReq crb, IN, Create request block.
[return] short, OUT, Byte size of CreateReq.

dsre_crb_vm The VM in this Create Request Block.
CreateReq crb, IN, Create request block.
[return] vmid, OUT, VM identifier for

CreateReq.

dsre_crb_args	The arguments in this Create Request Block.	
CreateReq		crb, IN, Create request block.
[return]		char [], OUT, Array of arguments in CreateReq.

IMPLEMENTATION FILES:

DSRE_Resource_i.h
DSRE_Resource_h.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
memh	Data Type	DSMM_Memory_Management
pach	Data Type	DSNE_Network
pach_st	Data Type	DSNE_Network
opcap	Data Type	DSOP_Operation
oper	Data Type	DSOP_Operation
proc	Data Type	DSPR_Process
sem	Data Type	DSS4_Semaphore
dss4_sem_count	Procedure	DSS4_Semaphore
vmid	Data Type	DSVM_Virtual_Machine
status	Data Type	UT_Utility
seqn	Data Type	UT_Utility
daddr	Data Type	UT_Utility
ut_offsetof	Procedure	UT_Utility

NOTES:

None.

DSRM REMOTE MODULE

PURPOSE:

Implement the data structures and data access functions for the remote operations.

DATA INTERFACE:

Name	Description

None.

DATA TYPE INTERFACE:

Name	Description

None.

INVOCATION INTERFACE:

Procedure **Description**
Parameters **Description** (Type,IN/OUT, etc.)

dsrm_rem_count
The number of available remote descriptors.
[return] int, OUT, Number of available descriptors.

IMPLEMENTATION FILES:

DSRM_Remote_i.h

IMPORTED ELEMENTS:

Name	Type	Module
sem	Data Type	DSS4_Semaphore
dss4_sem_count	Procedure	DSS4_Semaphore

NOTES:

None.

DSS4 SEMAPHORE MODULE

PURPOSE:

Implement the data structures and data access functions for the semaphore data type.

DATA INTERFACE:

Name	**Description**
None.	

DATA TYPE INTERFACE:

Name	**Description**
sem	Pointer to a semaphore data structure.

INVOCATION INTERFACE:

Procedure	**Description**	
Parameters		**Description** (Type,IN/OUT, etc.)
dss4_sem_count	The value of a semaphore counter.	
sema4		sem, IN, A semaphore.
[return]		int, OUT, Value of sema4's counter. If it is less than 0, then it gives the number of processes waiting on this semaphore. This value is zero if sema4 is not in use (free).

IMPLEMENTATION FILES:

DSS4_Semaphore.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
proc_queue	Data Type	DSPR_Process

NOTES:

None.

DSVM VIRTUAL MACHINE MODULE

PURPOSE:

Implement the data structures and data access functions for the virtual machine (VM) data types.

DATA INTERFACE:

Name	Description
sr_my_vm	Current virtual machine number.
sr_my_machine	Current physical machine number.
NULL_Virtual_Machine	Null VM capability.
NOOP_Virtual_Machine	Null VM capability.
sr_nu_vm cap	Null vm capability.
sr_no_vmcap	Noop vm capability.
VM_MAGIC	random number used to check that VMs are started by a valid SR program.
PROTO_VER	VERsion identifier. Used to check that two portions of SR code are compiled by the same SR compiler.

DATA TYPE INTERFACE:

Name	Description
pmid	Physical Machine IDentifier.
sr_pmdata	Physical Machine descriptor.
vmid	Virtual Machine IDentifier.
sr_vmdata	Virtual Machine descriptor.

INVOCATION INTERFACE:

Procedure / Parameters	Description / Description (Type,IN/OUT, etc.)
dsvm_vm_known	Is the given VM known to the VM data structure?
vm	vmid, IN, A VM identifier.
[return]	Bool, OUT, TRUE if VM is known, FALSE otherwise.
dsvm_vm_pm	Determine the physical machine id for the given VM.
dsvm_vm_addr	Determine the message address for the given VM.

IMPLEMENTATION FILES:

DSVM_Virtual_Machine.h

IMPORTED ELEMENTS:

Name	Type	Module
Bool	Data Type	UT_Utility

NOTES:

None.

RTS GENERIC LISTS SUBSYSTEM (GL) DESCRIPTION

Function of the Generic Lists Subsystem

Many of the SR entities are implemented using data structures called descriptors, eg. the resource and process descriptors. In turn, these descriptors are often stored in lists of various types, because of the SR requirement that the entities be created and destroyed dynamically. Since these list types have very little to do with the type of descriptor they contain, it is appropriate that the lists are implemented separately from the SR entities. For example, the resource descriptor is implemented by the LGRE_Resource module, but it uses a linked list which is implemented by the GLLL_Linked_List module. Therefore, the Generic Lists Subsystem has been created to implement modules for all the list types required by the RTS.

This subsystem has very few dependencies because it is usually only working with pointer fields. It initializes pointer fields, and assigns one field to another. GL_Generic_Lists does depend on MC_Machine for some generic data type definitions.

Generic Lists Subsystem Design

All of the Generic Lists (GL) modules are implemented using standard list manipulation algorithms. Therefore, this section merely describes some implementation techniques common to all the modules which affect the design and use of these modules.

Each Generic Lists (GL) module defines its own data type. However, this is little more than a syntactic convention. In fact, the procedures in these modules can work with any C record structure. This works because C has very loose type checking and all the GL procedures are implemented as #define statements.

The #define statements are processed by the C preprocessor. In essence, the GL procedures, implemented by #define statements, are 'invoked' before the code is compiled. Therefore, they can accept parameters containing C types, and C field names. These parameters allow the GL procedures to be more general than if they were implemented with the standard C functions.

Since all the modules in the GL subsystem are working on lists, they tend to supply very similiar procedures. To make this similiarity explicit, we have used the following standard procedure names:

create_list - Create a list and initialize it.

is_empty_list - Determine if a list is empty. Return

TRUE for an empty list, and FALSE otherwise.

pop	- Remove the node from the front of the list and return a pointer to it.
chop	- Remove a node from the end of the list and return a pointer to it.
delete	- Remove the given node from the list. The node may be anywhere in the list.
push	- Add a node to the front of the list.
append	- Add a node to the end of the list.
append_list	- Add a new list to the end of the old list.
insert	- Add a node after the given node in the list. The given node may be anywhere in the list.

Not all of the above procedures are implemented for all of the GL modules.

GLAR ARRAY MODULE

PURPOSE:

Implement generic data structures which support the use of arrays.

DATA INTERFACE:

Name	**Description**
"Descriptor fields"	
AD_LB1	Lower bound, if array.
AD_UB1	Upper bound, if array.
AD_LB2	Second lower bound, if two dimensional array.
AD_UB2	Second upper bound, if two dimensional array.

DATA TYPE INTERFACE:

Name	**Description**
tindex	Index for small tables.

INVOCATION INTERFACE:

Procedure **Description**

Parameters **Description** (Type,IN/OUT, etc.)

None.

IMPLEMENTATION FILES:

GLAR_Array.h

IMPORTED ELEMENTS:

Name **Type** **Module**

None.

NOTES:

None.

GLDD DOUBLE-ENDED, DOUBLE-LINKED LIST MODULE

PURPOSE:

Implement two-way (double), linked lists, with quick access to both ends of the list. These lists do not make as efficient use of memory as the other lists but they can quickly perform deletion operations at any position in the list. They can also quickly perform operations at both the head and the tail of the list.

DATA INTERFACE:

Name	**Description**
None.	

DATA TYPE INTERFACE:

Name	**Description**
gldd_list	Generic pointer type for this list structure.
gldd_node	Generic node type for this list structure.

INVOCATION INTERFACE:

Procedure / **Parameters**	**Description** / **Description** (Type,IN/OUT, etc.)
gldd_create_empty_list	Initialize List to be an empty list.
List	gldd_list, IN-OUT, A new list structure.
gldd_is_empty_list	Determine if List is an empty list.
List	gldd_list, IN, A list structure.
[return]	Bool, OUT, TRUE if List is empty. FALSE otherwise.
gldd_append	Add a node to the tail of the list.
Node	gldd_node, IN, The new node.
List	gldd_list, IN-OUT, An existing list structure.
gldd_append_list	Add a new list to the tail of an existing list.
NewList	gldd_list, IN, The new list.
OldList	gldd_list, IN-OUT, The existing list structure.
gldd_delete	Remove a node from the middle of the list.
Node	gldd_node, IN, The node to be removed.
List	gldd_list, IN-OUT, An existing list

structure.

IMPLEMENTATION FILES:
GLDD_Double_Double_List.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
Bool	Data Type	UT_Utility

NOTES:

None.

GLDE DOUBLE-ENDED LINKED LIST MODULE

PURPOSE:

Implement one-way, linked lists, with quick access to both ends of the list. These lists make efficient use of memory and quickly perform insertion and deletion operations to both ends of the list. Insertion and deletion operations performed on other parts of the list may be quite inefficient.

DATA INTERFACE:

Name	**Description**
None.	

DATA TYPE INTERFACE:

Name	**Description**
glde_list	Generic pointer type for this list structure.
glde_node	Generic node type for this list structure.

INVOCATION INTERFACE:

Procedure	**Description**	
Parameters		**Description** (Type,IN/OUT, etc.)
glde_create_empty_list	Initialize List to be an empty list.	
List		glde_list, IN-OUT, A new list structure.
glde_is_empty_list	Determine if List is an empty list.	
List		glde_list, IN, A list structure.
[return]		Bool, OUT, TRUE if List is empty. FALSE otherwise.
glde_push	Add a node to the head of the list.	
Node		glde_node, IN, The new node.
List		glde_list, IN-OUT, An existing list structure.
glde_append	Add a node to the tail of the list.	
Node		glde_node, IN, The new node.
List		glde_list, IN-OUT, An existing list structure.
glde_pop	Remove a node from the head of the list.	
List		glde_list, IN-OUT, An existing list structure.
Node		glde_node, OUT, The removed node.

glde_delete	Remove a node from the middle of the list.	
	Node	glde_node, IN, The node to be removed.
	List	glde_list, IN-OUT, An existing list structure.

IMPLEMENTATION FILES:

GLDE_Double_Ended.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
Bool	Data Type	UT_Utility

NOTES:

None.

GLDL DOUBLE LINKED LIST MODULE

PURPOSE:

Implement two-way (double), linked lists. These lists are not quite as efficient as other linked lists in their use of memory, but deletion operations are performed quickly for any position in the list.

DATA INTERFACE:

Name	Description
None.	

DATA TYPE INTERFACE:

Name	Description
gldl_list	Generic pointer type for this list structure.
gldl_node	Generic node type for this list structure.

INVOCATION INTERFACE:

Procedure	Description	
Parameters		Description (Type, IN/OUT, etc.)
gldl_create_empty_list	Initialize List to be an empty list.	
List		gldl_list, IN-OUT, A new list structure.
gldl_is_empty_list	Determine if List is an empty list.	
List		gldl_list, IN, A list structure.
[return]		Bool, OUT, TRUE if List is empty. FALSE otherwise.
gldl_push	Add a node to the head of the list.	
Node		gldl_node, IN, The new node.
List		gldl_list, IN-OUT, An existing list structure.
NextField		C field name, IN, Name of forward pointer field in gldl_node record structure.
PrevField		C field name, IN, Name of backwards pointer field in gldl_node record structure.
gldl_delete	Remove a node from the middle of the list.	
Node		gldl_node, IN, The node to be removed.
List		gldl_list, IN-OUT, An existing list structure.
NextField		C field name, IN, Name of forward pointer field in gldl_node record

	structure.
PrevField	C field name, IN, Name of backwards pointer field in gldl_node record structure.

IMPLEMENTATION FILES:

GLDL_Double_Link.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
Bool	Data Type	UT_Utility
C field name	Data Type	UT_Utility

NOTES:

This module is called macros.h in the UNIX implementation of SR.

GLLL LINKED LIST MODULE

PURPOSE:

Implement one-way, linked lists. These lists make efficient use of memory and quickly perform insertion and deletion operations to the head of the list. Insertion and deletion operations performed on other parts of the list may be quite inefficient.

DATA INTERFACE:

Name	**Description**
None.	

DATA TYPE INTERFACE:

Name	**Description**
glll_list	Generic pointer type for this list structure.
glll_node	Generic node type for this list structure.

INVOCATION INTERFACE:

Procedure	**Parameters**	**Description**	**Description** (Type, IN/OUT, etc.)
glll_create_empty_list		Initialize List to be an empty list.	
	List		glll_list, IN-OUT, A new list structure.
glll_is_empty_list		Determine if List is an empty list.	
	List		glll_list, IN, A list structure.
	[return]		Bool, OUT, TRUE if List is empty. FALSE otherwise.
glll_push		Add a node to the head of the list.	
	Node		glll_node, IN, The new node.
	List		glll_list, IN-OUT, An existing list structure.
glll_pop		Remove a node from the head of the list.	
	List		glll_list, IN-OUT, An existing list structure.
	Node		glll_node, OUT, The removed node.
glll_delete		Remove a node from the middle of the list.	
	Node		glll_node, IN, The node to be removed.
	List		glll_list, IN-OUT, An existing list structure.

IMPLEMENTATION FILES:
GLLL_Linked_List.h

IMPORTED ELEMENTS:

Name	Type	Module
Bool	Data Type	UT_Utility
C field name	Data Type	UT_Utility
C type	Data Type	UT_Utility

NOTES:

None.

RTS LANGUAGE SUBSYSTEM (LG) DESCRIPTION

Function

The Language (LG) Subsystem provides the functionality for SR Language-specific concepts, which are too complex to implement with in-line code. For example, the LG subsystem implements Virtual Machines, Resources, and Operations. Almost every module in LG implements an SR concept or statement directly.

Design

The dependencies between LG modules are fairly simple. Most modules only depend on one or two other LG modules. The two exceptions are LGMN_Main which calls almost every other module to initialize the RTS, and LGIV_Invoke which calls several other modules to implement the several different types of invocation.

The LG dependencies on other Subsystem modules are more complex. The LG modules only depend on two or three OS_Operating_System modules, but they often depend on six or seven DS_Data_Structure modules. The reason for the large number of DS modules is that the LG modules often must traverse the RTS data structure to find the information they need. In the course of traversing the data structure, they use the DS descriptors and data access procedures. Most LG modules also use several of the MC_Machine modules. Taken collectively, the LG modules use almost every other module in the RTS. This is not surprising since LG supplies most of the interface to the Generated Code (GC), and the rest of the RTS is written to support that interface.

There are two circular dependencies in the LG Dependency Diagram. Neither of them are cause for concern.

The circular dependency between LGVM_Virtual_Machine and LGRT_Remote_Tx occurs because the LGVM sr_create and sr_destroy procedures need to do sr_remote calls, and the LGRT sr_remote procedure needs to call sr_vm_connect in LGVM if the requested VM's communication address is unknown. Since the sr_vm_connect procedure does not depend on any other LG modules, there is no possibility of recursion or deadlock. We will need a stub for sr_vm_connect during the testing of LGRT_Remote_Tx.

The circular dependency between LGIV_Invoke and LGCO_Concurrent occurs because the LGIV sr_invoke procedure depends on LGCO to implement concurrent invocations, and LGCO must sometimes make a copy of an invocation descriptor, which it does by calling sr_dup_inv in LGIV. The sr_dup_inv procedure has no dependencies other than the obvious need to use the invocation descriptor. SR_dup_inv is a simple copy procedure. There is no possibility of recursion or deadlock. We will need a stub for sr_dup_inv during the testing of LGCO_Concurrent.

The internal design of some of the LG modules is quite complex. In particular the LGIV_Invoke and the LGIN_Input_Op modules must distinguish between many different types of invocations and implement each type as efficiently as possible. For more information on these design issues, refer to "An Overview of the SR Language and Implementation".

LGCL CLASS MODULE

PURPOSE:

Implement equivalence classes for input operations. A class stores information about the input operations and all the pending invocations on those input operations.

Section 4.2.2. The Input Statement, in "An Overview of the SR Language and Implementation", has a complete description of classes and their use in the SR RTS.

DATA INTERFACE:

Name	**Description**

None.

DATA TYPE INTERFACE:

Name	**Description**

None.

INVOCATION INTERFACE:

Procedure	**Description**	
Parameters		**Description** (Type,IN/OUT, etc.)
sr_init_class	Initialize this module.	
sr_make_class	Create a new class.	
[return]		class, OUT, The new class descriptor.
sr_free_class	Kill an old class.	
clap		class, IN, The class to be killed.

IMPLEMENTATION FILES:

LGCL_Class.c
LGCL_Class_i.h
LGCL_Class_h.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
ossf_declare_free_list	Procedure	OSSF_Safe_FreeList
ossf_init_free_list	Procedure	OSSF_Safe_FreeList
ossf_get_node	Procedure	OSSF_Safe_FreeList
ossf_free_node	Procedure	OSSF_Safe_FreeList
sr_class_count	Data (Update)	DSCL_Class
sr_max_classes	Data (Read)	DSCL_Class
class	Data Type	DSCL_Class

class_st	Data Type	DSCL_Class
create_invQ	Procedure	DSIN_Invoke
create_procQ	Procedure	DSPR_Process
sr_check_stk	Procedure	MCPR_Process
Bool	Data Type	UT_Util

NOTES:

None.

LGCO CONCURRENT MODULE

PURPOSE:

Implement the SR co statement. This statement executes a number of SR statements concurrently.

The "Revised Report on the SR Language" has more information about the SR co statement.

DATA INTERFACE:

Name **Description**

None.

DATA TYPE INTERFACE:

Name **Description**

None.

INVOCATION INTERFACE:

Procedure **Description**

Parameters **Description** (Type, IN/OUT, etc.)

sr_init_co — Initialize this module and this VM.

sr_co_start — Start a co statement by initializing a co descriptor and linking it to the current process.

sr_co_call — Initialize the invocation and co descriptors for a call from a co statement.

- ibp — invb, IN-OUT, The invocation descriptor.

sr_co_call_done — Finalize the invocation and co descriptors after a call from a co statement has completed. If the invoker is still interested in this event, notify him.

- ibp — invb, IN-OUT, The invocation descriptor.

sr_co_send — Initialize the invocation and co descriptors for a send from a co statement.

- ibp — invb, IN-OUT, The invocation descriptor.

sr_co_wait — Wait for a co invocation to terminate. Return a pointer to the original invocation descriptor so that the GC (Generated Code) can copy result parameters and find out which arm terminated.

[return]	invb, IN-OUT, The invocation descriptor.
sr_co_end	Finalize a co statement. Release the co descriptor.

IMPLEMENTATION FILES:

LGCO_Concurrent.c
LGCO_Concurrent_i.h
LGCO_Concurrent_h.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
sr_dup_invb	Procedure	LGIN_Invoke
sr_free	Procedure	OSMM_Memory
sr_kill_sem	Procedure	OSS4_Semaphore
sr_make_sem	Procedure	OSS4_Semaphore
P	Procedure	OSS4_Semaphore
V	Procedure	OSS4_Semaphore
ossf_declare_free_list	Procedure	OSSF_Safe_FreeList
ossf_init_free_list	Procedure	OSSF_Safe_FreeList
ossf_get_node	Procedure	OSSF_Safe_FreeList
ossf_free_node	Procedure	OSSF_Safe_FreeList
INIT_SEQ_CO	Data (Read)	DSCO_Concurrent
cob	Data Type	DSCO_Concurrent
cob_st	Data Type	DSCO_Concurrent
invb	Data Type	DSIN_Invoke
dsin_invb_co	Procedure	DSIN_Invoke
sr_max_co_stmts	Data (Read)	DSLM_Limits
sr_cur_proc	Data (Update)	DSPR_Process
dspr_proc_co_list	Procedure	DSPR_Process
tindex	Data Type	GLAR_Array
sr_check_stk	Procedure	MCPR_Process
daddr	Data Type	UT_Util

NOTES:

None.

LGIN INVOKE MODULE

PURPOSE:

Implement the SR invocation statements: call, send and reply. The SR invocation mechanisms are quite sophisticated. The implementation uses a sophisticated design to handle the different types of invocation, and the different types of invocation termination.

The "Revised Report on the SR Language" has more information about the SR invocation concepts.

DATA INTERFACE:

Name	Description
None.	

DATA TYPE INTERFACE:

Name	Description
None.	

INVOCATION INTERFACE:

Procedure	Parameters	Description	Description (Type,IN/OUT, etc.)
sr_invoke		Invoke a proc or input operation with either a call or a send.	
	ibp		invb, IN, The invocation descriptor.
	[return]		invb, OUT, The new invocation descriptor describing the current state of the invocation.
sr_reply		Send an early reply to the invoker of an operation. This implements the reply statement.	
	ibp		invb, IN, The invocation descriptor.
	[return]		invb, OUT, The new invocation descriptor describing the current state of the invocation.
sr_finished_input		Clean up a finished input operation.	
	ibp		invb, IN, The invocation descriptor.
sr_finished_proc		Clean up a finished proc operation.	
	ibp		invb, IN, The invocation descriptor.

sr_rej_inv — Reject an invocation because the operation was killed before the invocation was accepted.

- ibp — invb, IN, The invocation descriptor.

sr_dup_inv — Duplicate an invocation descriptor and return the address of the copy.

- ibp — invb, IN, The invocation descriptor.
- [return] — invb, OUT, The new invocation descriptor.

IMPLEMENTATION FILES:

LGIN_Invoke.c
LGIN_Invoke_i.h
LGIN_Invoke_h.h

IMPORTED ELEMENTS:

Name	Type	Module
sr_co_send	Procedure	LGCO_Concurrent
sr_co_call	Procedure	LGCO_Concurrent
sr_co_call_done	Procedure	LGCO_Concurrent
sr_invk_iop	Procedure	LGIP_Iop
sr_own_alloc	Procedure	OSMM_Memory
sr_activate	Procedure	LGPR_Process
sr_kill	Procedure	LGPR_Process
sr_remote	Procedure	LGRT_Remote_Tx
sr_kill_sem	Procedure	OSS4_Semaphore
sr_make_sem	Procedure	OSS4_Semaphore
P	Procedure	OSS4_Semaphore
V	Procedure	OSS4_Semaphore
INVOCATION_HEADER_SIZE	Data (Read)	DSIN_Invoke
invb	Data Type	DSIN_Invoke
in_type	Data Type	DSIN_Invoke
invk_argsize	Procedure	DSIN_Invoke
invk_opcap	Procedure	DSIN_Invoke
invk_type	Procedure	DSIN_Invoke
invk_wait	Procedure	DSIN_Invoke
RTS_OWN	Data (Read)	DSMM_Memory
pach	Data Type	DSNE_Network
ms_type	Data Type	DSNE_Network
sr_optab	Data (Read)	DSOP_Operation
oper	Data Type	DSOP_Operation
op_type	Data Type	DSOP_Operation
opcap_opindex	Procedure	DSOP_Operation
opcap_seqn	Procedure	DSOP_Operation
opcap_vm	Procedure	DSOP_Operation
oper_code	Procedure	DSOP_Operation
oper_inclass	Procedure	DSOP_Operation

oper_res	Procedure	DSOP_Operation
oper_seqn	Procedure	DSOP_Operation
oper_type	Procedure	DSOP_Operation
sr_cur_proc	Data (Update)	DSPR_Process
proc	Data Type	DSPR_Process
pr_type	Data Type	DSPR_Process
proc_intype	Procedure	DSPR_Process
proc_invoke	Procedure	DSPR_Process
proc_prtype	Procedure	DSPR_Process
proc_wait	Procedure	DSPR_Process
sr_cur_res	Data (Update)	DSRE_Resource
res_status	Data Type	DSRE_Resource
rint_capsize	Procedure	DSRE_Resource
rint_create	Procedure	DSRE_Resource
rint_rescap	Procedure	DSRE_Resource
rint_status	Procedure	DSRE_Resource
rint_varbase	Procedure	DSRE_Resource
sem	Data Type	DSS4_Semaphore
sr_my_vm	Data (Read)	DSVM_Virtual_Machine
sr_rtserror	Procedure	MCEX_Exception
sr_abort	Procedure	MCEX_Exception
sr_free	Procedure	MCMM_Memory
sr_check_stk	Procedure	MCPR_Process
NOOP_SEQN	Data (Read)	UT_Util
daddr	Data Type	UT_Util
Bool	Data Type	UT_Util
memcpy	Procedure	V-system

NOTES:

None.

LGIP INPUT OPERATIONS MODULE

PURPOSE:

The input statement is the most complicated statement in the SR language. This module implements the basic input operation processing: invoke input operations and retrieve input operation invocations (done by processes which execute input operations).

Section 4.2.2. The Input Statement in the "Overview of the SR Language and Implementation" has a good description of the input statement implementation and the use of equivalence classes (LGCL_Class module).

DATA INTERFACE:

Name **Description**

None.

DATA TYPE INTERFACE:

Name **Description**

None.

INVOCATION INTERFACE:

Procedure **Description**
Parameters **Description** (Type, IN/OUT, etc.)

sr_invk_iop	Invoke an input operation.	
ibp		invb, IN-OUT, The invocation descriptor.
clap		class, IN-OUT, The input operation's equivalence class.
sr_iaccess	Get access to an input operation class. This allows the Generated Code (GC) to start searching for an eligible invocation.	
clap		class, IN-OUT, The input operation's equivalence class.
sr_reaccess	Regain subsequent access to an input operation class.	
sr_rm_iop	Remove an invocation descriptor from the specified input operation queue. The Generated Code (GC) can service the invocation now.	
ibp		invb, IN-OUT, The invocation descriptor.

IMPLEMENTATION FILES:

LGIP_Input_Operation.c
LGIP_Input_Operation_i.c
LGIP_Input_Operation_h.c

IMPORTED ELEMENTS:

Name	Type	Module
awaken	Procedure	OSSH_Scheduler
block	Procedure	OSSH_Scheduler
sr_cswitch	Procedure	OSSH_Scheduler
class	Data Type	DSCL_Class
class_inuse	Procedure	DSCL_Class
class_newin	Procedure	DSCL_Class
class_newpr	Procedure	DSCL_Class
class_oldin	Procedure	DSCL_Class
class_oldpr	Procedure	DSCL_Class
class_pending	Procedure	DSCL_Class
invb	Data Type	DSIN_Invoke
invk_next	Procedure	DSIN_Invoke
append_invQ	Procedure	DSIN_Invoke
append_list_invQ	Procedure	DSIN_Invoke
delete_invQ	Procedure	DSIN_Invoke
sr_optab	Data (Read)	DSOP_Operation
oper	Data Type	DSOP_Operation
opcap_opindex	Procedure	DSOP_Operation
oper_pending	Procedure	DSOP_Operation
sr_cur_proc	Data (Update)	DSPR_Process
proc	Data Type	DSPR_Process
proc_class	Procedure	DSPR_Process
proc_next_inv	Procedure	DSPR_Process
proc_next	Procedure	DSPR_Process
sr_check_stk	Procedure	MCPR_Process
Bool	Data Type	UT_Util

NOTES:

None.

LGMN MAIN MODULE

PURPOSE:

This module initializes all the modules in the RTS. If this is the first RTS then it creates the main resource. Otherwise, it just waits for requests from remote VMs.

This module starts the RTS on each VM. The first RTS is invoked from the operating system command-line. This initial invocation is the **program startup** which may include program parameters. These parameters are ignored by the RTS and passed to the SR program. Every subsequent invocation is a **VM startup** which is the result of a VM create statement. In this case, all the parameters are used for the RTS initialization.

DATA INTERFACE:

Name	**Description**

None.

DATA TYPE INTERFACE:

Name	**Description**

None.

INVOCATION INTERFACE:

Procedure	**Parameters**	**Description**	**Description** (Type,IN/OUT, etc.)
main		Initialize the first RTS (program startup).	
	argc		int, IN, Number of arguments.
	argv		char **, IN, Array of arguments.
main		Initialize all the subsequent RTSs (VM startup).	
	argv[1]		char *, IN, A magic string (VM_MAGIC) indicating that this is a VM startup.
	argv[2]		char *, IN, Physical machine number for the new VM.
	argv[3]		char *, IN, Virtual machine number for the new VM.
	argv[4]		char *, IN, Network address of the of the srx [srx's PID].
	argv[5]		char *, IN, Debugging flags. Used to initialize the MCDE_Debug module.
	argv[6]		char *, IN, Program group communication address. Used by the OSGP_Group module.

IMPLEMENTATION FILES:

LGMN_Main.c
LGMN_Main_i.h
LGMN_Main_h.h

IMPORTED ELEMENTS:

Name	Type	Module
sr_init_class	Procedure	LGCL_Class
sr_init_co	Procedure	LGCO_Concurrent
sr_init_io	Procedure	LGIO_Input_Output
sr_argv	Data (Update)	LGMS_Miscellaneous
sr_argc	Data (Update)	LGMS_Miscellaneous
sr_init_oper	Procedure	LGOP_Operation
sr_init_proc	Procedure	LGPR_Process
sr_kill	Procedure	LGPR_Process
sr_create	Procedure	LGRE_Resource
sr_init_res	Procedure	LGRE_Resource
sr_init_remote_Rx	Procedure	LGRR_Remote_Rx
sr_init_remote_Tx	Procedure	LGRT_Remote_Tx
VM_MAGIC	Data (Read)	LGVM_Virtual_Machine
sr_init_mem	Procedure	OSMM_Memory
sr_own_alloc	Procedure	OSMM_Memory
sr_init_net	Procedure	OSNE_Network
sr_init_pool	Procedure	OSPL_Pool
sr_init_sem	Procedure	OSS4_Semaphore
sr_pgmgroup	Data (Update)	DSGP_Group
RTS_OWN	Data (Read)	DSMM_Memory
sr_cur_proc	Data (Read)	DSPR_Process
crbp	Data Type	DSRE_Resource
crb_st	Data Type	DSRE_Resource
rint	Data Type	DSRE_Resource
crb_rpatid	Data Type	DSRE_Resource
crb_vm	Data Type	DSRE_Resource
MAIN_VM	Data (Read)	DSVM_Virtual_Machine
sr_my_machine	Data (Update)	DSVM_Virtual_Machine
sr_my_vm	Data (Update)	DSVM_Virtual_Machine
sr_init_debug	Procedure	MCDE_Debug
sr_my_label	Data (Update)	MCEX_Exception
sr_trace_flag	Data (Read)	MCEX_Exception
stderr	Data (Update)	V-system
sprintf	Procedure	V-system

NOTES:

None.

LGMS MISCELLANEOUS MODULE

PURPOSE:

Implement a miscellaneous group of procedures which are useful for the Generated Code (GC). Included in this group of procedures are string manipulation procedures, max and min procedures, copy procedures, memory allocation procedures and command-line argument, access procedures.

DATA INTERFACE:

Name	**Description**
sr_argc	When an SR program is started, the command line may include several arguments. This variable gives the number of command line arguments.
sr_argv	This variable contains the command line arguments.

DATA TYPE INTERFACE:

Name	**Description**
sr_string	A string descriptor. All SR string variables are stored in this format.

INVOCATION INTERFACE:

Procedure	**Parameters**	**Description**	**Description** (Type,IN/OUT, etc.)
sr_cat		Concatenate all the string arguments and copy them to a new string. Return the address of the new string.	
	va_alist		va_dcl, IN, A list of argument pairs. The first argument in a pair is a char* pointer to a string, and the second argument is an int which gives the length of the string. The list of argument pairs is terminated with the pair: "NULL, 0".
	[return]		daddr, OUT, The address of the new string containing the concatenation of all the argument strings.

sr_strcmp — Compare two strings. Return a number indicating which string is larger. Return a negative number if the left string is less than the right; return a positive number if the left string is greater than the right; and return 0 if the two strings are equal.

laddr	char *, IN, The left string.
llen	int, IN, The left string length.
raddr	char *, IN, The right string.
rlen	int, IN, The right string length.
[return]	int, OUT, The comparison result.

sr_str_result — Copy a null terminated string to an SR string.

p	char *, IN, The null terminated string.
pstr	sr_string, IN-OUT, The SR string structure.
len	int, IN, The maximum length of p.

sr_max — Return the maximum of the integer arguments.

n	int, The number of integer arguments.
va_alist	va_dcl, IN, A list of n integer arguments.
[return]	int, The maximum integer in va_alist.

sr_min — Return the minimum of the integer arguments.

n	int, The number of integer arguments.
va_alist	va_dcl, IN, A list of n integer arguments.
[return]	int, The minimum integer in va_alist.

sr_clone — Make n copies (clones) of a memory block. All the clones are located in the memory area immediately after the original copy. I.e. the memory block from addr to (addr+len-1) is copied to (addr+len), (addr+ 2*len), (addr+ 3*len), etc.

addr	daddr, IN, The address of the original.
len	int, IN, The length of of the memory block.
n	int, IN, The number of clones to make.
[return]	daddr, OUT, Pointer to the memory location immediately after the last clone.

sr_swap — Swap two items in memory. If len is 0, then the items are strings, and the maximum of the current lengths is to be used.

- laddr — char *, IN-OUT, The left item.
- raddr — char *, IN-OUT, The right item.
- len — int, IN, The length of the items. 0 indicates that the maximum string length is to be used.

sr_new — Allocate memory for an SR new(type) call.

- len — int, IN, The length of the memory block.
- [return] — daddr, OUT, The address of the memory block.

sr_newfree — Deallocate a memory block allocated by sr_new.

- addr — daddr, IN, The address of the memory block. If this NULL, then do nothing.

sr_numargs — Return the number of command line arguments.

- [return] — int, The number of arguments.

sr_arg_bool — Interpret command line argument n as a Boolean literal. Assign its Boolean value to pBool. If this procedure is successful then return TRUE. Otherwise, return FALSE.

- n — int, IN, The argument number.
- pBool — Bool *, OUT, The Boolean value of the argument.
- [return] — Bool, OUT, Exit status of procedure.

sr_arg_int — Interpret command line argument n as an integer literal. Assign its value to pint. If this procedure is successful then return TRUE. Otherwise, return FALSE.

- n — int, IN, The argument number.
- pBool — int *, OUT, The integer value of the argument.
- [return] — Bool, OUT, Exit status of procedure.

sr_arg_char — Copy the n'th command line argument to an SR char array. If this procedure is successful then return TRUE. Otherwise, return FALSE.

- n — int, IN, The argument number.
- pstr — char *, OUT, The character array.
- len — int, IN, The maximum length of the string.
- [return] — Bool, OUT, Exit status of procedure.

sr_arg_string	Copy the n'th command line argument to an SR string. If this procedure is successful then return TRUE. Otherwise, return FALSE.	
n		int, IN, The argument number.
pstr		sr_string, OUT, The SR string variable.
len		int, IN, The maximum length of the string.
[return]		Bool, OUT, Exit status of procedure.

IMPLEMENTATION FILES:

LGMS_Miscellaneous.c
LGMS_Miscellaneous_i.h
LGMS_Miscellaneous_h.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
sr_own_alloc	Procedure	OSMM_Memory
va_alist	Data Type	OSVA_Variable_ArgList
va_dcl	Data Type	OSVA_Variable_ArgList
va_list	Data Type	OSVA_Variable_ArgList
va_start	Data Type	OSVA_Variable_ArgList
va_end	Data Type	OSVA_Variable_ArgList
va_arg	Data Type	OSVA_Variable_ArgList
DEBUG	Procedure	MCDE_Debug
sr_abort	Procedure	MCEX_Exception
sr_check_stk	Procedure	MCPR_Process
MAX_INTEGER	Data (Read)	UT_Util
MIN_INTEGER	Data (Read)	UT_Util
Bool	Data Type	UT_Util
daddr	Data Type	UT_Util
EOF	Data (Read)	V-system
free	Procedure	V-system
malloc	Procedure	V-system
memcpy	Procedure	V-system
sscanf	Procedure	V-system

NOTES:

None.

LGNP NETPATH MODULE

PURPOSE:

This module is responsible for building a path to the program's executable file. This path is needed whenever this VM attempts to start another VM.

Because of the limitations of some network operating systems, the path to a file is not the same on every machine. I.e. the path to file 'prog.exe' on machine X may not be the same as the path to file 'prog.exe' on machine Y. SR allows these different paths to be documented in a file called the **mapfile**. This module uses the mapfile to build the executable path for this machine.

For more information about mapfiles, refer to the example mapfiles in the main SR source directory.

DATA INTERFACE:

Name	**Description**

None.

DATA TYPE INTERFACE:

Name	**Description**

None.

INVOCATION INTERFACE:

Procedure / **Parameters**	**Description** / **Description** (Type,IN/OUT, etc.)
sr_netpath	Build a network path for the filename. Return a pointer to the network path or NULL if we can not build the path.
fname	char *, IN, The name of the file.
dir	char *, IN, The name of the directory containing fname. This directory path does not contain the hostname or any network information. This parameter is ignored if fname includes the full pathname.
mapfile	char *, IN, The network path for the mapfile.
result	char *, OUT, The network path for fname. Null if it can not be built.
[return]	char *, OUT, The network path for fname. Null if it can not be built.

IMPLEMENTATION FILES:

LGNP_Netpath.c
LGNP_Netpath_i.h
LGNP_Netpath_h.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
sr_open	Procedure	LGIO_Input_Output
sr_close	Procedure	LGIO_Input_Output
HOST_NAME_LEN	Data (Read)	DSOS_Operating_System
MAX_PATH	Data (Read)	DSOS_Operating_System
MAX_LINE	Data (Read)	DSOS_Operating_System
DEBUG	Procedure	MCDE_Debug
sr_rtserror	Data (Update)	MCEX_Exception
sr_rts_warn	Data (Update)	MCEX_Exception
sr_net_abort	Procedure	MCEX_Exception
FILE	Data Type	V-system
SystemCode	Data Type	V-system
fgets	Procedure	V-system
isspace	Procedure	V-system
perror	Procedure	V-system
strchr	Procedure	V-system
strlen	Procedure	V-system
strncpy	Procedure	V-system
sprintf	Procedure	V-system
QueryWorkstationConfig	Procedure	V-system

NOTES:

None.

LGOP OPERATION MODULE

PURPOSE:

Implement the procedures and data structures for SR operations. Construct the operation descriptors and capabilities when new operations are created, and remove the descriptors and capabilities when operations are killed. Find 'eligible' operation invocations in invocation lists.

The "Revised Report on the SR Language" has more information about SR operations.

DATA INTERFACE:

Name	**Description**
None.	

DATA TYPE INTERFACE:

Name	**Description**
None.	

INVOCATION INTERFACE:

Procedure — **Description**
Parameters — **Description** (Type, IN/OUT, etc.)

sr_init_oper — Initialize this module.

sr_make_resops

Add a list of new operations to a resource. Called during resource initialization.

- va_alist — va_dcl, IN, A list of operations. There are two or three arguments per operation. The first argument for an operation specifies the operation type (op_type). If the operation is a PROC_OP or a PROC_REP_OP, then the next argument is the proc code address (paddr). If the operation is an INPUT_OP then the next two arguments are the input class (class) and the number of operations of this type (int). The list is terminated by END_OP. Any other operation type causes a fatal error.

sr_kill_resops

Kill all the resource operations for the named resource instance. Remove any pending input invocations.

- res — rint, IN-OUT, The resource

instance.

sr_make_liop   Make a set of local input operations.

- clap — class, IN, The class for the input operations.
- opcp — opcap *, IN-OUT, Pointer to the first operation capability in an array of operation capabilities.
- count — int, IN, The number of input operations to be created.

sr_kill_liop   Kill local input operations. Purge any pending invocations from the class queues. If the killed operation is the last of its class, free the class as well.

- opcp — opcap *, IN, Pointer to the first operation capability in an array of operation capabilities.
- count — int, IN, The number of input operations to be killed.

sr_get_anyinv   Get the next eligible invocation descriptor for the GC (Generated Code) to check in processing an input statement. The current process must have access to the operation class. If no invocations are available, wait until more arrive.

- [return] — invb, OUT, The next eligible invocation descriptor.

sr_get_myinv   Get the next eligible invocation of the specified operation. If none are available, wait until more arrive.

- opc — opcap, IN, The operation capability descriptor for the operation to match on.
- [return] — invb, OUT, The next eligible invocation descriptor for the specified operation.

sr_receive   Get the next invocation for operations appearing in a single class with no synchronization or scheduling expressions. This is an optimization.

- clap — class, IN, The operation's class.
- [return] — invb, OUT, The next invocation for the operation.

sr_make_semop   Create an operation to act as a semaphore. I.e. a non-exported, parameterless, operation in its own class. This is an optimization.

- [return] — sem, OUT, The semaphore descriptor for the operation.

sr_query_iop   Return the number of pending invocations for an input operation.

opc	opcap, IN, The operation descriptor.
[return]	int, OUT, The number of pending invocations for opc.

IMPLEMENTATION FILES:

LGOP_Operation.c
LGOP_Operation_i.h
LGOP_Operation_h.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
sr_iaccess	Procedure	LGIP_Iop
sr_reaccess	Procedure	LGIP_Iop
sr_rm_iop	Procedure	LGIP_Iop
sr_rej_inv	Procedure	LGIN_Invoke
sr_kill_sem	Procedure	OSS4_Semaphore
sr_make_sem	Procedure	OSS4_Semaphore
ossf_declare_free_list	Procedure	OSSF_Safe_FreeList
ossf_init_free_list	Procedure	OSSF_Safe_FreeList
ossf_get_node	Procedure	OSSF_Safe_FreeList
ossf_free_list	Procedure	OSSF_Safe_FreeList
osva_va_alist	Data Type	OSVA_Variable_ArgList
osva_va_dcl	Data Type	OSVA_Variable_ArgList
osva_va_list	Data Type	OSVA_Variable_ArgList
osva_start	Procedure	OSVA_Variable_ArgList
osva_arg	Procedure	OSVA_Variable_ArgList
osva_end	Procedure	OSVA_Variable_ArgList
class	Data Type	DSCL_Class
class_num_ops	Procedure	DSCL_Class
class_oldin	Procedure	DSCL_Class
class_newin	Procedure	DSCL_Class
invb	Data Type	DSIN_Invoke
inv_queue	Data Type	DSIN_Invoke
is_empty_invList	Procedure	DSIN_Invoke
next_invList	Procedure	DSIN_Invoke
remove_invList	Procedure	DSIN_Invoke
is_empty_invQ	Procedure	DSIN_Invoke
top_invQ	Procedure	DSIN_Invoke
next_invQ	Procedure	DSIN_Invoke
pop_invQ	Procedure	DSIN_Invoke
remove_invQ	Procedure	DSIN_Invoke
invk_opcap	Procedure	DSIN_Invoke
END_OP	Data (Read)	DSOP_Operation
INIT_SEQ_OP	Data (Read)	DSOP_Operation
sr_max_operations	Data (Read)	DSOP_Operation
sr_no_ocap	Data (Update)	DSOP_Operation

sr_nu_ocap	Data (Update)	DSOP_Operation
sr_optab	Data (Update)	DSOP_Operation
END_OP	Data (Read)	DSOP_Operation
opcap	Data Type	DSOP_Operation
oper	Data Type	DSOP_Operation
oper_st	Data Type	DSOP_Operation
op_type	Data Type	DSOP_Operation
opcap_opindex	Procedure	DSOP_Operation
opcap_seqn	Procedure	DSOP_Operation
oper_code	Procedure	DSOP_Operation
oper_inclass	Procedure	DSOP_Operation
oper_res	Procedure	DSOP_Operation
oper_seqn	Procedure	DSOP_Operation
oper_type	Procedure	DSOP_Operation
is_empty_op_list	Procedure	DSOP_Operation
delete_oper	Procedure	DSOP_Operation
pop_oper	Procedure	DSOP_Operation
sr_cur_proc	Data (Update)	DSPR_Process
is_proc_else_leg	Procedure	DSPR_Process
proc_next_inv	Procedure	DSPR_Process
sr_cur_res	Data (Update)	DSRE_Resource
rescap	Data Type	DSRE_Resource
rint	Data Type	DSRE_Resource
rescap_opcap	Procedure	DSRE_Resource
rint_ops	Procedure	DSRE_Resource
rint_num_ops	Procedure	DSRE_Resource
rint_varbase	Procedure	DSRE_Resource
sem	Data Type	DSS4_Semaphore
sr_my_vm	Data (Read)	DSVM_Virtual_Machine
sr_abort	Procedure	MCEX_Exception
sr_alloc	Procedure	MCMM_Memory
paddr	Data Type	MCPR_Process
sr_check_stk	Procedure	MCPR_Process
NOOP_SEQN	Data (Read)	UT_Util
NULL_SEQN	Data (Read)	UT_Util
Bool	Data Type	UT_Util

NOTES:

None.

LGPR PROCESS MODULE

PURPOSE:

Implement the SR process module. SR processes are very lightweight. However, there is no time-slicing between SR processes. This means that an SR process will monopolize the cpu until it blocks itself. Refer to any operating systems text for more infomation about processes and the standard operations.

DATA INTERFACE:

Name	**Description**
None.	

DATA TYPE INTERFACE:

Name	**Description**
None.	

INVOCATION INTERFACE:

Procedure / **Parameters**	**Description** / **Description** (Type,IN/OUT, etc.)
sr_init_proc	Initialize the process module and start the specified code in an SR process context. This procedure never returns control to the calling procedure.
start_code	paddr, IN, Initial SR process to execute.
sr_spawn	Create a new process.
pc	paddr, IN, Process code address.
res	rint, IN, Resource which owns the process.
arg1	int, IN, Process's first argument.
arg2	int, IN, Process's second argument.
arg3	int, IN, Process's third argument.
arg4	int, IN, Process's fourth argument.
[return]	proc, OUT, New process descriptor.
sr_activate	Make a new process ready to execute.
pr	proc, IN-OUT, The new process.
sr_kill	Delete a process and all references to it.
pr	proc, IN-OUT, The process to be deleted.
do_rem_proc	Bool, IN, Is this process owned by a resource?

IMPLEMENTATION FILES:

LGPR_Process.c
LGPR_Process_i.h
LGPR_Process_h.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
sr_cswitch	Procedure	OSSH_Scheduler
osuf_declare_free_list	Procedure	OSUF_Unsafe_FreeList
osuf_init_free_list	Procedure	OSUF_Unsafe_FreeList
osuf_get_node	Procedure	OSUF_Unsafe_FreeList
osuf_free_node	Procedure	OSUF_Unsafe_FreeList
sr_num_blocked	Data (Update)	DSPR_Process
sr_cur_proc	Data (Read)	DSPR_Process
proc	Data Type	DSPR_Process
paddr	Data Type	DSPR_Process
sr_enqueue	Procedure	DSPR_Process
sr_dequeue	Procedure	DSPR_Process
dspr_delete_proc	Procedure	DSPR_Process
sr_cur_res	Data (Read)	DSRE_Resource
dsre_rint_mutex	Procedure	DSRE_Resource
dsre_rint_procs	Procedure	DSRE_Resource
rint	Data Type	DSRE_Resource
DEBUG	Procedure	MCDE_Debug
sr_abort	Procedure	MCEX_Exception
sr_alloc	Procedure	MCMM_Memory
sr_build_context	Procedure	MCPR_Process
Bool	Data Type	UT_Util

NOTES:

None.

LGRE RESOURCE MODULE

PURPOSE:

Implement the SR resource module. SR resources are very similiar to classes in Object-Oriented Programming System (OOPS). One resource implements the data structure and all the operations for an Abstract Data Type. Many copies of a resource may be created during runtime. Each resouce copy is called a resource instance.

The most important operations for a resource are create and destroy. The "Revised Report on the SR Programming Language" has a complete description of SR resources.

DATA INTERFACE:

Name **Description**

None.

DATA TYPE INTERFACE:

Name **Description**

None.

INVOCATION INTERFACE:

Procedure / **Parameters**	**Description** / **Description** (Type, IN/OUT, etc.)
sr_init_res	Initialize the resource module.
sr_create	Create a resource instance.
crbp	crb, IN-OUT, Create Resource descriptor. This procedure assumes that crbp points to an allocated and initialized descriptor.
sr_destroy	Destroy a resource instance.
rcp	rescap *, IN-OUT, Resource Capability descriptor of the resource to be destroyed.
sr_dest_all	Destroy all the resource instances on this VM.
sr_alloc_rv	Start the resource initial proc. Allocate memory for resource variables and initialize the ID part of the resource capability.
size	int, IN, Byte size of memory block required.
[return]	daddr, OUT, Memory block pointer.

sr_finished_init
: Finish the resource initial process. Initialize the operation capabilities in the resource capability.

sr_finished_final
: The resource's final code has completed. Notify the destroyer.

sr_build_rcap Create a null or noop resource capability.

rcp	rescap, IN-OUT, Resource capability.
size	int, IN, Size of rcp descriptor.
ocp	opcap, IN, Null or noop value.

IMPLEMENTATION FILES:

LGRE_Resource.c
LGRE_Resource_i.h
LGRE_Resource_h.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
sr_spawn	Procedure	LGPR_Process
sr_activate	Procedure	LGPR_Process
sr_kill	Procedure	LGPR_Process
sr_remote	Procedure	LGRT_Remote_Tx
sr_kill_res_ops	Procedure	LGOP_Operation
sr_own_alloc	Procedure	OSMM_Memory
sr_free	Procedure	OSMM_Memory
sr_res_free	Procedure	OSMM_Memory
sr_create_sem	Procedure	OSS4_Semaphore
P	Procedure	OSS4_Semaphore
V	Procedure	OSS4_Semaphore
sr_kill_sem	Procedure	OSS4_Semaphore
ossf_declare_free_list	Procedure	OSSF_Safe_FreeList
ossf_init_free_list	Procedure	OSSF_Safe_FreeList
ossf_get_node	Procedure	OSSF_Safe_FreeList
ossf_free_node	Procedure	OSSF_Safe_FreeList
RTS_OWN	Data (Read)	DSMM_Memory
memh	Data Type	DSMM_Memory
dsmm_push_mem	Procedure	DSMM_Memory
dest_st	Data Type	DSNE_Network
creb_st	Data Type	DSNE_Network
ms_type	Data Type	DSNE_Network
MIN_MESG_SIZE	Data (Read)	DSOS_Operating_System
opcap	Data Type	DSOP_Operation
dsop_opcap_seq	Procedure	DSOP_Operation

sr_cur_proc	Data (Read)	DSPR_Process
proc	Data Type	DSPR_Process
proc_type	Data Type	DSPR_Process
INIT_SEQ_RES	Data (Read)	DSRE_Resource
INIT_REPLY	Data (Read)	DSRE_Resource
FREE_SLOT	Data (Read)	DSRE_Resource
FINAL_REPLY	Data (Read)	DSRE_Resource
sr_cur_res	Data (Read)	DSRE_Resource
sr_max_resources	Data (Read)	DSRE_Resource
sr_noop_res	Data (Update)	DSRE_Resource
sr_null_res	Data (Update)	DSRE_Resource
rint	Data Type	DSRE_Resource
rint_st	Data Type	DSRE_Resource
rpat	Data (Read)	DSRE_Resource
rescap	Data Type	DSRE_Resource
sem	Data Type	DSS4_Semaphore
NULL_VM	Data (Read)	DSVM_Virtual_Machine
NOOP_VM	Data (Read)	DSVM_Virtual_Machine
sr_my_vm	Data (Read)	DSVM_Virtual_Machine
DEBUG	Procedure	MCDE_Debug
sr_net_abort	Procedure	MSEX_Exception
sr_rts_abort	Procedure	MCEX_Exception
sr_rts_warn	Procedure	MCEX_Exception
sr_check_stk	Procedure	MCPR_Process
NOOP_SEQN	Data (Read)	UT_Util
NULL_SEQN	Data (Read)	UT_Util
Bool	Data Type	UT_Util
daddr	Data Type	UT_Util
tindex	Data Type	UT_Util
paddr	Data Type	UT_Util
sr_maxof	Procedure	UT_Util

NOTES:

None.

LGRR REMOTE RX MODULE

PURPOSE:

This module executes remote requests from other VMs. It is responsible for hiding the details of communication, and executing the requested operation.

This module is closely related to the LGRT_Remote_Tx module.

DATA INTERFACE:

Name	Description
None.	

DATA TYPE INTERFACE:

Name	Description
None.	

INVOCATION INTERFACE:

Procedure		Description
	Parameters	**Description** (Type,IN/OUT, etc.)
sr_init_remote_Rx		Initialize this module.
sr_rmt_create		Service a request to create a resource on this VM.
	client	sender, IN-OUT, The sender descriptor. Includes all the information about the request.
sr_rmt_destroy		Service a request to destroy a resource on this VM.
	client	sender, IN-OUT, The sender descriptor. Includes all the information about the request.
sr_rmt_destvm		Service a request to destroy this VM.
	client	sender, IN-OUT, The sender descriptor. Includes all the information about the request.
sr_rmt_invk		Service a request to invoke an operation on this VM.
	client	sender, IN-OUT, The sender descriptor. Includes all the information about the request.

IMPLEMENTATION FILES:

LGRR_Remote_Rx.c
LGRR_Remote_Rx_i.h
LGRR_Remote_Rx_h.h

IMPORTED ELEMENTS:

Name	Type	Module
sr_invoke	Procedure	LGIN_Invoke
sr_create	Procedure	LGRE_Resource
sr_destroy	Procedure	LGRE_Resource
sr_dest_all	Procedure	LGRE_Resource
sr_kill	Procedure	LGPR_Process
sr_free	Procedure	OSMM_Memory
sr_own_alloc	Procedure	OSMM_Memory
sr_freesender	Procedure	OSMR_Message_Rx
sr_net_reply	Procedure	OSMT_Message_Tx
invb	Data Type	DSIN_Invoke
in_type	Data Type	DSIN_Invoke
inv_type	Procedure	DSIN_Invoke
RTS_OWN	Data (Read)	DSMM_Memory
CREP_HEADER_SIZE	Data (Read)	DSMS_Message
sender	Data Type	DSMS_Message
sender_is_seg	Procedure	DSMS_Message
sender_pid	Procedure	DSMS_Message
sender_server_seg	Procedure	DSMS_Message
sender_client_seg	Procedure	DSMS_Message
sender_mesg	Procedure	DSMS_Message
dest_st	Data Type	DSNE_Network
MIN_MESG_SIZE	Data (Read)	DSOS_Operating_System
sr_cur_proc	Data (Read)	DSPR_Process
crb	Data Type	DSRE_Resource
crb_rescap	Procedure	DSRE_Resource
crep_st	Data Type	DSRE_Resource
crep_rescap	Procedure	DSRE_Resource
DEBUG	Procedure	MCDE_Debug
sr_abort	Procedure	MCEX_Exception
Bool	Data Type	UT_Util
daddr	Data Type	UT_Util
status	Data Type	UT_Util
sr_maxof	Procedure	UT_Util
Message	Data Type	V-system
Copy	Procedure	V-system

NOTES:

None.

LGRT REMOTE TX MODULE

PURPOSE:

This module sends remote requests to the appropriate VM. It is responsible for hiding the network interface.

This module is closely related to the LGRR_Remote_Rx module.

DATA INTERFACE:

Name	**Description**
None.	

DATA TYPE INTERFACE:

Name	**Description**
None.	

INVOCATION INTERFACE:

Procedure / **Parameters**	**Description** / **Description** (Type,IN/OUT, etc.)
sr_init_remote_Tx	Initialize this module.
sr_remote	Send a request to the remote VM and wait for the reply.
dest	vmid, IN, Remote VM identifier.
type	ms_type, IN, Request type.
ph	pach, IN, Request descriptor.
size	short, IN, Message byte size.
[return]	pach, OUT, Reply message descriptor.

IMPLEMENTATION FILES:

LGRT_Remote_Tx.c
LGRT_Remote_Tx_i.h
LGRT_Remote_Tx_h.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
sr_vm_connect	Procedure	LGVM_Virtual_Machine
sr_net_send	Procedure	OSMT_Message_Tx
ms_type	Data Type	DSNE_Network
pach	Data Type	DSNE_Network
vmid	Data Type	DSVM_Virtual_Machine
sr_vm_known	Procedure	DSVM_Virtual_Machine
DEBUG	Procedure	MCDE_Debug

status	Data Type	UT_Util

NOTES:
None.

LGVM VIRTUAL MACHINE MODULE

PURPOSE:

Implement the Virtual Machine (VM) module. This module supplies the operations to create and destroy virtual machines. Each virtual machine has its own memory space, communication address, and RTS. Once a virtual machine is created, then resource instances may be started on it.

The "Revised Report on the SR Language" has more information about the SR concept of VMs.

DATA INTERFACE:

Name	**Description**
VM_MAGIC	When this parameter value is an argument to an RTS, the RTS knows that it is being started as a VM. I.e. it is not the initial program startup. Refer to LGMN_Main module for more information about the RTS startup.

DATA TYPE INTERFACE:

Name	**Description**
None.	

INVOCATION INTERFACE:

Procedure / **Parameters**	**Description** / **Description** (Type,IN/OUT, etc.)
sr_init_vm	Initialize this module.
rcvr_pid	Pid, IN, The communication address of this VM.
sr_locate	Specify the location of a physical machine. Register the location n on the specified phost with the executable path pexe. However, this location can only be referenced from this VM. Resources on other VMs must execute their own locate statements before using location n.
n	pmid, IN, The location identifier.
phost	char *, IN, The physical host name.
lhost	int, IN, The length of phost string.
pexe	char *, IN, The executable path of the program.
lexe	int, IN, The length of pexe string.

sr_crevm	Create a new virtual machine.	
vm_num		vmid *, IN-OUT, Identifier of the new VM.
pm_num		pmid, IN, Physical machine location of the new VM.
sr_destvm	Destroy a virtual machine.	
vm		vmid, IN, The virtual machine identifier.

IMPLEMENTATION FILES:

LGVM_Virtual_Machine.c
LGVM_Virtual_Machine_i.h
LGVM_Virtual_Machine_h.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
netpath	Procedure	LGNP_Netpath
remote	Procedure	LGRT_Remote_Tx
sr_free	Procedure	OSMM_Memory
sr_own_alloc	Procedure	OSMM_Memory
sr_invokeblockfunc	Procedure	OSPL_Pool
sr_acceptblockfunc	Procedure	OSPL_Pool
sr_termblockfunc	Procedure	OSPL_Pool
sr_freepid	Procedure	OSPL_Pool
P	Procedure	OSS4_Semaphore
va_alist	Data Type	OSVA_Variable_ArgList
va_dcl	Data Type	OSVA_Variable_ArgList
va_list	Data Type	OSVA_Variable_ArgList
va_start	Procedure	OSVA_Variable_ArgList
va_arg	Procedure	OSVA_Variable_ArgList
va_end	Procedure	OSVA_Variable_ArgList
sr_pgmgroup	Data (Read)	DSGP_Group
RTS_OWN	Data (Read)	DSMM_Memory
srx_addr	Data (Read)	DSNE_Network
sr_rcvr_pid	Data (Read)	DSNE_Network
sr_net_exe_path	Data (Read)	DSNE_Network
pach_st	Data Type	DSNE_Network
ms_type	Data Type	DSNE_Network
num_st	Data Type	DSNE_Network
srxreply	Data Type	DSNE_Network
system_errors	Data Type	DSOS_Operating_System
pidnode	Data Type	DSPL_Pool
blockfunc	Data Type	DSPL_Pool
sem	Data Type	DSS4_Semaphore
SRDIR	Data (Read)	DSVM_Virtual_Machine

SRLIB	Data (Read)	DSVM_Virtual_Machine
NOOP_VM	Data (Read)	DSVM_Virtual_Machine
NULL_VM	Data (Read)	DSVM_Virtual_Machine
SRX_VM	Data (Read)	DSVM_Virtual_Machine
MAX_VM	Data (Read)	DSVM_Virtual_Machine
VULTURE_PRIO	Data (Read)	DSVM_Virtual_Machine
VULTURE_STKSIZE	Data (Read)	DSVM_Virtual_Machine
sr_my_machine	Data (Read)	DSVM_Virtual_Machine
sr_my_vm	Data (Read)	DSVM_Virtual_Machine
sr_vmdata	Data (Update)	DSVM_Virtual_Machine
sr_vmpool	Data (Update)	DSVM_Virtual_Machine
pmid	Data Type	DSVM_Virtual_Machine
pmdata	Data Type	DSVM_Virtual_Machine
vmid	Data Type	DSVM_Virtual_Machine
sr_dbg_flags	Data (Read)	MCDE_Debug
DEBUG	Procedure	MCDE_Debug
sr_rtserror	Data (Update)	MCEX_Exception
sr_abort	Procedure	MCEX_Exception
sr_net_abort	Procedure	MCEX_Exception
Pid	Data Type	MCPR_Process
sr_check_sp	Procedure	MCPR_Process
SystemCode	Data Type	V-system
SelectionRec	Data Type	V-system
getenv	Procedure	V-system
getwd	Procedure	V-system
strcpy	Procedure	V-system
Create	Procedure	V-system
Ready	Procedure	V-system
ReceiveSpecific	Procedure	V-system
MapRemoteHost	Procedure	V-system
ExecProgram	Procedure	V-system
QueryWorkstationConfig	Procedure	V-system

NOTES:

None.

RTS MACHINE SUBSYSTEM (MC) DESCRIPTION

Function of the Machine Subsystem

The Machine Subsystem is the lowest level of the RTS. Every other subsystem in the RTS depends on it, either directly or indirectly.

This subsystem is a mixed collection of modules. There are two main reasons for including modules in this subsystem. Some modules are included because they are used by almost every other module in the RTS. Eg. the MCDE_Debug module. Others are included because they hide machine-specific details. Eg. the MCPR_Process module. In general, modules are put in this subsystem because they belong at the bottom of the RTS system dependency diagram.

Machine Subsystem Design

Most of the Machine subsystem design is straightforward. Each of the modules supplies a few procedures to manipulate their simple module.

MCDE DEBUG MODULE

PURPOSE:
Implement debugging support for the RTS modules.

DATA INTERFACE:

Name	**Description**
SRXDEBUG	UNIX Environment variable which can be used to specify the debug statements to be turned on.

DATA TYPE INTERFACE:

Name **Description**

None.

INVOCATION INTERFACE:

Procedure **Description**
Parameters **Description** (Type,IN/OUT, etc.)

mcde_init_debug
Specify which debug statements to be printing.
- s — char *, IN, A string of debug flags. If a flag is on that indicates that the associated group of debug statements is "turned on".

mcde_DEBUG
Print debugging values under format f, if this statement is "on".
- n — char *, IN, Debug group identifier. Only one of the flags in this string should be on.
- f — char *, IN, Format string for printf.
- v1 — int, IN, First debug value to be printed.
- v2 — int, IN, Second debug value to be printed.
- v3 — int, IN, Third debug value to be printed.

IMPLEMENTATION FILES:
MCDE_Debug.c
MCDE_Debug.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
getenv	Procedure	V-system

NOTES:

None.

MCEX EXCEPTION HANDLER MODULE

PURPOSE:

Implement a machine level exception handler for the RTS. This module handles all exceptions, including those which occur when the SR program is running on more than one VM. In this case, a program abort must stop every resource instance on each VM.

DATA INTERFACE:

Name	Description
sr_trace_flag	Indicates if tracing is turned on. Tracing causes some debug-type statements to print information about the current state of the program.
sr_rtserror	Contains a character string which describes the last error that occurred.
sr_my_label	Error label to indicate which VM the error message came from. It contains the vmid.

DATA TYPE INTERFACE:

Name	Description
None.	

INVOCATION INTERFACE:

Procedure **Description**
  Parameters **Description** (Type,IN/OUT, etc.)

mcex_error — Print an RTS error message.
  s — char *, IN, Error message string.

mcex_warn — Print an RTS warning message.
  s — char *, IN, Warning message string.

mcex_abort — Print a fatal error and abort.
  s — char *, IN, Message string.

mcex_net_abort — Print a fatal network communication error and abort.
  s — char *, IN, Message string.

mcex_stk_overflow
  Print a stack overflow message and abort.

mcex_stk_underflow
  Print a stack underflow message and abort.

mcex_stk_corrupted
  Print a corrupted stack message and abort.

mcex_stop	Stop execution of the SR program on all VMs.	
exitcode	int, IN, UNIX-style exit code.	

IMPLEMENTATION FILES:

MCEX_Exception.c
MCEX_Exception.h

IMPORTED ELEMENTS:

Name	Type	Module
sr_pgmgroup	Data (Read)	DSGP_Group
num_st	Data Type	DSNE_Network
ms_type	Data Type	DSNE_Network
system_errors	Data Type	DSOS_Operating_System
sr_exec_up	Data (Read)	DSSX_Srx
Bool	Data Type	UT_Utility
stdout	Data Type	V-system
stderr	Data Type	V-system
Send	Procedure	V-system
ErrorString	Procedure	V-system
fprintf	Procedure	V-system
fflush	Procedure	V-system

NOTES:

None.

MCMM MEMORY MANAGEMENT MODULE

PURPOSE:

Implement memory management for RTS modules. This module is just an interface to the machine memory management, but it is convenient to abstract the interface in order to hide machine differences.

DATA INTERFACE:

Name	**Description**

None.

DATA TYPE INTERFACE:

Name	**Description**

None.

INVOCATION INTERFACE:

Procedure	**Parameters**	**Description** (Type, IN/OUT, etc.)
mcmm_alloc		Allocate a chunk of contiguous memory.
	size	int, IN, Byte size of memory chunk desired.
	[return]	daddr, OUT, Pointer to allocated memory chunk.
mcmm_free		Free a chunk of contiguous memory.
	addr	daddr, IN, Pointer to allocated memory chunk.

IMPLEMENTATION FILES:

MCMM_Memory.h
MCMM_Memory.c

IMPORTED ELEMENTS:

Name	**Type**	**Module**
daddr	Data Type	UT_Utility
malloc	Procedure	V-system
mfree	Procedure	V-system

NOTES:

None.

MCPR PROCESS MODULE

PURPOSE:

Implement the process module at the machine level. This includes creating a process context, changing contexts, and context error checking. These operations can only be done at the machine level because they manipulate machine registers and the process stack.

DATA INTERFACE:

Name	**Description**
None.	

DATA TYPE INTERFACE:

Name	**Description**
paddr	A procedure address.

INVOCATION INTERFACE:

Procedure **Description**
  Parameters **Description** (Type, IN/OUT, etc.)

mcpr_build_context
Create a process context.

Parameter	Description
pc	paddr, IN, Process's initial program counter.
stack	daddr, IN, Pointer to the stack area.
stack_size	int, IN, Byte size of the stack.
arg1	int, IN, Process's first argument.
arg2	int, IN, Process's second argument.
arg3	int, IN, Process's third argument.
arg4	int, IN, Process's fourth argument.

mcpr_chg_context
Change to a new process context from the current process context.

Parameter	Description
stack	daddr, IN, Pointer to the new process's stack.

mcpr_check_stk
Check that the stack has not been corrupted.

IMPLEMENTATION FILES:

MCPR_Process.c
MCPR_Process.h
MCPR_M68k.s - Motorola 68000 Assembler code.

IMPORTED ELEMENTS:

Name	**Type**	**Module**
mc_stk_overflow	Procedure	MCEX_Exception
mc_stk_underflow	Procedure	MCEX_Exception
mc_stk_corrupted	Procedure	MCEX_Exception
daddr	Data Type	UT_Utility

NOTES:

None.

RTS OPERATING SYSTEM SUBSYSTEM (OS) DESCRIPTION

Function

The Operating System (OS) Subsystem provides the functionality that is normally associated with an Operating System. For example, it supplies Message passing, Memory Management, a Network interface, and SR Process Scheduling.

Design

The Operating System (OS) Subsystem is quite complex. There are over a dozen modules and many of these modules depend on ten or more other modules. To complicate the design further, this subsystem seems to have a tendency to develop circular dependencies. Fortunately, we have managed to break most of the circular dependencies. However, there is one circular dependency left.

The circular dependency that is left is 'caused' by the OSNE_Network module's dependency on several LG_Language modules. This particular dependency seems to be unavoidable. The OSNE_Network module has more information on this dependency.

The other modules are fairly simple when regarded in isolation.

There are several different types of Free Lists to manage the lists of descriptors. There are the OS-type modules like the Message modules, the OSSH_Scheduler module, the OSNE_Network module, and OSS4_Semaphore module. There are also several modules which are peculiar to SR or the V-system implementation. The OSSX_Srx module is peculiar to SR. It ensures that each VM number is unique. The OSPL_Pool module is peculiar to the V-system implementation. It supplies a pool of V-system processes to perform V-system blocking operations. Although the connections between these modules are complex, each module is straightforward.

OSGP GROUP MODULE

PURPOSE:

Implement the messages to process groups. There is a very close dependency on the VM data structures because the VM modules are the only modules that use process groups.

DATA INTERFACE:

Name	**Description**
sr_pgmgroup	the process GROUP identifier for this ProGram.

DATA TYPE INTERFACE:

Name **Description**

None.

INVOCATION INTERFACE:

Procedure **Description**
  Parameters **Description** (Type, IN/OUT, etc.)

sr_vm_connect — Connect to another VM by determining its communication address. This procedure is always successful. If there is any problem, then the program is aborted.
- vm — vmid, IN, VM to connect to.

sr_reply_findvm — Reply to a VM which is attempting to connect to another VM. This procedure is always successful. If there is any problem, then the program is aborted.
- client — sender, IN-OUT, The client VM's message descriptor.

sr_join_pgmgroup — Add the current VM to the SR program's process group. This procedure is always successful. If there is any problem, then the program is aborted.
- new_rcvr — Pid, IN, This VM's communication address.

IMPLEMENTATION FILES:

OSGP_Group.c
OSGP_Group_i.h
OSGP_Group_h.h

IMPORTED ELEMENTS:

Name	Type	Module
sr_group_send	Procedure	OSMT_Message_Tx
sr_my_vm	Data (Read)	DSVM_Virtual_Machine
sr_vmdata	Data (Update)	DSVM_Virtual_Machine
pach	Data Type	DSNE_Network
ms_type	Data Type	DSNE_Network
num_st	Data Type	DSNE_Network
findvm_reply	Data Type	DSNE_Network
system_errors	Data Type	DSOS_Operating_System
MCDE_DEBUG	procedure	MCDE_Debug
mcex_net_abort	Procedure	MCEX_Exception
sr_rtserror	Data (Update)	DSEX_Exception
Pid	Data Type	DSPR_Process
mcex_abort	Procedure	MCEX_Exception
SystemCode	Data Type	V-system
CreateGroup	Procedure	V-system
JoinGroup	Procedure	V-system

NOTES:

None.

OSIF INFINITE FREE LIST MODULE

PURPOSE:

Implement an infinite, unsafe free list of nodes.

A free list is a list of nodes that are currently unused. This module supplies the operations to create the list, get a node (from the free list), and free a node (return it to the free list).

It is an infinite list because if it ever runs out of nodes on the free list, it will allocate more nodes to make sure that the free list is never 'empty'.

It is an unsafe list because there is no mutual exclusion. The operations implemented by this module do not guarantee that only one process is modifying the list at any one time. It is up to the invoking module to guarantee mutual exclusion.

DATA INTERFACE:

Name **Description**

None.

DATA TYPE INTERFACE:

Name **Description**

None.

INVOCATION INTERFACE:

Procedure **Description**

Parameters **Description** (Type, IN/OUT, etc.)

osif_declare_free_list
Declare the data structures needed for a free list.

FreeList	C field name, IN-OUT, Free list name.
NodePtr	C type, IN, Pointer type of the list nodes.

osif_is_empty_list
Determine if FreeList is an empty list.

FreeList	C field name, IN, Free list name.
[return]	Bool, OUT, TRUE if List is empty. FALSE otherwise.

osif_init_free_list
Create a new FreeList and add TotalNodes number of nodes to the list.

FreeList	C field name, IN, Free list name.

NodePtr	C type, IN, Pointer type of the list nodes.
NodeStruct	C type, IN, Structure type of the list nodes.
TotalNodes	int, IN, Number of nodes in the new list.

osif_get_node — Get a node from the FreeList and return it to the caller. If there are no nodes available and we can not allocate more memory, the program is aborted.

FreeList	C field name, IN, Free list name.
ErrorMsg	char *, IN, Error message to be displayed if there are no nodes available.
Node	glll_node, OUT, The 'new' node.

osif_free_node — Return a node to the FreeList.

FreeList	C field name, IN, Free list name.
Node	glll_node, IN, The new node.

IMPLEMENTATION FILES:

OSIF_Infinite_Freelist.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
Bool	Data Type	UT_Util
C field name	Data Type	UT_Util
C type	Data Type	UT_Util
glll_list	Data Type	GLLL_Linked_List
glll_node	Data Type	GLLL_Linked_List
osuf_declare_free_list	Procedure	OSUF_Unsafe_FreeList
osuf_init_free_list	Procedure	OSUF_Unsafe_FreeList
osuf_is_empty_list	Procedure	OSUF_Unsafe_FreeList
osuf_push	Procedure	OSUF_Unsafe_FreeList
osuf_pop	Procedure	OSUF_Unsafe_FreeList
mcmm_alloc	Procedure	MCMM_Memory

NOTES:

This module does not depend on the existence of the 'next' field in the node record, as the OSUF_Unsafe_FreeList module does.

OSMM MEMORY MODULE

PURPOSE:

Implement a memory management for the RTS. This module implements RTS allocation and implicit SR program allocation. Explicit SR program allocation is handled by the LGMS_Miscellaneous module.

DATA INTERFACE:

Name	Description

None.

DATA TYPE INTERFACE:

Name	Description

None.

INVOCATION INTERFACE:

Procedure / Parameters	Description	Description (Type,IN/OUT, etc.)
sr_init_mem	Initialize Memory Managment module.	
sr_gen_alloc	Allocate memory. Called from Generated Code (GC).	
size		int, IN, Byte size of memory block.
owner		rint, IN, Resource owner of memory.
[return]		daddr, OUT, Memory block pointer.
sr_gen_free	Free memory. Called from Generated Code (GC).	
addr		daddr, IN, Memory block pointer.
sr_talloc	Allocate memory. Called from Generated Code (GC). Add memory descriptor to the given list.	
size		int, IN, Byte size of memory block.
MemList		memhdr, IN, Memory List.
[return]		daddr, OUT, Memory block pointer.
sr_own_alloc	Allocate memory.	
size		int, IN, Byte size of memory block.
owner		rint, IN, Resource owner of memory.
[return]		daddr, OUT, Memory block pointer.
sr_free	Free memory.	
addr		daddr, IN, Memory block pointer.
sr_res_free	Free all memory belonging to the specified resource.	
owner		rint, IN, Resource owner of memory.

IMPLEMENTATION FILES:

OSMM_Memory.c
OSMM_Memory_i.h
OSMM_Memory_h.h

IMPORTED ELEMENTS:

Name	Type	Module
sr_make_sem	Procedure	OSS4_Semaphore
P	Procedure	OSS4_Semaphore
V	Procedure	OSS4_Semaphore
sem	Data Type	DSS4_Semaphore
memh	Data Type	DSMM_Memory
memhdr	Data Type	DSMM_Memory
dsmm_create_empty_mem_list	Procedure	DSMM_Memory
dsmm_push_mem	Procedure	DSMM_Memory
sr_cur_res	Data(Read)	DSRE_Resource
rint	Data Type	DSRE_Resource
rint_memory	Procedure	DSRE_Resource
daddr	Data Type	UT_Util
mcpr_check_stk	Procedure	MCPR_Process
mcmm_alloc	Procedure	MCMM_Memory
mcmm_free	Procedure	MCMM_Memory
mcex_abort	Procedure	MCEX_Exception
mcde_DEBUG	Procedure	MCDE_Debug

NOTES:

None.

OSMR MESSAGE RECEIVE MODULE

PURPOSE:

Implement the message receive operations with the appropriate V-system operations: Receive, and Reply.

DATA INTERFACE:

Name	Description
None.	

DATA TYPE INTERFACE:

Name	Description
sender	Pointer to a sender descriptor. The sender descriptor is returned by the sr_net_recv procedure. It contains information about the message and the SENDER process.

INVOCATION INTERFACE:

Procedure	Parameters	Description	Description (Type, IN/OUT, etc.)
sr_msg_rx_start		Initialize this module.	
	max_clients		int, IN, Initial number of in-coming messages from client VMs, this module will service at any one time. If more messages arrive then more memory will be allocated as the messages arrive.
sr_net_recv		Receive a message. Suspend the VM until a message is received.	
	client		sender, IN, Blank message descriptor.
	[return]		sender, OUT, In-coming message descriptor.
sr_net_reply		Send a message in reply to a message received through sr_net_recv.	
	client		sender, IN, Out-going message descriptor.
	[return]		SystemCode, OUT, Status of reply operation.
sr_free_sender		Free up the resources associated with a message descriptor.	
	client		sender, IN, Message descriptor.

IMPLEMENTATION FILES:

OSMS_Message_Rx.c
OSMS_Message_Rx_i.h
OSMS_Message_Rx_h.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
osif_declare_free_list	Procedure	OSIF_Infinite_FreeList
osif_create_free_list	Procedure	OSIF_Infinite_FreeList
osif_get_node	Procedure	OSIF_Infinite_FreeList
osif_free_node	Procedure	OSIF_Infinite_FreeList
sr_cur_res	Data (Read)	DSRE_Resource
sr_cur_proc	Data (Read)	DSPR_Process
Pid	Data Type	DSPR_Process
pach	Data Type	DSNE_Network
system_errors	Data Type	DSOS_Operating_System
daddr	Data Type	UT_Util
Bool	Data Type	UT_Util
MCDE_DEBUG	Procedure	MCDE_Debug
mcex_net_abort	Procedure	MCEX_Exception
SEGMENT_PRESENT	Data	V-system
REPLY_RETURN_CODE	Data	V-system
SYS_REPLY_CODE	Data	V-system
REPLY_SEGMENT_BIT	Data	V-system
MsgStruct	Data Type	V-system
Receive	Procedure	V-system
Reply	Procedure	V-system
MoveTo	Procedure	V-system

NOTES:

This module is related to the OSMT_Message_Tx module.

OSMT MESSAGE TRANSMIT MODULE

PURPOSE:

Implement the message transmit operations with the V-system Send operation.

DATA INTERFACE:

Name	Description

None.

DATA TYPE INTERFACE:

Name	Description

None.

INVOCATION INTERFACE:

Procedure / Parameters	Description / Description (Type, IN/OUT, etc.)
sr_net_tx_start	Initialize this module.
max_requests	int, IN, Maximum number of out-going messages (request messages) this module will have outstanding at any one time.
sr_net_send	Send a message to another VM.
dest	vmid, IN, Destination VM.
type	ms_type, IN, Type of message.
packetH	pach, IN-OUT, Message packet header.
size	unsigned, IN, Byte size of the message.
[return]	SystemCode, OUT, Status of send operation.
sr_group_send	Send a message to a group of V-system processes.
dest	Pid, IN, Process group identifier.
type	ms_type, IN, Type of message.
packetH	pach, IN-OUT, Message packet header.
size	unsigned, IN, Byte size of the message.
[return]	SystemCode, OUT, Status of send operation.

IMPLEMENTATION FILES:

OSMS_Message_Tx.c
OSMS_Message_Tx_i.h
OSMS_Message_Tx_h.h

IMPORTED ELEMENTS:

Name	Type	Module
ossf_declare_free_list	Procedure	OSSF_Safe_FreeList
ossf_create_free_list	Procedure	OSSF_Safe_FreeList
ossf_get_node	Procedure	OSSF_Safe_FreeList
ossf_free_node	Procedure	OSSF_Safe_FreeList
InvokeMsg_st	Data Type	OSPL_Pool
blockfunc	Data Type	OSPL_Pool
sr_createprocpool	Procedure	OSPL_Pool
sr_invokeblockfunc	Procedure	OSPL_Pool
sr_acceptblockfunc	Procedure	OSPL_Pool
sr_termblockfunc	Procedure	OSPL_Pool
va_list	Data Type	OSVA_Variable_ArgList
va_arg	Procedure	OSVA_Variable_ArgList
pach	Data Type	DSNE_Network
ms_type	Data Type	DSNE_Network
sr_cur_proc	data (Read)	DSPR_Process
Pid	Data Type	DSPR_Process
sr_cur_res	data (Read)	DSRE_Resource
sr_vmdata	Data	DSVM_Virtual_Machine
MCDE_DEBUG	Procedure	MCDE_Debug
mcex_net_abort	Procedure	MCEX_Exception
SEGMENT_PRESENT	Data	V-system
MORE_REPLIES	Data	V-system
MsgStruct	Data Type	V-system
Send	Procedure	V-system

NOTES:

This module is related to OSMR_Message_Rx.

OSNE NETWORK MODULE

PURPOSE:

Implement a network interface. This module is responsible for receiving all messages from the network and calling the appropriate module to perform the requested operations.

DATA INTERFACE:

Name **Description**

None.

DATA TYPE INTERFACE:

Name **Description**

None.

INVOCATION INTERFACE:

Procedure **Description**
Parameters **Description** (Type,IN/OUT, etc.)

sr_init_net Initialize the network interface.
srx_addr Pid, IN, Address of SRX process.

sr_net_interface
Read all the outstanding messages from the network.

IMPLEMENTATION FILES:

OSNE_Network.c
OSNE_Network_i.h
OSNE_Network_h.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
sr_activate	Procedure	LGPR_Process
sr_spawn	Procedure	LGPR_Process
sr_rmt_create	Procedure	LGRM_Remote
sr_rmt_destroy	Procedure	LGRM_Remote
sr_rmt_destvm	Procedure	LGRM_Remote
sr_rmt_invk	Procedure	LGRM_Remote
sr_init_vm	Procedure	LGVM_Virtual_Machine
sr_reply_findvm	Procedure	LGVM_Virtual_Machine
sr_rtserror	Data(Update)	OSEX_Exception
sr_my_label	Data(Read)	OSEX_Exception
sr_stop	Procedure	OSEX_Exception
sr_pgmgroup	Data(Read)	OSGP_Group
sr_join_pgmgroup	Procedure	OSGP_Group

sender	Data Type	OSMS_Message
sr_freesender	Procedure	OSMS_Message
sr_net_start	Procedure	OSMS_Message
sr_net_recv	Procedure	OSMS_Message
sr_net_reply	Procedure	OSMS_Message
main	Procedure	OSSX_Srx
sr_max_rmt_reqs	Data(Read)	DSLM_Limits
ms_type	Data Type	DSNE_Network
num_st	Data Type	DSNE_Network
sr_exec_up	Data(Update)	DSNE_Network
stdout	Data(Read)	DSIO_IO
stdin	Data(Read)	DSIO_IO
SRXPATH	Data(Read)	DSSX_Srx
VM_MAGIC	Data(Read)	DSVM_Virtual_Machine
PROTO_VER	Data(Read)	DSVM_Virtual_Machine
sr_my_vm	Data(Read)	DSVM_Virtual_Machine
MCDE_DEBUG	procedure	MCDE_Debug
mcex_abort	procedure	MCEX_Exception
mcex_warn	procedure	MCEX_Exception
Pid	Data Type	MCPR_Process
SystemCode	Data Type	V-system
getenv	Procedure	V-system
ExecProgram	Procedure	V-system

NOTES:

This module is a 'design problem'. It is called from the OSSH_Scheduler module, which is in the middle of the OS Dependency Diagram, but it calls several of the LG_Language modules, which depend on the OS subsystem. Unfortunately, there does not seem to be any way to avoid this circular dependency.

This circular dependency is unavoidable because OSNE must be called from OSSH_Scheduler and it must call the LG modules. Before we go any further, we will explain why the OSSH_Scheduler must call OSNE and why OSNE must call the LG modules.

The OSSH_Scheduler module is responsible for scheduling tasks. Since the OSNE module must periodically check for messages on the network, OSSH_Scheduler is responsible for scheduling OSNE periodically. Therefore, OSSH_Scheduler must call OSNE_Network.

The OSNE module must call the LG_Language modules because OSNE is responsible for ensuring the operations requested by the in-coming messages are executed. Unfortunately, all these operations are implemented in the LG_Language subsystem. Therefore, OSNE must call the LG_Language modules.

Fortunately, the circular dependency is not as serious as it appears. OSNE spawns SR processes to perform most of the message operations. Therefore, very little of the LG_Language code is actually executed when OSNE calls the LG_Language modules. Furthermore, the code that is executed never calls OSNE either directly or indirectly. Therefore, we do not have to worry about infinite recursion.

However, this dependency does make testing more difficult. OSNE can not be completely tested until the LG_Language subsystem is working, but it must be working in order to test the OS_Operating_System subsystem. We suggest that a special test program with stubbed procedures be set up to test the OSNE_Network module by itself. Then it can be used with confidence in the OS_Operating_System tests.

OSPL POOL MODULE

PURPOSE:

Implement a process pool module. This module is implemented to accommodate the V-system blocking operations. In the V-system, if you want to execute a blocking operation without blocking the current process, then you must put the code for the blocking operation in another process, called a helper process, and send a message to the helper process. The message contains the blocking operation code and any parameters required for the operation.

In the V-system implementation of SR, we follow this V-system model of one main process, and many helper processes. However, the main process is also receiving messages from other VMs as well as the helper processes. Plus, there are different types of helper processes. There are helper processes to perform IO operations, processes for Message operations, and processes for VM operations.

This module simplifies the implementation by containing all the code to create a V-system process pool, report process pool errors, and synchronize with the other in-coming messages.

This module supplies the operations to communicate with process pools, and the operations used to implement the process pools.

DATA INTERFACE:

Name	**Description**
None.	

DATA TYPE INTERFACE:

Name	**Description**
pool	Pointer to a process POOL descriptor.
InvokeMsg	INVOKE MeSsaGe to a pool process.
blockfunc	BLOCKing operation codes. Values are: REMOTE_SEND CREATE_Virtual_Machine FILE_FLUSH FILE_READ FILE_OPEN FILE_CLOSE FILE_SEEK FILE_UNLINK

INVOCATION INTERFACE:

Procedure	**Description**
Parameters	**Description** (Type, IN/OUT, etc.)
sr_init_pool	Initialize the Process Pool module.
sr_createprocpool	Create a Process Pool.
NumProcess	unsigned, IN, Number of processes to be in the pool.
func	paddr, IN, Procedure to execute in the process.
priority	short, IN, Process priority.
StkSize	unsigned, IN, Byte size of process stack.
[return]	pool, OUT, The new pool descriptor.
sr_invokeblockfunc	Invoke a blocking function implemented in a process pool.
poolptr	pool, IN, The process pool.
func_num	blockfunc, IN, The blocking function to be executed.
argList	va_list, IN, Pointer to an argument list.

Pool Process Implementation Operations

sr_acceptblockfunc	Accept a blocking function invocation.
message	InvokeMsg, IN-OUT, The invocation msg.
func_num	blockfunc, OUT, The operation code.
param_ptr	va_list, OUT, The argument list.
sr_termblockfunc	Terminate the blocking function invocation.
message	InvokeMsg, IN-OUT, The invocation msg.

IMPLEMENTATION FILES:

OSPL_Pool.c
OSPL_Pool_i.h
OSPL_Pool_h.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
sr_alloc	Procedure	OSMM_Memory
ossf_declare_free_list	Procedure	OSSF_Safe_FreeList
ossf_create_free_list	Procedure	OSSF_Safe_FreeList

ossf_get_node	Procedure	OSSF_Safe_FreeList
ossf_free_node	Procedure	OSSF_Safe_FreeList
sr_make_sem	Procedure	OSS4_Semaphore
P	Procedure	OSS4_Semaphore
V	Procedure	OSS4_Semaphore
va_list	Data Type	OSVA_Variable_ArgList
va_dcl	Data Type	OSVA_Variable_ArgList
va_start	Data Type	OSVA_Variable_ArgList
va_end	Data Type	OSVA_Variable_ArgList
sr_rtserror	Data (Update)	DSEX_Exception
pach_st	Data Type	DSNE_Network
sr_cur_proc	Data (Read)	DSPR_Process
Pid	Data Type	DSPR_Process
sr_cur_res	Data (Read)	DSRE_Resource
dss4_sem_count	Procedure	DSS4_Semaphore
sem	Data Type	DSS4_Semaphore
paddr	Data Type	MCPR_Process
Message	Data Type	V-system
SystemCode	Data Type	V-system
Create	Procedure	V-system
Ready	Procedure	V-system
ReceiveSpec	Procedure	V-system
Reply	Procedure	V-system
Send	Procedure	V-system
GetTeamRoot	Procedure	V-system

NOTES:

None.

OSS4 SEMAPHORE MODULE

PURPOSE:

Implement a semaphore module with the standard operations. Semaphores are used to control process synchronization. Any operating systems text will have an explanation of semaphores.

DATA INTERFACE:

Name	**Description**
None.	

DATA TYPE INTERFACE:

Name	**Description**
sem	Pointer to a semaphore descriptor.

INVOCATION INTERFACE:

Procedure	**Description**	
Parameters		**Description** (Type, IN/OUT, etc.)
sr_init_sem	Initialize the semaphore module.	
sr_make_sem	Return a new, initialized, semaphore descriptor.	
sr_init_val		int, IN, Initial value of semaphore counter.
[return]		sem, OUT, New semaphore descriptor.
sr_kill_sem	Destroy the semaphore.	
sp		sem, IN, Pointer to semaphore descriptor.
V	Increment semaphore counter or unblock a waiting process.	
sp		sem, IN, Pointer to semaphore record.
P	Decrement semaphore counter or block the calling process.	
sp		sem, IN, Pointer to semaphore record.
sr_query_sem	Return the value of the semaphore counter. This is used by GC (Generated Code) to determine the number of pending invocations on a semaphore op.	
sp		sem, IN, Pointer to semaphore record.
[return]		int, OUT, The semaphore counter value.

IMPLEMENTATION FILES:

OSS4_semaphore.c
OSS4_semaphore_i.h
OSS4_semaphore_h.h

IMPORTED ELEMENTS:

Name	Type	Module
awaken	Procedure	OSSH_Scheduler
block	Procedure	OSSH_Scheduler
sr_cswitch	Procedure	OSSH_Scheduler
osuf_declare_free_list	Procedure	OSUF_Unsafe_FreeList
osuf_is_empty_list	Procedure	OSUF_Unsafe_FreeList
osuf_init_free_list	Procedure	OSUF_Unsafe_FreeList
osuf_get_node	Procedure	OSUF_Unsafe_FreeList
osuf_free_node	Procedure	OSUF_Unsafe_FreeList
sr_cur_proc	Data (Read)	DSPR_Process
sr_cur_res	Data (Read)	DSRE_Resource
MCDE_DEBUG	Procedure	MCDE_Debug
mcex_abort	Procedure	MCEX_Exception
mcex_warn	Procedure	MCEX_Exception
sr_check_stk	Procedure	MCPR_Process

NOTES:

None.

OSSF SAFE FREE LIST MODULE

PURPOSE:

Implement a safe free list of nodes. It is a safe list because each operation on a free list is protected by mutual exclusion. The operations implemented by this module guarantee that only one process is modifying the list at any one time.

A free list is a list of nodes that are currently unused. This module supplies the operations to create the list, get a node (from the free list), and free a node (return it to the free list).

DATA INTERFACE:

Name	**Description**

None.

DATA TYPE INTERFACE:

Name	**Description**

None.

INVOCATION INTERFACE:

Procedure **Description**
Parameters **Description** (Type,IN/OUT, etc.)

ossf_declare_free_list
Declare the data structures needed for a free list.

FreeList	C field name, IN-OUT, Free list name.
NodePtr	C type, IN, Pointer type of the list nodes.

ossf_is_empty_list
Determine if FreeList is an empty list.

FreeList	C field name, IN, Free list name.
[return]	Bool, OUT, TRUE if List is empty. FALSE otherwise.

ossf_init_free_list
Create a new FreeList and add TotalNodes number of nodes to the list.

FreeList	C field name, IN, Free list name.
NodePtr	C type, IN, Pointer type of the list nodes.
NodeStruct	C type, IN, Structure type of the list nodes.
TotalNodes	int, IN, Number of nodes in the new list.

ossf_get_node	Get a node from the FreeList and return it to the caller. If there are no nodes available, the program is aborted.	
FreeList		C field name, IN, Free list name.
ErrorMsg		char *, IN, Error message to be displayed if there are no nodes available.
Node		glll_node, OUT, The 'new' node.
ossf_free_node	Return a node to the FreeList.	
FreeList		C field name, IN, Free list name.
Node		glll_node, IN, The new node.

IMPLEMENTATION FILES:

OSSF_Safe_Freelist.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
oss4_make_sem	Procedure	OSS4_Semaphore
P	Procedure	OSS4_Semaphore
V	Procedure	OSS4_Semaphore
osuf_declare_free_list	Procedure	OSUF_Unsafe_FreeList
osuf_is_empty_list	Procedure	OSUF_Unsafe_FreeList
osuf_init_free_list	Procedure	OSUF_Unsafe_FreeList
osuf_get_node	Procedure	OSUF_Unsafe_FreeList
osuf_free_node	Procedure	OSUF_Unsafe_FreeList
glll_list	Data Type	GLLL_Linked_List
glll_node	Data Type	GLLL_Linked_List
Bool	Data Type	UT_Util
C field name	Data Type	UT_Util
C type	Data Type	UT_Util

NOTES:

None.

OSSH SCHEDULER MODULE

PURPOSE:

Implement the OS-level Scheduler module. This module controls the processor. It assigns the processor to the ready process which has been waiting the longest.

DATA INTERFACE:

Name	Description
sr_ready_list	LIST of processes that are READY to run.
sr_max_c_switch_per_msg	MAXimum number of Context SWITCHes between attempts to read MeSsaGes from the network.
sr_cur_proc	CURrent PROCess that is running.
sr_num_blocked	NUMber of BLOCKED processes. They may be blocked waiting for a semaphore, an io operation, etc.

DATA TYPE INTERFACE:

Name	Description
None.	

INVOCATION INTERFACE:

Procedure / Parameters	Description / Description (Type,IN/OUT, etc.)
sr_cswitch	Process context switch. Execute the next process which is ready to run.
block	Block the current process and place it on the process queue.
procQ	proc_queue, IN-OUT, The process queue.
awaken	Awaken the next process on the process queue.
procQ	proc_queue, IN-OUT, The process queue.
sr_enqueue	Add a process to the given queue.
procQ	proc_queue, IN-OUT, The process queue.
procDesc	proc, IN-OUT, Process added to procQ.
sr_dequeue	Remove a process from the given queue.
procQ	proc_queue, IN-OUT, The process queue containing procDesc.
procDesc	proc, IN, The process descriptor.

IMPLEMENTATION FILES:

OSSH_Scheduler.c
OSSH_Scheduler_i.h
OSSH_Scheduler_h.h

IMPORTED ELEMENTS:

Name	Type	Module
sr_net_interface	procedure	OSNE_Network
sr_stop	procedure	OSEX_Exception
dscl_class_count	procedure	DSCL_Class
dsco_co_count	procedure	DSCO_Concurrent
dsop_oper_count	procedure	DSOP_Operation
sr_cur_proc	data (Update)	DSPR_Process
sr_ready_queue	data (Update)	DSPR_Process
proc	data type	DSPR_Process
proc_queue	data type	DSPR_Process
dspr_append_procQ	procedure	DSPR_Process
dspr_delete_procQ	procedure	DSPR_Process
dspr_free_proc	procedure	DSPR_Process
dsrm_rem_count	procedure	DSRM_Remote
dsre_rint_count	procedure	DSRE_Resource
sr_cur_res	data (Update)	DSRE_Resource
sr_exec_up	data (Read)	DSSX_Srx
sr_chg_context	procedure	MCPR_Process
sr_rtserror	Data	MCEX_Exception
rts_warn	procedure	MCEX_Exception
MCDE_DEBUG	procedure	MCDE_Debug

NOTES:

None.

OSSX SRX MODULE

PURPOSE:

Supply a unique VM number for each new VM.

Currently this module is implemented as a separate V-system process. This implementation affects the interface. This module is initialized by starting the process rather than by calling a procedure, and operations are 'called' by sending messages to the process. Therefore, some of the 'procedures' listed in the Invocation Interface have the word 'Message' appended to indicate they are really messages, not procedures.

DATA INTERFACE:

Name	Description
SRXPATH	filename PATH for the SRX executable file.

DATA TYPE INTERFACE:

Name	Description
None.	

INVOCATION INTERFACE:

Procedure	Parameters	Description	Description (Type,IN/OUT, etc.)
main		Initialize this module.	
	vm_magic		char *, IN, This string should match the VM_MAGIC constant. If it does, we can be fairly certain that this process has been correctly started by an SR program.
	version		char *, IN, This string should match the PROTO_VER constant. If it does, we can be certain that this code is the same version as the SR program code.
	programGroup		int, IN, The program group number identifies the communication group that this SR program belongs to. By belonging to this group, we will ensure that this process receives all the broadcast messages.
REQ_Virtual_MachineNUM Message		Return a unique VM identifier.	
	[return]		vmid, OUT, A unique VM identifier.
MSG_EXIT Message		Program has terminated. Time to exit.	

IMPLEMENTATION FILES:

OSSX_Srx.c
OSSX_Srx_i.h
OSSX_Srx_h.h

IMPORTED ELEMENTS:

Name	Type	Module
sr_pgmgroup	Data (Update)	OSGP_Group
sr_join_pgmgroup	Procedure	OSGP_Group
sender	Data Type	OSMS_Message_Rx
sr_net_start	Procedure	OSMS_Message_Rx
sr_net_recv	Procedure	OSMS_Message_Rx
sr_net_reply	Procedure	OSMS_Message_Rx
srxreply	Data Type	DSNE_Network
ms_type	Data Type	DSNE_Network
MAX_Virtual_Machine	Data (Read)	DSVM_Virtual_Machine
VM_MAGIC	Data (Read)	DSVM_Virtual_Machine
PROTO_VER	Data (Read)	DSVM_Virtual_Machine
init_debug	Procedure	MCDE_Debug
MCDE_DEBUG	Procedure	MCDE_Debug
Bool	Data Type	UT_Util
SystemCode	Data Type	V-system

NOTES:

None.

OSUF UNSAFE FREE LIST MODULE

PURPOSE:

Implement an unsafe free list of nodes. It is an unsafe list because there is no mutual exclusion. The operations implemented by this module do not guarantee that only one process is modifying the list at any one time. It is up to the invoking module to guarantee mutual exclusion.

A free list is a list of nodes that are currently unused. This module supplies the operations to create the list, get a node (from the free list), and free a node (return it to the free list).

DATA INTERFACE:

Name **Description**

None.

DATA TYPE INTERFACE:

Name **Description**

None.

INVOCATION INTERFACE:

Procedure **Description**

Parameters **Description** (Type, IN/OUT, etc.)

osuf_declare_free_list

Declare the data structures needed for a free list.

FreeList	C field name, IN-OUT, Free list name.
NodePtr	C type, IN, Pointer type of the list nodes.

osuf_is_empty_list

Determine if FreeList is an empty list.

FreeList	C field name, IN, Free list name.
[return]	Bool, OUT, TRUE if List is empty. FALSE otherwise.

osuf_init_free_list

Create a new FreeList and add TotalNodes number of nodes to the list.

FreeList	C field name, IN, Free list name.
NodePtr	C type, IN, Pointer type of the list nodes.
NodeStruct	C type, IN, Structure type of the list nodes.
TotalNodes	int, IN, Number of nodes in the new list.

osuf_get_node — Get a node from the FreeList and return it to the caller. If there are no nodes available, the program is aborted.

FreeList	C field name, IN, Free list name.
ErrorMsg	char *, IN, Error message to be displayed if there are no nodes available.
Node	glll_node, OUT, The 'new' node.

osuf_free_node — Return a node to the FreeList.

FreeList	C field name, IN, Free list name.
Node	glll_node, IN, The new node.

IMPLEMENTATION FILES:

OSUF_Unsafe_Freelist.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**
Bool	Data Type	UT_Util
C field name	Data Type	UT_Util
C type	Data Type	UT_Util
glll_list	Data Type	GLLL_Linked_List
glll_node	Data Type	GLLL_Linked_List
glll_create_empty_list	Procedure	GLLL_Linked_List
glll_is_empty_list	Procedure	GLLL_Linked_List
glll_push	Procedure	GLLL_Linked_List
glll_pop	Procedure	GLLL_Linked_List
mcmm_alloc	Procedure	MCMM_Memory

NOTES:

This module assumes that the name of the NextField pointer in the node structure is always 'next'. This simplifies the interface and it happens to be true for the current version of the SR RTS.

Currently (Feb/91), this module is only used by the OSS4_Semaphore and OSPR_Process. Therefore, only the Semaphore and Process data structures have to use the 'next' fieldname.

OSVA VARIABLE ARGUMENT LIST MODULE

PURPOSE:

Implement a variable argument list for C functions. This allows calling functions to invoke a function with any number of arguments.

DATA INTERFACE:

Name	**Description**
va_alist	The variable name of the argument list. The last argument in the C function header must have this name.

DATA TYPE INTERFACE:

Name	**Description**
va_dcl	Declare the va_alist variable.
va_list	Pointer to a variable argument. This is used to declare the current argument pointer.

INVOCATION INTERFACE:

Procedure / **Parameters**	**Description** / **Description** (Type,IN/OUT, etc.)
va_start	Initialize the current argument pointer.
list	va_list, IN-OUT, Current argument pointer.
va_arg	Remove the current argument from the argument list.
list	va_list, IN-OUT, Current argument pointer.
mode	C type, IN, The type of the current argument.
va_end	Release all resources in use.
list	va_list, IN-OUT, Current argument pointer.

IMPLEMENTATION FILES:

OSVA_Variable_ArgList.c
OSVA_Variable_ArgList_i.h

IMPORTED ELEMENTS:

None.

NOTES:

Refer to the LGMI_Miscellaneous, sr_max function code for an example of the use of this module.

SRSYS MODULE

PURPOSE:

Gather together a group of types which are used by SR generated code.

DATA INTERFACE:

Name	Description
None.	

DATA TYPE INTERFACE:

Name	Description
sem	Pointer to semaphore data record.

INVOCATION INTERFACE:

Procedure **Description**
Parameters **Description** (Type, IN/OUT, etc.)

None.

IMPLEMENTATION FILES:

srsys.h

IMPORTED ELEMENTS:

Name	Type	Module
None.		

NOTES:

None.

UT UTILITY MODULE

PURPOSE:

Implement utility procedures and utility data types.

DATA INTERFACE:

Name	Description
NULL_SEQN	Sequence number of null resource or operation capability.
NOOP_SEQN	Sequence number of noop resource or operation capability.

"Descriptor fields"

AD_MAXL	String maximum length.
AD_ADDR	Address.
AD_SIZE	Size.

DATA TYPE INTERFACE:

Name	Description
Bool	Boolean type. Values are: TRUE, FALSE.
status	Exit status code for SR primitive functions such as create and invoke.
seq	Sequence number for dynamic objects.
daddr	Generic data address pointer.
C field name	Name of a field name in a C record structure. This name is stored in a text string.
C type	A C type definition. This name is stored in a text string.

INVOCATION INTERFACE:

Procedure **Description**
    Parameters **Description** (Type, IN/OUT, etc.)

ut_maxof — Return the maximum of two numbers.
- first — int, IN, First number.
- second — int, IN, Second number.
- [return] — int, OUT, Maximum of first and second.

ut_offsetof — Return the byte offset of a field within a struct.
- type — C type, IN, Struct declaration.
- id — C field name, IN, Field name in type.
- [return] — int, OUT, Byte offset of id in type.

ut_fieldsize — Return the size of a field in a struct.
- type — C type, IN, Struct declaration.

id	C field name, IN, Field name in type.
[return]	int, OUT, Size of id field in type.

IMPLEMENTATION FILES:
UT_Utility.h

IMPORTED ELEMENTS:

Name	**Type**	**Module**

None.

NOTES:

None.